OSF DCE SIG **D. Mackey (OSF)**
Request For Comments: 46.0 **R. Salz (OSF)**
October 1993

DCE ACL LIBRARY — FUNCTIONAL SPECIFICATION

1. INTRODUCTION

DCE ACL's are a very powerful mechanism, and the ability to do secure RPC's are a compelling reason to adopt DCE. Unfortunately, application developers quickly find three hurdles that must be crossed before they can effectively use these mechanisms. First, they must write an ACL storage facility. Second, they must implement the RDACL interface. Third, they must write routines to determine if an ACL grants the necessary rights to the requesting client. Each task is difficult enough, and taken together they are a daunting amount of work.

The DCE daemon [RFC 47] is a replacement for `rpcd` that will be part of DCE 1.1. Since `dced` needs to have an ACL manager, we will have to address the three hurdles. We would like to provide a library that will do almost all of the work needed to add an ACL manager to a DCE server.

The first hurdle can be solved by using the Backing Store Library [RFC 45]. Based on the experience we gained writing the DCE 1.1 Serviceability ACL manager (which, in turn, was based on the `dpeacl` code in DTS[1]), we believe it is possible to solve the second hurdle such that an application need only write a small, well-defined, ''callback'' function. The third hurdle can be solved by providing the support routine mentioned above: compare a PAC (more properly, a DCE 1.1 EPAC [RFC 3]) against an ACL and a desired permission set and return a boolean answer.

This document specifies a library to be part of DCE 1.1 that leaps all hurdles in a single API.

2. TERMINOLOGY

The first part of this section gives an introduction to the DCE ACL datatypes. Readers familiar with the `rdaclif.idl` and `aclbase.idl` files can skip it. The second part gives an introduction to the DCE Backing Store Library. Readers familiar with DCE RFC 45 can skip it.

2.1. DCE ACL Datatypes

An ACL permission set, `sec_acl_permset_t`, is a 32-bit word. Each server can interpret the permission set as it wants, but by convention each bit is interpreted individually. To further encourage standardized use, several public constants (of the form `sec_acl_perm_XXX`) are also defined.

Within a single C program, a programmer can hopefully read the source to understand how certain bits are used, but for distributed editing by humans more information is needed. The `sec_acl_printstring_t` datatype maps a set of permission bits to a short mnemonic and a human-readable description. Using this datatype generic tools like `acl_edit` can be written so a user can type ''rw'' knowing it means ''read and write'' rather than just entering the number ''3''. The strings are limited to the DCE Portable Character Set.

A server often manages different types of objects. For example, a name service will have directories and leaf nodes. The different types will interpret their permission bits differently and have different printstrings. The ''manager type'' is a UUID that distinguishes among the types. The manager type UUID is an input parameter for most RDACL operations.

---

1. The `dpeacl` library is an ACL manager based on the example ACL manager developed by Hewlett-Packard provided as ''pass-through'' code in the DCE distribution. It was intended to be used by both DTS and CDS although it ended up that CDS has its own ACL code. The ACL Library described here would not be based on the `dpeacl` code.

A single object can also have multiple ACL types. The most common use of this is probably to provide more permission bits than can be contained in a single permission set. When an object has multiple ACL's, they are said to be ''chained''; the manager type UUID distinguishes among them.

An ACL, `sec_acl_t`, contains the manager type UUID and a list of ACL entries, `sec_acl_entry_t`. Each entry has permission bits, a type, and type-specific data. For example, a `sec_acl_e_type_user` entry identifies the DCE principal for whom the permission bits apply.

The ACL entry types are defined by OSF; they are specified by the `sec_acl_entry_type_t` enumeration. There are several types of ACL entries, including groups and masks. It can be very difficult to correctly write the code that applies the client's privileges to an ACL and determine what permissions the client actually has. DCE assumes POSIX.6 D12 interpretation rules. The `sec_acl_posix_semantics_t` flag indicates whether or not an implementation supports the ''mask_obj'' semantics specified by POSIX.

Most objects have one ACL. If an object is a ''container,'' however, it will probably have two additional ACL's. The initial ACL, or IACL, specifies the ACL that is given to objects created within the container. (The `aclbase.idl` file gives IACL the misleading name `sec_acl_type_default_object` and container IACL the misleading name `sec_acl_type_default_container`.) The container IACL specifies the ACL given to sub-containers. (Sub-containers inherit their parent's IACL and container IACL.) For example, in a filesystem the ACL on directory ''/foo/bar'' specifies who can modify the directory's state. The IACL specifies the permissions given to files created within that directory such as ''/foo/bar/myfile'', and the container IACL specifies the permissions given to directories created within ''/foo/bar'' such as ''/foo/bar/subdir''.

2.2. Backing Store Library

The Backing Store Library provides simple storage for DCE application programmers. The term ''database'' is used in place of the more accurate term ''backing store'' because it is less cumbersome.

The `dce_db_open` routine opens an existing database or creates a new one, returning a handle of type `dce_db_handle_t` that should be used in subsequent calls. Items in a database are indexed either by UUID or string; the database index type is determined when it is created.

The data stored in the database is converted between internal and external formats by using the IDL Encoding Services, or ''pickling.'' When a program opens a database it must specify the (IDL-generated) function to use. Application programmers are encouraged to use a standard header as the first element in their database objects; parts of this ACL library require the standard header.

A program may open several databases; the Backing Store Library properly handles concurrency. Application programs may lock the database for their exclusive use.

3. TARGET

This document describes a convenience library for programmers writing DCE servers.

4. GOALS AND NON-GOALS

The goals of the ACL Library are:

(a) Provide stable storage for ACL's using the `dce_db_xxx` API.

(b) Implement the RDACL interface, including support for multiple object types and IACL's and containers.

(c) Implement the full access algorithm, including masks and delegation once it is developed.

(d) Provide DCE developers with a set of convenience functions so that servers can easily perform common styles of access control with minimal effort.

(e) Support copy-on-write ACL's by indicating to application code when an operation is modifying an ACL object. The application code may then create a new ACL object if it has a reference count greater than one.

The following are not goals (for this first release):

(a) The initial implementation of this library will allow only one ACL manager for each object type. In particular, an object will be limited to 32 permission bits. The library design and API should allow for more than this, however.

(b) Extended ACL entries (`sec_acl_e_type_extended`) will not be supported.

(c) Replication or referrals to ''master'' servers will not be supported.

(d) It is not a goal that application developers be able to use this library but replace the storage mechanism with one of their own. Vendors might be able to do this with source code changes.

5. REQUIREMENTS

The library should provide a complete and correct implementation of the RDACL interface. No protocols may be changed; existing servers with their own ACL managers and existing client programs (e.g., `acl_edit`) should continue to interoperate. The interface between the RDACL code and the server code should be small, simple, and clean.

The most common test a server will want to perform is ''Does this client have the necessary permissions?'' It should be trivial for a server developer to make a library call to perform this test. Related operations, such as ''What permissions does this client have?'' should be equally simple.

Servers should be able to treat the security data structures as abstract data types as much as possible.

6. FUNCTIONAL DEFINITION

The ACL Library provides simple and practical access to the DCE security model.

ACL's are objects, stored using the DCE Backing Store Library [RFC 45]. They are indexed by a DCE UUID; a second database provides access by name. The standard data prolog defined in the Backing Store Library includes a reference count field. If a server uses this field, it can share ACL objects, creating a new one only when needed. The library supports this by indicating to the server code when an ACL ''write'' operation is being done.

The library provides a routine so that a developer can make one call to see if a client has the right permissions to perform an operation. A server can also easily retrieve the full set of permissions granted to a client by an object's ACL.

The library provides the complete RDACL remote interface. The implementation uses a single function to map the remote parameters to the local ACL object. Standard routines are provided to map either a UUID attached to a handle or a residual name specified as one of the parameters.

The combination of these capabilities means that most servers will not have any need to use the DCE ACL datatypes directly.

7. DATA STRUCTURES

There are no user-visible data structures added by this library.

8. USER INTERFACES

There are no user interfaces added or changed by this library.

The existing `acl_edit` program and planned DCE shell [RFC 42] programs can be used to access the RDACL interface.

9. API'S

The ACL library can be divided into the following parts:

(a) Initialization routines, where the server registers each ACL manager type.

(b) RDACL implementation and server callback, where the server maps RDACL parameters into a specific ACL object.

(c) Server queries, where a server can perform various types of access checks.

(d) Creating ACL objects, where servers can create ACL's without worry about some of the low-level datatype details.

The next four sections explain each part.

9.1. Initialization Routines

A manager must first define the types of objects it manages. For example, a simple directory service would have directories and entries, and each type of object would have a different ACL manager. On a practical level, if a server has different types of objects, the most common difference between the ACL managers is the printed representation of its permission bits — that is, only the `sec_acl_printstring_t`'s are going to be different; the algorithm for evaluating permissions will be the same.

The ACL library will provide a global printstring that specifies the ''read'', ''write'', and ''control'' bits. Application developers should be encouraged to use this printstring whenever possible.

The following routine is called to register an object type:

```
void
dce_acl_register_object_type(
    dce_db_handle_t             db,
    uuid_t                      *manager_type,
    unsigned32                  printstring_size,
    sec_acl_printstring_t       *printstring,
    sec_acl_printstring_t       *mgr_help,
    dce_acl_resolve_func_t      resolver,
    void                        *resolver_arg,
    error_status_t              *st
);
```

The `db` parameter specifies the database where the ACL objects are stored; it must be indexed by UUID. The `uuid` parameter points to the UUID identifying the ACL manager. The server developer will have to provide the UUID. The ''printstring'' parameters identify the internal and printed representation of the permissions for the objects. The `mgr_help` parameter is a printstring that describes the object type. The `resolver` function, in conjunction with the specified argument, is used to map an RDACL request into an ACL object. It is described in detail below.

The `dce_acl_register_object_type` routine should be called once for each type of object that the server manages. A typical call is shown below. The sample code defines three variables: the manager printstring, the ACL printstrings, and the ACL database. Note that the manager printstring does not define any permission bits; they will be set by the library to be the union of all permissions in the ACL printstring. The code also uses the global `my_uuid` as the ACL manager type UUID. The ACL printstring uses the standard `sec_acl_perm_XXX` bits.

```
#include <dce/dceacl.h>

/* Manager help. */
sec_acl_printstring_t my_acl_help = {
    "me", "My manager"
};
```

```
/*
 * ACL permission descriptions; deliberately non-standard for
 * use as an example.
 */
sec_acl_printstring_t my_printstring[] = {
    { "r", "read", sec_acl_perm_unused_00000080 },
    { "f", "foobar", 0x01 },
    { "w", "write", sec_acl_perm_write },
    { "d", "delete, sec_acl_perm_delete },
    { "c", "control", sec_acl_perm_control }
};

dce_db_open("my_acldb", NULL,
    dce_db_c_std_header | dce_db_c_index_by_uuid,
    (dce_db_convert_func_t)dce_acl_convert_func,
    &dbh, &st);

dce_acl_register_object_type(dbh, &my_manager_uuid,
    sizeof my_printstring / sizeof my_printstring[0],
    my_printstring, &my_acl_help, xxx_resolve_func,
    NULL, &st);
```

Note that the server must register the RDACL endpoint by calling `rpc_ep_register` itself.

9.2. RDACL Implementation and Server Callback

As previously stated, the ACL Library provides a complete implementation of the RDACL interface. Most of this will be transparent to the rest of the server code.

The operations in the RDACL interface share an initial set of parameters that specify the ACL object being operated upon:

```
handle_t                        h
sec_acl_component_name_t        component_name
uuid_t                          *manager_type
sec_acl_type_t                  sec_acl_type
```

The `sec_acl_type` parameter indicates if the ACL or an IACL is desired. It does not appear in the ‘‘access’’ operations as it must have the value `sec_acl_type_object`.

In order to implement the RDACL interface, the server must provide a ‘‘resolution’’ routine that maps these parameters into the UUID of the desired ACL object; the library includes some default routines.

The resolution routine is required because servers use the namespace differently. For example, some servers will only export their binding information and use a single ACL. In this case the resolution parameters are not needed. Other servers will have many objects in the namespace, putting a UUID in each entry. In this case, calling `rpc_binding_inq_object` on the handle to obtain the object UUID; they would use this same UUID as the index of the ACL object. Servers with many objects will use a junction or similar architecture so that the component name (also called the ‘‘residual’’) will specify the ACL object. DTS is an example of the first type of server, we believe many application servers will be of the second type, and security is essentially of the third type.

The following `typedef` specifies the signature for a resolution routine. The first four parameters are the common RDACL parameters mentioned above.

```
typedef void (*dce_acl_resolve_func_t)(
/* [in] parameters */
    handle_t                    h,
    sec_acl_component_name_t    component_name,
    sec_acl_type_t              sec_acl_type,
    uuid_t                      *manager_type,
    boolean32                   writing,
    void                        *resolver_arg
/* [out] parameters */
    uuid_t                      *acl_uuid,
    error_status_t              *st
);
```

The ACL Library will include resolver functions that match the common paradigms described above. Application developers can use these functions or provide their own.

For example, a server has several objects and the state for each object is stored in a backing store database. Part of the standard header for each object is a structure that contains the UUID of the ACL for that object. (The standard header is not intended to be an abstract type, but rather a common prolog provided to ease server development.) The resolution function for this server should retrieve the object UUID from the handle, use that as an index into its own backing store, and use the `sec_acl_type` parameter to retrieve the appropriate ACL UUID from the standard data header. A sample implementation of this routine, `dce_acl_resolve_by_uuid`, is given in Appendix A. This routine must know the database handle for server's object storage. This handle would be specified as the `resolver_arg` parameter in the `dce_acl_register_object_type` call.

Servers that use the residual name to resolve an ACL object can use `dce_acl_resolve_by_name`. This routine requires a DCE database that maps names into ACL UUID's. This database must be maintained by the server application. That is, whenever objects are created they must have a name, and that name must be a key into a database that stores the UUID that identifies the object. The `resolver_arg` parameter given in the `dce_acl_register_object_type` call should be a handle for that database.

9.3. Server Queries

The ACL library provides a few routines to automate the most common use of DCE ACL's. The following routine ensures that the client's credentials are authenticated and, if so, that they grant the desired access. This will probably be the routine that is invoked most often.

```
void
dce_acl_is_client_authorized(
    handle_t                    h,
    uuid_t                      acl,
    sec_acl_permset_t           desired_perms,
    boolean32                   *authorized,
    error_status_t              *st
);
```

The `h` parameter is the server's client handle. The `acl` parameter is the UUID of the ACL protecting the object being operated upon and the `desired_perms` specifies the permissions that the application requires the client to have. The routine will fill in the `authorized` value with `TRUE` if the client has the necessary permissions or `FALSE` if not. If permissions cannot be verified, or an other error occurs, it will be left unchanged. A typical call would look like this:

```
/* Server determines object being operated on, and fetches
 * UUID of the ACL for that object into "aclobj." */
dce_acl_is_client_authorized(h, aclobj, sec_acl_perm_write,
        &authorized, &st);
if (st != error_status_ok || !authorized)
        return;
/* Server performs desired operation. */
```

A server that wants to know all the permissions a client has can call the following routine:

```
void
dce_acl_inq_client_permset(
    handle_t                    h,
    uuid_t                      acl,
    sec_acl_permset_t           *permset,
    error_status_t              *st
);
```

The h parameter is the server's client handle. The acl parameter identifies the UUID of the ACL object. If no errors occurred while determining the permissions, the library will fill in the permset parameter with the client's permissions.

The following convenience routine returns the certified credentials of the client, or NULL if there are none. It is a wrapper around the rpc_binding_inq_auth_client routine.

```
void
dce_acl_inq_client_pac(
    handle_t                    h,
    sec_id_pac_t                **pac,
    error_status_t              *st
);
```

This routine does not handle DCE 1.1 EPAC's.

Sometimes an application will only want an implementation that applies the POSIX rules to the DCE objects. The following lower-level routine provides this capability:

```
void
dce_acl_inq_permset_for_pac(
    sec_id_pac_t                *pac,
    sec_acl_t                   acl,
    sec_acl_posix_semantics_t   posix_semantics,
    sec_acl_permset_t           *permset,
    error_status_t              *st
);
```

The meaning of the pac and acl parameters should be obvious. The posix_semantics parameter is for future expansion and is currently ignored. If no errors are found, then the routine will fill in the permset parameter with the permissions. The dce_acl_is_client_authorized routine is implemented in terms of dce_acl_inq_client_pac and this routine. It may also be useful to programmers using the GSSAPI facility or that do not wish to use the storage features provided by this library. This routine does not handle DCE 1.1 EPAC's.

9.4. Creating ACL Objects

The following convenience functions may be used by an application programmer to create ACL objects. An object is created by using dce_acl_obj_create and entries are added to it by using dce_acl_obj_add_xxx_entry. The ACL can then be used as needed, for example it can be stored in the ACL database. Once the application is finished with it, its resources can be released by calling dce_acl_obj_free.

The ‘‘name’’ parameter may be NULL.

The following routine creates an ACL:

```
void
dce_acl_obj_create(
    uuid_t                      *manager_type,
    sec_acl_t                   *acl,
    error_status_t              *st
);
```

The manager_type parameter is the UUID of the ACL manager. It will be the same as the UUID in the dce_acl_register_object_type call.

The following two routines add a user or group entry to an ACL.

```
void
dce_acl_obj_add_user_entry(
    sec_acl_t                   *acl,
    sec_acl_permset_t           permset,
    uuid_t                      user,
    error_status_t              *st
);

void
dce_acl_obj_add_group_entry(
    sec_acl_t                   *acl,
    sec_acl_permset_t           permset,
    uuid_t                      group,
    error_status_t              *st
);
```

Both of these routines are wrappers around the following routine:

```
void
dce_acl_obj_add_id_entry(
    sec_acl_t                   *acl,
    sec_acl_entry_type_t        entry_type,
    sec_acl_permset_t           permset,
    uuid_t                      id,
    error_status_t              *st
);
```

The following routine adds an ACL entry for unauthenticated access. It is a wrapper around the dce_acl_obj_add_obj_entry routine that follows it.

```
void
dce_acl_obj_add_unauth_entry(
    sec_acl_t                   *acl,
    sec_acl_permset_t           permset,
    error_status_t              *st
);
```

```
void
dce_acl_obj_add_obj_entry(
    sec_acl_t                   *acl,
    sec_acl_entry_type_t        entry_type,
    sec_acl_permset_t           permset,
    error_status_t              *st
);
```

The following routine adds an ACL entry for a foreign user or group. Again, the ‘‘name’’ parameters may be NULL.

```
void
dce_acl_obj_add_foreign_entry(
    sec_acl_t                   *acl,
    sec_acl_entry_type_t        entry_type,
    sec_acl_permset_t           permset,
    uuid_t                      realm,
    uuid_t                      id,
    error_status_t              *st
);
```

When the application is finished with the created ACL the following routine will free all allocated memory associated with it:

```
void
dce_acl_obj_free(
    sec_acl_t                   *acl,
    error_status_t              *st
);
```

10. REMOTE INTERFACES

The library will contain a complete implementation of the RDACL interface. It will not register endpoints or export entries to the namespace.

11. MANAGEMENT INTERFACES

There is no management interface.

12. RESTRICTIONS AND LIMITATIONS

The initial implementation is limited to 32 permission bits and no manager chaining.

13. OTHER COMPONENT DEPENDENCIES

The ACL Library uses the Backing Store Library API [RFC 45].

Implementing the access algorithm will require sample code from HP that shows how delegation chains affect the code.

14. COMPATIBILITY

This is a new facility; compatibility with previous releases is not applicable.

For compatibility with the current RDACL interface, all text arguments are limited to the DCE PCS.

15. STANDARDS

Permissions are interpreted according to POSIX.6 D12 interpretation rules.

16. OPEN ISSUES

This library implements a subset of the full RDACL functionality. We will need to determine if the current example ACL manager should be removed; this would have additional documentation impact.

The Backing Store Library should have a routine that reads just the standard object header.

APPENDIX A. SIMPLE CALLBACK IMPLEMENTATION

The following code implements a resolution routine:

```
void
dce_acl_resolve_by_uuid(
    handle_t                    h,
    sec_acl_component_name_t    name,
    sec_acl_type_t              sec_acl_type,
    uuid_t                      *manager_type,
    boolean32                   writing,
    void                        *resolver_arg,
    uuid_t                      *acl_uuid,
    error_status_t              *st
)
{
    dce_db_handle_t             db_handle;
    error_status_t              *st2;
    dce_db_header_t             dbh;
    uuid_t                      obj;

    /* Get the object. */
    rpc_binding_inq_object(h, &obj, st);
    if (*st != error_status_ok)
        return;

    /* Get the object header using the object database.
     * The handle was passed in as the resolver_arg in the
     * dce_acl_register_object_type call. */
    db_handle = (dce_db_handle_t)resolver_arg;
    dce_db_fetch_header(db_handle, &obj, &dbh, st);
    if (*st != error_status_ok)
        return;

    /* Get the appropriate ACL based on the ACL type. */
    dce_acl_inq_acl_from_header(dbh, sec_acl_type, acl_uuid,
        st);
    if (*st != error_status_ok)
        return;

    /* Release the object header. */
    dce_db_free_header(dbh, &st2);

    /* If "writing" was true, we could retrieve the header of
     * the ACL object and return a newly-generated UUID if its
     * reference count was greater than one.  We assume each
     * object has its own ACL object and don't bother. */
}
```

The `dce_acl_inq_acl_from_header` function is a convenience routine that retrieves the UUID of the appropriate ACL object from a standard object header. Its implementation is given below to illustrate the standard use of the Backing Store Library object header:

```
void
dce_acl_inq_acl_from_header(
    dce_db_header_t             dbh,
    sec_acl_type_t              sec_acl_type,
    uuid_t                      *acl_uuid,
    error_status_t              *st
)
{
    *st = error_status_ok;
    switch (sec_acl_type) {
    default:
        *st = sec_acl_invalid_acl_type;
        break;
    case sec_acl_type_object:
        *acl_uuid = dbh->h->uuid;
        break;
    case sec_acl_type_default_object:
        if (!dbh->is_container)
            *st = sec_acl_invalid_acl_type;
        else
            *acl_uuid = dbh->h->iacl_uuid;
        break;
    case sec_acl_type_default_container:
        if (!dbh->h->is_container)
            *st = sec_acl_invalid_acl_type;
        else
            *acl_uuid = dbh->h->h->cont_iacl_uuid;
        break;
    }
}
```

REFERENCES

[RFC 3] J. Pato, DCE-RFC 3.0, ‘‘Extending the DCE Authorization Model to Support Practical Delegation (Extended Summary),’’ June, 1992.

[RFC 42] H. Melman, DCE-RFC 42.0, ‘‘DCE Shell Functional Specification,’’ June, 1993.

[RFC 45] D. Mackey, R. Salz, DCE-RFC 45.0, ‘‘DCE 1.1 Backing Store Library Functional Specification,’’ August, 1993.

[RFC 47] D. Mackey, R. Salz, DCE RFC 47.0, ‘‘DCE 1.1 dced Functional Specification,’’ to appear.

AUTHORS’ ADDRESS

Richard Mackey
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
USA

Internet email: dmackey@osf.org
Telephone: +1-617-621-8924

Rich Salz
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
USA

Internet email: rsalz@osf.org
Telephone: +1-617-621-7253