A GENERIC INTERFACE FOR EXTENDED REGISTRY ATTRIBUTES

1. INTRODUCTION

The DCE 1.0 user registry facility stores principal data and attributes using a static schema. This RFC provides a proposal for a generic attribute facility that employs a dynamic schema. A network interface is defined for manipulating both the schema and attributes. Rgy_edit is extended to present a user interface for manipulating the schema and attributes stored in the security name space.

1.1. Statement of the Problem

DCE 1.0 provides a repository for principal, group, and account data. This repository stores the network privilege attributes used by the DCE as well as account data used by local operating systems. The local account attributes defined in the current system, however, are only appropriate for UNIX operating systems.

In a heterogeneous distributed environment, it is inappropriate to assume that all machines are running some variant of the UNIX operating system. A mechanism for storing user and group attributes for arbitrary operating systems is necessary in the DCE. In addition it is necessary to allow the controlled creation and modification of arbitrary attributes associated with principals.

1.2. Outline of the Proposal

This proposal builds on the dynamic attribute facility present in the predecessor to the DCE user registration system. It adds a set of operations for creating and maintaining an attribute schema. The interface for manipulating the schema and for manipulating attributes has been designed to mirror the naming junction protocol used by the DCE ACL interfaces. This allows, but does not require, the interface to be considered for use by DCE components other than the security services.

2. ATTRIBUTE SCHEMA

The attribute schema provides a catalog for attributes known to the system. The catalog may be dynamically updated to create or destroy schema entries that define the characteristics of attributes supported by a system. Access to the schema is controlled by a DCE access control list.

2.1. Schema Entries

A schema entry defines the characteristics of an attribute class known to the system. Once a schema entry has been created for an attribute class, instances of that attribute can be created on nodes that are dominated by the schema. The primary identifier for an attribute is a unique identifier (UUID), but the schema entry for the attribute also contains a string name that can be used as a secondary key. We expect that application programs reading and writing attributes from the repository will always use the UUID key while interactive user interfaces will allow users to specify the string name.

In addition to defining the identifier for an attribute class, each schema entry contains the following control information for the attribute class:

(a) attribute_encoding: defines the legal encodings for the attribute class. Encodings are specified from a small set of primitive data types, a set of self-describing constructed datatypes (or pickles [Harr 92]) and a dynamic encoding tag which specifies that each instance of the attribute may have a different encoding.

(b) acl_manager_type: defines the acl manager for the object to which the attribute may be attached. This field field provides a well-defined context for evaluating the query and update permission bits needed for operating on the attribute.

(c) query_permset: defines the permission bits needed to allow reads of the attribute's value.

(d) update_permset: defines the permission bits needed to allow writes to the attribute's value.

(e) test_permset: defines the permission bits needed to use the attribute as a control field for test and update operations.

(f) delete_permset: defines the permission bits needed to delete an attribute instance.

2.2. The Format of Attributes

Attributes consist of a unique attribute id and a value. The value is a discriminated union controlled by an encoding tag and the appropriate data for the encoding.[1] Encodings are defined from the following set:[2]

(a) any: the attribute can take on any encoding. This value is only legal for the definition of an attribute in a schema entry. All instances of an attribute must provide a concrete encoding from the remainder of this set.

(b) non existent: the attribute has no value. It is simply a marker that is either present or absent.

(c) printstring: the attribute is a printable IDL character string.

(d) integer: the attribute is a signed (32 bit) integer.

(e) bytes: the attribute is a string of bytes. The byte string is assumed to be a pickle or is otherwise a self describing type.

(f) confidential bytes: the attribute is an encrypted string of bytes. Like a plaintext byte string this is assumed to be a self describing type. Typically this encoding type is only useful when stored with the user registry where it will be decrypted and re-encrypted every time the password associated with the node is changed.

3. INTEGRATION WITH NAMING

The attribute interface uses the same junction protocol defined by the DCE ACL interface. All operations are performed on the object selected by the combination of the object id in the RPC handle, and the component name specified in the remote operation. The component name is used to disambiguate references to small grained objects that are not independently exported to the endpoint mapper or the cell naming system.

4. NETWORK PROGRAMMING INTERFACE

The attribute programming interface is divided into two components: an interface for manipulating a schema and an interface for manipulating attribute instances. A server may choose to implement only the attribute interface if it does not need to provide a flexible access control model for manipulating attributes and if it does not need to yield access to attributes via names instead of UUIDs.

4.1. Schema Manipulation Interface

The schema manipulation interface is outlined in Appendix B. It provides operations for creating, deleting and reading schema entries and for iterating over the schema contents.

4.2. Attribute Manipulation Interface

The attribute manipulation interface is outlined in Appendix C. It provides operations for writing, reading and deleting attribute instances. In addition it provides a mechanism for an atomic write to a set of attributes if a control set of attributes have not changed value.

---

1. An alternative approach suggested by Bill Sommerfeld is to abandon primitive encodings entirely and store all attribute values as a pickled byte string. This strategy simplifies the definition of an attribute type, and the work performed by a repository at the expense of additional pickle header information for all simple attributes. This RFC does not follow that recommendation, but could be trivially changed if the OSF deems the pickle approach is better.

2. This is not an exhaustive set of primitive encodings. This set can be extended to cover the range of primitive attributes supported by CDS.

5. USER INTERFACE

The initial implementation of this attribute interface will be in the security services to allow for extended security attributes. The user interface to this facility will be provided through rgy_edit.

6. ACKNOWLEDGEMENTS

Bill Sommerfeld and Liza Martin of HP provided helpful suggestions in rounding out this proposal.

APPENDIX A. sec_attr_base.idl

```
/*
** Copyright (c) Hewlett-Packard Company 1992
** Unpublished work. All Rights Reserved.
**
**      Attributes base type definitions
**
*/

[
    uuid(825719e8-d1b9-11ca-8687-08001e01dc6c)
]

interface sec_attr_base {

    import "dce/aclbase.idl";

    /*
     * Attribute encoding tags:
     *  Defines the encoding of the attribute.
     *
     *      any                 Attribute can take on any encoding.
     *                          This is only legal in a schema entry
     *                          an attribute entry must contains a
     *                          concrete encoding type.
     *
     *      non_existent        A marker without value.
     *      printstring         A printable string.
     *      integer             A signed integer.
     *      byte                A string of bytes (presumably a
     *                          pickle).
     *      confidential byte   A string of bytes (presumably a
     *                          pickle) that have been encrypted in
     *                          the principal’s key.  This type is
     *                          normally only supported by the
     *                          security service.
     */
    typedef unsigned32  sec_attr_encoding_t;
    const unsigned32    sec_attr_enc_any                = 0;
    const unsigned32    sec_attr_enc_non_existent       = 1;
    const unsigned32    sec_attr_enc_printstring        = 2;
    const unsigned32    sec_attr_enc_integer            = 3;
    const unsigned32    sec_attr_enc_bytes              = 4;
    const unsigned32    sec_attr_enc_confidential_bytes = 5;

    /* s e c _ a t t r _ v a l u e _ t;
     *
     * An attribute value is a union of the various known encodings.
     */

    typedef struct {
        unsigned32      length;
        [size_is(num_bytes)]
            byte_p_t    pickled_data;
```

```
} sec_attr_byte_encoding_t;

typedef union sec_attr_u
    switch (sec_attr_encoding_t     attr_encoding)
    tagged_union {
        case sec_attr_enc_non_existent:
            ;

        case sec_attr_enc_confidential_bytes:
        case sec_attr_enc_bytes:
            sec_attr_byte_encoding_t   bytes;

        case sec_attr_enc_printstring:
            [string] char              *printstring;

        case sec_attr_enc_integer:
            signed32                   signed_integer;

        default:
            /* This is an error case! */
            ;
    }
} sec_attr_value_t;

/* s e c _ a t t r _ t;
 *
 * An attribute is a structure containing the id and value for
 * the attribute.
 */

typedef struct {
    uuid_t              attribute_id;
    sec_attr_value_t    attribute_value;
} sec_attr_t;

/* s e c _ a t t r _ s c h e m a _ e n t r y _ t;
 *
 * Schema catalog
 *
 *  Attributes are:
 *      Name <key>, Attribute UUID <key>,
 *                  attribute encoding,,
 *                  acl manager type, read permset, write
 *                  permset, test_permset, delete permset,
 *                  comment
 *
 *  Each attribute is named by both a string name and an
 *  attribute uuid - either can be used as a retrieval key and
 *  both must be unique.  The name should only be used for
 *  interactive access to the attribute the attribute id uuid
 *  should be used for programmatic access.
 *
 *  The attribute id uuid is used to identify the semantics of
 *  the attribute.
 *
```

```
 *  Fields:
 *      attr_encoding       Defines the legal encodings for the
 *                          attribute If a concrete encoding is
 *                          specified, the attribute may only
 *                          exist with that encoding.  If the
 *                          "dynamic" encoding is specified, the
 *                          each instance of the attribute may
 *                          take on any encoding.
 *
 *      acl_manager_type    Defines the acl manager for the
 *                          object that attribute is attached to.
 *                          This field will provide a
 *                          well-defined context for evaluating
 *                          the permission bits needed for
 *                          operating on the attribute.
 *
 *      query_permset       The permission bits needed to access
 *                          the attribute's value.
 *
 *      update_permset      The permission bits needed to update
 *                          the attribute's value.
 *
 *      test_permset        The permission bits needed to test
 *                          the attribute's value.
 *
 *      delete_permset      The permission bits needed to delete
 *                          an attribute instance.
 */

typedef struct {
    [string] char           *attribute_name;
    uuid_t                  attr_id;
    sec_attr_encoding_t     attr_encoding;
    uuid_t                  acl_manager_type;
    sec_acl_permset_t       query_permset;
    sec_acl_permset_t       update_permset;
    sec_acl_permset_t       test_permset;
    sec_acl_permset_t       delete_permset;
    [string] char           *comment;
} sec_attr_schema_entry_t;


/* s e c _ a t t r _ c o m p o n e n t _ n a m e _ t
 *
 * A string for disambiguating operations on objects.  The
 * component name further specifies the entity to which the
 * attribute is attached.  This is analogous to a
 * sec_acl_component_name_t in the acl interface.
 */
typedef [string, ptr] unsigned char *sec_attr_component_name_t;

/* s e c _ a t t r _ s c h e m a _ c u r s o r _ t;
 *
 * Schema scan cursor.  This must be something real at the wire
 * level - probably don't need a full context handle, but that
```

```
 * would do.  The cursor must minimally represent the
 * sec_attr_component_name that was provided to the cursor
 * initialization.
 */
typedef void * sec_attr_schema_cursor_t;
```

APPENDIX B. sec_attr_schema.idl

```
/*
** Copyright (c) Hewlett-Packard Company 1992
** Unpublished work. All Rights Reserved.
**
**      Attributes Schema Manipulation Interface
**
** Binding model:
**
**  The attribute binding model is the same as the ACL binding model.
**  An RPC handle is obtained to an object, and any residual portion
**  of the name is passed as the component name.  This is the
**  mechanism that the ACL interface uses for implementing a junction
**  point in the name system.
**
**  The schema is accessed in the same way - but it is possible that
**  an attribute service will choose to implement a single schema
**  rather than a schema per nameable component.  Some attribute
**  services (like the security service) implement a single schema
**  for all nodes in the namespace and will ignore the component name
**  when operating on the schema.
**
*/

[
    uuid(b47c9460-567f-11cb-8c09-08001e04de8c)
]

interface sec_attr_schema {

    import "dce/sec_attr_base.idl";

    /*
     * Private Datatypes for the schema interface
     */

    /* sec_attr_schema_twr_ref_t, sec_attr_schema_tower_set_t
     *
     * These types enable the client to pass in an unallocated array
     * of towers and have the server allocate/fill the correct
     * ammount.
     */

    typedef [ref] twr_t *sec_attr_schema_twr_ref_t;

    typedef [ptr] struct {
        unsigned32          count;
        [size_is(count)]
        sec_attr_schema_twr_ref_t   towers[];
    } *sec_attr_schema_tower_set_t;

    /*
     * SCHEMA OPERATIONS
     */
```

```
/* s e c _ a t t r _ s c h e m a _ c r e a t e _ e n t r y
 *
 * Create a new schema entry.  Presumably you must be authorized
 * to add entries to the schema - that is handled by the target
 * server.
 *
 * Errors:
 *  Redundant name/uuid
 *  Unauthorized
 *  Server read only
 *  Server unavailable
 *  Invalid/Unsupported attribute type
 *  Invalid acl manager type
 *  Invalid permission set
 */
void sec_attr_schema_create_entry (
    [in]        handle_t                       h,
    [in]        sec_attr_component_name_t      schema_name,
    [in]        sec_attr_schema_entry_t        *schema_entry,
    [out]       error_status_t                 *st
);

/* s e c _ a t t r _ s c h e m a _ d e l e t e _ e n t r y
 *
 * Delete a schema entry.  This is a radical operation that will
 * delete or invalidate any existing attributes on nodes
 * dominated by the schema.  Access to this operation should be
 * severely limited.
 *
 * Errors:
 *  Unauthorized
 *  No such entry
 *  Server read only
 *  Server unavailable
 */
void sec_attr_schema_delete_entry (
    [in]        handle_t                       h,
    [in]        sec_attr_component_name_t      schema_name,
    [in]        uuid_t                         *attr_id,
    [out]       error_status_t                 *st
);

/* s e c _ a t t r _ s c h e m a _ c u r s o r _ i n i t
 *
 * Initialize a scan cursor
 */
void sec_attr_schema_cursor_init (
    [in]        handle_t                       h,
    [in]        sec_attr_component_name_t      schema_name,
    [out]       unsigned32                     *cur_num_entries,
    [out]       sec_attr_schema_cursor_t       *cursor,
    [out]       error_status_t                 *st
);

/* s e c _ a t t r _ s c h e m a _ c u r s o r _ r e l e a s e
```

```
 *
 * Release any state associated with a scan cursor.  (Is this
 * just a local operation?)
 */
void sec_attr_schema_cursor_release (
    [in]        handle_t                        h,
    [in, out]   sec_attr_schema_cursor_t        *cursor,
    [out]       error_status_t                  *st
);

/* s e c _ a t t r _ s c h e m a _ s c a n
 *
 * Read K entries from the schema - useful for a browser.
 */
void sec_attr_schema_scan (
    [in]        handle_t                        h,
    [in]        sec_attr_component_name_t       schema_name,
    [in, out]   sec_attr_schema_cursor_t        *cursor,
    [in]        unsigned32                      num_to_read,
    [out]       unsigned32                      *num_read,
    [out] [size_is(num_to_read), length_is(num_read)]
                sec_attr_schema_entry_t        *schema_entries[],
    [out]       error_status_t                  *st
);

/* s e c _ a t t r _ s c h e m a _ l o o k u p _ n a m e
 *
 * Read a schema entry by name - useful for an interactive
 * editor.
 */
void sec_attr_schema_lookup_name (
    [in]        handle_t                        h,
    [in]        sec_attr_component_name_t       schema_name,
    [in] [string] char                          *name,
    [out]       sec_attr_schema_entry_t         *schema_entry,
    [out]       error_status_t                  *st
);

/* s e c _ a t t r _ s c h e m a _ l o o k u p _ i d
 *
 * Read a schema entry by ID.  Useful for programmatic access.
 */
void sec_attr_schema_lookup_id (
    [in]        handle_t                        h,
    [in]        sec_attr_component_name_t       schema_name,
    [in]        uuid_t                          *attr_id,
    [out]       sec_attr_schema_entry_t         *schema_entry,
    [out]       error_status_t                  *st
);

/* s e c _ a t t r _ s c h e m a _ g e t _ r e f e r r a l
 *
 * Obtain a referral to an schema update site.  This function is
 * used when the current schema site yields a
 * sec_schema_site_readonly error.  Some replication managers
```

```
     * will require all updates for a given object to be directed to
     * a given replica.  Clients of the generic schema interface may
     * not know they are dealing with an object that is replicated in
     * this way.  This function allows them to recover from this
     * problem and rebind to the proper update site.
     */
    void sec_attr_schema_get_referral (
        [in]        handle_t                     h,
        [in]        sec_attr_component_name_t   component_name,
        [in]        uuid_t                       *attr_id,
        [out]       sec_attr_schema_tower_set_t *towers,
        [out]       error_status_t               *st
    );

}
```

APPENDIX C. sec_attr_manip.idl

```
/*
** Copyright (c) Hewlett-Packard Company 1992
** Unpublished work. All Rights Reserved.
**
**      Attributes Manipulation Interface
**
** Binding model:
**
**  The attribute binding model is the same as the ACL binding model.
**  An RPC handle is obtained to an object, and any residual portion
**  of the name is passed as the component name.  This is the
**  mechanism that the ACL interface uses for implementing a junction
**  point in the name system.
**
*/

[
    uuid(a71fc1e8-567f-11cb-98a0-08001e04de8c)
]

interface sec_attr_manip {

    import "dce/sec_attr_base.idl";

    /*
     * Private Datatypes for the attribute manipulation interface
     */

    /* sec_attr_manip_twr_ref_t, sec_attr_manip_tower_set_t
     *
     * These types enable the client to pass in an unallocated array
     * of towers and have the server allocate/fill the correct
     * ammount.
     */

    typedef [ref] twr_t *sec_attr_manip_twr_ref_t;

    typedef [ptr] struct {
        unsigned32          count;
        [size_is(count)]
        sec_attr_manip_twr_ref_t   towers[];
    } *sec_attr_manip_tower_set_t;

    /*
     * ATTRIBUTE OPERATIONS
     */

    /* s e c _ a t t r _ m a n i p _ c u r s o r _ i n i t
     *
     * Initialize a scan cursor
     */
    void sec_attr_manip_cursor_init (
        [in]        handle_t                        h,
```

```
        [in]        sec_attr_component_name_t       component_name,
        [out]       unsigned32                      *cur_num_entries,
        [out]       sec_attr_manip_cursor_t         *cursor,
        [out]       error_status_t                  *st
    );

    /* s e c _ a t t r _ m a n i p _ c u r s o r _ r e l e a s e
     *
     * Release any state associated with a scan cursor.  (Is this
     * just a local operation?)
     */
    void sec_attr_manip_cursor_release (
        [in]        handle_t                        h,
        [in, out]   sec_attr_manip_cursor_t         *cursor,
        [out]       error_status_t                  *st
    );

    /* s e c _ a t t r _ l o o k u p _ b y _ i d
     *
     * Read an attribute by ID.  Useful for programmatic access.
     * Multi-valued attributes are returned as independent attributes
     * sharing the same attribute id.
     *
     * If the number of query attribute keys is 0, this function will
     * return all attributes that the caller is authorized to see.
     *
     * If a list cursor is supplied it is used to establish the point
     * in the attribute list from which the server should start
     * processing the query.  List cursors should be initialized with
     * the cursor_init function.  If the list cursor parameter is
     * NULL, the server will begin processing the query with the
     * first attribute that satisfies the search criteria.
     *
     * If the output parameter num_left is larger than 0, then the
     * output buffer supplied was not big enough.  Num_left is a hint
     * at the number of attributes that were not possible to return
     * due to space constraints.  This number may be inaccurate if
     * the server allows updates between succesive query calls.
     *
     * Warnings:
     *  Not_all_available       Not all of the requested attributes
     *                          were available.
     *
     * Errors:
     *  Unauthorized
     *  Server Unavailable
     */
    void sec_attr_lookup_by_id (
        [in]        handle_t                        h,
        [in]        sec_attr_component_name_t       component_name,
        [in]        unsigned32                      num_attr_keys,
        [in]        unsigned32                      space_avail,
        [in] [ length_is(num_attr_keys) ]
                    uuid_t                          attr_id[],
        [in, out, ptr] sec_attr_manip_cursor_t      *list_cursor,
```

```
        [out]       unsigned32                      *num_returned,
        [out] [ size_is(space_avail), length_is(num_returned) ]
                    sec_attr_t                      attr[],
        [out]       unsigned32                      *num_left,
        [out]       error_status_t                  *st
    );

    /* s e c _ a t t r _ l o o k u p _ b y _ n a m e
     *
     * Read an attribute by name.  Useful for an interactive editor.
     *
     * Warnings:
     *  More Available          Named attribute was a multi-valued
     *                          attribute.  Re-submit request using
     *                          ID returned in single output
     *                          parameter to the lookup_by_id
     *                          function.
     *
     * Errors:
     *  Unauthorized
     *  Server Unavailable
     */
    void sec_attr_lookup_by_name (
        [in]        handle_t                        h,
        [in]        sec_attr_component_name_t       component_name,
        [in] [string] char                          *name,
        [out]       sec_attr_t                      *attr,
        [out]       error_status_t                  *st
    );

    /* s e c _ a t t r _ u p d a t e
     *
     * Write/Create an attribute.  This is an atomic operation.  All
     * attributes are written/created or none are modified.  The
     * first attribute causing the update to fail is identified in
     * "failure_index".  If the failure cannot be attributed to a
     * given attribute, then -1 is returned in the failure index.
     *
     * Errors:
     *  Unauthorized
     *  Database read only
     *  Server unavailable
     *  Invalid/Unsupported attribute type
     *  Site read only
     */
    void sec_attr_update (
        [in]        handle_t                        h,
        [in]        sec_attr_component_name_t       component_name,
        [in]        unsigned32                      num_to_write,
        [in] [ length_is(num_to_write) ]
                    sec_attr_t                      attr[],
        [out]       signed32                        *failure_index,
        [out]       error_status_t                  *st
    );
```

```
/* s e c _ a t t r _ t e s t _ a n d _ u p d a t e
 *
 * Update attributes if a set of control attributes retain
 * specified values.  This is an atomic operation - if any of the
 * test values do not match, then none of the updates are
 * performed.  If the update should be performed, but the write
 * cannot occur to any member of the update set, the entire
 * update fails and the problematic attribute is identified in
 * the output failure index.  If the update fails and cannot be
 * attributed to a given update attribute, -1 is returned as the
 * failure index.
 *
 * Matching rules are based on exact matches of each primitive
 * datatype.
 *
 * Errors:
 *  Control attribute has changed
 *  Unauthorized
 *  Database read only
 *  Server unavailable
 *  Invalid/Unsupported attribute type
 *  Site read only
 */
void sec_attr_test_and_update (
    [in]        handle_t                    h,
    [in]        sec_attr_component_name_t   component_name,
    [in]        unsigned32                  num_to_test,
    [in]    [length_is(num_to_test) ]
                sec_attr_t                  test_attr[],
    [in]        unsigned32                  num_to_write,
    [in] [ length_is(num_to_write) ]
                sec_attr_t                  update_attr[],
    [out]       signed32                    *failure_index,
    [out]       error_status_t              *st
);

/* s e c _ a t t r _ d e l e t e
 *
 * Delete an attribute.  This is an atomic operation.  All
 * attributes are deleted or none are modified.  The first
 * attribute causing the delete to fail is identified in
 * "failure_index".  If the failure cannot be attributed to a
 * given attribute, then -1 is returned in the failure index.  A
 * non-existent attribute does not induce a failure on delete.
 *
 * Errors:
 *  Unauthorized
 *  Database read only
 *  Server unavailable
 *  Invalid/Unsupported attribute type
 *  Site read only
 */
void sec_attr_delete (
    [in]        handle_t                    h,
    [in]        sec_attr_component_name_t   component_name,
```

```
        [in]        unsigned32                      num_to_delete,
        [in] [ length_is(num_to_delete) ]
                    uuid_t                          attr_id[],
        [out]       signed32                        *failure_index,
        [out]       error_status_t                  *st
    );

    /* s e c _ a t t r _  g e t _ r e f e r r a l
     *
     * Obtain a referral to an schema update site.  This function is
     * used when the current schema site yields a
     * sec_schema_site_readonly error.  Some replication managers
     * will require all updates for a given object to be directed to
     * a given replica.  Clients of the generic schema interface may
     * not know they are dealing with an object that is replicated in
     * this way.  This function allows them to recover from this
     * problem and rebind to the proper update site.
     */
    void sec_attr_get_referral (
        [in]        handle_t                    h,
        [in]        sec_attr_component_name_t   component_name,
        [in]        uuid_t                      *attr_id,
        [out]       sec_attr_manip_tower_set_t  *towers,
        [out]       error_status_t              *st
    );

}
```

REFERENCES

[Harr 92] J. Harrow, ‘‘Proposed Enhancements for DCE 1.1 IDL’’, DCE-RFC 2.0, June 1992.

AUTHOR’S ADDRESS

Joseph N. Pato
Distributed Object Computing Program
Hewlett-Packard Co.
250 Apollo Drive
Chelmsford, MA 01824
USA

Internet email: pato@apollo.hp.com
Telephone: +1-508-436-4350