Integrating Policy-Driven Role Based Access Control with the Common Data Security Architecture

Along Lin
Extended Enterprise Laboratory
HP Laboratories Bristol
HPL-1999-59
April, 1999

E-mail: alin@hplb.hpl.hp.com

policy-driven management, RBAC, CDSA, policy description language

This paper shows how Policy-Driven Role-Based Access Control (PDRBAC) techniques can be used to extend the Common Data Security Architecture (CDSA). The extensions provide constraint-based access control and are implemented using a flexible policy description language and a new trust policy enforcement mechanism. The expressiveness of the policy description language is demonstrated by examples and the integration of the policy enforcement mechanism with CDSA is described.

Integrating Policy-Driven Role Based Access Control With The Common Data Security Architecture

Along Lin

Hewlett-Packard Laboratories
Filton Road, Stoke Gifford
Bristol BS34 8QZ, U.K.
Email: alin@hplb.hpl.hp.com

Abstract

This paper shows how Policy-Driven Role-Based Access Control (PDRBAC) techniques can be used to extend the Common Data Security Architecture (CDSA). The extensions provide constraint-based access control and are implemented using a flexible policy description language and a new trust policy enforcement mechanism. The expressiveness of the policy description language is demonstrated by examples and the integration of the policy enforcement mechanism with CDSA is described.

Keywords: Policy-Driven Management, RBAC, CDSA, Policy Description Language.

1 Introduction

Due to the wide acceptance of the Internet and the Web, it has become much easier for users to access various resources such as files, directories, databases, web pages or even services using CGI-bins, servlets over the Internet. However, there are some challenging issues such as authentication, integrity, privacy and authorization, which must be addressed in a manageable and cost-effective manner.

Existing access control mechanisms are inflexible and do not help in alleviating the management task of administrating users accesses to resources. Discretional Access Control (DAC) can be used to restrict users to perform authorized operations on specified resources. Each time either a user is added into/removed from a user list or a resource is added /removed, security managers have to administer relevant ACLs, user and resource lists, for the internet applications - implying DAC is not an ideal solution. Mandatory Access Control (MAC) can be used to alleviate the management task by labeling the sensitivity levels of resources. However, when constraints-based or more fine-grained accesses to resources are required, MAC also has limitations.

Users privileges in accessing resources are based on an organizations security policy. The goal of this work is to provide mechanisms whereby any change in organizational policy will drive a change in the security policy, thus allowing flexible security management [14].

Role-Based Access Control (RBAC) can be adopted to alleviate the administration of users and resources [1, 3, 5, 6, 7]. In RBAC, each role has a set of privileges for operating on some resources. Although a user may play several roles during a particular period of time, his/her privileges must comply with the organizational policies. A user's roles may reflect his/her current position, responsibility and job requirements in the organization. Because access permissions on resources are only associated with roles rather than individual users, the complexity of administrating users and managing resources is greatly simplified. Any change to users has no effect on the managed resources, and vice versa (see Figure 1).

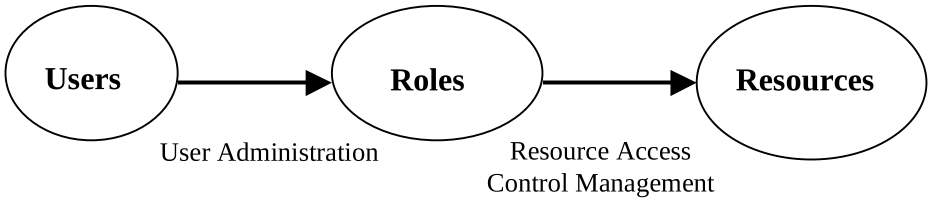


Figure 1. **Role-Based Access Control**

Policy-Driven RBAC (PDRBAC) is useful in this situation for a number of reasons. Firstly, the Internet supports large numbers of both users and resources, and the mapping of users to resources can change dynamically. Thus, if the traditional DAC were used this would generate a significant amount of administration. Secondly, within organizations, users can be categorized into a set of comparatively stable roles based on organization's business processes. Thirdly, because security policies vary from one corporation to another and may change, they must be deployed flexibly and dynamically rather than being hard coded in advance. Finally, roles may form a hierarchy, in which a role's privileges may be inherited by other roles. Compared with DAC, this can be a big advantage in mapping the business requirements to the security mechanisms.

CDSA is an open, standards-based, industry accepted framework, supporting extensible, interoperable, cross-platform secure applications development. Since the Open Group adopted CDSA 2.0 specification in December 1997, it has been widely supported. It has the following four layers:

- Applications
- Layered Services, Middleware, Language Interface-adapter and tools
- Common Security Services Manager (CSSM)

- Security Add-in Modules such as Cryptographic Service Provider (CSP), Certificate Library (CL), Trust Policy Module (TP) and Data Storage Library (DL)

TP modules implement policies defined by authorities or institutes. The semantics of a trust policy is completely defined by its module developer. Once a TP module is released, users are unable to add new policies or even to modify existing policies. From the users perspective, this is not ideal since they may want to define and enforce some specific policies themselves in their application domains. To achieve this, it would be useful to extend the current CDSA architecture to include a separate policy interpreter. The advantage of the extension is that both the CSSM and other add-in modules can enforce their own policies.

In the following chapters, we first discuss Policy-Driven RBAC by a database access control example. The underlying ideas, however, can be generalized to control the accesses to other resources. Secondly, a policy description language is introduced and its expressiveness is demonstrated by writing a set of policy examples. Finally, a policy based CDSA framework extended by trust policy enforcement mechanism is presented.

2 Policy-Driven RBAC

Existing commercial Data Base Management Systems (DBMSs) suffer from a common difficulty in specifying and enforcing access control policies in a flexible and dynamic way[2]. For business requirements or commercial reasons, we need fine-grained access controls on those legacy databases containing sensitive information especially as it is now accessible to Internet users. Even though some DBMSs have provided security managers with fine-grained access control mechanisms, security managers are still unable to specify and enforce policies flexibly and dynamically with controls logic being evaluated at run-time. An authorized user may be given several privileges on a specific database, such as ***create***, ***open***, ***close*** a database and ***delete***, ***insert***, ***select***, ***update*** database records.

Let us take a simplified Performance Evaluation (PE) database as an example. Each record has several fields. ID, containing the information about an employee such as employee number, name, rank, hired date, job title and the date of last PE, which only Human Resource (HR) staff can modify. Furthermore, only HR staff are authorized to change the fields Evaluating Manager (EM) and Reviewing Manager (RM) of a PE. If a manager has completed an employee's PE, he will mark it as Finished (F). Employee Comments (EC) can be optionally filled in by an employee and Employee Signature (ES) can only be signed if the employee has agreed to the written PE before the evaluating manager signs it by filling in the field EMS. The Reviewing manager signs it by filling in the field Reviewing Manager's Signature (RMS). An employee's PE will not become valid until it has been signed by its reviewing manager.

During the processing of a PE, the following policies need to be enforced:

1. An employee can modify the comments of his/her PE which has not been signed by the reviewing manager of the PE.
2. An employee can only sign his/her PE which has been finished.
3. A PE's evaluating manager can update an employee's PE until it is finished.
4. An evaluating manager signs an employee's PE after the employee has signed it.
5. A reviewing manager can not sign a PE before both the employee and the evaluating manager have signed it.
6. An HR staff can change an employee's ID, evaluating manager or reviewing manager but s/he can not change these fields of his/her own PE.

From the policy descriptions, it is observed that a user of a particular role accesses different fields under different constraints, which means any policy description language must be expressive enough to describe those constraints. Generally, policies to be enforced are evaluated dynamically by a policy interpreter at run-time. Following is another policy example on a bank transaction:

A bank clerk (User) is authorized to transfer (Amount) pounds from account (From) to account (To) only if the amount is less than 100,000 pounds.

If we want the bank transaction to be more secure, it is desirable to add two more constraints under which the specified operation is performed.

- The transferred amount of money must be less than the balance of From account.
- If the transferred amount is more than 5,000 pounds, it must be done between 9:00AM and 18:00PM.

Therefore, besides being expressive, high-level and easy-to-use, a policy description language must be flexible enough to allow the extension of a policy by adding or modifying its constraints easily without affecting other policies. Policy-driven access control is more flexible than any other existing access control mechanisms. Reducing the burden of administering users and resources is possible by using RBAC. The next section shows how the policies needed for policy-driven management and RBAC are represented within prototypes recently developed as part of our ongoing research..

3 Policy Representation

Whatever the resources are, an access control policy description language must be able to express constraint templates, containing variables. Of all existing high-level programming languages, PROLOG (PROgramming in LOGic) is the most suitable policy writing language due to its following features:

- It is declarative. A rule in PROLOG defines a relationship between several objects. Writing a policy in PROLOG is almost the same as describing what the policy is.
- It provides procedural readings, which supports writing more powerful policies.
- It is based on a subset of the First Order Logic, thus having a solid mathematical foundation, some properties such as soundness and completeness can be guaranteed.
- It supports backtracking and can express non-deterministic constraints.
- It is a unification-based language, which allows writing policy templates.
- It is a productive modeling language supporting incremental policy writing and refinement.
- It is able to reason from a set of PROLOG rules and supports meta-level reasoning, thus making policy conflict detection possible.
- It supports recursive programming, infinite and recursive data structures. It can do garbage collection automatically and frees programmers from having to deal with memory allocation.
- All of the data structures can be uniformly expressed as a structure, thus simplifying its processing.

Now, let us start to model the access control policies on the PE database in PROLOG. In order to use RBAC, employees in an organization are first categorized into a set of basic roles such as Employee, Evaluating Manager (Manager), Reviewing Manager (Reviewer) , HR Staff. Because the PE database requires more fine-grained access control, the fields of its records are the minimum resource units to be controlled. For simplicity, it is assumed that a PE database has been established and we just focus on the operation ***update***. The issues of role hierarchy and privilege inheritance will not be covered here.

As described before, each PE database record contains following fields:

ID	PE	F	EM	RM	EMS	RMS	ES	EC

Role	**Operation**	**Fields**	**Constraints**
employee	***update***	EC	employeeCanUpdateEc(User, Record)
employee	***update***	ES	employeeCanUpdateEs(User, Record)
manager	***update***	PE, F	managerCanUpdatePeF(User, Record)
manager	***update***	EMS	managerCanUpdateEms(User, Record)
reviewer	***update***	RMS	reviewerCanUpdateRms(User, Record)
hRStaff	***update***	ID, EM, RM	hRStaffCanUpdateIdEmRm(User, Record)

The semantics of predicates corresponding to constraints will be defined later. It should be noted that on the one hand, the constraints change for different combinations of roles, operations and fields. On the other hand, users of different roles can perform the same operation on the same set of fields in a record under different constraints. In PROLOG, variables begin with a capital letter or an underscore. Anonymous variables are represented by "_". All other identifiers are treated as atoms. There are two ways of expressing policies: facts and rules. A fact is a rule whose body is true. It should be pointed out that RBAC is a special case of PDRBAC where there are no access constraints, in which case policies are facts.

First, we define a predicate of five arguments as follows, declaring that a user of a particular role can operate on a set of fields in a record if constraints are evaluated true.

```
can_Perform_Operation_On_Resource(User, Role, Operation, Fields, Record) :-
        constraints(User, Record).
```

Then, a relation userIsThePEFieldSpecifiedPerson is defined, which says that a user is the person whose id is the same as that of the specified field of the PE being processed.

```
 userIsThePEFieldSpecifiedPerson (User, Field, Record):-
       getFieldValueFromRecord(Record, Field, Value), // get the Field value
       getEmployeeNo(Value, EmpNo),        // get the employee No. of Field
       getEmployeeNo(User, EmpNo).         // test if the PE record is the user's PE
```

Now, access control policies for the PE database can be represented in PROLOG as follows:

```
can_Perform_Operation_On_Resource(User, employee, update, [ec], Record) :-
        employeeCanUpdateEc(User, Record).
can_Perform_Operation_On_Resource(User, employee, update, [es], Record) :-
        employeeCanUpdateEs(User, Record).
can_Perform_Operation_On_Resource(User, manager, update, [pe, f], Record) :-
        managerCanUpdatePeF(User, Record).
can_Perform_Operation_On_Resource(User, manager, update, [ems], Record) :-
        managerCanUpdateEms(User, Record).
can_Perform_Operation_On_Resource(User, reviewer, update, [rms], Record) :-
        reviewerCanUpdateRms(User, Record).
can_Perform_Operation_On_Resource(User, hRStaff, update, [id, em, rm], Record) :-
        hRStaffCanUpdateIdEmRm(User, Record).

employeeCanUpdateEc(User, Record)      :-
    getFieldValueFromRecord(Record, rms, unsigned),
    // check if the PE record has not been signed by its reviewing manager
    userIsThePEFieldSpecifiedPerson (User, id, Record).
employeeCanUpdateEs(User, Record)      :-
    getFieldValueFromRecord(Record, f, finished),    // if PE has been finished
    getFieldValueFromRecord(Record, es, unsigned), //and PE has not been signed
    userIsThePEFieldSpecifiedPerson (User, id, Record).
```

```
managerCanUpdatePeF(User, Record) :-
    getFieldValueFromRecord(Record, f, unfinished), // check PE is unfinished
    userIsThePEFieldSpecifiedPerson (User, em, Record).
managerCanUpdateEms(User, Record) :-
    getFieldValueFromRecord(Record, f, finished), // check PE has been finished
    getFieldValueFromRecord(Record, es, signed), // Employee of the PE has signed it
    getFieldValueFromRecord(Record, ems, unsigned),
    // check if the PE has not been signed by the PE evaluating manager
    userIsThePEFieldSpecifiedPerson (User, em, Record).
reviewerCanUpdateRms(User, Record) :-
    getFieldValueFromRecord(Record, ems, signed),//evaluating manager has signed it
    getFieldValueFromRecord(Record, rms, unsigned),
    // check if the PE has not been signed by its reviewing manager
    userIsThePEFieldSpecifiedPerson (User, rm, Record).
hRStaffCanUpdateIdMRm(User, Record)        :-
    not userIsThePEOwner(User, id, Record).
```

Therefore, at some decision-making point, if we can guarantee that a specified operation is executed only if the constraints associated with it are evaluated true, the policy can be enforced. For securing database accesses, the control point is when a record is read, where a trusted program calls a policy interpreter. If the policy is evaluated false, the operation is denied. By adding the trusted program over existing DBMSs, we can make all legacy databases as secure as we want them to be.

The policy on a bank transaction can be expressed as follows:

```
isAuthorizedToTransferBetweenAccounts(User, Amount, From, To) :-
    Amount < 100000.00 , // The amount to be transferred must be less than 100,000.00
    getBalanceOFAccount(From, Balance),              // get the balance of account From
    Amount < Balance,              // The transferred amount must be less than the balance
    isValidTimeForTheAmount(Amount).  // Check the time validity
isValidTimeForTheAmount(Amount) :-
    Amount <= 5000.00 .   // no time restrictions
isValidTimeForTheAmount(Amount) :-
    getCurrentTime(Now), // retrieve the current time from the system
    isWithinTimePoints(Now, "09:00:00", "18:00:00").
    // test if the current time is between 9:00AM and 18:00 PM
```

In this case, three PROLOG built-in predicates are used. The semantics of each of them is clear from the relation signature (e.g. consisting of a predicate name and its arguments). The predicates getBalanceOfAccount and getCurrentTime are used to retrieve information about a bank account and the system time. When they are evaluated true, the variables Balance and Now will be unified to the real balance of a bank account and real system time. Most PROLOG systems (e.g. B-PROLOG) allow users to define as many system built-in predicates like these as they want.

It is evident that expressing policies will require the modeling of a system or an application domain. Fortunately, PROLOG is a powerful modeling language. For a restricted domain, it is helpful to have a domain modeling tool. A more detailed discussion of modeling a system or a domain based on logic can be found in [8, 9].

4 Extending CDSA by PDRBAC

CDSA 2.0 (see Figure 2) is an open, standards-based, industry-accepted framework. It has four layers and supports a complete set of security services provided by add-in modules. It is a flexible architecture allowing software/hardware vendors to provide new add-in security service modules, and it also supports interoperability between applications.

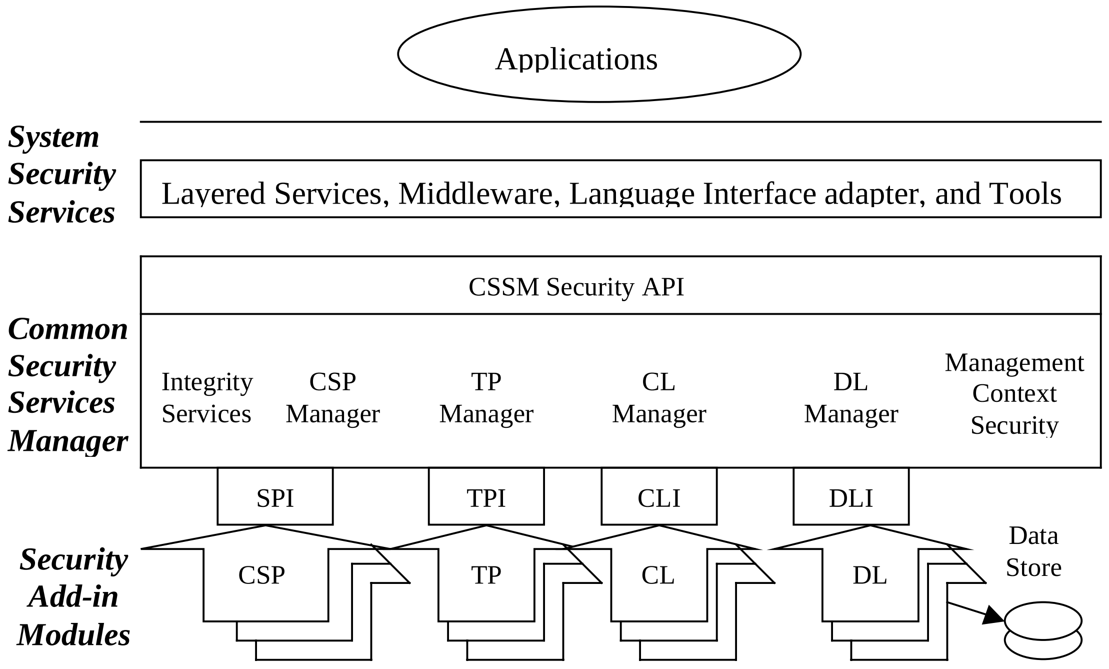
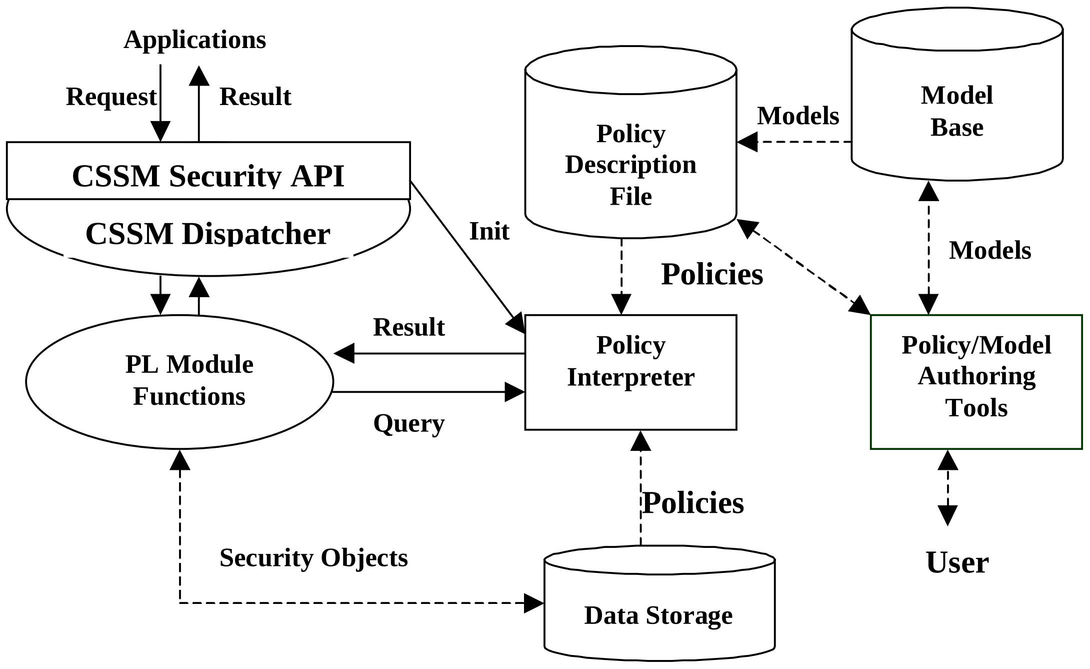


Figure 2. Common Data Security Architecture

Each Trust Policy (TP) module implements policies defined by its developer. However, it would be better if administrators were able to define policies for their application domains and/or modify deployed policies. One approach is to develop a new module called Policy Library (PL) which enforces policies provided by different authorities or users. PL provides a set of standard TP services and the semantics of each policy will be completely defined by an external policy description file or policy database so that a TP module can be completely replaced by PL and some specific Policy Description File (PDF) or data storage. This can be expressed as follows:

$$TP_i = PL + PDF_i$$

The policy enforcement mechanism of PL is shown in figure 3 below. Because there will be no TP modules, TP manager may be removed from the CDSA framework, which will be discussed below. All a trust policy module developer needs to do is to write his/her own trust policies. PL will be developed once for all, which greatly simplifies current TP module development. The issues of trust policy writing and policy integrity checking can be easily solved.



Where, directed solid lines mean control flows and dotted lines are data flows.

Figure 3. Policy Enforcement in PL

Within applications, users first call a CSSM initialization API, then ask the CSSM to attach itself to the PL (the policy interpreter can either be initialized during the PL attachment or when CSSM is initialized). The PL supports a PassThrough API which can evaluate a query based on a set of policies. In functions defining each of other APIs of a standard TP module, the PassThrough function will be called first to enforce the policies associated with the particular API. In this way, both add-in modules and CSSM are capable of enforcing their own security policies by calling a PL PassThrough API.

This is a significant advantage over the original CDSA framework. To integrate PDRBAC with CDSA, security managers need firstly to model the roles and privileges of an organization in a model base, then write access control policies in a policy description file.

As an alternative approach, we can provide a DLL (on NT) or a shared library (on HP-UX) so that CSSM and all add-in modules can enforce their own policies by calling the policy interpreter directly rather than by calling the PL PassThrough API. One advantage of this solution is its high efficiency. In addition to this, the PL module manager will not be needed if the trust policy enforcement mechanism becomes a part of CSSM. In this case, CSSM will be extended to have some APIs dealing with trust policy description files in order to replace current TP modules. However, from users point of view and for legacy CDSA applications, the interface should be backwards compatible. Therefore, it is worth providing both a shared library of policy interpreter and a PL.

It has been observed that some of the APIs in CSSM, CSP, CL and DL can be policy-based. Some examples are:

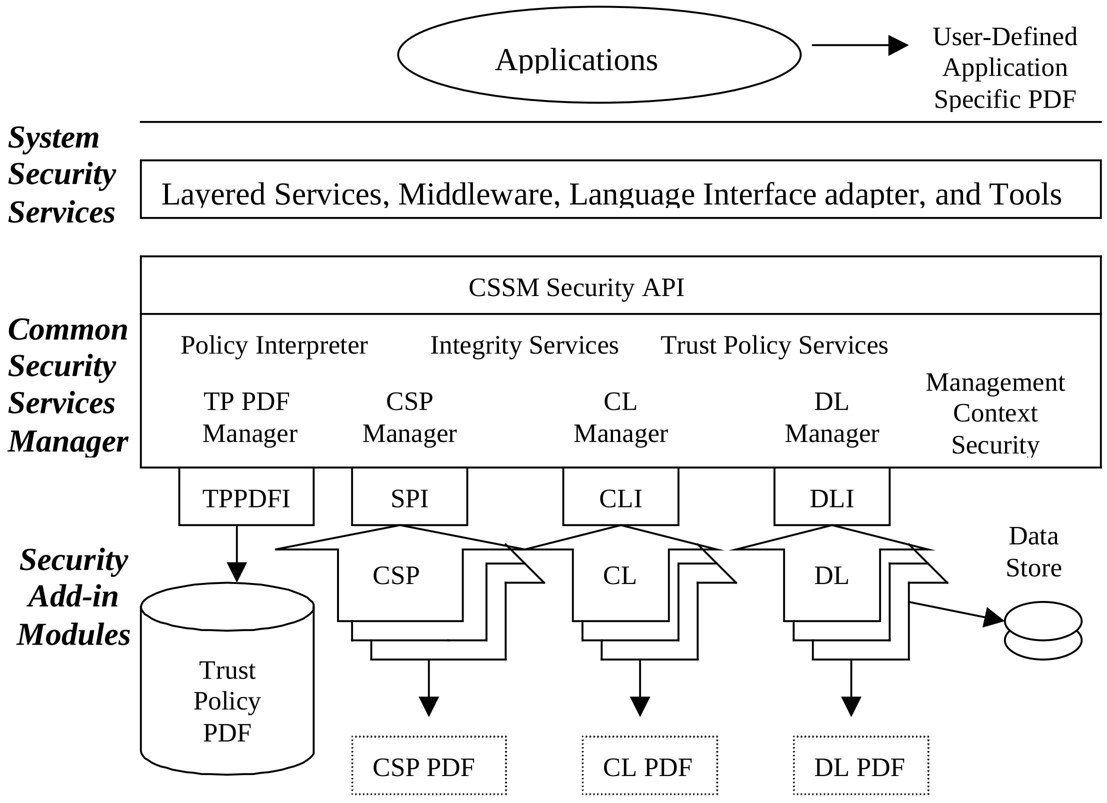
- Private key storage in a CSP needs to enforce the policy that the CSP must not reveal key material unless it has been cryptographically protected.
- When policies are put in a data store, they must be at least password protected.
- Before the CSSM can load a Trust Policy Description File (TPDF) into a system, it must verify the signatures on the TPDF.

In a Policy-Based CDSA implementation (shown in Fig. 4), PL is dropped and only its PDF remains. TP PDF manager provides TP PDF management services such as installing, uninstalling a TP PDF, attaching / detaching a TP PDF to/from CSSM. After a TP PDF has been installed in a system, it can be attached by CSSM and will be used by a set of TP services, which are now provided by CSSM based on the attached TP PDF. What really happens in the CSSM is that policy interpreter will be informed of the attached TP PDF so that the trust policies within it will be enforced. When the TP PDF is detached, CSSM will provide a set of default trust policy services.

In current CDSA, if the CSSM does not attach to any TP module, there will be no trust policy services provided. Obviously, the same idea can be applied to CSSM, CSP, CL, DL and even applications, which can be achieved either by providing each of them a PDF or extending the APIs by an extra PDF parameter.

Regarding the PDF provided by a user at the application level, one possibility is that a user's application will be provided with a parameter, which is a PDF for the particular application domain. Even though CSP PDF, CL PDF and DL PDF are provided by their corresponding developers, as is the TP PDF, they are invisible to users - which is different from the TP PDF. This is because a TP can be replaced with a generic PL and a TP PDF whereas it is hard for a CSP, CL or DL to be dealt with in the same way.

Therefore, there is only one TP PDF manager rather than a PDF manager. For some of APIs in a CSP, CL or DL, it may be better to extend them with an extra parameter of PDF rather than developing the whole add-in module as policy-based. Comparatively speaking, the PDL used to write a PDF for a CSP, CL, DL or some other APIs does not need to be as expressiveness as PROLOG.



Where, PDF denotes Policy Description File, and TPPDFI means TPPDF Interface. The parts enclosed by dashed lines are optionally provided by add-in module developers and are invisible to users.

Figure 4. Policy-Based Common Data Security Architecture

Now let us address the example problem, from the Human Resources example presented earlier – that of processing performance evaluations. As assumed before, a PE database has been established. Users want to access different fields in a PE record according to their roles based on a well-defined set of security policies over the Internet. For simplicity, the issues of users registration and authentication are not discussed. In order to control users accesses to the PE database remotely, a client/server architecture may be adopted. Users can access the PE processing service by launching a web browser and going to some URL. After the server receives the request and constructs a query based on the user's role and requirements, a trusted CGI-bin or a Java servlet (if it is a Java web server) accesses the database based on the query. During the processing of each record, the trusted program calls CSSM PL API with the query and a policy description file, returning *true* or *false*. Based on the result, the trusted program makes a decision whether it should allow the user to perform the specified operation on particular fields in the record. If later some security polices on the users accesses to the database need to be changed, nothing will happen except that new security policies will be enforced. Actually, the trusted program is an application of our extended CDSA framework and the access control policies are at the application-level and user-definable, domain specific. Most gateways and firewalls can be adapted to use the proposed mechanisms to enforce their security policies.

5 Conclusions

In this paper, the ideas behind policy-driven role based access control and a proposal on how to extend the current CDSA framework using it are proposed. The constraint policies for accessing resources are designed and expressed in PROLOG. By integrating policy-based management and RBAC, the administration of users accesses to resources becomes less of a burden to system administrators. CDSA 2.0 is an open, standards-based, industry-accepted framework, and has been adopted by the Open Group. It supports interoperability and a consistent, comprehensive set of security services. Its trust policy add-in modules implement policies defined by module developers. Having been extended, policy-based CDSA will allow users to define and enforce domain-specific policies. One advantage of this extension is that the CSSM itself and all add-in security modules are capable of defining and enforcing their own policies. Furthermore, because any TP can be replaced by a generic TP module PL and a particular trust policy description file, the improved CDSA framework will greatly simplify current TP development. Once the policy enforcement service becomes an important extension of CSSM, TP manager and TP will no longer be needed. At that time, CDSA will be totally policy-based and security management will be much more flexible and easier. The extra benefit of policy-based CDSA is that it can provide a set of default trust policy services without attaching a TP. The capability of the new framework has been demonstrated by an example.

In this paper, the issue of distributing policy description files is not discussed. However, it is important for an organization to deploy the same set of trust policies consistently. One possible approach is to use LDAP service. Another approach is to provide a trust policy service based on CORBA architecture.

Possible future research topics include:

- developing a knowledge-based authoring tool for policy/model writers [11],
- making some APIs of DL, CL, CSP and CSSM policy-based,
- investigating efficient approaches to policy conflict-detecting [10, 12],

- researching the possibility of automatically generating policies from high level policy descriptions,
- deploying trust policy services across a business community based on CORBA architecture.

Acknowledgement

The work described in this paper has been part funded by the European Union's ESPRIT Programme within Project MultiPLECX (ESPRIT No. 26810).

I would like to thank following people for their reviews and comments: Dave Clarke, Richard Brown, Dirk Kuhlmann, Christopher Milsom, Sandy Johnstone, Jörg Bartholdt, at Hewlett-Packard Laboratories, United Kingdom. Particularly, I would like to thank Dave Clarke, without whose encouragement and strong support this paper would not have been finished.

References

[1] R. Anderson, A security policy model for clinical information systems. In *Proc. of the Symp. on Security and Privacy, 1996.*
[2] R. W. Baldwin, Naming and Grouping Privileges to Simplify Security Management in Large Databases, In *Proc. of the IEEE Symposium on Computer Security and Privacy,* 1990.
[3] J. Barkley, Implementing Role Based Access Control Using Object Technology, *First ACM/NIST Workshop on Role-Based Access Control*, 1995.
[4] D. Clark and R. Wilson, A Comparison of Commercial and Military Computer Security Policies, In *Proc. of the Symp. on Computer Security and Privacy*, 184-194, 1987.
[5] David F. Ferraiolo and Richard Kuhn, Role-Based Access Control, In *Proc. of the 15th NIST-NSA Nat. (U.S.) Comp. Security Conf.*, 554-563, 1992.
[6] David F. Ferraiolo, Janet A. Cugini, D. Richard Kuhn, Role-Based Access Control (RBAC): Features and Motivations, In *Proc. of 11th Annual Computer Security Applications Conf.*, 1995.
[7] L. Giuri and P. Iglio, A formal model for role based access control with constraints, In *Proc. of the Computer Security Foundations Workshop*, 136-145, 1996.
[8] A. Lin, A Logic-Based Approach to Automated System Management, In *Proc. of the 6th Int. Conf. on Practical Applications of PROLOG* (*PAP98*), 417-425, March 1998.
[9] A. Lin, A Model-Based Automated Diagnosis Algorithm, *11th Int. Conf. on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems* (*IEA/AIE-98*), LNAI 1415, 848-856, June 1998.
[10] F. Massacci, Reasoning about Security: a Logic and a Decision Method for Role-Based Access Control, In *Proc. of the International Joint Conference on Qualitative and*

Quantitative Practical Reasoning (ECSQARU/FAPR-97), Vol. 1244 of *Lecture Notes in Artificial Intelligence*, 421-435, 1997.
[11] J. Moffett and M. Sloman, Policy Hierarchies for Distributed Systems Management, *IEEE JSAC*, *Special issue on Network Management*, 11(9), December 1993.
[12] J. Moffett and M. Sloman, Policy Conflict Analysis in Distributed System Management, *Journal of Organizational Computing*, 1994.
[13] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, Role-based access controls models, *IEEE Computer*, 29(2), February 1996.
[14] M. Sloman, Policy Driven Management for Distributed Systems, *Journal of Network and Systems Management*, 2(4), 333-360, 1994.