SIREN: A SQL-based Implementation of Role-based access control (RBAC) for Enterprise Networks

by

Arundhati Singh

S.B., Electrical Engineering and Computer Science (2001)

Massachusetts Institute of Technology

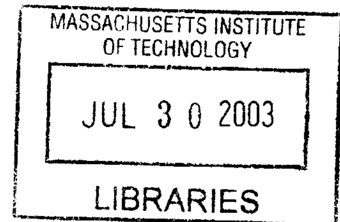
Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

July 26, 2002



© Massachusetts Institute of Technology, 2002. All Rights Reserved.

Author.........................

Department of Electrical Engineering and Computer Science

July 26, 2002

Certified By.................................. ....

Sanjay E. Sarma

Associate Professor of Mechanical Engineering

Thesis Supervisor

Accepted By.......................

Arthur C. Smith

Chairman, Department Committee on Graduate Theses

BARKER

SIREN: A SQL-based Implementation of Role-based access control (RBAC) for Enterprise Networks

by

Arundhati Singh

Submitted to the

Department of Electrical Engineering and Computer Science

July 26, 2002

In partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Increasingly, enterprises are collecting and aggregating key business information in distributed database networks. Doing so allows data about the company's sales, organizational structure, logistics practices, pricing, customer base and more to be made available online. With the capability to dynamically query this data, enterprises can automate and streamline many important business processes. Due to the business critical and potentially sensitive nature of this information, the data must be adequately protected from inappropriate access. In this thesis we present SIREN, an access control framework for the unique design constraints of these distributed enterprise environments. Our implementation provides maximum expressive power for privilege specification and the guarantee of system-wide security policy coherency in such database networks by unifying the concepts of Role Based Access Control (RBAC) and query rewriting. We also examine several key considerations in actually deploying SIREN within an enterprise environment.

Thesis Supervisor: Sanjay E. Sarma

Title: Associate Professor of Mechanical Engineering

ACKNOWLEDGEMENTS

Foremost, I owe my thanks to Sanjay Sarma for placing his faith in my abilities. As an advisor and mentor, he has constantly encouraged me to think in different ways and push the envelope of creativity. His keen insight, patient guidance, boundless energy and humor have made working with him a great opportunity and true pleasure.

Many thanks also to all the other members of the Auto-ID Center whom I have had the privilege of knowing this past year. In particular, Dan Engels was a wonderful source of practical advice about research and life in general. I am also very grateful to Prasad Putta and his colleagues at OATSystems, who have so often given generously of their time and knowledge to further my research along.

To my labmates Amit Goyal, Junius Ho and Nosh Petigara – Over this past year, I have come to value your friendship greatly. You guys were simply the best part of coming into lab every day. Thanks for all the laughter and camaraderie, the intellectual conversations and the silly ones, and the research advice and suggestions.

I owe many thanks to the friends who have given balance to my life throughout my time at MIT. In particular, my heartfelt gratitude goes to Jaspal Sandhu for his unfailing support, for listening and for sharing, and for a smile on cloudy days. Thanks also to Shounak, Erick, Liz, Mona, the Rearhousers, and my fellow Warehousers, especially Priya and Satwik. You have all truly given me a home away from home.

In this, as with every other endeavor I have undertaken, my family has always supported me with so much love and understanding. Any real credit for my successes has always been due in large measure to them. To Dad, Mom, Achal and our three beloved doggies – all my love and deepest appreciation.

TABLE OF CONTENTS

1 INTRODUCTION 7

1.1 The AutoID Center 7

1.2 AutoID Technology 7

1.2.1 Tags and Readers 7

1.2.2 Electronic Product Code (EPC) & Physical Markup Language (PML) 9

1.2.3 Savants 10

1.3 Motivation 12

1.4 Design Considerations 13

2 ACCESS CONTROL PARADIGMS 15

2.1 Mandatory Access Control (MAC) 16

2.1.1 MAC Model 16

2.1.2 Implementing MAC 18

2.1.3 Evaluating MAC 19

2.2 Discretionary Access Control (DAC) 20

2.2.1 DAC Model 20

2.2.2 Implementing DAC 21

2.2.3 Evaluating DAC 24

2.3 Role-Based Access Control (RBAC) 25

2.3.1 RBAC Model 26

2.3.2 Implementing RBAC 28

2.3.3 Evaluating RBAC 31

2.4 Selecting a Model 32

3 IMPLEMENTING RBAC: SIREN 34

3.1 Concept 34

3.2 Implementation 38

3.2.1 Leveraging Database Views 40

3.2.2 Session Activation and Multiple Roles 44

3.2.3 Savant Integration 47

3.2.4 Completeness 48

3.3 Summary 50

4 DEPLOYING SIREN 51

4.1 Role Engineering 51

4.2 Authentication 53

4.2.1 Server-Pull Architecture 53
4.2.2 User-Pull Architecture 54
4.3 Privilege Management 56
4.3.1 Local Domains 56
4.3.2 Distributed Queries 59
4.3.3 Inter-organization Access Privileges 61
4.4 Integration With Other Access Control Systems 63
5 CONCLUSIONS 64
5.1 Evaluation 64
5.2 Further Work 66
REFERENCES 69
APPENDIX: PROTOTYPE DETAILS 72

LIST OF FIGURES

Figure 1.1: Tagging an object 8
Figure 1.2: Electronic product code partitions 9
Figure 1.3: Reader reads data and stores it to savant 10
Figure 1.4: Savant hierarchy 11
Figure 2.1: A sample multilevel relation 18
Figure 2.2: Starships relation as it appears to users with Confidential access 19
Figure 2.3: Starships relation as is appears to users with Secret access 19
Figure 2.4: Schema for System R's ACL relation 23
Figure 2.5: Relationship between NIST RBAC models 27
Figure 3.1: Query rewriting as suggested by Stonebraker 35
Figure 3.2: Excerpt from conceptualized Role-Privileges relation 36
Figure 3.3: The same query is rewritten differently for different users 37
Figure 3.4: Modified query rewrite architecture in SIREN 38
Figure 3.5: Creating a role view to restrict access to Products table 41
Figure 3.6: How rewriting occurs in our prototype 42
Figure 3.7: Sample rules for update behavior of Products_SalesClerk role view 43
Figure 4.1: Server-Pull Architecture 53
Figure 4.2: User-Pull Architecture 54
Figure 4.3: Local domains allow decentralized administration 59
Figure 4.4: External domain exposes inter-organization services 61
Figure 4.5: Role server certificates can be used by multiple RBAC modules 63
Figure A.1: Screen shot of web interface to rewriter 73

CHAPTER 1

INTRODUCTION

This thesis presents SIREN, an access control framework for enterprise data networks. The ideas presented here were developed within the larger context of the current research at the Auto-ID Center, an international research program headquartered at the Massachusetts Institute of Technology. In this chapter, an overview of the Auto-ID Center and its research is provided to understand the need for such an access control framework and its key design requirements.

1.1 The Auto-ID Center

Recent advances in radio frequency identification (RFID) have allowed us to begin envisioning a world that is interconnected via embedded wireless technology. The Auto-ID Center is creating the standards and technologies that will transform ordinary, every day objects into "smart products" that can communicate with each other, with businesses and with consumers. This network of wirelessly interconnected physical objects will revolutionize our interactions with even the simplest of supermarket items. Every such object will be uniquely identified by Auto-ID technologies and relevant information about each will be stored and retrieved on demand. An immediate, significant application of this research is to provide greater automation and intelligence capabilities to global supply chains [14], which account for approximately 75% of product cost [23].

1.2 Auto-ID Technologies

1.2.1 Tags and Readers

The first step in networking physical objects is to replace traditional UPC barcodes on product packaging with very inexpensive embedded chips, called *tags*. An important

research goal the Center is working towards is developing RFID tags with a production cost of only 5¢ each [29]. Such a tag would necessarily be very simple in design, holding only an identifying *electronic product code (EPC)* number, and a small amount of logic for communication via the chip's RF antenna.

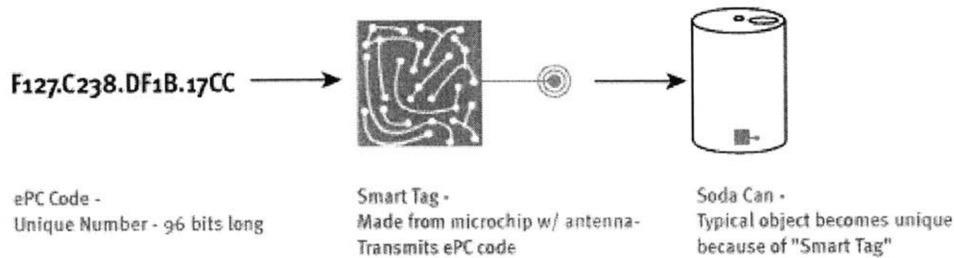


Figure 1.1: Tagging an object

A *reader* is a device that can repeatedly poll any tags within its read range every second. Read ranges vary among different reader models and transmission frequencies, but a typical reader might be used to monitor all the items on several supermarket shelves. A key advantage of RFID technology is that readers are not line-of-sight technology, so they do not require that objects be manually scanned by a human operator to be registered. A reader will simply register the EPC information of any tag in its read range. Based on the different RF readers that recorded a particular chip's presence within their field at different times, the location history of an object can be tracked as it is moved around a warehouse or shipped between distribution locations. Readers may also take other environmental records, such as the room temperature at the time of a particular set of reads. In this way, other information pertinent to an object is also recorded in its traceable history.

1.2.2 Electronic Product Code (EPC) and Physical Markup Language (PML)

Similar in concept to an Ethernet MAC address, the Electronic Product Code (EPC) is a 96-bit number that can uniquely identify over 1 billion tagged items [5]. Each tag stores its own unique EPC value in on-chip memory. The current proposal is to break the EPC down into the following 4 partitions:

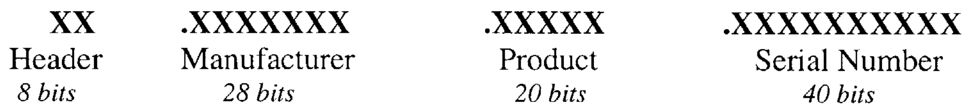


Figure 1.2: Electronic Product Code Partitions

The Header bits are used for version control, so that any future changes to the partition structure can be transparently adopted. The Manufacturer bits uniquely identify the particular manufacturer for the product with this EPC. These bits tell us where to look for general information related to the product with this EPC. The Product code identifies a class of items, and the Serial bits uniquely identify distinct items within that product class. The Serial bits of the EPC number are what allow us to distinguish between two different instances of the exact same product. This capability is another key advantage of EPC technology over traditional barcodes, which can only identify items down to product-level, not individual instance-level, granularity.

There are many other features of an object that we may be interested in recording, apart from those captured by the EPC partitions. For instance, we may want to know an object's size, weight, position, color, price or temperature at various times. The *Physical Markup Language (PML)* is an XML-like language that provides a general, standardized format for describing the characteristics of any physical object [6]. PML is the lingua franca of the Auto-ID System, providing objects, consumers and businesses a common

format to represent information of interest. Any two entities expect to exchange information about an object's history or features in PML representation, though either entity could then choose to transform that information to its own proprietary format.

1.2.3 Savants

The functionality provided on tags is limited so that they can be manufactured cheaply enough to be a feasible alternative to barcodes. Since the tags themselves are so simple, a supporting network infrastructure must be able to intelligently process the large quantity of simple data snapshots collected from various readers. The PML data captured by the readers is sent to a network of servers known as *savants* [25]. The savant network is the backbone of the Auto-ID infrastructure. Within a savant, incoming PML data is parsed and stored into a relational database.

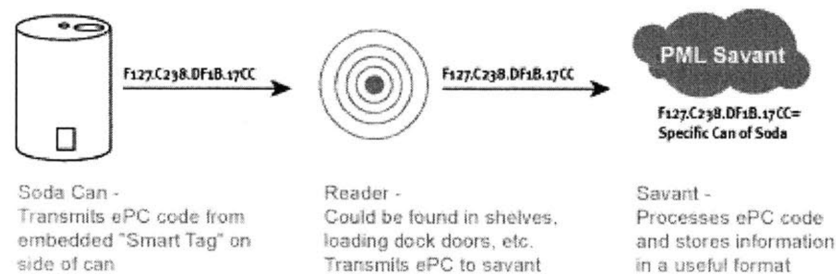


Figure 1.3: Reader reads data and stores it to the savant

These savants can be thought of as Lego™ blocks, which can be stacked together in various configurations. A tree structure of savants, illustrated below, provides different depths and breadths of information at each level. In this typical configuration for a retail organization, each store-level savant holds detailed inventory and sales information for that day at that location. In contrast, a regional-level savant may only store weekly data, but aggregated over all stores in that region. Finally, the national-level savant would likely aggregate data into a monthly granularity.

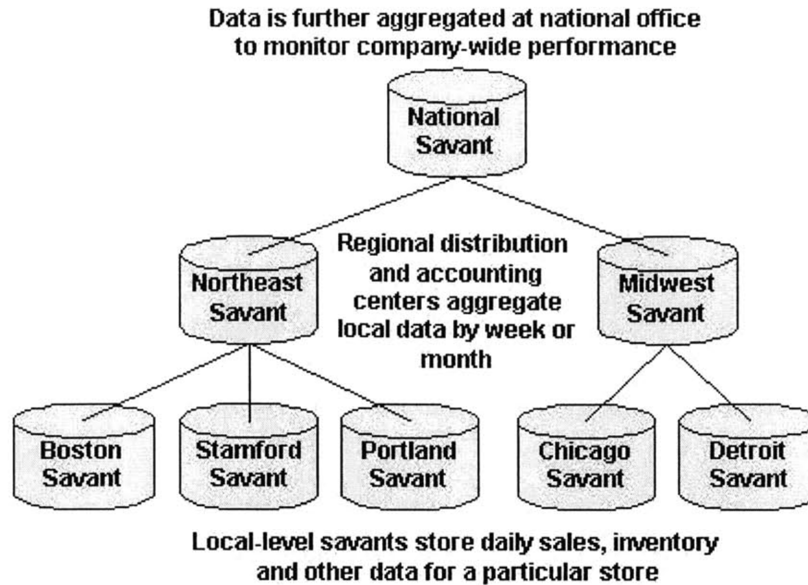


Figure 1.4: Savant hierarchy

Hierarchical arrangement of savants is an intuitive organization for PML data, corresponding to the structure of the organization. However, to be useful this data must be available to be monitored, aggregated or queried by other applications. Typical applications include real-time monitoring programs such as theft-detection for store shelves, online analytical processing (OLAP) applications for inventory and sales trend analysis and batch processing programs to aggregate and transfer data up along the savant hierarchy. These programs access the information in a savant's database via the *Task Management System (TMS).* The TMS is a software component within each savant.

The TMS exposes a Simple Object Access Protocol (SOAP) interface that allows applications to run tasks on the savants. Tasks are customizable and can be created to perform any action on the PML data, or to interface savant data with other applications in an enterprise's network. The TMS supports tasks that cascade down the savant hierarchy. For instance, an SQL query task against a national-level savant may require data from each regional level savant. In response to such a task, the TMS component of

the national-level savant would automatically spawn sub-queries to each regional savant and compile the results. Recurring tasks can be set to automatically repeat on some specified schedule.

1.3 Motivation

Much of the data collected and aggregated by the savant database network is potentially sensitive business data, providing insight into a company's sales, organizational structure, logistics practices, pricing, customer base and more. Ensuring the security of this information is absolutely vital to the acceptance of the Auto-ID system. Data security incorporates three main goals: protection against imposters, protection against eavesdroppers and protection against unauthorized actions by legitimate users. Therefore a truly secure system must provide: authentication to verify the identity of a user, encryption to protect the privacy and integrity of user transactions, and access control to limit the actions allowed by different users.

Authentication and encryption are lower level problems in that providing these services on a communications channel requires no knowledge of the specifics of a requested transaction. Any message sent across the channel is encrypted and signed in the same way, regardless of the actual message contents. Access control, on the other hand, inherently requires examining the contents of each message, analyzing whether the user is allowed to make that particular request.

This thesis focuses on developing an efficient and flexible framework to address the access control problem for enterprise databases in such a large, distributed environment. A protocol for authentication and encryption is assumed to be present and available as a service.

1.4 Design Considerations

Database access control is a broad and widely researched area. However, there are specific challenges and unique requirements that constrain the realm of feasible solutions for a system of the scale envisioned by the Auto-ID Center. We attempt to find a good solution within our constraints, with the expectation that such a solution will generalize well to other large, hierarchical data networks. In evaluating any solution, the following factors are key considerations.

Performance: The PML data stored in the savants is often business critical information. Ideally, access control should affect the speed of queries and other tasks as minimally as possible.

Scalability: The sheer number of users and amount of data for which access must be monitored is one of most challenging aspects of this problem. An effective solution should support compartmentalizing access control in domains, each of which regulates a smaller subset of the users and data.

Flexibility: Any solution must be easily adaptable to the needs of disparate organizations and capable of expressing a range of access control policies. For a single element of data, different types of access, such as read, modify or delete permission, may be required for different users. PML data is also more complex because of it is 3-dimensional in nature, in the sense that all data has three attributes:

- Datatype: such as EPC data, Sales data, Environmental/Sensor data, etc.
- Topology: the location in the savant network (local, regional, national, etc.)
- Time Range: can range from minutes to years

The access control solution should support policies that define any arbitrary region in this 3-dimensional space as permissible for a particular user and type of access.

Application Agnosticism: The access control solution must provide a suitable level of service to a variety of different applications: ah-hoc queries, batch processing or online analytic processing programs.

Platform Independence: Many large organizations operate in heterogonous computing environments, with different protocols, database software, operating platforms and hardware resources. To be viable at an enterprise level, a suitable access control solution must be as platform independent as possible.

Ease of Administration: Access control for a large enterprise may potentially involve monitoring privileges for thousands of users. A feasible solution must ease the administrative burden of creating, reviewing and updating privileges as much as possible.

CHAPTER 2

ACCESS CONTROL PARADIGMS

A variety of access control frameworks have been proposed by security researchers over the years. The most widely accepted models in the literature are Mandatory Access Control, Discretionary Access Control and Role Based Access Control. In this chapter, we review the major concepts behind each of these models and existing implementation strategies for each. The different models all share a basic conceptual framework in which *users* require *privileges* to perform actions on system *resources*. System resources are "containers of information", and might typically be operating system directories and files, or database relations and tuples.

The goal of access control is to protect these system resources from improper actions by users. The definition of "improper actions" is derived from an organization's *security policy*, which defines the principles on which access to resources is granted or denied to different users. The existence of a coherent security policy is a prerequisite to any meaningful data protection, as access control mechanisms are merely tools set up to enforce these policies. There are two important constraints that any viable security policy must encompass to ensure data integrity:

Separation of Duties: Roles and responsibilities are divided so that a single user does not have the privileges to compromise critical procedures. Separation of Duties constraints are usually specified by administrators as pairs of job functions or privileges that can not be simultaneously granted to a user. The access control system must then be able to enforce these constraints by ensuring that no user is assigned a privilege or group pair in violation of them.

Principle of Least Privilege: Each user shall have exactly as much access as necessary to fulfill his job function, but no additional privileges. In practice, implementing the Principle of Least Privilege translates to giving system administrators enough flexibility and expressive power to set fine-grained access control policies on different data for different users.

Therefore an important consideration in examining access control models is whether they support enforcement of these constraints effectively and efficiently.

2.1 Mandatory Access Control (MAC)

The Mandatory Access Control (MAC) model is based on the lattice framework and information flow models of Denning [8], and was first formalized by Bell and LaPadula [4]. Sandhu presents a summary of MAC concepts as well as several more recent extensions of this framework [27].

2.1.1 MAC Model

The principle behind the MAC model is that every user and every resource is assigned to some pre-defined *security class*. The security class to which a user is assigned defines that user's privileges. A security class is a combination of a hierarchical *sensitivity level* and any subset of non-hierarchical *categories* that define what domains the clearance level applies to. The hierarchical nature of MAC sensitivity levels is why this approach is sometimes referred to as the multilevel security (MLS) model. Whenever information is transferred from one resource to another, or between a user and a resource, the information is said to be *flowing* from the security class of the original resource or user to the security class of the requesting resource or user. The MAC model requires the formal

definition of a precise *information flow policy* that regulates the information flow between any two security classes.

As a concrete example, MAC systems are typically found in defense-related applications, where the security class is composed of a military clearance (Confidential, Secret, Top Secret, etc.) as the sensitivity level plus a set of departments or projects as the categories in which the user has this clearance. Any file or other system resource is classified as belonging to one of these security classes, and every user is also assigned to a security class. A user can access any resources in his security class. A resource is in a user's security class if they share the same sensitivity level, and the user has category permissions for the department to which the resource belongs. The set of resources outside his own security class that a user may access is determined by the information flow policy. A basic information flow policy in a MAC system would incorporate the Bell-LaPadula restrictions [4]:

Simple Security Property: A user can only read a resource if the security class of the user is greater than or equivalent to the security class of the resource. This means a user can read any information of a lower or equivalent security class. However, he can not access information that is more highly classified than his own clearance level permits.

****-Property:*** A user can only append a resource if the security class of the user is less than or equivalent to the security class of the resource. Note that this restriction means that users can not write to resources at *lower* classification levels. While this may seem unintuitive because presumably the user's clearance level ensures his trustworthiness, this is an information flow restriction intended to prevent highly classified users from accidentally writing highly classified information in less classified resources. That

accidentally written information would then be viewable by other users who should not have had access to it. However, the *-Property does allow a user to append information to files of a *higher* security class, despite the fact that he can not read the contents of the rest of the file. Such capability can be useful in classified systems, where intelligence analysts can append notes into classified files for their supervisors, without having read access to the rest of the classified file's contents. However, other systems find it more intuitive to use a modified *-Property in which users can only write to resources within their own security class. This prevents a user from appending contradictory information to a file because he was unable to access its other contents.

2.1.2 Implementing MAC

The actual implementation of the MAC framework in database systems is based on the notion of *multilevel relations* [15, 28]. A multilevel relation is similar to a normal database relation, except that the value of each field in a tuple is actually a pair: (field data, security class). The example Starships relation below contains information from three security classes: Confidential [C], Secret [S] and Top Secret [TS]. Such a relation is multilevel in the sense that it appears differently when queried by users with different security levels. The security class of the fields requested by a query must be authorized by the security policy to the requesting user's security class.

Starships Relation		
Starship	**Objective**	**Destination**
Enterprise [C]	Exploration [C]	Talos [C]
Voyager [C]	Spying [S]	Mars [TS]

Figure 2.1: A sample multilevel relation

If a user with Confidential access requested to read this relation, they would see output shown in Figure 2.2, whereas a user with Secret access would see the output shown in Figure 2.3. A user with Top Secret access would see the full table, as it appears above.

Starships Relation		
Starship	**Objective**	**Destination**
Enterprise [C]	Exploration [C]	Talos [C]
Voyager [C]	NULL	NULL

Figure 2.2: Starships relation as it appears to users with Confidential access

Starships Relation		
Starship	**Objective**	**Destination**
Enterprise [C]	Exploration [C]	Talos [C]
Voyager [C]	Spying [S]	NULL

Figure 2.3: Starships relation as it appears to users with Secret access

2.1.3 Evaluating MAC

The key advantage of the MAC model is that it ensures consistent enforcement of the access control policy system-wide. This is due to the centralized nature of MAC; the information flow policy and all security class assignments are centrally administered with no access control policy decisions delegated to users. Among the frameworks we will discuss, MAC systems generally rate higher on the Department of Defense's Trusted Computer System Evaluation Criteria [33].

A disadvantage of the MAC model is that it is not as flexible as other approaches in terms of the range of access policies that can be expressed within this framework. The centralized nature of MAC administration, which provides much of the model's security, also makes it unsuitable for many non-military environments. In a typical commercial enterprise, there are often no uniform clearance levels that are used throughout the

organization. Even if users can be classified into a clearance hierarchy, it is usually more difficult to fit system resources into such classifications. Additionally, it is often more desirable and practical for the users in various departments to determine the appropriate access restrictions on their resources and privileges for their staff members, rather than a having centralized security administrator assign security classes. In large, non-military organizations, the overhead costs to centrally administer a MAC system would be considerable. Finally it is possible to imitate a MAC-like security policy within newer, more flexible access control frameworks by setting up appropriate constraints [19] and this is often a better alternative than a strict MAC implementation for commercial enterprises. In practice, MAC remains mostly limited to military and defense systems.

2.2 Discretionary Access Control (DAC)

Discretionary Access Control (DAC) is the most prevalent model for regulating data access in civilian businesses and non-military organizations. The term DAC is applied as an umbrella term to a group of related approaches that share common principles. The National Computer Security Center provides an overview of several variations of the DAC model [11].

2.2.1 DAC Model

The central concept of DAC is that every resource has some user who "owns" it. This user can then set the permissions for others to access that resource at his discretion. DAC completely contrasts the MAC model in that the overall access policy is decentralized among all the resource owners. One feature of DAC models is the concept of delegation, whereby a user who does not own a resource but has some permissions for that resource

can assign whatever permissions he has to another user, without the intervention of the resource owner.

One important point on which various DAC models differ is the delegation of privilege-granting authority. *Hierarchical control* gives ownership and delegation authority to the head of each organizational unit in the hierarchy. That unit's head user then handles delegation of access granting privileges to various resources to the appropriate sub-groups of users within that unit. In this way, access control is partitioned into hierarchical domains, easing administrative overhead and allowing policy decisions to correlate to the organizational power structure. A more flexible approach is *Laissez-Faire control*, which allows any user with access granting privileges on a resource to delegate that access granting privilege to *any* other user. In contrast, *Strict Ownership* control means that only the creator of a resource can set access permissions on an object and can not delegate his permission granting capabilities to any other users.

2.2.2 Implementing DAC

Although there are many variations in details among DAC implementations, most systems fall in one of two general categories: *Access Tokens* or *Access Control Lists*.

2.2.2.1 Access Tokens

The idea behind Access Tokens, sometimes called Capabilities, is that when a user wishes to access some resource, he must present a token of proof to verify his privileges on the resource. The token is an identifier that specifies the resource for which it is valid, and the access permissions which its possessor is entitled to for that resource. When a user attempts to access a resource, the system will ask for the requisite token before granting access. Tokens may be passed from one user to another, and the contents of a

token can not be modified without the intervention of the system's token granting facility. In this type of system, a user has a set of tokens granting him access to various system resources. Each user's tokens are stored in a special file that is protected by encryption or other hardware or software safeguards.

In theory, access tokens could be useful in creating systems that adhere to the Principle of Least Privilege and Separation of Duties, because each user can be given only those tokens which he requires and no other tokens, and it is easy to check whether the set of tokens possessed by a user gives him unchecked permissions on any critical procedures. Tokens can also result in faster processing since the access control system needs only quickly check the user's requested action against the user's token, rather than a potentially expensive permission table look-up.

In practice, a token based systems are rare because in these systems it is difficult to list all the users who posses a particular token, making it difficult for administrators to get a clear picture of the overall security policy. Also, the question of token revocation is tricky when an object is deleted, or if a user needs to revoke all delegated tokens. One alternative to revocation is to simply have tokens expire after a fixed interval. However, this approach is still vulnerable to temporary *privilege creep*, where a user retains old permissions after changing job functions because his old tokens are still valid for some time. Redell [22] provides a more complete discussion of token revocation issues.

2.2.2.2 Access Control Lists

Access Control Lists (ACLs) are the inverse approach of Access Tokens. Instead of associating a collection of resource permissions with each user, an ACL is a list of users associated with a particular resource. For each user in the list, the ACL specifies what

actions the user may perform on the resource. The owner of a particular resource has edit privileges on the resource's ACL.

An important feature of ACLs is support for user groups [1]. Rather than listing permissions for many individual users, an ACL may contain an entry detailing permissions for a group name, and any users in that group will then automatically have the associated permissions. The use of groups allows ACLs to scale to a large number of users without creating files of unwieldy size. This also eases the administrative burden of maintaining ACLs, since simply removing a user from a group will automatically remove his access privileges on any ACLs which list that group.

In databases, ACLs are actually implemented as separate relations that store information about user access privileges for all other relations. This relation-based ACL approach was first suggested by Griffiths and Wade for the System R database system [13]. In their paper, they describe a SYSAUTH relation to store this ACL information.

SYSAUTH TABLE		
Column Name	**Data Type**	**Significance**
UserID	String	Database username or groupname
TableName	String	Another relation in the database; this row defines the specified user's privileges on this table
Type	Character	"T" = table; "V" = view
READ	Boolean	Indicates user can read specified table
INSERT	Boolean	Indicates user can append specified table
DELETE	Boolean	Indicates user can delete from specified table
UPDATE	Boolean	Indicates user can update specified table
GRANT	Boolean	Indicates user can grant any privileges he possesses to another user

Figure 2.4: Schema for System R's ACL relation

Figure 2.4 illustrates the SYSAUTH relation schema. Similar approaches are still used in many popular modern databases, such as the open-source PostgreSQL database, where access information for all other relations is stored in the PG_CLASS relation [21].

Note that an "ACL relation" differs from the multilevel relations described earlier in that access control information is not stored *within* each relation. Rather, the access control information is stored in a *separate* relation, such as SYSAUTH. Another difference is that since "ACL relations" are DAC implementations, grant permissions are supported so that users can modify the ACL information by granting some of their permissions to other users, which is not supported in MAC systems. However, multilevel relations can support a finer granularity of access control (attribute/field level), whereas ACL relations can only support access control at the relation or column level.

2.2.3 Evaluating DAC

The major benefit of DAC is that it affords much more flexibility than the MAC model. The approach naturally supports different domains of access, where access policy can be set at resource-by-resource level granularity by users within those domains. This eases the burden on system administrators, who no longer have to set access policy to meet the disparate needs of various units across a large enterprise. Presumably, in a DAC approach, the access policy for a resource is set by the user who is best able to judge the minimal subset of other users who need access to the object. Every resource can have slightly different access permissions in the DAC model, whereas in the MAC model each resource must be force fitted into one of the pre-existing security classes. The resulting security class assignment may give access to a larger subset of users than the minimal

subset of users who actually require access. In theory, this means that the Principle of Least Privilege should be more easily implemented in DAC systems.

ACL-based approaches are the most widely implemented DAC solution and offer the distributed policy-making ability required by many non-military organizations. Though these ACL implementations are generally considered a better access control alternative than token-based systems, they still suffer from the certain fundamental problems with the DAC framework.

The major drawback of the DAC model is the difficulty of monitoring whether users are setting appropriate access permissions on resources within their control. Because the access decisions are distributed throughout the organizations, it becomes difficult to ensure that some minimal access control policy is enforced system-wide. In particular, it is hard for system administrators to ascertain whether the Separation of Duties constraint holds when many different users own critical resources and are setting access permissions on those resources independently of each other.

2.3 Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC) is a relatively new framework that has been attracting attention from the research community and security vendors in recent years. It is intended to better suit commercial requirements than the MAC model does, while still providing guaranteed enforcement of a coherent security policy across the enterprise. The RBAC framework incorporates the MAC concepts of centralized policy definition and centralized (as opposed to user) ownership of resources. It also draws upon the concept of user groups and ACLs from DAC. The research community has only recently begun to converge on a standardized, formal definition of RBAC [10].

2.3.1 RBAC Model

RBAC is centered around the concept of *roles*. A role is an access control category that maps to a particular job function in an organization. In a hospital, typical roles might include Surgeon, Physician, Resident, Nurse, Clerk, Pharmacist and so on. Users can then be assigned to one or more of these role categories based on their job functions. Each role is defined to have certain *permissions*, which are available to all users in the role. One difference between roles and MAC security classes are that a role's privileges are not defined by classification of system resources into different classes. The question of how privileges are defined is left to the implementation. By not classifying system resources into security classes, slightly different access policies can be set for each resource and role rather than specifying a monolithic information flow policy for all resources in the same class. Role-based permissions are attractive because they provide a layer of abstraction between users and privilege assignment. In most organizations, the security policy defining privileges for any job function changes much less often than the assignments of users to various job functions.

The proposed standard for RBAC [10] submitted by the National Institutes of Standards and Technology (NIST) defines 4 levels within the RBAC model. The first level, $RBAC_0$, represents the simplest feature set required for a system to be considered an RBAC implementation. $RBAC_0$ requires the ability to make many-to-many user/role assignments and role/permission assignments. It also includes the concept of a session, where a user must log-in and activate some subset of his roles before making access requests.

A higher level of functionality in this model is $RBAC_1$, which supports role hierarchies. The hierarchy is a seniority ranking of roles in which more senior roles automatically inherit the permissions of their juniors. Continuing with the hospital example, some set of actions can be defined for the role of Resident. To avoid redundancy, the role of Doctor does not have to redefine all these privileges, but can instead inherit them from Resident. Additional privileges available only to the Doctors can then be defined. $RBAC_1$ is an administrative enhancement that simplifies role/privilege assignment.

$RBAC_2$ supports for constraint setting and checking. *Static* checking of a Separation of Duties constraint makes certain that no user is assigned to a set of roles that would violate this constraint. Any attempt to assign a user to two conflicting roles would fail. *Dynamic* constraint checking allows such conflicting role assignments to be made to a user, but enforces Separation of Duties at session initialization by never allowing two conflicting roles to be activated in the same session.

$RBAC_3$ support combines the features of $RBAC_1$ and $RBAC_2$, and by transitivity $RBAC_0$. The relationship between the models is shown below.

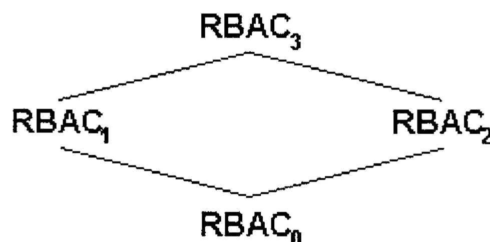


Figure 2.5: Relationship between NIST RBAC models

Providing the features of the higher-level RBAC models is often very difficult in existing decentralized DAC frameworks.

2.3.2 Implementing RBAC

Many existing RBAC implementations are research prototypes and do not satisfy the requirements for all 4 levels of the NIST RBAC standard. Security vendors have only recently started to incorporate parts of RBAC framework into their products. The existing implementations tend to fall into one of 2 categories: ACL-based or Object-Oriented.

2.3.2.1 ACL-based Approach

Several existing implementations of RBAC systems use ACLs under the sheets as the actual privilege authorization mechanism. Users are isolated from the details of maintaining these ACLs by application-level code that manages them and checks that no user interactions with the application produce ACLs in violation of the RBAC constraints.

Ferraiolo, *et al.* present such an implementation for a web server, where access to any URL is governed by an ACL for that URL [9]. The ACL contains a list of roles that have the requisite permissions. The web server is configured so that a CGI script maps any URL request to the appropriate ACL. User sessions are also implemented via an ACL mechanism, which lists the possible roles that a user may choose to activate at session initialization. Once a user has established a session on the server, any request made by the user results in checking the URL's ACL to see whether it includes any of the roles in the user's ARS ACL. Each URL's ACL just has a list of user names in that ACLs can only be modified via the Administrative interface, which enforces certain consistency checks.

A more sophisticated but conceptually similar approach is outlined in [26]. In this implementation, every time a user opens an application, that application queries an access control server for information about the user's privileges for that particular application. The access control server looks up the user's privileges in an ACL and returns the information to the requesting application. The applications are then responsible for ensuring that the user is not allowed to execute any privileges beyond those returned by the access control server. Similar functionality can also be implemented by leveraging the security features built-in to the Java programming language, as detailed in [12].

The question of what differentiates RBAC roles and role/privilege assignments from DAC user groups and ACLs is a common point of confusion. The two models actually provide equivalent functionality in terms of the security policies they can express when considering simple RBAC implementations [2]. These implementations take the view that RBAC is simply a set of guidelines and tools for setting up and maintaining groups and ACLs in a structured way across the entire system.

2.3.2.2 Object-Oriented Approach

Barkley suggested a different implementation strategy for RBAC systems that leverages object-oriented concepts [3]. Each role is represented by its own class, and the methods defined within that class represent the actions authorized for that role. While ACL-based RBAC systems simply provide a better framework to administer ACL functionality, an object-oriented implementation actually provides more flexibility in the types of policies that it supports. This is because within each object's method, conditional checks can be performed or data can be filtered to provide arbitrarily complex, fine-grained control over what permissions are allowed and under what circumstances. As an example, a

getRecordData(patientID) method in the role class *Nurse* might include code to perform the following checks:

- Is this nurse currently marked as on-duty in the scheduling program?
- Does the patientID correspond to a patient in the same department as this nurse is assigned to?
- Filter the returned record data so that no billing/financial information from the patient record is shown to the nurse.

However, the *getRecordData(patientID)* method in the role class *Financial Clerk* could provide different checks and data filters for the same patient records. It would be difficult to support such complex access restrictions on top of an ACL-based implementation, whereas in an object-oriented implementation, this involves only a few lines of code for each method.

In [16], Neumann and Strembeck describe an object-oriented RBAC implementation using the extended Object Tcl (XOTcl) scripting language. A single instance of a *RoleManager* class is the application interface to this RBAC system. This *RoleManager* object then manipulates relationships between *User*, *Role* and *Permission* objects. This approach differs slightly from Barkley's proposal in that it creates separate *Permission* objects, rather than defining permissions within Role class methods. Different subclasses of *Permission* objects can be created for different types of system resources. Each such object can be coded to represent as coarse or fine grained a privilege on the resource as desired. This implementation takes advantage of certain dynamic class inheritance features of XOTcl known as *per-object mixins* [18] to register different *Permission* objects with different *Role* objects, and different *Role* objects with different *User* objects. The use of per-object mixins, which allow users to modify the class inheritance order on an object-by-object basis, is also used to support the formation

of role hierarchies. Methods for privilege review and specification of constraints of privilege-role and role-role relationships are also provided by the *RoleManager* object.

Object-oriented implementation strategies provide fine-grained control and support a wider range of security policies than implementations that rely on ACLs under the sheets. However, there are a variety of drawbacks that make this approach less desirable for large enterprise networks. The primary concern is performance; in [16] the authors note that the type of XOTcl or Java based object oriented implementations they suggest are better suited to small or mid-sized environments. Another issue with object orientation implementations is maintenance. Changes to the permissions require the system administrators to modify the source code for the relevant classes and redeploy them, which may be impractical in large organizations that already place many demands on their system administrators. Administrative effort is also required to verify the correctness of the permission code against the security policy; this again requires a system administrator to step through the code in each of the relevant class files.

2.3.3. Evaluating RBAC

The real value of the RBAC framework is not based solely on its ability to provide some additional functionality as the extent of additional functionality varies based on the implementation strategy. Rather, the real advantage that the RBAC framework provides is ease of administration and policy verification. This is an important issue for large enterprises where accurately maintaining access control information for thousands of users and resources is a major factor in overall security. RBAC allows an organization's existing structure and security policy to be directly mapped to the access control

mechanism. In effect, it is a methodology to reduce the semantic gap between the description of the policy and the underlying implementation mechanism.

There is no need to determine what groups, ACLs or objects must be created to correctly implement the security policy, because the structured framework of RBAC provides this information intuitively. If a user changes positions within the organization, a system administrator only needs to make a change in one place: the user's assigned roles. There is no need to delete the user from multiple lists all over the system. Privilege review and modifications to the security policy are easy because they can be done simply by checking each role's permissions, independently of any other roles. Overall, the RBAC approach is a good compromise between DAC's flexibility and MAC's security.

2.4 Selecting a Model

For the needs of enterprise environments, Role-Based Access Control seems to be the best alternative as a fundamental model. In fact, RBAC systems can actually be configured to imitate either MAC or DAC policies [19]. We note that the RBAC model does not specify the nature of privileges. This key detail varies widely among implementation strategies and greatly affects their expressive power, performance and ease of administration. One drawback of both the ACL-based and object-oriented implementations discussed is that they require system administrators to define a finite set of methods or actions that can be performed on system resources. While this may be feasible when defining an API for standard applications to utilize, it will generally limit the ability to run very complex database queries. However, key applications such as online analytic processing (OLAP) to identify data trends often require complex multi-

level queries. Another drawback is that defining a fixed API to access data also prevents users from running ad-hoc queries directly in SQL against the database. To run ad-hoc queries, the users must be aware of the API calls that are provided, meaning that the access control implementation is not transparent to the end-user.

In the next chapter, we present SIREN, an implementation strategy that provides more flexibility than traditional ACL-based access control but that we believe is easier to administer than object-oriented approaches. It also addresses the access control needs of OLAP applications and ad-hoc queries while maintaining transparency of the access control implementation to the end user.

CHAPTER 3

IMPLEMENTING RBAC: SIREN

As discussed in the previous chapter, Role Based Access Control is a useful model for thinking about access control in enterprise environments because it maps intuitively from the organization's written security policy. In designing an effective implementation strategy for RBAC in database networks, we emphasize transparency and flexibility: the end user should not need to be aware of any details of the access control application or be constrained to use a particular API when expressing data queries. In other words, a user or application should be free to run any SQL query without regard to permissions and assume that an underlying service is prepared to handle the query appropriately. In this chapter we describe SIREN, an implementation strategy for RBAC in database networks that adheres to these principles.

3.1 Concept

An implementation strategy that performs access control directly on an SQL query eliminates the semantic gap between the way an application or user expresses its request and the way the access control system expresses its restrictions. Stonebraker first proposed query rewriting to perform database access control [31]. The idea is simply to take an incoming query and modify it so that it no longer requests access to data the end user doesn't have privileges for. The result set can then never contain data that the end user is not authorized to see. Stonebraker suggests that such query rewriting be implemented within a separate module of a database. This preserves the transparency of the solution to end users, who connect to the database normally. The query is then modified internally by the rewriter and passed on to another module for execution.

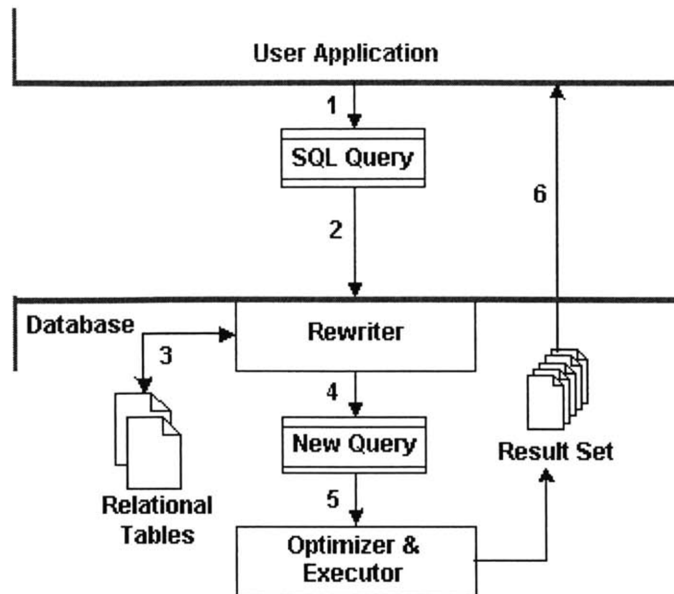


Figure 3.1: Query rewriting as suggested by Stonebraker [31]

1. An end-user (application or user) generates some SQL query.
2. End user establishes a connection to the database and transmits the query.
3. A rewrite module looks up rewrite information, stored in special system relations, for each table mentioned in the query.
4. The rewrite module then uses this information to add conditional clauses that restrict the scope of the query.
5. The rewritten query is then passed on to the database's optimizer and executor modules.
6. The result set is returned to the end-user.

As described above, query rewriting is a local solution for access control on a single database. The implementation proposed in this chapter generalizes the query rewrite concept to an enterprise environment of interconnected database servers, such as the savant network outlined in Chapter 1. The Stonebraker approach presents the query rewrite system from a Discretionary Access Control perspective. Rewrite rules are seen as a tool similar to ACLs, but providing a finer-grained specification of privilege. In this view, each relation is owned by some user who created it. That user may then specify portions of the table as accessible to other users [32]. However, we noted previously that in the DAC model it is difficult to verify that the security policy is enforced system-wide.

SIREN is an implementation strategy that unifies query rewriting with RBAC principles; it provides a framework that leverages query rewriting in a structured way to provide the benefits of RBAC. As specified in the RBAC model, permissions are still granted to specific roles and users are then assigned to those roles. However, the RBAC model leaves defining the nature of permissions up to the implementation. Our suggestion is that the permissions be conditions *specified in SQL* that limit what data any role may access for any table. For an incoming query, these conditional expressions are retrieved based on the requestor's role. The query is then rewritten to take into account the SQL restrictions specified in the permission.

The permission information for the different roles can be conceptualized as a relation. The combination of [*Role*, *Table*] is the primary key of the relation. The *Privilege* attribute in this relation is a SQL SELECT statement which describes the portion of the data in the specified *Table* available to a certain *Role*. Consider the following relation:

PRODUCTS: pid, name, price, quantity, discount

Privilege information for this table can be conceptualized as the following relation:

Role-Privileges Relation		
Role	**Table**	**Privilege**
SalesClerk	Products	*Select pid, name, price, discount from Products where quantity > 0*
HumanResources	Products	*Select NULL from Products*
Stockroom	Products	*Select pid, name, quantity from Products*

Figure 3.2: Excerpt from conceptualized Role-Privileges relation

The rewriter module then takes an incoming query and rewrites it to run within the restricted scope specified by the privileges for this role. Running the same query produces different results if run by users with different roles depending on the conditions specified in their privileges.

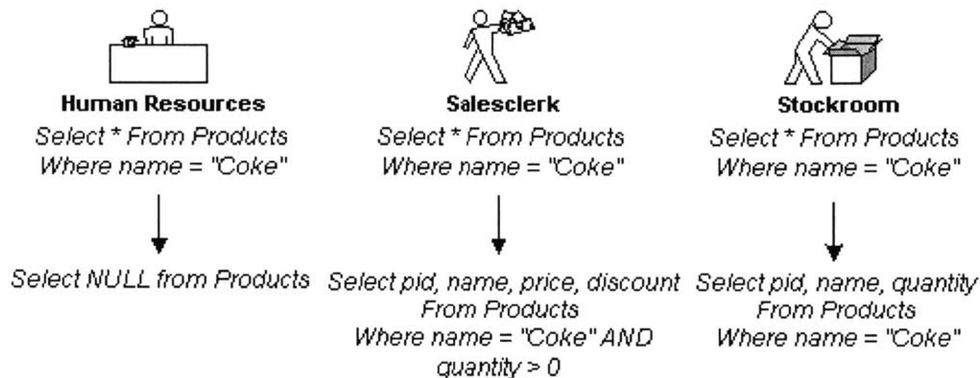


Figure 3.3: The same query is rewritten differently for different users

The example above is based on the privileges given in Figure 3.2.

This approach may seem similar to the "ACL relations" mentioned in the previous chapter, but is in fact more expressive. SIREN provides very fine-grained control since any constraint expressible in a SQL conditional can be used to specify privileges, meaning that any arbitrary section of the table can be specified for each role rather than simple column-wise constraints. Additionally, the SQL conditional can include any functions or system calls that the database administrator wishes to make available to users from the database. An example of this could be an *isOnDuty(employeeID)* function that makes an external call to the company's electronic scheduling program and returns a Boolean value indicating whether the indicated employee is scheduled for the current shift. Including such a restriction in the privilege conditional dynamically changes an employee's access rights, making the information available only at certain times. Finally,

this approach works for complex queries involving joins or multiple subquery levels without the need for defining special functions in an API to handle these cases.

3.2 Implementation

In this section, we outline design issues in implementing SIREN. Given the heterogeneity of computing resources in large enterprises, the ability to be platform agnostic is important. To support access control across a connected network of database servers, the rewrite system should be moved up out of the database. The architecture shown below gives us the flexibility to implement the same rewrite interface for different backend databases. If we changed the backend, only the rewrite module is affected and the change is transparent to the applications. This architecture can also take advantage of connection pooling between the rewrite module and the database, as discussed in Section 3.2.3.

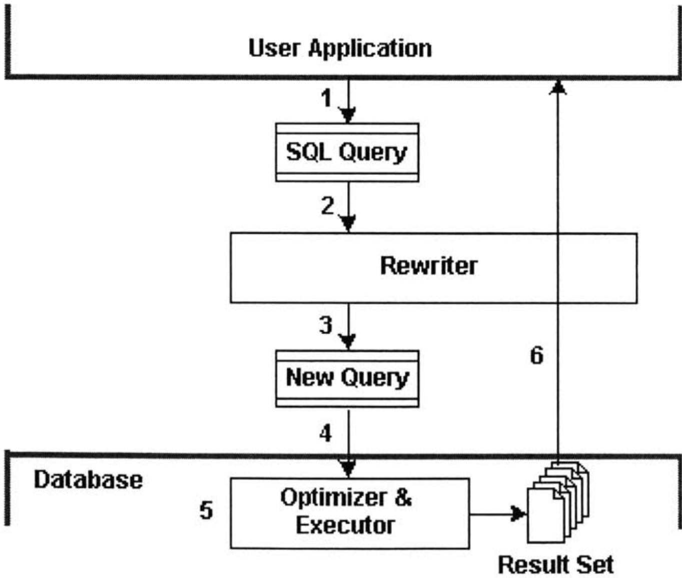


Figure 3.4: Modified query rewrite architecture in SIREN

1. An end-user (application or user) generates some SQL query.
2. End user connects to Rewriter and transmits query.

3. Rewriter modifies query based on stored rewrite information and produces a new query.
4. The Rewriter opens a database connection and transmits the new query.
5. The query is passed to the database's optimizer and executor modules, producing a result set.
6. The result set is returned to the end-user via the Rewriter.

In this architecture, the application must be aware of the presence of the rewriter, since it connects to the rewriter and not the database directly. However, we still preserve a great deal of transparency because the application does not have to tailor its queries in any way; it can still send any SQL query to the rewriter. To the application, the rewriter interface is simply a connection proxy.

The Rewriter module is the central component of the SIREN architecture. An industrial strength implementation of the Rewriter would probably be based on an internal parser and rules engine that examines incoming queries and applies appropriate rewrite rules to them. Rewrite rules for the various roles and their permissions would be created by an administrator via a grammar defined for the rules engine. However, our implementation is intended to be a proof-of-concept prototype that demonstrates the feasibility of RBAC via query rewriting.

The Rewriter module is written in Java for a PostgreSQL database. Java was chosen for its platform independence. PostgreSQL is a powerful, extensible open-source database system and is the database around which the Auto-ID savant is built. Since this is a prototype system, we leverage database views as a tool to simplify the Rewriter implementation.

3.2.1 Leveraging Database Views

Base relations are those relations that are physically stored by the database in memory. Views are virtual relations composed from these base relations. Views are generally *non-materialized*, meaning that the view is not stored as a separate relation. Rather, they are recreated on-the-fly from its definition when it is needed. A view can be defined to be the result of any SQL SELECT statement. Since we have defined our notion of a privilege to be any portion of a table described by a SELECT statement, each such privilege definition is really just a view definition. Our prototype implementation represents a structured methodology for view administration as the basis for RBAC.

No access privileges to the base data tables are given to any roles. Instead for each role, we create a view for each table that role can access. This "role view" specifies exactly which parts of the table a user of that role is allowed to read. The standard naming convention used for role views in our implementation is to append the role name to the table name this view is defined for. For example, a view defining a portion of the Products table accessible to a SalesClerk would be called "Products_SalesClerk".

Note that the view definition must include every column of the base relation, even if the column simply returns NULL data. This allows us to perform certain logical operations on views such as the UNION operation, which are required to support multiple active roles as discussed later. It also preserves transparency in that operations such as JOINs are performed automatically by the database, without the need for additional checks in the rewriter to ensure that a column is actually present in both views. PostgreSQL will generate an error if an end user attempts to join on a column that is not

present in his role view, but the database *will* handle a case in which the join column is simply full of NULL entries.

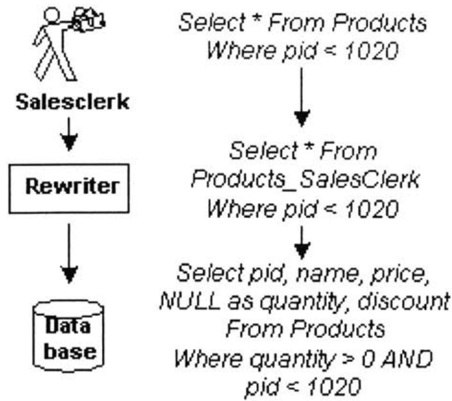
Products (Base Table)				
pid	**name**	**price**	**quantity**	**discount**
1000	Soda	$2	100	10% off
1001	Diet Soda	$2	75	10% off
1002	Caffeine-free Soda	$2	0	None
1050	Orange Juice	$3	0	2 for $5
1060	Apple Juice	$2.50	65	None

CREATE VIEW Products_SalesClerk AS
SELECT pid, name, price, NULL as quantity, discount
From Products WHERE quantity > 0

Products_SalesClerk (View)				
pid	**name**	**price**	**quantity**	**discount**
1000	Soda	$2	NULL	10% off
1001	Diet Soda	$2	NULL	10% off
1060	Apple Juice	$2.50	NULL	None

Figure 3.5: Creating a role view to restrict access to Products table

Standard naming of role views allows the rewrite module to automatically rewrite any query on the base tables to be a query against the appropriate role views of those tables. A query against a role view is automatically circumscribed by the access control restrictions of the view definition. The view system of the database automatically takes care of the rest of the query rewriting based on the view definition, as shown in Figure 3.6 below.



Query Result Set				
pid	**name**	**price**	**quantity**	**discount**
1000	Soda	$2	NULL	10% off
1001	Diet Soda	$2	NULL	10% off

Figure 3.6: How rewriting occurs in our prototype

The framework outlined above applies to SELECT queries and provides a way to specify read privileges. To perform access control for other types of SQL commands in our prototype, we take advantage of *updateable views.* A view is updateable if the database can take the actions specified in an INSERT, UPDATE or DELETE statement on that view and perform those actions on the appropriate underlying base tables. Updateable views are supported by most major databases, including Oracle, DB2/Informix, Sybase, MS SQL Server and PostgreSQL.

In PostgreSQL, updateable views are supported via an internal rule system [32, 21]. The PostgreSQL Rules System (PRS) is a module that lies between the query parser and the planner/optimizer modules of the database. Using PRS, we create rules that specify how actions that update columns in a view are broken down into actions on the actual base table. Therefore, in addition to defining each view with the naming

convention noted above, we also define three rules for each role view. These rules detail how the role view responds to INSERT, UPDATE and DELETE queries.

SAMPLE UPDATE RULES	
CREATE RULE update_products_salesclerk AS ON **UPDATE** TO products_salesclerk DO INSTEAD UPDATE products SET pid = NEW.pid, name = NEW.name, price = NEW.price WHERE pid = OLD.pid AND quantity > 0;	The SalesClerk role is allowed to update only 3 fields of an existing, in-stock product If the incoming query attempts to update quantity or discount, that portion of the query is discarded by the rewrite rule. If the incoming query attempts to update an out-of-stock product, the query is disregarded
CREATE RULE delete_products_salesclerk AS ON **DELETE** TO products_salesclerk DO INSTEAD DELETE FROM products WHERE pid = OLD.pid AND quantity > 0;	The SalesClerk role is allowed to delete any existing, in-stock product If the query attempts to delete an out-of-stock product, the query is disregarded
CREATE RULE insert_products_salesclerk AS ON **INSERT** INTO products_salesclerk DO INSTEAD NOTHING;	The SalesClerk role is not allowed to insert any new products. Any INSERT attempt by this role is discarded

Figure 3.7: Sample rules for update behavior of Products_SalesClerk role view

This approach gives even more fine-grained control over access privileges since we can specify different behavior for the different types of SQL statements. For example, using appropriate rules we can specify that a user has append privileges (INSERT) but can not otherwise write data (DELETE or UPDATE). Since PostgreSQL supports a conditional clause in each rule, update rules can be even more fine-grained in that a user may have a particular update privilege only under certain conditions. Note that this view-based framework assumes that a user will always have read privileges (by the view definition) on any data that he has write privileges on (by update rules for that view). In existing standard security policies, this is usually a basic assumption.

3.2.2 Session Activation and Multiple Roles

The RBAC standard proposed by NIST includes the notion of a user session and multiple active roles in the definition of a basic $RBAC_0$ system. Session establishment is necessary before any data queries can be made. To establish a session, a user must authenticate himself to the access control service and designate a current *Active Role Set (ARS).* Authentication is a separate implementation issue and several architectural options are discussed further in Chapter 4. Our prototype uses simple username and password authentication. The rewriter uses this information to retrieve roles assigned to that username from a UserRoles relation in the database.

In the simple scenario where each user is only assigned one role, that role forms the user's ARS. In our implementation, if the user is assigned to multiple roles, all roles are automatically added to the user's ARS. Query rewriting in the presence of multiple active roles is supported through the use of the UNION operator. UNION is a standard SQL keyword that combines the result set of two or more queries into a single result set that includes all rows from each of the original result sets. As noted earlier, a view definition is really the result set of a SQL SELECT statement, and hence we can take the UNION of any two role views. Again, the database system then takes care of the actual rewriting of the UNION conditions for us. Since we required that every role view is defined to include every column of the base table, even if that column contains only NULL elements, the UNION of two different role views for the same table will not lose any data due to missing columns.

One problem with the UNION operator is the case of role views with different permissions for the same column. As an example, the Products_SalesClerk view defines

the Quantity column as NULL. Another role view, such as Products_Stockroom, might allow the user to see the actually values in the Quantity column. If a user is assigned to both the SalesClerk and Stockroom roles, we would want that user to be able to see the quantity information, since a user's privileges should be the sum of all privileges of his roles. However, in this situation, the UNION operator creates a result set in which the rows are duplicated. Half of the rows in the result set would contain a NULL in the quantity column. The remaining rows of the result set would contain data identical to the first half, except that the actual quantities would be shown in the QUANTITY column. This reduces the coherency of the result set, though it does contain all the necessary data. A more sophisticated query rewriter could avoid this problem by explicitly rewriting the combined role view definition, rather than relying on the database system to do this by using the UNION operator on role views. Such a rewriter would examine the role definitions and intelligently combine which column definitions and conditional clauses to use from each role view to ensure that the combined view includes all privileges from each of the individual views.

Note that a large active role set would considerably slow down query time in our implementation as it would require the UNION of a large number of role views. Using a more sophisticated rewriter would also mitigate this performance penalty. Such a rewriter produces a more efficient role definition query by eliminating overlapping or contradictory conditionals, resulting in fewer time-consuming conditionals than a UNION of role views would have. However, large active role sets may not be an issue in actual systems. The implementers of a real-world RBAC system for a bank noted that in

most cases, their users only required a single role assignment, and no users required more than four [26].

Role hierarchies are simply an administrative extension of multiple active role support. Forming role hierarchies is a way to reduce the effort of role/permission assignment by allowing a role to inherit the permissions of other roles. An additional benefit is that such hierarchies more intuitively reflect an organization's structure and security policy. Support for role hierarchies would require creating an administrative application that could automatically compute a senior role's view by taking the UNION of privileges of junior roles and any additional privileges manually specified by the administrator. Users are only explicitly assigned to the senior-most roles they need to be authorized for; membership in the junior roles is granted automatically by the UNION of permissions that occurred when the administrator first set up the hierarchy. Because the set of explicitly assigned roles is kept small this way, query rewriting is simplified. At run time, only the explicitly assigned roles are included in the user's ARS. Additionally, this makes user privilege review and revocation more straightforward.

Finally, the proposed RBAC standard includes static and dynamic Separation of Duties constraint enforcement. Again, these features are intended as an administrative aid. Once a Separation of Duties constraint is specified by the administrator, it prevents him from inadvertently allowing a user access to two potentially conflicting roles. Static Separation of Duties (SSD) means that this constraint checking occurs when the administrator is assigning roles to the user. If a constraint has previously been specified between two roles, the administrator can not accidentally assign both roles to the user. Implementing SSD is again simply a matter of creating an appropriate administrative

application within which administrators could specify constraints and make user/role assignments. Any time a new user/role assignment is attempted, the application would then enforce any existing constraints.

With Dynamic Separation of Duties (DSD), the administrator is allowed to make the assignment of two potentially conflicting roles to a single user. However, the constraint is checked at every session establishment, and the user can never request an ARS in which the two conflicting roles are both active at once. To implement DSD, we would have to modify our prototype so that it no longer defaulted to including all assigned roles in a user's ARS. Instead, whenever a user had multiple assigned roles, they would be prompted to select some subset of those roles to activate at session establishment. The rewrite system would then need to check the requested ARS against the existing constraints before initializing the session. In this way, our prototype of SIREN could be extended to support the full $RBAC_3$ model in NIST's proposed standard.

3.2.3 Savant Integration

The savant network is intended to support both ad-hoc queries from users as well as queries generated by applications. The Rewriter is made available as a SOAP service for applications that wish to run queries against the savant database. The service exposes methods to establish a session, run a query and close a connection. The applications view this SOAP service simply as a proxy to the database. For ad-hoc queries, a simple servlet application was created to allow users to enter their username, password and a query against the savant via a web browser. The servlet calls the Rewriter interface to pass the query to the savant database. The result set is displayed in the user's browser window.

Connection pooling is another important integration issue. The Rewriter can make use of a connection pool to more efficiently connect to the database. The Rewriter connects to the database on its login, and not with the login information of the user sending the query. We can also utilize connection pooling between the end user's application and the Rewriter. However, if the applications use a connection pool to the Rewriter, they must be able to transmit the username information as a separate argument so that the Rewriter can look up the appropriate role information. This can easily be supported by exposing another SOAP method that takes this login information as a separate parameter.

3.2.4 Completeness

There are several issues related to the completeness of our implementation with respect to the SIREN concept. The first is the use of NULL to mask unauthorized data from a user in a role view. Because NULL is also a valid value for normal field data, there may be ambiguity as to what semantic meaning a particular instance of NULL has; the user may not be able to distinguish whether he simply doesn't have authorization for a particular piece of data or whether the value of that data is simply NULL. However, it is not clear what a more appropriate label to mask unauthorized data would be in our implementation. Because databases enforce type-checking, we can not simply use a text string label such as "Not Viewable" to mask data in non-text columns, such as columns of a numerical data type. The use of any other values to mask this data for columns of different datatypes has the same ambiguity problem as NULL in that a user might not be able to discern between the masking label and real data. In fact, using NULL in our

prototype is less complicated than any other possible masking label since a NULL value can be used in a column of any data type.

Eventually, the correct way to handle this situation is to *not* mask data with any label. Rather, using a more sophisticated rewriter module with rewrite rules for different roles, a query is rewritten such that the only data displayed is the data that a user specifically has access privileges for. Unlike in role views where every column is included in the view, with the more sophisticated approach no unauthorized data would be included in the display, and therefore no data has to be masked. If a user requests to view data he doesn't have privileges for, the rewriter either ignores that portion of the request, or rejects the request entirely and tells the user what portion of the request was illegal.

Another important completeness issue is the whether every possible query can be handled by a particular implementation. Our prototype can easily handle all SQL queries because the rewriter is simply changing base table names to role view names. The underlying view implementation performs the rest of the rewriting and this view system is designed to appropriate rewrite any valid query against a valid view.

Finally, there is the issue of policy verification with our implementation. In other words, how does an administrator know whether a set of privilege assignments and constraints accurately reflects the intended security policy? Our implementation relies on a framework in which for every role, for every table, some role view is defined on that table. This approach forces system administrators to do privilege assignment in a systematic way, decreasing the likelihood that there are holes in the implemented policy. Additionally, a simple application could easily look at the database schema information

to verify that a role view is actually defined for each role for each table. This would further help administrators identify and correct gaps in the privilege definitions.

3.3 Summary

The concept of query rewriting for access control is not new, but what we have presented here is a structured way to use this concept to achieve the benefits of RBAC model for database networks. Using the SIREN framework provides a security policy that is enforced system wide and intuitively mapped from the organization's written security policy. There are a number of other benefits of this implementation strategy.

This implementation has a high degree of transparency because an application can safely send **any** SQL query to the database. Query rewriting increases the flexibility of RBAC by allowing a wider range of security restrictions to be easily expressed. Any combination of restrictions that are representable in SQL can define a user's privilege space. Additionally, it is platform agnostic in that the Rewriter can be implemented for multiple database back-ends. Higher performance can be achieved by implementing a more sophisticated Rewriter instead of relying on existing database functionality. And ease of administration requires only a few simple administrative applications as outlined above. In short, the SIREN implementation strategy meets the key considerations for access control in a large enterprise. In the next chapter, we discuss the issues involved in deploying and maintaining SIREN in a distributed environment.

CHAPTER 4

DEPLOYING SIREN

In the previous chapter, we detailed the basis for a new implementation strategy for RBAC. A functional overview of SIREN for a single database was presented. In this chapter, we survey a variety of related design issues that must be addressed when deploying such a solution in a distributed enterprise environment. The discussion is intended to provide appropriate context for enterprise-wide deployment of RBAC implementations and as a reference to existing research in these areas. While we explore feasible solutions, it is not our intention to provide definitive answers for these design challenges.

4.1 Role Engineering

Deploying any RBAC system requires defining roles that accurately reflect the activities, responsibilities and privileges within an organization. Role Engineering is the process of defining an appropriate role set, defining the requisite Separation of Duties constraints between the roles and defining the correct permissions for each role [7]. In effect, Role Engineering is the process of mapping a security policy into the required RBAC components to implement that policy. This is a complex task for large, distributed enterprises. A limited amount of research exists for formal processes to design appropriate RBAC components within a specific enterprise.

Nuemann and Strembeck [17] take a software engineering approach, demonstrating how to define a role set based on usage scenarios for resources. A scenario is composed of a series of steps that require some particular access. Different scenarios can be grouped into classes of related actions. A preliminary role set is then

defined from these classes. A role must have permissions to complete each step for every usage scenario in its job definition. The role set can then be refined to take to account inheritance relations and separation constraints. However, the authors note that scaling such a scenarios-based process to large enterprises is often very difficult. In particular, they suggest that Role Engineering be a collaborative process between centralized security officers and domain experts throughout the enterprise. The issue of local domains for role and privilege administration is discussed further in Section 4.3.

In [24], the authors define several different classes of roles. Organizational Roles corresponding to various departments can be determined easily. Special Roles are for temporary or special case privileges and must be created manually. However, most roles fall into the Functional Roles category, whose members represent specific job functions. Access to resources requires users to have some combination of Organizational and Functional (and possibly Special) roles assigned to them. The authors describe a *RoleFinder* software tool to help derive the required Functional Roles from a model of enterprise processes using a specific top-down Role Engineering process.

Although research continues on stream-lined methodologies and software tools to accurately model organizational structure in RBAC components, the sheer size of the undertaking in enterprise environments introduces a substantial coordination problem across distributed departments and physical locations. In the Auto-ID system, we expect that certain basic roles can be globally specified with general privileges. However, a manageable solution to the problem of role definition will require support for creation of local roles within specific domains. This allows privileges to be specified by users with greater knowledge of the true business requirements of users in their domain. Domain-

based role administration must be limited so that local permissions can only be assigned within the restricted sphere of permission allowed at a local site by the global security policy.

4.2 Authentication

To make access restrictions specified in SIREN meaningful, the system must verify that a particular user is who he claims to be and determine what roles have been assigned to him. Although authentication is considered a lower level service which the access control implementation can take advantage of, the design of the authentication service affects the overall architecture of the access control deployment. There are two basic architectures for authentication in RBAC for web systems: server-pull vs. user-pull [20]. These architectural principles can be adapted to a deployment of a SIREN system.

4.2.1 Server-Pull Architecture

In a server-pull architecture, the end-user sends his authentication information, usually a username and password, to the database via the rewriter proxy. It is the rewriter's responsibility to verify the user's identity based on this login information. The rewriter then retrieves the user's assigned roles from a database table or role server.

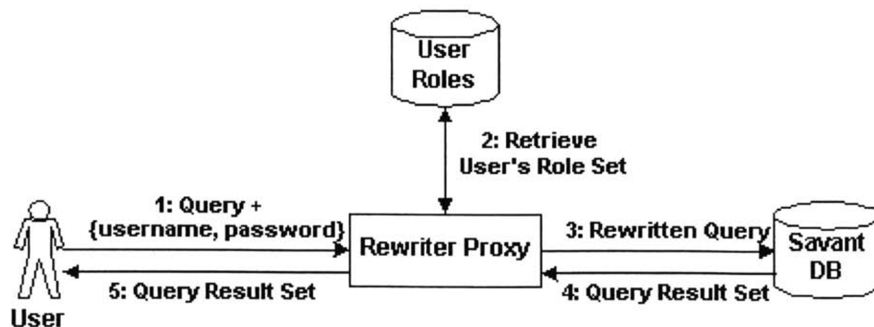


Figure 4.1: Server-Pull Architecture

One interesting design issue in this architecture is implementing the notion of a user session. Certainly it would be inconvenient to have the user submit a username and password every time he wanted to query the database. In this case, each query would be a separate user session. This approach would have a high overhead because the user is reauthenticated and roles re-retrieved for each query. Another option is to have the end-user's application maintain a single authenticated connection, perhaps via SSL, to the rewriter for the duration of the user's session on the application. While this reduces set-up overhead, it also reduces the overall availability of the rewriter module since each connection persists longer.

4.2.2 User-Pull Architecture

In contrast, the user-pull architecture requires the end-user to first authenticate himself to an independent role server, which then grants him some sort of credentials certifying which roles he belongs to. Applications can then present these credentials to the rewriter on behalf of the user when making queries against the data. The user-pull architecture can be easily implemented using PKI-based certificates to store the role credentials. Park, *et al.* [20] describe an implementation of such role certificates by using the extension fields of standard X.509 web certificates.

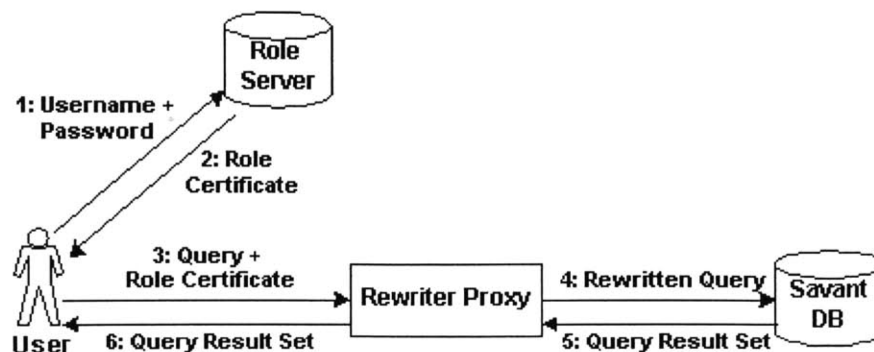


Figure 4.2: User-Pull Architecture

In this architecture, a single certificate represents a user session. The set of roles granted in the certificate is the user's active role set for the session. One issue to be addressed is the lifetime of the user session. It may not be unreasonable to have certificates expire daily, requiring an employee to obtain a fresh one at the start of their work day. Even if the standard role certificate lifetime is longer, support for temporary certificates increases flexibility. For instance, temporary certificates can grant additional privileges to a user when he is filling in for another user on leave.

Since the rewriter is responsible for role retrieval, the server-pull architecture for RBAC is more transparent to end-user applications. The applications are only aware of the rewriter as a proxy to the database, and do not need to know any details about the role information and where it is stored. User-pull architectures are less transparent in that they require the user to have knowledge of a credential-granting authority and obtain privileges as a separate step prior to initiating data queries. However, the user-pull architecture is conceptually cleaner because it separates authentication and role verification logic from the access control logic, resulting in more modular and reusable components. A benefit of a separate authentication step is that the authentication scheme does not have to be standardized throughout the enterprise, while the access control module can be. Different domains may choose to have users authenticate themselves to the role server with a simple username and password, or with more sophisticated authentication devices such as smartcards with pin numbers. Separating the functionality may also reduce the size of the overall deployed code base since a single role certificate server could potentially serve several rewriters, or even other applications. A user-pull architecture seems to be the better choice to support SIREN in distributed enterprises.

4.3 Privilege Management

Data access privileges must be managed throughout the enterprise in a manner that provides for local flexibility while ensuring that enterprise-wide security policies are not circumvented. A framework for privilege management makes the necessary central and local cooperation possible. Such a framework must also address how privilege information is shared when a query requires data that is distributed across the enterprise. We assume a hierarchical organization of data within the enterprise, as in the savant network outlined in Chapter 1.

4.3.1 Local Domains

Centralized access control management is not feasible in large, distributed environments due to the administrative overhead required by the sheer size of the task. Nor is centralized administration necessarily desirable since local administrators often have greater expertise in what permissions are really required for users at their site and can more effectively enforce the Principle of Least Privilege. For this reason, the notion of local *domains* of administration is necessary for enterprise access control.

Administrative domains could be created along any arbitrary divisions; one reasonable approach is to have each physical site within the enterprise comprise a separate domain under the control of a local site administrator. We assume that every user is primarily associated with a specific home domain, and that domain administrator is responsible for maintaining the user's role profile. Each local administrator can further delegate administrative responsibilities within the local site. For instance, each department within a specific domain may have a separate department administrator, who has a subset of administrative privileges granted by the domain administrator. In [29],

the delegation of administrative privileges within a single domain is implemented using role ranges and conditional restrictions to specify the actions allowed to each local administrator.

However, the drawbacks of a traditional decentralized approach such as DAC have already been discussed. Effective enterprise security administration entails a combination of centralized and decentralized control. A successful implementation of local domains requires an architecture that allows the scope of local administrative privileges to be bounded by a centralized security policy. In other words, a generalized set of *global roles* and privileges are centrally created. These centrally defined privileges for global roles limit access to *global data*. Global data refers to data that is collected throughout the enterprise, such as inventory, sales or customer data. This is in contrast to *local data*, which is unique to some specific domain. An example of such local data might be internal employee performance notes collected by department managers at a specific site to help write their personal reviews. If this data is separate from the enterprise's formal employee ratings data, then all department managers at other sites do not need privileges defined for the data specific to this single site.

Local administrators then assign users within their domains to these global roles. To maintain consistency of the base security policy, local administrators should not be able to modify the privileges assigned to global roles in any way. However, the local administrators can create *local roles* valid within their specific domain. Locally created roles can be specified as inheriting from some global role. In this case, the local role inherits all privileges of that global role. The domain administrator can then further *restrict* the global privileges by adding extra conditionals, but can not otherwise modify

the inherited privileges. The local administrator can now assign his users to this more restricted local role instead of the global role it inherits from. This gives local administrators the discretion to provide even greater security of global data if they deem this necessary to enforce Separation of Duties or the Principle of Least Privilege. Locally created roles that do not inherit from any global role can only be granted privileges on local data specific to that administrative domain. This prevents domain administrators from accidentally defining local roles that ignore the privilege restrictions of centrally created roles on global data. Local role names should be qualified by adding a domain identifier to the name, thereby avoiding any potential naming conflicts with global roles.

A malicious domain administrator could still violate enterprise security by creating local roles that inherit from very high-level global roles and then assigning them to inappropriate users. One way to further enhance security is for central security officers to restrict the set of global roles which a local administrator can assign to his users. The local administrator would also not be able to create local roles that inherit from these restricted global roles. For example, in most cases there will be no need for local administrators to create a role that inherits directly from the global CEO role. The subset of global roles that are restricted can vary among administrators of different local domains, so a regional domain administrator might have access to different global roles than a local domain administrator.

In a user-pull architecture, each domain maintains its own domain role server. The central security officers would create the definitions of global roles at a single global role server. This centralized global role server could then push the role information out to each of the domain role servers whenever changes are made. Since the central

administrators can also set which global roles are available to a particular domain, the global role server only pushes the appropriate role definitions for a particular domain. The local administrator can then add local roles to the role server in his domain.

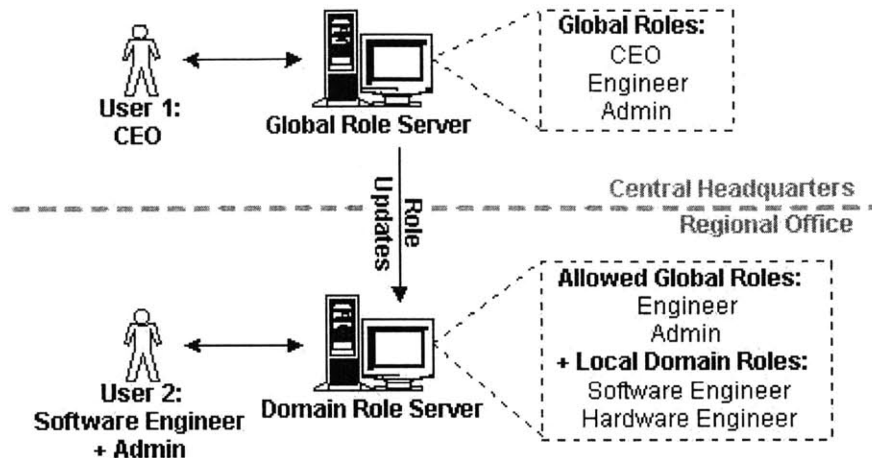


Figure 4.3: Local domains allow decentralized administration

Distinguishing between global and local scope supports the Role Engineering notion that defining appropriate roles and privileges across a large enterprise requires collaboration between central security officers and local domain experts. The approach outlined here provides flexibility to local administrators to fine-tune privileges while guaranteeing that enterprise-wide policies are not violated.

4.3.2 Distributed Queries

The savant network supports distributed queries, in which a single query at one savant spawns multiple subqueries that run on remote savants. The results are then aggregated at the original machine. Such distributed queries must work within the domain-based framework outlined above; the key issue is to ensure that the subqueries use consistent access control information when gathering data across several domains.

One solution is to require that users always initiate queries at a savant within their home domain. This ensures that any local roles assigned to the user will be locatable and well-defined to the rewriter that handles the query. Assuming a user-pull approach, each user obtains role certificates for global and local roles from the role server for his home domain. The query is rewritten to take into account both globally and locally defined privileges assigned to this user's roles. The rewritten query is then passed on by the rewriter to a savant. The savant spawns any appropriate subqueries and forwards them to other savants. Note that once the rewriter modifies the original query and passes it to the savant, the query is permanently out of the control of the rewrite system. The subqueries will automatically have the appropriate access control restrictions and they are passed to remote savants directly, not via the remote rewriters. Each subquery therefore runs with a consistent set of access control privileges without any complicated privilege coordination between the domains.

The privileges used by the distributed subqueries are based on the global and local roles assigned to a user by his home domain. The privilege management framework discussed earlier ensures that this combination of local and global roles will not violate the global security policy. Therefore it is safe to run queries with the privileges specified in the home domain on remote savants anywhere in the enterprise network. This distributed query framework is simple in that it avoids syncing of local privilege definitions across each savant that runs a subquery. This requires that queries are always initiated within a user's home domain. This restriction is likely to be feasible in typical enterprise situations. However, a more sophisticated system in which a query can be initiated anywhere in the enterprise can be envisaged. Efficiently locating and retrieving

the user's role and privilege information from the appropriate domain would be the main design issue in such an approach.

4.3.3 Inter-organization Access Privileges

To automate the supply chain and other key processes, two organizations must be able to share relevant data with each other. For example, a retail organization would like each supplier to be able to dynamically monitor overall inventory levels at its distribution centers to help automate the reordering process. However, that supplier should not be able to access data about any other supplier's products.

The simple solution to this problem is just an extension of the local domain concept. Each organization can create a separate *external domain* composed of a set of servers beyond its firewall, as shown below.

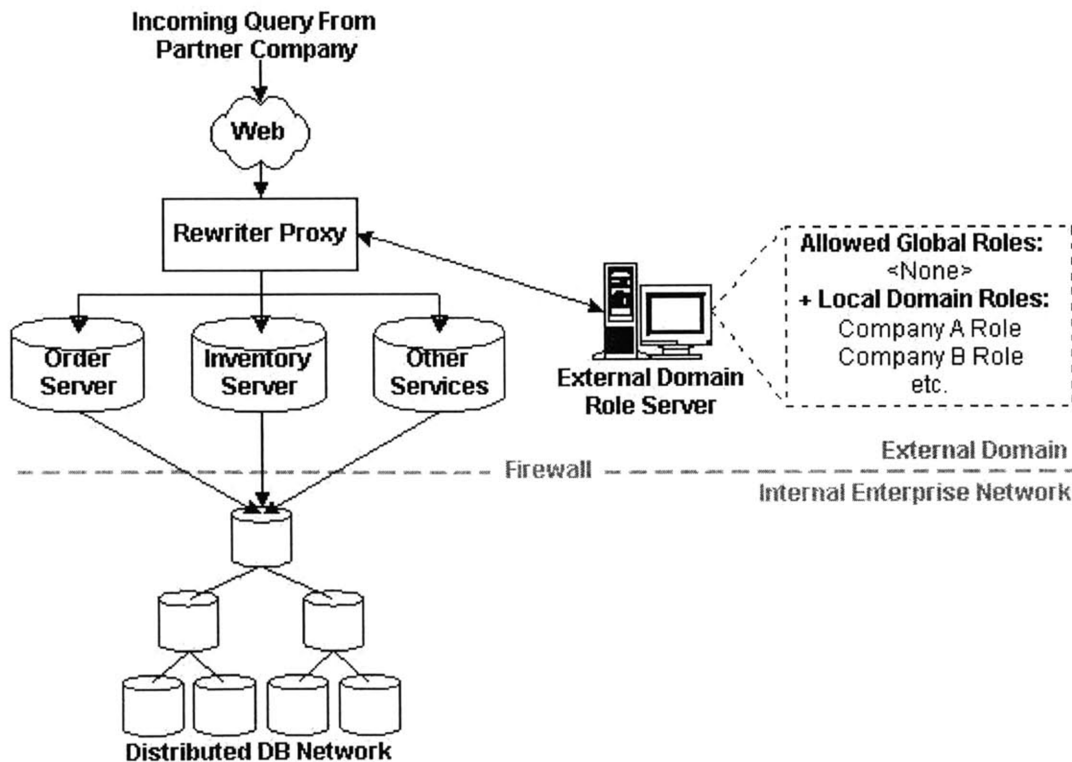


Figure 4.4: External domain exposes inter-organization services

The servers in this external domain provide the interface that other companies use to access this organization's data. Different services and data can be exposed by different servers. A tracking server could allow customers to track the progress of their orders within a supplier's enterprise, while an inventory server would allow suppliers to monitor their product sales at a specific customer enterprise.

The administrator of this external domain can create local roles and assign them to partner companies. There is one important difference between the external domain and any other domains within an enterprise. Whereas standard domain administrators can not grant privileges on global data to local roles, the administrator of the external domain may do so. This gives him the freedom to define different roles for each company as local roles, but grant those roles privileges on any relevant enterprise data. Since each organization will only have a single external domain, the actions of this domain administrator can be easily audited to ensure he does not abuse the ability to grant global permissions to local roles. It makes more sense to create local roles for each partner company, rather than global roles; these companies will only ever run queries from servers in the external domain so no other local domains should need to store this role information.

If a partner company runs a query that requires data from within the enterprise firewall, that query will first be rewritten based on the access control restrictions set in its external domain role definition. Assuming that the external domain administrator set appropriate role privileges, the rewritten query can then be safely passed to the savant's Task Management System, which runs the distributed subqueries on the organization's internal savant hierarchy.

4.4 Integration With Other Access Control Systems

Our primary purpose has been to describe an effective access control implementation and system-wide deployment strategy for enterprise database networks. However, we note that elements of the SIREN architecture can be reused to provide access control over non-database resources such as file systems. Since the RBAC specification leaves the definition of privileges up to the implementation, separate access control modules could be built for file systems and other resources.

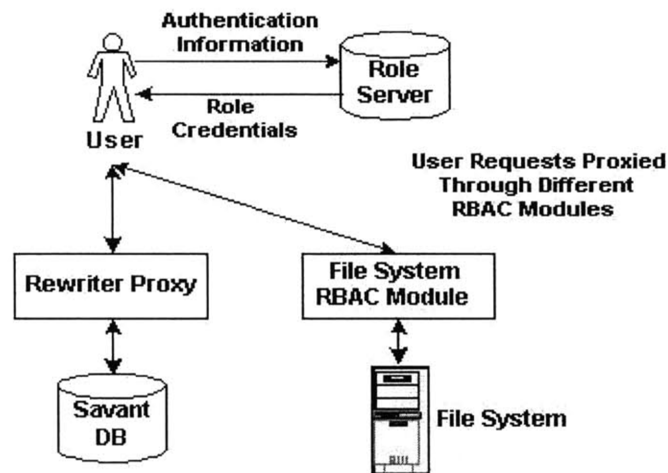


Figure 4.5: Role server certificates can be used by multiple RBAC modules

These other modules may be based on ACLs or some other underlying definition of privileges. If separate role servers are maintained in each domain, these other access control modules may also leverage the role certificates they distribute. Once they have obtained a user's role certificates, these other systems can perform access control as specified by their own definition of privileges for these roles. The reuse of the same global and local roles by different access control modules is very important because it reduces the overall administrative overhead and allows for a consistent security policy based on a single set of roles.

CHAPTER 5
CONCLUSIONS

5.1 Evaluation

In this thesis we have presented SIREN, a framework for efficient database access control using RBAC principles. Key management and architectural issues for deployment in enterprise environments were also outlined. In Chapter 1, we enumerated several important design goals and will now briefly evaluate our solution against these criteria.

With respect to performance, we note that our implementation is only intended to be a prototype, relying on the database to do much of the rewriting automatically. While sufficient for a proof-of-concept, this is not the most efficient approach. As discussed in Chapter 3, performance concerns can be addressed by designing a more sophisticated rewriter that optimizes the way in which queries are rewritten. We believe that a sophisticated query rewrite solution will provide good performance when compared to DAC or MAC solutions that require checking column-wise or element-wise restrictions for each query.

The concept of domains increases the scalability of the system in several ways. First, the amount of user, role or permission data maintained within a single domain is smaller, limiting storage requirements within each domain and increasing the look-up speed against this data. Second, the ability to create local roles that inherit from global roles allows us to limit the size of the global role set, since it doesn't have to encompass every possible role needed anywhere in the enterprise. Local role servers within each domain prevent a bottleneck at the central role server, increasing scalability. Domains

also greatly reduces the complexity of the Role Engineering process, which can be quite complex in large organizations.

The flexibility to support many different security policies is the biggest benefit provided by our solution. Because the access control permissions are expressed directly in SQL, we have maximum expressive power in setting fine-grained permission, including the use of conditionals and function calls in privilege definition. Additionally, the ability to create local roles and privileges further increases flexibility by allowing domains to express their unique security needs within the overall enterprise access control framework. Yet, central security administrators also have the flexibility to limit the administrative privileges of different domain administrators by restricting the set of global roles available to their domains.

Since the rewriter is a separate module outside of the database, it is independent of the specific database or server platforms being used in the backend. Using tools such as Java and JDBC increase the platform independence of the rewriter code. Additionally, because our solution performs access control directly on the SQL query, it works equally well for ad-hoc, OLAP or batch processing queries and is application agnostic in this sense.

Finally, ease of administration is an important issue in large enterprises. As discussed in Chapter 2, RBAC approaches have the benefit that roles serve as a layer of abstraction between users and privileges. While users may change roles fairly frequently, the permissions assigned to a specific role are usually pretty static. Having roles as an intermediate abstraction greatly reduces administrative effort since privileges need only be modified for a single role, and not for each user in that role individually. Additionally,

using the concept of domains, the administrative duties can be reasonably divided over the enterprise. And as mentioned in Chapter 3, GUI-based applications can be created to help administrators manage user/role/privilege assignments using hierarchies and constraint checking.

We conclude that SIREN is an appropriate access control solution for enterprise data networks. While there are a variety of deployment issues, the potential solutions discussed in Chapter 4 meet many of the design requirements of enterprise environments.

5.2 Further Work

We have selected an appropriate access control model, laid out a new implementation framework aimed at database networks and discussed several key enterprise deployment issues. However, further research is still necessary in several important areas.

The next key step in developing SIREN is the design and implementation of a more sophisticated rewriter module. We expect that such a rewriter would be based on an internal rules engine that applies rewrite rules to incoming user queries. This also entails designing an appropriate grammar and user interface to create the rewrite rules. Implementing a more sophisticated rewriter is basically a software engineering project. However, it is a substantial one. In implementing such a rewriter, performance and expressiveness would be the key design goals. Another software design issue is creating the administrative software to assist in user/role/permission management by allowing administrators to manage role hierarchies and assignment constraints via a GUI.

Much of the deployment discussion in Chapter 4 is based on our assumptions of how enterprises would structure and utilize their savant networks. Once such networks are deployed, feedback from these enterprises would provide further clarity about the

deployment issues that need to be addressed. The trade-offs of various architectural configurations and authentication schemes need to be examined in greater depth.

Another interesting question is the possibility of using the access control modules to provide scheduling capabilities. Because of the business critical nature of the data in these enterprise networks, we expect that there will be a high load on the system from user queries. However, certain types of queries should have higher priority. For example, a batch process that runs for a while to transfer large amounts of data from one savant to be aggregated at another should not prevent real-time queries from accessing key data resources. Since the access control module already has all the necessary information about who is making a data request and what resources they are requesting, then this module may also be able to assign priorities to these requests. The challenge is that query priority does not necessarily correspond to role hierarchy. A company's chief financial officer may run large analytic queries over a large data set. A quick response to this query is not as urgent as a real-time query on incoming product information is to stockroom workers unloading a shipment at the dock door. Designing a framework for specifying priority rules and applying to incoming queries within the SIREN framework would be an interesting research topic.

Finally, there are many possible variations that could be imagined in the framework itself. One interesting idea is a distinction between inheritable and non-inheritable privileges in role hierarchies. Whether such functionality would be truly useful and how to implement it efficiently are interesting questions. There may be other such possible extensions to the basic model presented here that need to be examined more

thoroughly. Again, feedback from enterprises that deploy SIREN systems will assist in revising the model and fine-tuning implementation features.

References

[1] Baldwin, R.W. *Naming and Grouping Privileges to Simplify Security Management in Large Databases*. Proceedings of the Symposium on Security and Privacy. Los Alamitos, California. IEEE Press, 1990. pp 116--132.

[2] Barkley, J. *Comparing Simple Role Based Access Control Models and Access Control Lists*. Proceedings of the second ACM workshop on Role-based Access Control. Fairfax, Virginia. ACM Press, 1997. pp 127-132.

[3] Barkley, J. *Implementing Role Based Access Control Using Object Technology*. Proceedings of the first ACM workshop on Role-based Access Control. Gaithersburg, Maryland. ACM Press, 1996.

[4] Bell, D.E. and L. J. LaPadula. *Secure computer systems: Mathematical foundations and model*. Technical Report M74-244. The MITRE Corporation, 1973.

[5] Brock, D.L. *The Electronic Product Code (EPC) – A Naming Scheme For Physical Objects*. Technical Report MIT-AUTOID-WH-002. The Auto-ID Center, MIT. Cambridge, Massachusetts. Published Jan 1, 2001. http://www.autoidcenter.org/research.asp

[6] Brock D.L., T. P. Milne, Y.Y. Kang and B. Lewis. *The Physical Markup Language*. Technical Report MIT-AUTOID-WH-005. The Auto-ID Center, MIT. Cambridge, Massachusetts. Published Jun 1, 2001. http://www.autoidcenter.org/research.asp

[7] Coyne, E.J. *Role Engineering*. Proceedings of the first ACM Workshop on Role-based Access Control. Gaithersburg, Maryland. ACM Press, 1996.

[8] Denning, D.E. *A Lattice Model of Secure Information Flow*. Communications of the ACM. Vol. 19, No. 5 (May 1976): 236-242.

[9] Ferraiolo, D., J. Barkley and D.R. Kuhn. *A role-based access control model and reference implementation within a corporate intranet*. ACM Transactions on Information and System Security (TISSEC). Vol. 2, No. 1 (February 1999): 34-64.

[10] Ferraiolo, D., R. Sandhu, S. Gavrila, D.R. Kuhn and R. Chandramouli. *Proposed NIST Standard for Role-Based Access Control*. ACM Transactions on Information and System Security (TISSEC). Vol. 4, No. 3 (August 2001): 224-274

[11] *A Guide to Understanding Discretionary Access Control in Trusted Systems*. National Computer Security Center. September 1987. http://www.radium.ncsc.mil/tpep/library/rainbow/NCSC-TG-003.html

[12] Giuri, L. *Role-Based Access Control on the Web Using Java.* Proceedings of the fourth ACM workshop on Role-based Access Control. Fairfax, Virginia. ACM Press, 1999. pp 11-18.

[13] Griffiths, P.P. and B.W. Wade. *An Authorization Mechanism for a Relational Database System*, ACM Transactions on Database Systems (TODS). Vol. 1, No 3 (1976)

[14] Koh, R., Y.Y. Kang, D. McFarlane, V. Agarwal, A.A. Zaharudin and C.Y. Wong. *The Intelligent Product-Driven Supply Chain.* Technical Report CAM-AUTOID-WH-005. The AutoID Center, Cambridge University. Cambridge, England. Published Feb 1, 2002. http://www.autoidcenter.org/research.asp

[15] Lunt, T.F., D. Denning, R. R. Schell, M. Heckman and W. R. Shockley. *The SeaView Security Model.* IEEE Transactions on Software Engineering (TOSE), Vol. 16, No. 6 (1990): 593-607.

[16] Neumann, G. and M. Strembeck. *Design and Implementation of a Flexible RBAC-Service in an Object-Oriented Scripting Language.* Proceedings of the Conference on Computer and Communications Security, 2001. Philadelphia, Pennsylvania. pp 58-67.

[17] Neumann, G. and M. Strembeck. *A Scenario-driven Role Engineering Process for Functional RBAC Roles.* Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies. Monterey, California. ACM Press, 2002. pp 33-42.

[18] Neumann, G. and U. Zdun. *Implementing object-specific design patterns using per-object mixins.* Proceedings of the Second Nordic Workshop on Software Architecture (NOSA), August 1999.

[19] Osborn, S., R. Sandhu and Q. Munawer. *Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control.* ACM Transactions on Information and System Security (TISSEC). Vol. 3, No. 2 (May 2000): 85-106.

[20] Park, J.S., R. Sandhu and G. Ahn. *Role-Based Access Control on the Web.* ACM Transactions on Information and System Security (TISSEC). Vol. 4, No. 1 (February 2001): 37-71

[21] PostgreSQL Interactive Documentation. http://www.postgresql.org/idocs/

[22] Redell, D.D. *Naming and Protection in Extensible Operating Systems.* AD-A001721, MIT Press. Cambridge MA. November 1974.

[23] *Reinventing Technology.* The Auto-ID Center. http://www.autoidcenter.org/technology_reinventing.asp

[24] Roeckle, H., G. Schimpf and R. Weidinger. *Process-Oriented Approach for Role Finding to Implement Role-Based Security Administration in a Large, Industrial Organization*. Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies. Chantilly, Virginia. ACM Press, 2001. pp 103-110.

[25] *The Savant Installation Guide*. Internal Technical Report. The Auto-ID Center, MIT. Cambridge, MA.

[26] Schaad, A., J. Moffett and J. Jacob. *The Role-Based Access Control System of a European Bank: A Case Study and Discussion*. Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies. Chantilly, Virginia. ACM Press, 2001. pp 3-9.

[27] Sandhu, R. *Lattice-Based Access Control Models*. IEEE Computer. Vol. 26, No. 11 (November 1993)

[28] Sandhu, R. and F. Chen. *The Multilevel Relational Data Model*. ACM Transactions on Information and System Security (TISSEC). Vol. 1, No. 1 (November 1988).

[29] Sandhu, R. and J. Park. *Decentralized User-Role Assignments for Web-based Intranets*. Proceedings of the third ACM workshop on Role-based Access Control. Fairfax, Virginia. ACM Press, 1998. pp 1-12.

[30] Sarma, S. *Towards the 5 cent tag*. Technical Report MIT-AUTOID-WH-006. The AutoID Center, MIT. Cambridge, Massachusetts. Published Nov 1, 2001. http://www.autoidcenter.org/research.asp

[31] Stonebraker, M. and E. Wong. *Access Control in a Relational Data Base Management System by Query Modification*. Proceedings of the 1974 ACM Annual Conference. ACM Press, 1974. pp. 180-186.

[32] Stonebraker, M., A. Jhingran, J. Goh and S. Potamianos. *On Rules, Caching, Procedures and Views in Data Base Systems*. Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data. Atlantic City, New Jersey. ACM Press, 1990. pp 281-290.

[33] *Trusted Computer System Evaluation Criteria*. United States Department of Defense. December 1985. http://www.radium.ncsc.mil/tpep/library/ rainbow/5200.28-STD.html

Appendix

Prototype Details

Our prototype rewriter is written in Java. The rewriter is based on the open-source freeware package SQL4J developed by Jianguo Lu [1]. SQL4J examines string representations of SQL queries, parses them and fills in Java structures representing these statements and expressions. Conceptually, an incoming SQL query is represented as a Java *SQLStatement* object. Actually, *SQLStatement* is an abstract class, and the actual instances created by the parser are instances of more detailed subclasses such as *InsertStatement*, *UpdateStatement*, *CreateTableStatement*, etc. The fields of these objects are other Java objects that represent the various clauses, predicates and variables in the SQL query, such as *WhereClause*, *LikePredicate*, *Table*, etc. Some of these field objects, like *WhereClause* further contain other objects such as *AtomicWhereCondition* or *CompoundWhereCondition*.

First, the source code of the SQL4J package was modified by the addition of a rewrite method, *rewriteTableNames(rolename)*. This method is called on a specific instance of a *SQLStatement* object, and takes a role name as a parameter. It examines the fields of the *SQLStatement* and recursively examines the clause objects that form this *SQLStatement*. In each clause object, any reference to a database table is rewritten as a reference to the role view for that table. After the method has recursed through all the components of the *SQLStatement*, the query represented by this *SQLStatement* has been rewritten in accordance with the RBAC framework described in Chapter 3.

Next, to demonstrate the rewriter's functionality we created a Java servlet-based application that allows a user to input his login information and a database query in an

html form using a web browser. The web application was deployed using the Tomcat servlet engine. After a user fills out the form data and hits the "Submit Query" button, the Tomcat engine would call our Java servlet, *DBQuery*. Using the login information, the servlet first retrieves the user's role information. In our implementation, this we do this by establishing a JDBC connection to a PostgreSQL database and querying a *UserRoles* table. However, a more sophisticated implementation might choose to cache some of this role information in the rewriter so that a database lookup is not always necessary.

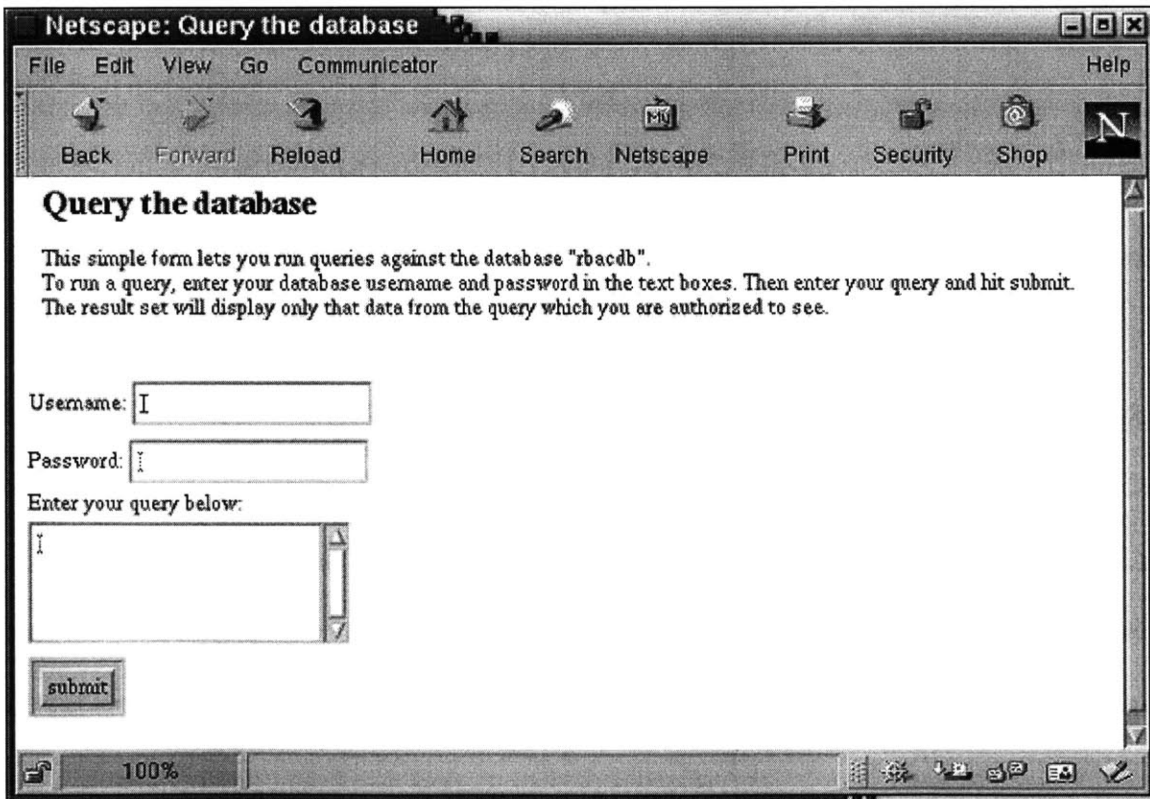


Figure A.1: Screen shot of web interface to rewriter

The *DBQuery* servlet then calls the SQL4J package to create a *SQLStatement* object based on the query string entered by the user. The servlet then calls the *rewriteTableNames(rolename)* method on this object, with the user's assigned role as a

parameter. This method call modifies the *SQLStatement* so that the query now attempts to access only the appropriate role views. Finally, the servlet code uses a *toString()* method to translate the *SQLStatement* back to a string representation and runs this query against the database. The servlet then displays the result set in the user's browser window.

The web servlet is intended as a demo interface to our rewriter. The same functionality can also be exposed as an API for applications to call directly. To do this, we create a SOAP service that such applications can use. The SOAP service exposes an interface method that takes login information and a query. Using a deployment descriptor, this interface method name is associated with the method of a specific Java class. When an application invokes the SOAP interface method, this Java class method is actually called. Similarly to the servlet described above, the method looks up the user's role information, calls the SQL4J package to create a *SQLStatement* object, calls the rewrite function using the appropriate rolename, then runs the modified query and returns a result set. The result set is then passed by the SOAP service back to the requesting application.

[1] Lu, Jiango. *SQL4J*. http://www.cs.toronto.edu/~jglu/sql4j/index.htm