An Approach of Security for Handling the Security Threats for Distributed Systems

Farman Ali[#1], *Syed Nusrat* [*2], *Dr. Sachin Kumar**[3]

[#] *Department of Computer Science, SJJT University*
Churu-Bishau Road, Chudella,Jhunjhunu, Rajasthan, India

[1] raofarmanmca@gmail.com

[2]nusrat.syed@gmail.com

[3]*imsachingupta@rediffmail.com*

Abstract

With the rapid growth of the information age, open distributed systemshave become increasingly popular. The need for protection andsecurity in a distributed environment has never been greater. Theconventional approach to security has been to enforce a system-widepolicy, but this approach will not work for large distributed systemswhere entirely new security issues and concerns are emerging. Weargue that a new model is needed that shifts the emphasis from"system as enforcer" to user-definable policies. Users ought to be ableto select the level of security they need and pay only the necessaryoverhead. Moreover. Ultimately, they must be responsible for theirown security.This research is being carried out in the context of the Legion project.We start by describing the objectives and philosophy of the overallproject and then present our conceptual model and design decisions. Aset of technical challenges and related issues are also addressed.

Keywords: Distributed Kernel; Heterogeneity;Legion Project;Authentication;Delegation; Legion Object Model; System Philosophy;

1. Introduction

High speed networking has significantly changed the nature ofcomputing, and specifically gives rise to a new set of securityconcerns and issues. The conventional security approach hasbeen for "the system" to mediate all interactions between usersand resources, and to enforce a single system-wide policy.This approach has served us well in the environment of acentralized system because the operating system implementsall the key components and knows who is responsible for eachprocess.

However, in a large distributed system several things have changed:

- Distributed Kernel: There is no clear notion of a single protected kernel. The path between any two objects mayinvolve several machines that are not equally trusted.
- System Scope and Size: The system is usually much larger than a centralized one. It may very well be a federation of distinct administrative domains with separate authorities.
- Heterogeneity: The system may involve many subdomains with distinct security policies, channels that are secured in several ways, and platforms with different operation systems.

The intricate nature of distributed system has fundamentallychanged the requirement of system security. We areinvestigating a new model of computer security - a modelappropriate to large distributed systems in the context ofLegion - a system described below.

Users of Legion-like systems must feel confident that theprivacy and integrity of their data will not be compromised -either by granting others access to their system, or by runningtheir own programs on an unknown remote computer. Creatingthat confidence is an especially challenging problem for anumber of reasons; for example:

- We envision Legion as a very large distributed system; at least for purposes of design, it is useful to think of it as running on millions of processors distributed throughout the galaxy.
- Legion will run on top of a variety of host operating systems: it will not have control of the hardware or operating system on which it runs.
- There won't be a single organization or person that "owns" all of the systems involved. Thus no one can be trusted to enforce security standards on them; indeed, some individual owners might be malicious.

No single security policy will satisfy all users of a huge system.We cannot even presume a single "login"mechanism - some situations will demand a far morerigorous one than others. Moreover we cannot anticipate allthe policies or login mechanisms that will emerge; both will beadded dynamically. And, for both logical and performancereasons, the potential size and scope of Legion suggest thatwe should not have distinguished "trusted" components thatcould become points of failure/penetration or bottlenecks.

Running "on top of" host operating systems has manyimplications, but in particular it means that in addition to theusual assumption of insecure communication, we must assumethat copies of the Legion system itself will be corrupted (rogueLegionnaires), that some other agent may try to impersonateLegion, and that a person with "root" privileges to acomponent system can modify the bits arbitrarily.

The assumption of "no owner" and wide distributionexacerbates these issues, of course. Since Legion cannot replace existing host operating systems, the idea of securingthem all is not a feasible option. We have to presume that atleast some of the hosts in the system will be compromised, andmay even be malicious.

These problems pose new challenges for computer security.They are sufficiently different from the prior problems facedby single-host systems that some of the assumptions that havepervaded work on computer security must be re-examined.Consider just two such assumptions. The first is that security isabsolute; a system is either secured or it is not. A second is that"the system" is the enforcer of security.

In the physical world, security is never absolute. Some safesare better than others, but none is expected to withstand an arbitrary attack. In fact, safes are rated by the time they resistparticular attacks. If a particular safe isn't good enough, itsowner has the responsibility to get a better one, hire a guard,string an electric fence, or whatever. It isn't "the system",whatever that may be, that provides added security.

Note that we said that users must feel "confident"; we did notsay that they had to be "guaranteed" of anything. Securityneeds to be "good enough" for a particular circumstance. Ofcourse, what's good enough in one case may not be in another- so we need a mechanism that first lets the user know howmuch confidence they are justified in having, and secondprovides an avenue for gaining more when required.

The phrase "the trusted computing base" (TCB) is commonwhen referring to systems that enforce a security policy. Themental image is that "the system" mediates all interactionsbetween users and resources, and for each interaction decidesto permit or prohibit it based on consulting a "trusted database"; the Lampson access matrix [] is the archetype of suchmodels. Even communications, which is inherently insecure,is usually presumed to be inside the perimeter and the systemis considered to be responsible for implementing securecommunication on top of the insecure base.

As with the previous assumption, this one just doesn't work ina Legion-like context. In the first place there isn't a singlepolicy, new ones may emerge all the time, and thecomplexities of overlapping/intersecting security domains blurthe very notion of a perimeter to be protected. In the secondplace, since we have to presume that the code might bereverse-engineered and modified, we cannot rely on thesystem enforcing security - or very much of anything, forthat matter.

Moreover, security has a cost in time, convenience, or both.The intuitive determination of how much confidence is "goodenough" is moderated by cost considerations. As weobserved many times, one reason that extant computer systemshave not paid more attention to security is that the cost,especially in convenience, is too high. These prior systemstook the "security is absolute" approach, and everyone paidthe cost regardless of their individual needs. To succeed, ourmodel must scale - it must have essentially zero cost if nosecurity is needed, and the cost must increase in proportion tothe extra confidence one gains.

The above observation calls for rethinking some very basic,often stated assumptions - that is, a change in the way ofthinking and a shift in security paradigm. In the rest of thepaper, we suggest a new security model that differs from thetraditional approach. We also illustrate ideas to deal with theissues raised above, as well as others. Before proceeding todescribe our plan of attack, the following describes the Legionsystem to provide context.

2 Backgrounds - The Legion Project

The Legion project at the University of Virginia is an attemptto provide system services that create the illusion of a singlevirtual machine, a machine that provides secure shared objectand shared name spaces, high performance via both task anddata parallelism, application adjustable fault-tolerance,improved response time, and greater throughput. Legion istargeted towards wide-area assemblies of workstations,supercomputers, and parallel supercomputers. Such a system,if constructed, will unleash the integrated potential of manydiverse, powerful resources which may very wellrevolutionize how we work, how we play, and in general, howwe interact with one another.

The potential benefits of Legion are enormous. We envision(I) more effective collaboration by putting co-workers in thesame virtual workplace; (2) higher application performancedue to parallel execution and exploitation of off-site resources;(3) improved access to data and computational resources; (4)improved researcher and user productivity resulting frommore effective collaboration and better applicationperformance; (5) increased resource utilization; and (6) aconsiderably simpler programming environment for theapplications programmers. Indeed, it seems probable to us thatthe NII can reach its full potential only with a Legion-likeinfrastructure.

2.1 The Legion Object Model and System Philosophy

Legion is an object-oriented meta-system'. The principles ofthe object-oriented paradigm are the foundation for theconstruction of Legion; all components of interest in Legionare objects, and all objects, including classes, are instances ofclasses. Use of the object-oriented paradigm enables us toexploit the paradigm's encapsulation and inheritanceproperties, as well as benefits such as software reuse, faultcontainment, and reduction in complexity.

Hand-in-hand with the object-oriented paradigm is one of ourdriving philosophical themes: we cannot design a system thatwill satisfy every user's needs; therefore we must design anextensible system. This philosophy manifests itselfthroughout, particularly in our use of delayed binding andwhat we call "service sliders". Consider security. There is atrade-off between security and performance (due to the cost ofauthentication, encryption, etc.). Rather than providing a fixedlevel of security - with the result that no one will be happy, weallow users to choose their own trade-offs by implementingtheir own policies or using existing policies via inheritance.Similarly users can select the level of fault-tolerance that theywant - and pay for only what they use. By allowing users toimplement their own or inherit services from library classeswe provide the user with flexibility while at the same timeproviding a menu of existing choices.

2.2 Design Objectives and Restrictions

We have the following design objectives, against which wemeasure our success; site autonomy; an extensible core;scalability; easy-to-use, seamless computational environment:high performance via parallelism; single, persistentnamespace; security for both users and resource providers;manage and exploit resource heterogeneity, and faulttolerance.

In addition to the goals above, two constraints restrict ourdesign - we cannot replace host operating systems, and wecannot legislate changes to the interconnection network.

To accomplish the goals, many technical, political,sociological, and economic issues need to be resolved. In thispaper we attempt to address the security aspect of the Legionproject.

3 The Security Model

In this section we describe a design for the security model inLegion. The model, following closely to the Legionphilosophy, responds to the issues raised in the introduction.We first present the design guidelines and principles. Wediscuss the trade-offs and our design decisions. We thenexplain how the model works, in particular how it can be usedto enforce discretionary policies.

The premise here is that we cannot, and indeed should not,provide a guarantee of security. What we can and should do is(1) be as precise as possible about the degree of confidence auser can have, (2) make that confidence "good enough" and"cheap enough" for an interestingly large selection of users,and (3) provide a context that allows the user to gain theadditional confidence they require with a cost that isintuitively proportional to the added confidence they get.

3.1 Design Principles

The Legion Security model is based on three principles:

- First, as in the Hippocratic Oath, do no harm!
- Second, caveat emptor let the buyer beware.
- Third, small is beautiful.

Legion's first responsibility is to minimize the possibility thatit will provide an avenue via which an intruder can domischief to a remote system. The remote system is, by thesecond principle, responsible for ensuring that it is running a valid copy of Legion - but subject to that, Legion should not permit its corruption.

The second principle means that in the final analysis users areresponsible for their own security. Legion provides a modeland mechanism that make it feasible, conceptually simple, andinexpensive in the default case, but in the end the user has theultimate responsibility to determine what policy is to beenforced and how vigorous that enforcement will be. This, wethink, also models the "real world"; the strongest door with thestrongest lock is useless if the user leaves it open.

The third principle simply means, given that one cannotabsolutely, unconditionally depend on Legion to enforcesecurity, there is no reason to invest it with elaboratemechanisms. On the contrary, at least intuitively, the simplerthe model and the less it does, the lower the probability that acorrupted version can do harm. The remainder of the paperdescribes such a simple, albeit evolving model. Thedescription is discursive, but a much shorter, formal definitionwill be forthcoming.
As noted above, Legion is an object-oriented system. Thus,

- the unit of protection is the object, and
- the "rights" to the object allow invocation of its member functions (each member function is associated with a distinct right).

This is not a new idea; it dates to at least the Hydra system inthe mid 1970's 161 and is also in some proposed CORBAmodels. Note, however, that it subsumes more commonnotions such as protection at the level of file systems. InLegion, files are merely one type of user-defined object that happen to have methods read/write/seek/etc. Directories arejust another type of object with methods such as lookup/enter/delete/etc. There is no reason why there must be only one typeof file or one type of directory and, indeed, these need not bedistinguished concepts defined by, or even known to Legion.

The basic concepts of the Legion Security Model are minimal;there are just four:

- Every object provides certain known member functions(that may be defaulted to NIL); the ones we willdescribe here are "MayI," "Jam," and "Delegate. ".
- There is a "responsible agent" (RA) associated witheach operation. The RA is someone who can be

heldaccountable for the particular operation. There are a certainset of member functions associated with an RA object.User-defined objects can play the role of RA by supplyingthese member functions.

- Every invocation of a member function is performedin an environment consisting of a pair of (unique) objectnames - those of the operative responsible agent, and“calling agent”, CA.
- There are a small set of rules for actions that Legionwill take, primarily at member function invocation. Theserules are defined informally here.

The general approach is that Legion will invoke the knownmember functions (MayI, etc.), thus giving objects theresponsibility of defining and ensuring the policy. Preciselyhow this happens is detailed in the following sections.

3.2 Protecting Oneself – Privacy

In Legion users are responsible for their own security. Theyare the ones, who decide how secure their applications ought tobe, and from there, which policy is to be enforced and howrigorous the enforcement should be.

For example, a truly paranoid user’s object can (and should, ifthey deem it important) include code in every method toauthenticate the caller and to determine whether that caller hasthe right to make this call. This cautious user most likely willnot be satisfied unless some elaborate authentication scheme isused to identify the caller.

For many users, however, this degree of caution isunnecessary and some delegation to the Legion mechanism isappropriate - for example, rather than engaging in anauthentication dialog with the caller, an object might trust thatthe CA field of the environment is correct. In the followingwe’ll describe how the model facilitates appropriate, situation-specific delegation; for readability we’ll precede in severalsteps, each of which adds a bit more detail and refinement.

Our first objective is to have policies defined by the objectsthemselves. At the same time, we don’t want to have toinclude policy-enforcement code in every member functionunless the object is particularly sensitive. So, instead, werequire that every class define a special member function,“MayI” (this can be defaulted, but we’ll ignore that for now).MayI defines the security policy for objects of that class.Conceptually at least, Legion will automatically call the MayIfunction before every member function invocation, and willpermit that invocation only if MayI sanctions it (see figure 1).

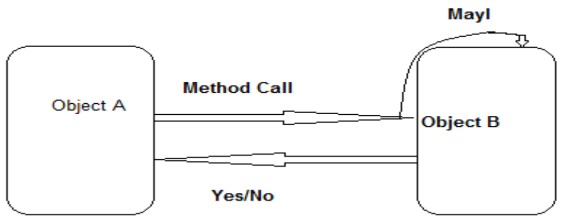

Figure 1

We’ll refine this in a moment to be both more efficient andmore powerful -but note how this simple idea begins to meetour objectives. First, it permits the creator of an object class todefine the privacy policy for objects of that class; there is nosystem-wide policy. Second, it is fully extensible - when auser defines a new class its member functions become the“rights” for that class and its MayI function/policy determineswho may exercise those rights. Third, it is fully distributed;there is no distinguished trusted data base (each MayI mayconsult a database if it chooses, but there is no “distinguished”one(s)). Fourth, it is not particularly burdensome; users candefault MayI to “always OK”, inherit a MayI policy from aclass they trust, or write a new policy if the situation warrantsit. Fifth, the code for implementing the security policy islocalized to the MayI function rather than distributed amongthe member functions. Finally, the default “always OK” policycan be optimized so that there is no overhead at all associatedwith the mechanism.

3.3 Authentication

The previous discussion left one question unanswered: who orwhat is the “I” that the MayI function grants access to? Indeed,the request must first be authenticated to identify the principalthat uttered it, and then authorized only if the principal has theright to perform the operation on the object. The principalbehind the request could be human users, software programs,or compound identities such as delegations, roles and groups.

Authentication in Legion is aided by the use of Legionenvironment. Recall that the environment contains two objectidentifiers, namely the calling agent (CA) and the responsibleagent (RA). The CA is the object that initiated the currentmethod call. The RA is a generalization of the “user id” inconventional systems; for the moment it is adequate to think ofit as identifying the user or agent who was responsible for thesequence of method invocations that lead to the current one.

In the general spirit of our approach, the authentication of thecaller and caller’s context can be anything that the MayIfunction demands - and in sensitive cases, that is just as itshould be. In most cases, however, “I” will be simply CA, orRA, or any subset of the two. Indeed, by analogy with familiarsystems where "I" is the user, that subset may be just RA.

Legion makes a specified level of effort to assure theauthenticity of the environment IDS; this effort should beadequate for most purposes. However, in the spirit of thesecond principle, we expect that MayI functions withextraordinary security concerns will code their ownauthentication protocols by, for example, calling back to thecaller, and/or responsible agent. To make this possible, werequire every Legion object to supply a special public memberfunction - "lam" for authentication purposes. In the sameprinciple as "MayI ", "lam" could be optimized to NIL.

Legion bases authentication on public-key cryptography in thedefault case. Knowledge of the private key is the proof ofauthenticity. In addition, a set of general principleauthentication protocols will be provided as the systemstandard. Yam" can choose to support all or none of them.Other more elaborate protocols could be negotiated betweenobjects and made known to the "lam" function. Objectsunprepared to adequately authenticate themselves are ipsofacto not to be trusted. The result of "Iam" can be cached forfuture reference, but that is an implementation choice and isbeyond the scope ofthis paper.

3.4 Login

The avenue via which Legion users authenticate themselves toLegion is the Login procedure. Login establishes user'sidentity as well as creating a responsible agent object for theuser. The login procedure is therefore the building block forfuture authentication, delegation and creating of compoundidentities.

By the same design principle, Legion should not mandate asingle "Login" mechanism. Typically, there is a login objectthat will be invoked when a user first logs in. This login objectengages in a login dialog with the user and, if satisfied,declares itself to be the responsible agent. Actually, anyI.egion object may declare itself to be the current responsibleagent should it choose. It simply does so by executing a "RA =me" command (environments are stacked, so that a returnfrom an object executing this command will revert to theprevious RA).

There are many advantages to why we shouldn't make this"login" mechanism universal. For example, logging on toLegion in UVa may require only a simple password whileLegion in CIA might demand their users to submit fingerprintsor retinal scan information. Users can define their own loginclass with varying degrees of rigor in the login dialog, specificto their needs. The "login" mechanism can also be easilyinherited or defaulted to some simple scheme.

How do we know that a particular login class is to be trusted?We don't, in general. The MayI function of another class neednot believe the login! After interrogating the class of theresponsible agent the MayI function may reject the call if thelogin is either insufficiently rigorous, or simply unknown to this MayI. As in the infamous "real world", trust can only beearned.

3.5 Delegation

In all security models one must consider how rights propagate;can a principal hand all or some of its authority to another,and how can a principal restrict its authority? For example, auser on a workstation may wish to delegate the "read" right onher files to the C compiler. The compiler can then access fileson her behalf as long as the delegation still stands, much in thesame way the user may wish to delegate. Just as the basicsecurity policy is embedded in MayI and not in Legion, ourmodel does not answer this question - but it does provide auniform way for the user to answer it.

We require every Legion object to have another publicmethod, "Delegate." The parameters to Delegate are the ID ofthe object to which rights should be delegated, and a set ofrestrictions that limit those rights. For example, a user objectsA wants to invoke a compiler C and pass the "read-only" righton file F to C. To accomplish this, A must invoke the"Delegate" function of F to request such a delegation. Using aC++ like notation, but prefixing it with the name of theexecuting object and a colon, this is:

A: F.Delegate(C, read);

F, upon receiving the above request, can grant thedelegation, reject it, or grant delegation of a more restrictedauthority than what is requested. Granting delegation mayresult in storing some information locally or in creation of anew entry in some database (for example, an access control list) known to MayI.

A then instructs C to compile he file by passing it the ID of F.

A: C.Compile (F)

When C attempts to read F, F's MayI is invoked. MayIrecognizes this delegated authority either by looking up somelocal information or consulting some external database. Theoperation is thus permitted. However, if C attempts to invokeany of F's other methods, F will disallow this.

Our philosophy is that delegation policy is a part of thediscretionary policy which should be defined by the objectitself. Indeed, delegation policies can be arbitrarily complex orlight weight. Classes that want to take extreme precautionsagainst delegation may choose not to support delegation at all- this is the default. Alternatively, users can write their owndelegation functions or inherit appropriate ones from existingclasses - for example, by including a time limit as part of theaccess database, delegation can be made to expire after certaintime period.

So far we have discussed three security-related functions:MayI, lam and Delegate. They are user-defined functions,together, quite elegantly; they form a guard or referencemonitor upon which any discretionary policy can be defined.In addition.

- "MayI". "lam" and "Delegate" can be defaulted toNIL and hence will impose no overhead. And indeed, manyclasses will favour the default case for performance reasons.

When these functions are non-NIL, they enforceuser-definable policies rather than some global Legiondefinedone,

These functions can be as simple or as elaborate asthe user feels necessary to achieve their comfort level – the"service slider" approach again.

4 Mandatory Policies

Mandatory policies, such as multi-level security, presume thatthe parties involved may be conspirators and impose some sortof check by a third party - usually "the system" – betweencaller and called objects. Generally this imposition iscompletely dynamic - every call is checked.In the Legion context, of course, we eschew the idea of asystem-wide policy. Thus we need a safe mechanism thatinterposes an arbitrary enforcer of an arbitrary policy betweencaller and called object. Interestingly, when combined withinheritance, the MayI function already discussed provides halfthe answer, albeit in a somewhat different way.

Imagine that a new mandatory security regime is to be created.An obvious consideration is that the enforcer, which we'll callthe "security agent" must know about all of the kinds ofobjects in its domain -it cannot enforce "no write down" if itdoesn't know what a "write" to a specific object is, forexample. Thus we'll begin with the presumption that a goodsecurity agent simply won't allow calls on objects of unknownpedigree.Given that, it is reasonable to presume that the security agentcan derive subclasses for the objects that it does know about;in these subclasses the security agent can inherit a MayI function of its choosing - and specifically one that invokesthe security agent to verify the validity of each inward call. Allthe objects and only the objects that are instances of thesederived classes will be permitted in this security agent'sregime.Derived classes will be permitted in this security agent'sregime.

As noted above, this solves half the problem - the securityagent is invoked whenever an object under its control is called.We need to add the symmetric capability for outward calls;thus we add a method I want to that, if non-null is invoked byLegion whenever an object attempts to make a call on anotherobject. Now, by deriving a class that defines both the MayI andI want to methods, the security agent can be ensured that itgets invoked on every call involving one of the objects underits control. Finally, although we won't discuss it here, obviously we candefine a license mechanism for I want to that is analogous tothat for MayI, with the analogous benefit – I want to can getinvolved as much or as little as it deems appropriate.

6. Conclusion and Opportunities for Further Research

We have discussed database security issues in general and how the database model affects database system security in particular. We have seen that security protections for OODBMS and RDBMS are quite different. Each model has significant strengths and weaknesses. Currently, the RDBMS is the better choice for a distributed application. This is due to the relative maturity of the relational model and the existence of universally accepted standards.

The recent emergences of hybrid models that combine the features of the two models discussed raise many new security questions. For example, Informix's Illustrate combines a relational database schema with the capability to store and query complex data types. They call this system an "object-relational database." Informix claims that their system has all the capabilities of a RDBMS, including "standard security controls" with the principle advantage of an OODBMS: encapsulation, inheritance, and direct data access through the use of data IDs. This hybrid and similar systems offered by Oracle and others raise many new questions. For example, do the relational database security controls work well with complex data types and objects? How well do these security controls interface with encapsulation and object methods? What new avenues of attack have been opened by the combination of these two seemingly different concepts? What special security problems will arise when the object relational system is extended to the distributed environment?

In addition to the questions raised above, there are also opportunities for research in several other areas. They include subject authorization strategies for heterogeneous distributed systems, inference prevention strategies for both centralized and distributed database systems, and distributed object-oriented database security standards.

ACKNOWLEDGEMENTS

This research paper is the part of my PhD work. I am doing PhD in Computer Science. This paper will help in the Design and Implementation of a Heterogeneous DistributedDatabase Inference Controller. One of the most important of these factors are single level and multilevel access controls, protection against inference, and maintenance of integrity. For determining that which distributed database model will be more secure for a particular application, the decision should not be made purely on the basis of available security features.

REFERANCES

- C. Batini, S. Navathe, Conceptual database design.
- R.V. Binder, Testing Object-Oriented Systems—Models, Patterns, and Tools.
- M. Blaha, W. Premerlani, Object-Oriented Modeling and Design for Database Applications.

G. Dhillon, Information Security Management.

- R. Elmasri, S. Navathe, Fundamentals of Database Systems.
- IEEE Standards for software verification and validation.
- M.R. Adam, Security-control methods for statistical database: a comparative study, ACM Computing Surveys.
- BELL751 Bell, D., and L. LaPadula, July 1975, Secure Computer Systems: Unified Exposition And Multics Interpretation, TechnicalReport NTIS AD-A023588, the MITRE Corporation,
- FORD901 Ford, W., J. OKeeffe, and B. Thuraisingham, August 1990, ***Database Inference Controller: An Overview,*** Technical Report MTR 10963, Vol. 1, The MITRE Corporation, Bedford, MA (a version accepted for publication in Data and KnowledgeEngineering Journal- North Holland).
- RUB1901 Rubinovitz, H. H., and B. Thuraisingham, August 1990, ***Secure Distributed Query Processor Overview,*** MTR 10969, Vol.1, The MITRE Corporation, Bedford, MA (version published inthe Journal of Systems and Software, Vol. 21).
- Thuraisingham, B., July 1990, ***Handling ssociation-based Constraints in Multilevel Database Design,*** Workingaper, The MITRE Corporation (a version presented at the 4th **RADC** Database Security Workshop).