Implementing Role Based Access Control using Object Technology

John Barkley
NIST
B266 Tech
Gaithersburg MD 20899
(301) 975-3346
jbarkley@nist.gov

November 28, 1995

With Role Based Access Control (RBAC), each role is associated with a set of operations which a user in that role may perform. The power of RBAC as an access control mechanism is the concept that an *operation* may theoretically be anything. This is contrasted to other access control mechanisms where bits or labels are associated with information blocks. These bits or labels indicate relatively simple operations, such as, read or write, which can be performed on an information block. *Operations* in RBAC may be arbitrarily complex, e.g., "a night surgical nurse can only append surgical information to a patient record from a workstation in the operating theater while on duty in that operating theater from midnight to 8 AM." A goal for implementing RBAC is to allow operations associated with roles to be as general as possible while not adversely impacting the administrative flexibility or the behavior of applications.

Consider the possible activities associated with defining and modifying roles:

- Add a role and its associated operations.[1]
- Remove a role and its associated operations.
- Modify an existing role:
  - Add an operation.
  - Remove an operation.
  - Modify an existing operation.

Information is usually accessed by applications based on a fixed set of operations defined by the mechanism or processor which is used to access the information. Applications are

[1]Some operations may be available to more than one role, e.g., a credit account may be read by both a bank teller and a bank supervisor.

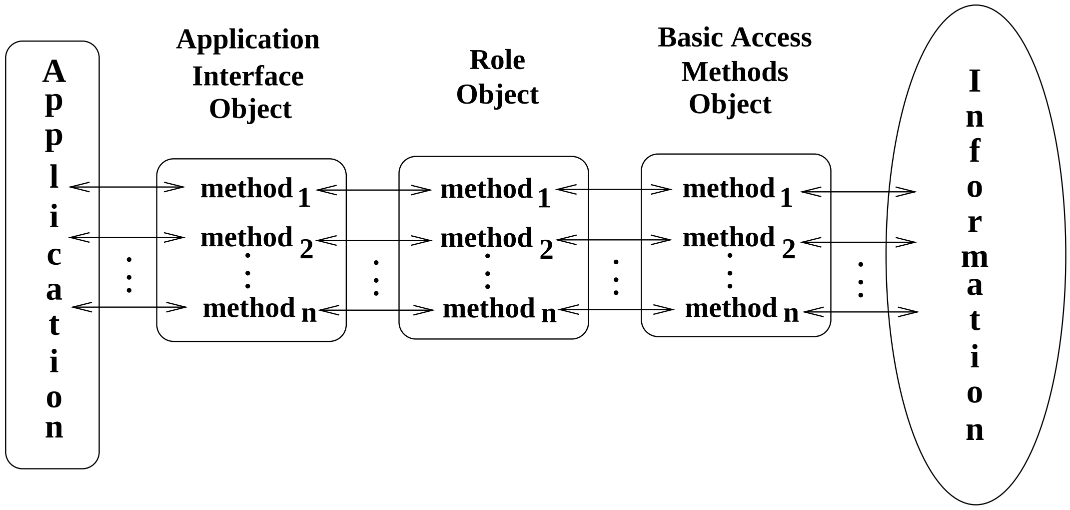

Figure 1: Implementing RBAC with layered objects

built based on a fixed set of operations which they routinely perform. For example, Unix files are accessed by the operations defined by the procedures: *open(), close(), read(), write(), fseek(), etc.*; tables in a relational data base are accessed by the operations defined by SQL.

Modifying the operations available to an application can have a great impact on an existing application. Removing an operation or modifying the semantics of an operation seriously affects an application's functioning and can produce very unpredictable results.

One approach which can be used to maintain flexible administration, minimize impact on applications, and maintain a significant capability for defining complex role operations is to use Object Technology in the following manner (see fig. 1). A complete set of operations based on access methods associated with the information storage mechanism is defined and held fixed. These are the operations that are made available to an application. These operations become the methods in a *basic access methods* class.

Access control for the basic access methods class is provided by *role* classes, one for each defined role. The methods of the role classes have the same names, types and parameters as the methods of the basic access methods class. Access control to the information accessed by the basic access methods class is located exclusively in the role classes and not in any other part of the application. The bodies of the methods in the role classes are restricted to:

- conditionals which determine access for the role associated with that role class; and/or
- filters which constrict the flow of information between the application interface and the basic access methods.

If access is permitted for a role, the methods of the role class then invoke the corresponding methods of the basic access methods class. If not all information obtained by the basic access methods is permitted to a role, then the parts of the information not permitted can be filtered out. Filtering may be more desirable in a application rather than generating an access violation for the entire information block.

```
class Access_PRDBO{
        public:
                Idlist GetIdinfo();
                Patrec GetPR(Patid pid);
};
```

Figure 2: Example basic access methods class for accessing patient information

The methods of the application interface class also have the same names, types and parameters as the methods of the basic access methods class. The methods of the application interface class invoke the corresponding methods of the role classes. It is the methods of an application interface object which the application invokes. Given the current role associated with the application, the methods of the application interface object select the appropriate role object.

This approach has the following advantages:

- Applications need not change when access conditions for roles are changed.

  Applications use the methods of the application interface class whose methods have the same names, types, and parameters as the methods in the basic access methods class. The methods of the application interface class and the methods of the basic access methods class are fixed and remain constant over time. When access conditions for roles change, applications fail only because of access violations. This type of failure is comparable to the failures that typically occur when information protection bits or labels are changed. Applications are normally implemented to be able to handle access violations.

- Access conditions for roles are easily changed.

  Access conditions for roles are located exclusively within the role classes. Consequently, role policy changes do not require modifications to the applications themselves. One can conceive of a simple language, suitable for use by data and security administrators, for expressing access conditions restricted to conditionals and filters. A processor for such a language could generate the role objects and place them in the libraries used by applications. Most environments today support dynamically linked libraries which link when an application is loaded into memory for execution. Thus, applications do not need to be relinked when role classes are changed. This ability to easily change access conditions associated with roles permits rapid response to policy changes.

The following example in C++ illustrates the approach. See:

http://waltz.ncsl.nist.gov/rbac/vision/proj/applint.cc.txt for the complete C++ example which may be compiled and run. In actual practice, RBAC roles, operations, and policy can be numerous and complex. In order to simplify this example, only a small subset of the roles, operations, and policy that would normally be required are illustrated.

This example has the following operations which can be performed by applications on a patient record database:

Get patient ID list This operation obtains a complete list of patient names and their IDs.

Get patient record This operation obtains the patient record given the patient ID.

Figure 2 shows C++ code for a basic access methods class (*Access_PRDBO*) which has methods (*GetIDinfo()*, and *GetPR()*) for performing these operations.

Figure 3 shows C++ code for role classes associated with a patient (*Pat_PRDBO*) and doctor role (*Doc_PRDBO*). These role classes inherit from a base class (*Role_PRDBO*) which defines the names, types, and parameters for the methods which correspond to the methods in the basic access methods class. The patient and doctor role classes together implement the following RBAC policy:

- Only Doctors are permitted to read the list of patient names and IDs.
- Doctors are permitted to read the records for all patients.
- Patients are only permitted to read their own record.

In order to ensure that patients only access their own records, the patient role object (*Pat_PRDBO*) calls a system procedure which returns the patient ID for the user.

Figure 4 shows the application interface class (*PRDBO*) used by applications. When an object of this class is instantiated and a method of that object is called, that method first calls a system procedure (*get_role()*) which returns the user's current role. The method then calls another system procedure (*get_role_obj()*) which returns a pointer to the role object for that role. This procedure is shown in Figure 5. Finally, the method calls its corresponding method in the role object passing its input arguments to the role object method.

```
class Role_PRDBO{
        public:
                virtual Idlist GetIdinfo()=0;
                virtual Patrec GetPR(Patid patid)=0;
};


class Pat_PRDBO:public Role_PRDBO{
        public:
                virtual Idlist GetIdinfo(){
                   return("ERROR: patient cannot access patient id list\n");
                };
                virtual Patrec GetPR(Patid pid){
                   if (pid == get_user_pid())
                        return(access_prdbo.GetPR(pid));
                   else
                        return("ERROR: patients cannot get other´s records\n");
                };
};

class Doc_PRDBO:public Role_PRDBO{
        public:
                virtual Idlist GetIdinfo(){
                   return(access_prdbo.GetIdinfo());
                };
                virtual Patrec GetPR(Patid pid){
                   return(access_prdbo.GetPR(pid));
                };
};
```

Figure 3: Example role classes for accessing patient information

```
class PRDBO{
        public:
                Idlist GetIdinfo(){
                   char * role_name;
                   Role_PRDBO *roleobj;
                   role_name = get_role();
                   roleobj = get_role_obj(role_name);
                   if (roleobj == (Role_PRDBO *)NULL)
                                        return("ERROR: no such role\n");
                   return(roleobj->GetIdinfo());
                };
                Patrec GetPR(Patid patid){
                   char * role_name;
                   Role_PRDBO *roleobj;
                   role_name = get_role();
                   roleobj = get_role_obj(role_name);
                   if (roleobj == (Role_PRDBO *)NULL)
                                        return("ERROR: no such role\n");
                   return(roleobj->GetPR(patid));
                };
};
```

Figure 4: Example application interface class for accessing patient information

```
Role_PRDBO *get_role_obj(char *role_name){
        struct{
        char role_name[ROLE_NAME_LENGTH];
        Role_PRDBO *role_object;
        } role_tab[NUMBER_OF_ROLES] =
                {
                  {"patient", &pat_prdbo},
                  {"doctor", &doc_prdbo}
                };
        for(int i=0; i<NUMBER_OF_ROLES; i++)
                if (strcmp(role_name, role_tab[i].role_name) == 0)
                        return(role_tab[i].role_object);
        return((Role_PRDBO *) NULL);
};
```

Figure 5: Example procedure to locate the proper role object