Open Software Foundation
Request For Comments: 83.0
May 1995
M. Hondo (HP)

DCE 1.2 REGISTRY SYNCHRONIZATION — FUNCTIONAL SPECIFICATION

1. INTRODUCTION

The DCE registry is the repository for information pertaining to users (and other principals, such as servers), groups and organizations. In order to support **registry synchronization** of user information between DCE and other ''network registries'' (Novell, NIS, etc.), it is necessary to allow these other network registries to have access to the information which DCE stores.

In pre-DCE 1.2 systems, the only way for the information in the registry to be retrieved was through a limited ''pull'' of the data via a read of all of the database items. Within this model, however, there was still no way to retrieve the password information for an account.

To provide a secure and timely mechanism for registry synchronization, DCE 1.2 will support a new category of replicas, called **secondary replicas**. Like ordinary **primary** replicas, secondary replicas will be servers needing authentication and authorization to communicate with a master. But unlike primary replicas, secondary replicas will not support external interfaces for clients to bind to, and will not support the full set of internal interfaces that primaries do.

An additional requirement for registry synchronization is support for the storage and propagation of cleartext passwords.

The DCE 1.2 registry synchronization infrastructure will consist of the following:

(a) Definition of and support for a new type of replica (**secondary replicas**, mentioned above).

The master replica will know about the existence of the new type of replicas and will propagate updates of registry data to the secondary replicas. Secondary replicas will be responsible for co-ordinating the non-DCE registry synchronization process in a secure manner.

(b) Support for storage of **cleartext passwords**.

Currently the DCE Registry service (1.0.* & 1.1) stores two one-way transformed versions of the password for a principal. These transformations occur at the time of a password change when the new plaintext password is received by the master encrypted under a key derived from the password of the **originator**. (The originator is defined to be the person/principal changing the password — usually the owner or an administrator with the appropriate authorization.) After performing the transformations and storing the UNIX `crypt` form and the DCE DES forms of the password, the plaintext password is not stored. On requests for authentication, the DES form is used.

For DCE 1.2 to fully support synchronization, the plaintext is needed since other repositories may require the plaintext form of the password in order to perform their own (non-DCE, non-UNIX-`crypt`) transformations. The plaintext form must be stored securely in the registry in a non-readable form. ''Reading'' the cleartext from the Registry database will not be allowed. It is important to note that storage of a plaintext password even using a two-way transformation is contentious. This functionality will therefore only be activated when the `PLAINTEXT` ERA is set; sites using this functionality should be aware of the security implications of storing two-way transformed plaintext passwords.

(c) Support for **plaintext protocol on authentication**. (This functionality will be a DCE 1.2.2 deliverable.)

There was an addition to the preauthentication protocol in DCE 1.1 which allows a variation of third party preauthentication to be performed on principals who have the `PLAINTEXT` ERA set. This preauthentication protocol transfers the plaintext form of the password encrypted under the session key established between the machine principal and the Authentication Server for the purpose of authentication.

The purpose of this preauthentication protocol was to allow the registry to authenticate principals whose accounts have been loaded via `passwd_import` with a UNIX form of the password, not a DES form. The protocol was only implemented on the client side for DCE 1.1 .

For DCE 1.2.2, the server side will be implemented. If an account is marked with the `PLAINTEXT` ERA and the client attempts to login, the KDC will check to see if there is a DES form of the password available for the named principal. If there is, it will use the DES form for authentication. If there is not, then the KDC fails the request and the client side of the login protocol will retry the request with this ''plaintext preauthentication protocol''. When the KDC receives the plaintext, it performs a UNIX `crypt` transformation on the plaintext comparing the result with the UNIX form previously loaded into the registry. If the comparison is successful, the KDC then performs the DES transformation on the plaintext, storing the new DES form in the registry and securely storing the plaintext.

(d) Support for **plaintext protocol on password change**. (This functionality will be a DCE 1.2.2 deliverable.)

On a password change operation, a principal is prompted for the new password. If plaintext is passed as input from the client, the ciphertext of the new password is generated using the key of the originator (the owner or admin). The ciphertext is passed to the registry via calls to `sec_rgy_acct_replace_all()` to update the account record with the new password. The KDC has access to all principal keys and is able to retrieve the key of the person requesting the change and decrypt the plaintext form of the new password. If the account is not marked with the `PLAINTEXT` ERA, only the DES form of the password is stored in the database. If the account is marked with the `PLAINTEXT` ERA, the DES form of the password is generated. The plaintext is also stored in the database.

1.1. Changes Since Last Publication

This is the first publication.

2. TARGET

The target audience for this functionality is vendors supporting heterogeneous network registry environments.

3. GOALS AND NON-GOALS

It is a goal of this work to have DCE support an additional class of replicas (secondary replicas).

It is a goal of this work to isolate the **secondary replica application logic** from the propagation and administration model that currently exists for primary replicas.

It is *not* a goal of this work to provide a replica to support a specific instance of a ''foreign'' network registry with which DCE needs to be synchronized.

[In this context, ''foreign'' means ''non-DCE'' — for example, a ''legacy'' network registry. In other contexts within DCE, ''foreign'' means ''other cell''. The intended meaning will always be clear from context.]

4. TERMINOLOGY

New terminology in this RFC is introduced in context. (It has not been gathered into a list here.)

5. FUNCTIONAL DEFINITION

The new secondary replicas will be a combination of DCE-supplied secondary replica infrastructure code (source from OSF) and vendor-supplied code.

A ''vendor'' is here defined as a supplier of secondary replicas. To construct a secondary replica a vendor will need source code from OSF for the basic infrastructure, and a link-time library containing the secondary replica ''application logic'' functions coded to the interfaces defined below.

The interfaces supplied by DCE 1.2 infrastructure are the administration and state interfaces, pictured and listed below. The admin interface, state mgmt, and propagation mgmt functions provide the basic services necessary for a secondary replica. Vendors will need to compile the source code supplied and link it with their own (vendor-supplied) ''backend'' translation library (`sec_rep_lib.a`) to generate an instance of a secondary replica. See picture below.

```
                               Primary Replicas
                              /       ^
                             /        |
                            /+--------V--------------+
  Master  --+------------> / |                       |
            |             /  |  SEC REPLICA          |
+-----------V----------+/    |                       |
|   existing replist   |     |  comm/state mgmt      |
+----------------------+     |  admin interfaces     |
            |               /|                       |
            |              / |       +---------------+
+-----------V----------+ /   | P     |   sec_rep_lib.a
|   secondary replist  |/    | r     |  +------------+
+----------------------+     | o     |  | secondary  |
                             | p     |  |  replica   |
                             |       |  |            |
                             | M     |sec_rep_prop_acct_add()
                             | g     +->|            |
                             | m     |  |            |
                             | t     |  |            |
                             +-------+  |            |
                                        +------------+
```

5.1. Vendor Secondary Replica Support — sec_rep_lib.a

DCE 1.2 infrastructure will provide a library interface for vendors to code to which will isolate them from the underlying DCE mechanisms, allowing them to focus on the data rather than managing the replica.

For example, a new secondary replica would support the new library API interface:

```
PUBLIC  void  sec_rep_prop_acct_add
    (num_accts, accts, status)
    unsigned32              num_accts;    /* [in] */
    sec_rep_acct_add_data_t accts[];      /* [in, size_is
                                             (num_accts)] */
    error_status_t          * status;     /* [out] */
```

The vendor-supplied code would take the data supplied through the argument `accts[]` (an array of `sec_rep_acct_add_data_t` data structures), extract the necessary information and make a call to construct the ''account add'' operation for a foreign registry. Since the ''infrastructure'' will allocate the memory for the `accts[]` array, it will also be responsible for deallocating this space when the call completes.

By being separate from the 1.2 infrastructure, secondary replica application code will not be dependent on any particular wire interface.

The 1.2 infrastructure will not create a persistent database for storing the data which has been propagated. If an implementation requires the data to be kept in persistent storage it would need to supply the code to perform those operations.

The 1.2 master code will strip out ''architectural information'' from the propagations sent to secondary replicas. This architectural information consists of information in the registry not related to the management of principals and their accounts, but keys and other replica information which needs to be known by the master (and other primary replicas which are capable of becoming masters), but does not need to be known by secondary replicas.

There will be nothing architectural restricting the ability to run multiple secondary replicas on a machine. In the current 1.1 model, the UUID in the endpoint map registered by the replica is a global UUID (`cell_sec_id` is generated on the master and passed to replicas so they all share the same object UUID in the PAC), thus prohibiting multiple instances of primary replicas from running on the same host. The secondary replicas will generate a UUID which will be stored in the state file and used until a secondary replica is deleted.

Anyone interested in building a secondary replica will need to provide the following ‘‘backend’’ interface logic, through coding the following functions in `sec_rep_lib.a`.

The following are used to receive propagation information:

(a) Struture used for holding information:

```
typedef struct {
    sec_rgy_login_name_t    login_name;
    sec_rgy_acct_user_t     user_part;
    sec_rgy_acct_admin_t    admin_part;
    sec_passwd_rec_t        * key;
    [ptr] sec_passwd_rec_t  * unix_passwd; /* may be
                                              NULL */
    sec_rgy_foreign_id_t    client;  /* originator
                                        of update */
    sec_passwd_type_t       keytype; /* currently only
                                        DES is valid */
} sec_rep_acct_add_data_t;
```

(b) Take account add and translate it to foreign add operation:

```
sec_rep_prop_acct_add(
    [in]   unsigned32                 num_accts,
    [in, ref, size_is(num_accts)]
           rs_prop_acct_add_data_t    accts[],
    [out]  error_status_t             * status
)
```

(c) Take account delete and translate it to foreign delete operation:

```
sec_rep_prop_acct_delete(
    [in]   sec_rgy_login_name_t       * login_name,
    [out]  error_status_t             * status
)
```

(d) Take account rename and translate it to foreign rename operation:

```
sec_rep_prop_acct_rename(
    [in]   sec_rgy_login_name_t       * old_login_name,
    [in]   sec_rgy_login_name_t       * new_login_name,
    [out]  error_status_t             * status
)
```

(e) Take account replace and translate it to foreign replace operation:

```
sec_rep_prop_acct_replace(
    [in]        sec_rgy_login_name_t    * login_name,
    [in]        rs_acct_parts_t         modify_parts,
    [in]        sec_rgy_acct_user_t     * user_part,
    [in]        sec_rgy_acct_admin_t    * admin_part,
    [in, ptr]   sec_passwd_rec_t        * key,
                                        /* may be NULL */
    [in, ref]   sec_rgy_foreign_id_t    *client,
    [in]        sec_passwd_type_t       new_keytype,
                                        /* only DES */
    [in, ptr]   sec_passwd_rec_t        * unix_passwd,
                                        /* may be NULL */
    [out]       error_status_t          * status
)
```

(f) Take account add key and translate it to foreign add key operation:

```
sec_rep_prop_acct_add_key_version(
    [in]    sec_rgy_login_name_t            * login_name,
    [in]    unsigned32                      num_keys,
    [in, ref, size_is(num_keys)]
            sec_passwd_rec_t                keys[],
    [out]   error_status_t                  * status
)
```

(g) Take ACL replace and translate it to foreign replace operation:

```
sec_rep_prop_acl_replace(
    [in]    unsigned32                          num_acls,
    [in, size_is(num_acls)]
            rs_prop_acl_data_t                  acls[],
    [out]   error_status_t                      * status
)
```

(h) Take attribute update and translate it to foreign update operation:

```
sec_rep_prop_attr_update(
    [in]    unsigned32                    num_prop_attrs,
    [in, ref, size_is(num_prop_attrs)]
            rs_prop_attr_data_t           prop_attrs[],
    [out]   error_status_t                * status
)
```

(i) Take attribute delete and translate it to foreign delete operation:

```
sec_rep_prop_attr_delete(
    [in]    unsigned32                    num_prop_attrs,
    [in, ref, size_is(num_prop_attrs)]
            rs_prop_attr_data_t           prop_attrs[],
    [out]   error_status_t                * status
)
```

(j) Take attribute schema create and translate it to foreign attribute schema create operation:

```
sec_rep_prop_attr_schema_create(
    [in]   unsigned32                      num_schemas,
    [in, ref, size_is(num_schemas)]
           rs_prop_attr_sch_create_data_t  schemas[],
    [out]  error_status_t                  * status
)
```

(k) Take attribute schema delete and translate it to foreign attribute schema delete operation:

```
sec_rep_prop_attr_schema_delete(
    [in, ref]  rs_prop_attr_sch_update_data_t * schema,
    [in]       uuid_t                         * attr_id,
    [out]      error_status_t                 * status
)
```

(l) Take attribute schema update and translate it to foreign attribute schema update operation:

```
sec_rep_prop_attr_schema_update(
    [in, ref]  rs_prop_attr_sch_update_data_t * schema,
    [out]      error_status_t                 * status
)
```

(m) Take login activity update and translate it to foreign login activity:

```
sec_rep_prop_login_reset(
    [in]       sec_rgy_login_name_t       * login_name,
    [in]       sec_rgy_login_activity_t   * login_part,
    [out]      error_status_t             * status
)
```

(n) Take pgo add and translate it to foreign pgo add:

```
sec_rep_prop_pgo_add(
    [in]        sec_rgy_domain_t        domain,
    [in]        unsigned32              num_pgo_items,
    [in, size_is(num_pgo_items)]
                rs_prop_pgo_add_data_t  pgo_items[],
    [out]      error_status_t           * status
)
```

(o) Take pgo delete and translate it to foreign pgo delete:

```
sec_rep_prop_pgo_delete(
    [in]       sec_rgy_domain_t          domain,
    [in, ref]  sec_rgy_name_t            name,
    [in]       sec_timeval_sec_t         cache_info,
    [out]      error_status_t            * status
)
```

(p) Take pgo rename and translate it to foreign pgo rename:

```
sec_rep_prop_pgo_rename(
    [in]       sec_rgy_domain_t           domain,
    [in, ref]  sec_rgy_name_t             old_name,
    [in, ref]  sec_rgy_name_t             new_name,
    [out]      error_status_t             * status
)
```

(q) Take pgo replace and translate it to foreign pgo replace:

```
sec_rep_prop_pgo_replace(
    [in]        sec_rgy_domain_t        domain,
    [in, ref]   sec_rgy_name_t          name,
    [in, ref]   sec_rgy_pgo_item_t      * item,
    [out]       error_status_t          * status
)
```

(r) Take pgo add member and translate it to foreign pgo add member:

```
sec_rep_prop_pgo_add_member(
    [in]        sec_rgy_domain_t        domain,
    [in]        sec_rgy_name_t          go_name,
    [in]        unsigned32              num_members,
    [in, size_is(num_members)]
                sec_rgy_member_t        members[],
    [out]       error_status_t          * status
)
```

(s) Take pgo delete member and translate it to foreign pgo delete member:

```
sec_rep_prop_pgo_delete_member(
    [in]        sec_rgy_domain_t        domain,
    [in, ref]   sec_rgy_name_t          go_name,
    [in, ref]   sec_rgy_name_t          person_name,
    [out]       error_status_t          * status
)
```

(t) Take properties and translate it to foreign properties operation:

```
sec_rep_prop_properties_set_info(
    [in, ref]   sec_rgy_properties_t    * properties,
    [out]       error_status_t          * status
)
```

(u) Take policy and translate it to foreign policy operation:

```
sec_rep_prop_plcy_set_info(
    [in, ref]   sec_rgy_name_t          organization,
    [in, ref]   sec_rgy_plcy_t          * policy_data,
    [out]       error_status_t          * status
)
```

(v) Take auth policy and translate it to foreign auth policy operation:

```
sec_rep_prop_auth_plcy_set_info(
    [in, ref]   sec_rgy_login_name_t    * account,
    [in, ref]   sec_rgy_plcy_auth_t     * auth_policy,
    [out]       error_status_t          * status
)
```

To allow communication between the vendor code and the DCE 1.2 infrastructure there are two mechanisms:

(a) Each `sec_rep_*()` function returns a status code. These codes (listed below) indicate success with a zero (0) return code, and failure with a non-zero return code. The infrastructure will continue to attempt to contact the application periodically until it succeeds, or until the application requests a ‘‘re-initialization’’. An application would request a re-initialization by returning a `sec_rep_request_reinit` error.

   (i) `sec_rep_success` (0)

   (ii) `sec_rep_cant_process`

(iii) `sec_rep_request_reinit`

(b) The following functions will be called by the DCE 1.2 infrastructure during the initialization/destruction of a secondary replica. These interfaces allow ‘‘vendors’’ to implement these calls to do any foreign work necessary at these junctures. Vendors may choose to implement a ‘‘no-op’’ function which just returns when called.

(i) Vendor callout to do any necessary foreign authentication: `sec_rep_auth_init()`.

(ii) Vendor callout to notify foreign of interruption of service: `sec_rep_stop()`.

(iii) Vendor callout to do any cleanup: `sec_rep_destroy()`.

(iv) Vendor callout to do any work needed before re-initializing: `sec_rep_init_reinit()`.

5.2. DCE 1.2 Infrastructure for Secondary Replicas

The intent of providing infrastructure for secondary replicas within the core DCE is to provide a framework within which vendors will be able to build solutions for heterogeneous enterprise environments. The requirements for such solutions are constantly changing and the design of the infrastructure is focused on providing a solid foundation without restricting the implementations.

With this aim in mind, the DCE 1.2 infrastructure will provide the following support mechanisms to vendors of secondary replics:

(a) Administration Support:

(i) The ability to add/delete secondary replicas to new secondary replist.

(ii) The ability to start/stop secondary replicas.

(iii) The ability to configure a secondary replica.

(iv) The ability to start secondary replicas through `dced`.

(b) Server Initialization:

(i) The ability to notify the master that a secondary replica is now ready to receive propagations.

(ii) The ability to negotiate for initialization with a surrogate replica.

(iii) The ability to request re-initialization.

(c) Error Handling:

(i) The ability to notify the master that it cannot currently process propagations. Error conditions on propagation will be handled by the 1.2 infrastructure so that vendors need not be aware of failure conditions and the protocol between the master and its replicas.

(ii) The applications will be able to return a success (0) or failure (non-zero) status to the 1.2 infrastructure which will then return an appropriate error to the master.

(iii) The ability to notify the master that a secondary replica requests re-initialization (request will come via application return code). This will be handled by returning an error, `sec_rgy_rep_must_init_slave`, to the master which will cause the replicas state to be set to `rs_c_replica_prop_init`.

(d) Propagation Logic:

(i) The ability to receive bulk or individual propagations from the master.

[There is an additional proposal to support a single bulk interface (`rs_prop_bulk()`) for receiving propagation instead of the individual wire interfaces that currently exist in 1.1. Support for bulk propagation requires significant new code on the master and initially the ‘‘bulking’’ of records may not be implemented. The existence of a bulk wire propagation interface would be transparent to the `sec_rep_*()` library interfaces which vendors will code to. Vendors will need to check the `num_accts` field and be prepared to deal with an array of records or a single instance of a record.]

(e) Authentication:

(i) The ability to establish a login context.

5.2.1. General

The DCE 1.2 infrastructure code source files and directories will be documented in the implementation spec. Porting to other non-UNIX-based operating systems or modifications to the skeletal source code is the responsibility of vendors supplying secondary replicas.

Since the initialization and argument parsing aspects of process creation may vary based on the underlying operating system, there will be two aspects of the DCE 1.2 deliverables:

(a) There will be ''reference 1.2 infrastructure'' which will consist of a skeletal main program (`srs_main`) with a sample argument parsing function (`process_args_replica()`). These OS-specific functions will be UNIX-based and need to parse command line arguments and do process initialization before calling the `create_replica()` function.

(b) The base DCE 1.2 infrastructure will define the `create_replica()` function, and all other functions needed to provide the infrastructure defined above will be called from `create_replica()`:

```
void create_replica
(
    sec_id_t                 *rgy_local_cell,
    uuid_p_t                 rep_id,
    rpc_binding_vector_p_t   rep_bindings,
    rs_replica_twr_vec_p_t   rep_twrs,
    unsigned_char_p_t        princ_name,
                             /* principal name
                                (default: root) */
    unsigned_char_p_t        path_name,
                             /* pathname prefix for location
                                of locally generated files
                                (default:
                                /opt/dcelocal/var/srs) */
    unsigned_char_p_t        keytab,
                             /* alternate keytab
                                (default: DCE keytab) */
    unsigned_char_p_t        master_str,
    error_status_t           *st
)
```

The reference implementation default will be to integrate the `create_replica()` function into the `srs_main` program and compile and link with the ''application library'' (`sec_rep_lib.a`) to produce a `sec_replica` executable.

5.2.2. Authentication

A secondary replica needs to authenticate itself both to the local operating system on which it runs and to DCE. Authentication to the local operating system is documented under ''Porting Issues''.

When authenticating to DCE, it can authenticate as either:

(a) Root (inheriting the context from the machine principal, as primaries do).

(b) Any principal defined by an administrator (principal name is one of the arguments at secondary replica initialization).

In either case, it will need to follow the following steps to authenticate successfully to DCE. The DCE 1.2 infrastructure code will include the following support to allow the master and secondary replicas to mutually authenticate:

(a) A master must know the identity of its secondary replicas. It finds its secondary replicas through reading the secondary replica list. A new node will be created at the root level for the secondary replica list (e.g., `/.:/sec/replist2`). This node will have an ACL to control access to the secondary replist.

(b) Establishing the identity of a secondary replica includes:

  (i) The definition of the principal identity in the registry under which the secondary replica will run (admin action through `sec_rgy_acct_add()`).

  (ii) The registration of the replicas name in a CDS server entry (e.g., for replica `SR1`, the name is `/subsys/dce/sec/SR1`).

  (iii) The registration of the replicas and principal name in the secondary replica list (admin action through `sec_rgy_sec_rep_add_replica()`). (A principal name is required when adding a named replica to the secondary replica list via `sec_rgy_sec_rep_*()` APIs.)

  (iv) The ACL on the secondary replica list (which indicates which replicas are allowed to receive propagations from the master) must contain the principal identity of the secondary replica.

  (v) A keytab file may be created for the storage of the replica's long term key through the `dcecp` interface, or if not specified, the default DCE keytab will be used.

(c) When starting a secondary replica, a principal name and a password (supplied via a keytab file) can be supplied to identify the secondary replica.

(d) During initialization, each secondary replica will need to authenticate to the master via `sec_login_*()` calls and establish a login context for the process. If no ID and keytab are supplied, the ''default'' principal identity will be used (root), and the context will be inherited from the host machine's login context.

(e) During the propagation sequence the master tries to contact the replica to begin the propagation sequence:

  (i) The master contacts the slave requesting its authentication information, through the `rs_rep_mgr_get_info_and_creds()` interface. (This is essentially user-to-user authentication, and happens when the master calls the slave to establish a valid binding.)

  (ii) A secondary replica provides its PTGT (created during its initialization sequence through calls to `sec_login_setup_identity()` and `sec_login_validate_identity()`, and stored in the krb5 credential cache) to the master. This PTGT is encrypted under the long term key of the new secondary replica (which was added to the registry at config-time) and contains a session key that is known to the replica.

  (iii) On receipt of the slave PTGT the master decrypts the ticket (which it can do because it has all keys), and uses the session key obtained to encrypt additional tickets for use when authenticating the slave.

  (iv) To authenticate as the security service, name-based authentication is used. A secondary replica will ask the authenticated RPC for the caller's principal name and check that name against the architecturally-defined security service name (`dce-rgy`). Only the master can generate authenticated RPC's as `dce-rgy`. Slaves trust all such operations originating from the `dce-rgy` principal. The secondary replica will authenticate as the principal defined.

5.2.3. Authorization

All primary master-to-slave operations use name-based DCE authorization. All primary replicas identify to the runtime as `dce-rgy`. The secondary replica will authenticate as the principal defined.

All slave-to-master operations use standard PAC-base DCE authorization.

Access is controlled by the appropriate permission bit in the secondary replica list ACL.

5.2.4. Password Encryption

There are two cases where passwords are encrypted for transmission between replicas:

(a) At the initialization of a new replica, the new replica receives data from the ''surrogate'' which is helping it to bootstrap. The surrogate and the new replica share a session key and this session key is used to encrypt all password records over the wire.

(b) Once a replica is in service, the master may propagate a single record containing a password. In this case the master encrypts the password record in the key of the originator. On primary replicas this works because primarys maintain a complete copy of the registry and are able to lookup the key of any originator and use that key to decrypt the password record received.

Secondary replicas will support the same method of encryption in case (a). In case (b), secondary replicas need the password encrypted in a known key because they do not store the complete registry and do not have access to the originator's key. The master and the secondary replicas will use the session key for decryption of passwords during the propagation of principal DES keys. The semantics of the protocol do not change. Only the actual encryption/decryption key changes.

When the master is composing a propagation record from the log data, it knows whether the target replica is a primary or secondary replica because this information is stored in the volatile replica list. If the master is propagating to a secondary record it will retrieve the ''originator's key'' from the registry database and decrypt the key. It will then encrypt the password under the session key which is established by the master and the secondary replicas at initialization. The secondary replica will use the session key to decrypt passwords.

Once the ''bulk propagation'' logic is implemented on the master, a different key may be derived from the session key and it will be used to encrypt blocks of data records rather than individual records.

5.2.5. Replication

The mechanism for supporting different kinds of replicas will be built upon the existing propagation mechanism, which is not an external interface. A new data structure, the `secondary_replist`, will be added to the replication mechanism to identify secondary replicas. Administrative operations will allow for the addition, modification, and deletion of these replicas at the master.

(a) New admin APIs:

(i) `sec_rgy_sec_rep_add_replica()`

(ii) `sec_rgy_sec_rep_replace_replica()`

(iii) `sec_rgy_sec_rep_delete_replica()`

(iv) `sec_rgy_sec_rep_read()`

(v) `sec_rgy_sec_repadm_stop()`

(vi) `sec_rgy_sec_repadm_destroy()`

(vii) `sec_rgy_sec_repadm_info()`

(b) For administration (supplied by DCE 1.2) (`sec_rgy` operations on the master, translating to actions at the secondary replica):

(i) Notify replica of interruption: `rs_sec_repadm_stop()`.

(ii) Read local information about replica: `rs_sec_repadm_info()`.

(iii) Notify replica of termination: `rs_sec_repadm_destroy()`.

(c) For communication (supplied by DCE 1.2) (master initiates action to secondary replica):

(i) Master asks secondary replica for its credentials: `rs_rep_mgr_get_info_and_creds()`.

(ii) Master tells secondary replica to initialize itself: `rs_rep_mgr_init()`.

(iii) Master transmits bulk propagation: `rs_prop_bulk()`.

[The bulk propagation will be a performance enhancement in a future release. The intent of adding a bulk wire interface now is to prevent code changes when the bulk propagation code is supplied on the master. If the secondary replicas support the bulk interface, `rs_prop_bulk()`, to receive their records now they will not require any code change when the master is modified to do bulk propagations. One of the arguments to the `rs_prop_bulk()` call is `rs_replica_master_info_t`. This data structure contains the `update_seqno` and the `previous_update_seqno`. The DCE 1.2 secondary replica code will check that the previous sequence number is equal to the `last_update_seqno` to make sure the propagations are in sync.]

(d) For communication (supplied by DCE 1.2) (secondary replica initiates action to master):

(i) `rs_sec_rep_add_replica()`

(ii) `rs_sec_rep_replace_replica()`

(iii) `rs_sec_rep_delete_replica()`

(iv) `rs_sec_rep_read_replica()`

(v) Tell master I am here: `rs_rep_mgr_i_am_slave()`.

(vi) Tell master I am initialized: `rs_rep_mgr_init_done()`.

(e) For communication (supplied by DCE 1.2) (secondary replica initiates action to surrogate replica):

(i) Establish contact with surrogate for initialization sequence: `rrs_rep_adm_info()`.

(ii) Tell surrogate to begin copying its database: `rrs_rep_mgr_copy_all()`.

(f) For propagation (supplied by DCE 1.2) (surrogate replica initiates action via `rrs_*()`, secondary replica receives data via `rs_*()`):

(i) Process all pgos from surrogate: `rs_prop_pgo_add()`.

(ii) Process all accounts from surrogate: `rs_prop_acct_add()`.

(iii) Process all policy from surrogate: `rs_prop_policy()`.

(iv) Process all ACLs from surrogate: `rs_prop_acl_replace()`.

(v) Process all schemas from surrogate: `rs_prop_attr_schema_create()`.

(vi) Process all attributes from surrogate: `rs_prop_attr_update()`.

5.3. Administrative Interfaces

A new set of interfaces will be added to `dcecp` to manage secondary replica replication list (‘‘replist’’) entries. When replicas initialize themselves, they will communicate to the master that they are a secondary replica through the add interface and the master will store that information in a separate list of secondary replica information.

All admin operations that currently list or display replicas must be modified to display the new replicas as ‘‘secondary’’.

5.4. Relationship With Other Replicas

The new type of secondary replicas will not be on the same replist as full primary replicas, so the functions that try to establish the validity of a replica during change master (`chk_bind_to_new_master()`) will not see secondary replicas and not allow them to be used as a master.

`rs_m_replist_get_init_frm_reps()` is the function that is called to find a replica to initialize from. This function reads the volatile replist on the master (which contains secondary replicas). It will be modified to pass over secondary replicas as not valid to be used for initialization.

6. NEW SECONDARY REPLICA INTERFACES

6.1. idl/rs_sec_repadm.idl

```
[
 uuid(),
 version(1.0),
```

```
 pointer_default(ptr)
]
interface rs_sec_repadm
{
    import "dce/rgynbase.idl";
    import "dce/rplbase.idl";

    /*
    /*
     *  r s _ s e c _ r e p a d m _ s t o p
     *
     *  Stop the secondary  replica identified by this handle.
     */
    void rs_sec_repadm_stop(
        [in]    handle_t            h,
        [out]   error_status_t      *status
    );

    /*
     *  r s _s e c _  r e p a d m _ i n f o
     *
     *  Get basic information about a secondary replica such
     *  as its state, UUID, latest update sequence
     *  number and timestamp.
     */
    void rs_sec_repadm_info(
        [in]    handle_t               h,
        [out]   rs_sec_replica_info_t  *rep_info,
        [out]   error_status_t         *status
    );

    /*
     *  r s _ s e c _ r e p  a d m _ d e s t r o y
     *
     *  A drastic operation which tells a secondary replica
     *  to destroy its database and exit.
     */
    void rs_sec_repadm_destroy(
        [in]    handle_t            h,
        [out]   error_status_t      *status
    );

    /*
     *  s e c _ r g y _ s e c _ r e p a d m  _ s t o p
     *
     *  Stop the secondary replica identified by this handle.
     */
    void sec_rgy_sec_repadm_stop (
        [in]    sec_rgy_handle_t    context,
        [out]   error_status_t      *status
    );

    /*
     *  s e c _ r g y _ s e c _ r e p a d m _ i n f o
     *
```

```
     *  Get basic information about a secondary replica such
     *  as its state, UUID, latest update sequence
     *  number and timestamp.
     *  Also get the replica's information about the master's
     *  UUID and the sequence number when the master was
     *  designated.
     */
    void sec_rgy_sec_repadm_info(
        [in]    sec_rgy_handle_t    context,
        [out]   rs_sec_replica_info_t   *rep_info,
        [out]   error_status_t      *status
    );

    /*
     *  s e c _ r g y _ s e c _ r e p a d m _ d e s t r o y
     *
     *  A drastic operation which tells a secondary replica
     *  to destroy its database and exit.
     */
    void sec_rgy_sec_repadm_destroy(
        [in]    sec_rgy_handle_t    context,
        [out]   error_status_t      *status
    );
}
```

6.2. idl/rs_prop_misc.idl

```
[
 uuid(),
 version(1.0),
 pointer_default(ptr)
]
interface rs_prop
{
    import "dce/rgynbase.idl";
    import "dce/rplbase.idl";
    import "dce/rsbase.idl";

    /*
     * rs_prop_bulk
     */
    void  rs_prop_bulk (
        [in]       handle_t                    h,
        [in]       unsigned32                  num_rec,
        [in, ref, size_is(num_rec)] idl_pkl_t  rec[],
        [in, ref]  rs_replica_master_info_t    * master_info,
        [in]       boolean32                   propq_only,
        [out]      error_status_t              * status
    );
}
```

7. IMPLEMENT EXISTING REPLICA INTERFACES

7.1. idl/rs_repmgr.idl

```
interface rs_repmgr
{
```

```
import "dce/rgynbase.idl";
import "dce/rplbase.idl";
import "dce/rsbase.idl";

/*
 *  rs_rep_mgr_get_info_and_creds
 *
 *  Get a replica’s basic state information
 *  and credentials to authenticate to it.
 */
void rs_rep_mgr_get_info_and_creds(
    [in]    handle_t                    h,
    [out]   rs_replica_info_t           *rep_info,
    [out]   rs_replica_auth_p_t         *rep_auth_info,
    [out]   error_status_t              *st
);

/*
 *  rs_rep_mgr_init
 *
 *  Master tells slave to initialize itself from
 *  one of the "init_from_reps".  "init_id" identifies
 *  the initialize event and prevents redundant
 *  initializations.
 *
 *  The slave returns the ID of the replica it will
 *  init from and the last update sequence number and
 *  timestamp of the "init_from_rep".
 */
void rs_rep_mgr_init(
    [in]    handle_t                    h,
    [in]    uuid_p_t                    init_id,
    [in]    unsigned32                  nreps,
    [in, size_is(nreps)]
            uuid_p_t                    init_from_rep_ids[],
    [in, size_is(nreps)]
            rs_replica_twr_vec_p_t      init_from_rep_twrs[],
    [in]    rs_replica_master_info_p_t  master_info,
    [out]   uuid_t                      *from_rep_id,
    [out]   rs_update_seqno_t           *last_upd_seqno,
    [out]   sec_timeval_t               *last_upd_ts,
    [out]   error_status_t              *st
);

/*
 *  rs_rep_mgr_init_done
 *
 *  Slave tells master that it is finished initializing
 *  itself from "from_rep_id".
 */
void rs_rep_mgr_init_done(
    [in]    handle_t                    h,
    [in]    uuid_p_t                    rep_id,
    [in]    uuid_p_t                    init_id,
    [in]    uuid_p_t                    from_rep_id,
```

```
        [in]    rs_update_seqno_t           *last_upd_seqno,
        [in]    sec_timeval_t               *last_upd_ts,
        [in]    error_status_t              *init_st,
        [out]   error_status_t              *st
    );
}
```

8. NEW RGY INTERFACES — IMPLEMENTED AT MASTER ONLY

8.1. idl/sreplist.idl

```
[
 local
]
interface sec_rgy_sec_replist
{
    import "dce/rgynbase.idl";
    import "dce/binding.idl";
    import "dce/rplbase.idl";

    /*
     *  sec_rgy_sec_rep_add_replica
     *
     *  Add a replica to the secondary_replica list.
     *
     *  Master-only operation.
     */
    void sec_rgy_sec_rep_add_replica(
        [in]    sec_rgy_handle_t        context,
        [in]    rs_sec_replica_item_t   srep_info,
        [out]   error_status_t          *status
    );

    /*
     *  sec_rgy_sec_rep_read
     *
     *  Read the replica list.
     *
     *  To start reading at the beginning of the secondary
     *  replica list, set marker to uuid_nil.
     *  To read information about a specific replica, set
     *  marker to its uuid and max_ents to 1.
     *
     *  The returned marker contains the uuid of the next
     *  secondary replica on the list.  Marker contains uuid_nil
     *  when there are no more secondary replicas on the list.
     */
    void sec_rgy_sec_rep_read(
        [in]        sec_rgy_handle_t    context,
        [in, out]   uuid_t              *marker,
        [in]        unsigned32          max_ents,
        [out]       unsigned32          *n_ents,
        [out, length_is(*n_ents), size_is(max_ents)]
                    rs_sec_replica_item_t   sreplist[],
        [out]       error_status_t      *status
    );
```

```
    /*
     *  sec_rgy_sec_rep_replace_replica
     *
     *  Replace information about replica "rep_id" on the
     *  secondary replica list.
     *
     *  Master-only operation.
     */
    void sec_rgy_sec_rep_replace_replica(
        [in]    sec_rgy_handle_t        context,
        [in]    rs_sec_replica_item_t   srep_info,
        [out]   error_status_t          *status
    );

    /*
     *  sec_rgy_sec_rep_delete_replica
     *
     *  Delete the replica identified by "rep_id".
     *  If "force_delete" is false, send the delete
     *  to the replica identified by "rep_id" as
     *  well as the other replicas.
     *  If "force_delete" is true, do not send the
     *  delete to the replica identified by "rep_id";
     *  it has been killed off some other way.
     *
     *  The master may NOT be deleted with this operation.
     *
     *  Master-only operation.
     */
    void sec_rgy_sec_rep_delete_replica(
        [in]    sec_rgy_handle_t        context,
        [in]    rs_sec_replica_item_t   srep_info,
        [in]    boolean32               force_delete,
        [out]   error_status_t          *status
    );
}
```

9. NEW MANAGEMENT INTERFACES — MASTER ONLY

9.1. idl/rs_sec_replist.idl

```
[
 uuid(),
 version(1.0),
 pointer_default(ptr)
]
interface rs_sec_replist
{
    import "dce/rgynbase.idl";
    import "dce/rplbase.idl";

    /*
     *  rs_sec_replist_add_replica
     *
     *  Add a replica to the secondary replica list.
     *
```

```
     *  Master-only operation.
     */
    void rs_sec_replist_add_replica(
        [in]    handle_t                h,
        [in]    rs_sec_replica_item_t   srep_info,
        [out]   error_status_t          *status
    );

    /*
     *  rs_sec_replist_replace_replica
     *
     *  Replace information about replica "srep_info" on the
     *  replica list.
     *
     *  Master-only operation.
     */
    void rs_sec_replist_replace_replica(
        [in]    handle_t                h,
        [in]    rs_sec_replica_item_t   srep_info,
        [out]   error_status_t          *status
    );

    /*
     *  rs_sec_replist_delete_replica
     *
     *  Delete the replica identified by "srep_info".
     *
     *  Master-only operation.
     */
    void rs_sec_replist_delete_replica(
        [in]    handle_t                h,
        [in]    rs_sec_replica_item_t   srep_info,
        [out]   error_status_t          *status
    );

    /*
     *  rs_sec_replist_read
     *
     *  Read the replica list
     *
     *  To start reading at the beginning of the secondary
     *  replica list, set marker to uuid_nil.
     *  To read information about a specific replica, set
     *  marker to its uuid and max_ents to 1.
     *
     *  The returned marker contains the uuid of the next
     *  replica on the secondary list.  Marker contains uuid_nil
     *  when there are no more replicas on the list.
     */
    void rs_sec_replist_read(
        [in]        handle_t            h,
        [in, out]   uuid_t              *marker,
        [in]        unsigned32          max_ents,
        [out]       unsigned32          *n_ents,
        [out, length_is(*n_ents), size_is(max_ents)]
```

```
                    rs_sec_replica_item_t   sreplist[],
        [out]       error_status_t      *status
    );
)
```

10. LOCAL INTERFACES

Discussed elsewhere in this document.

11. RESTRICTIONS AND LIMITATIONS

It is important to note that storage of a cleartext password even using a two-way transformation is contentious. This functionality will only be activated when the PLAINTEXT ERA is set and sites using this functionality should be aware of the security implications of storing cleartext passwords.

12. OTHER COMPONENT DEPENDENCIES

12.1. Audit

There will be several new audit events similar to the events that exist for primary replicas:

(a) SECREPADMIN_Stop

(b) SECREPADMIN_Maint

(c) SECREPADMIN_Destroy

(d) SECREPADMIN_Init

Also these events will be added to the default filters.

13. COMPATIBILITY

There was concern about how migration from a 1.1 to a 1.2 configuration will occur. The 1.2 master will be capable of supporting propagation of cleartext passwords but 1.1 servers are not aware of cleartext passwords or secondary replicas.

After reviewing the current migration strategy and the changes needed to support the storage of cleartext we do not think it will be feasible to support a mixture of 1.2 and 1.1 replicas in a cell.

Regardless of how cleartext is stored, it needs to be transmitted to secondary replicas encrypted in a session key established between the master and the secondary replica at initialization (details above).

If cleartext were stored in an ERA, it could potentially be propagated to 1.1 replicas since 1.1 replicas already know about ERAs. But it would need to be stored encrypted in the database using the long term database key and 1.1 replicas would not know that the data was encrypted or know not to allow query on this ‘‘special ERA’’ which would be a security issue. And, if a 1.2 master crashes and there are no other 1.2 replicas in the cell capable of becoming the master, a 1.1 replica will become the master. When a 1.1 replica becomes master, the information about the secondary replica list and the ability to support the storage of cleartext passwords is lost. If a 1.2 replica then became the master again, all the information propagated from the former master would be in some intermediate state.

For this reason, using secondary replicas will require all replicas to migrate to 1.2.

14. STANDARDS

No standards have been established to which registry synchronization must adhere.

15. OPEN ISSUES

(a) How will this functionality work with Public Key? Will there be new prop API’s for public key?

(b) Audit:

 (i) The audit model needs to be explored to determine whether or not there is existing support for multiple secondary replicas sharing an audit trail or whether each should maintain its own trail. Also whether these secondaries should be like primaries and share a trail with the master.

 (ii) There is a current limitation with the existing audit mechanism which does not allow a server id to be carried in an audit record. Support for this should be considered.

(c) Make sure support is added for inter-domain checking for name-based authorization.

(d) Are new permission bits needed in ACL to discriminate between changing towers and names?

16. PORTING ISSUES

(a) The reference operating system is UNIX and the source code supplied relys on concepts like `argv`, `argc` for parsing of command line arguments and running as the ‘‘root identity’’. Should/can secondary replicas run as unprivileged processes?

APPENDIX A. BACKGROUND ON REPLICATION

To understand the functioning of the new secondary replicas, a description of the current initialization sequence is given. Areas of change for new secondary replica behavior are indicated with ‘‘*’’.

(a) During the initialization of a slave replica, there is a bootstrapping process. The slave replica processes command line arguments, gets a binding to the master, gets some default information about the cell from the master and then sets up its own skeletal database.

* (b) When initializing a secondary replica, command line arguments may indicate a principal name and key. Secondary replicas get a binding to the master through the namespace and get default information about the cell from the master. Secondary replicas setup their own state and master files under a default path of `/opt/dcelocal/var/srs` unless overridden by a command line argument.

(c) The slave then calls the master via `sec_rgy_replist_add_replica()` to let the master know its replica ID, name, and the towers by which it can be contacted. It then sets its own local state to be `rs_c_state_uninitialized`.

* (d) Secondary_replicas will call `sec_rgy_secondary_replist_add_replica()` to add an entry at the master on the secondary replica list. The replica principal name will be stored in local replica state information in addition to setting its local state to uninitialized. The secondary replica will be able to be run as root (default) (inheriting the context from the machine principal, as primaries do), or as a principal supplied by a command line argument.

(e) It then registers itself with the name service, creates a `pe_site` file and exits.

* (f) Secondary replicas will setup their authentication information through a call to `rs_rep_auth_init()`. Secondary replicas will register their server entry name (e.g., `/.:/subsys/dce/sec/SR1`), but not add themselves as part of the `/.:/sec` group.

(g) On the master, the `sec_rgy_replist_add_replica()` translates into an replist add (`rsdb_replica_add()` of the named replica with a state of `rs_c_replica_prop_init`), and an addition of the replica to the masters volatile copy of the replist (`rs_m_replist_add_replica()`).

* (h) On the master, the `sec_rgy_secondary_replist_add_replica()` translates into a secondary replist add (`rsdb_replica_add()` of the secondary replica with a state of `rs_c_replica_prop_init`), and an addition of the replica to the masters volatile copy of the replist (`rs_m_replist_add_secondary_replica()`), marking the replica as a secondary replica.

(i) On the master there are several tasks called `prop_driver()`. There are multiple tasks for types of response from replicas:

 (i) `Long_incommunicado` — Any replica that could not be contacted with four or more consecutive retries.

(ii) `Short_incommunicado` — Any replica that could not be contacted the first time.

(iii) `Communicado` — Any replica that responded to the last communication. Each sleeps on requests for service.

(j) When an event is written to the log file, or an administrator takes action to ''reinitialize'' a replica (`rs_rep_admin_init_replica()`) or if a change master event occurs, the `prop_tasks` are sent a signal and wake up.

(k) The next time a `prop_driver()` wakes, it loops through the replica list looking for things to do.

* (l) It will now find secondary replicas on the volatile master replist.

(m) One of the things the master does is facilitate the initialization of new replicas. The master checks for a valid binding handle for the replica.

(n) In the initialization case it doesn't have a valid binding handle and there is an unauthenticated call made to the new replica to acquire its `rs_replica_auth_t` info. Primary replicas use the machine principal's credentials inherited from the root credentials and retrieved by a call to `rs_login_get_host_login_context()`. The `auth_info` is then passed to the master so the master may authenticate to the slave using the session key sealed in the auth info.

* (o) In the initialization of secondary replicas, the master doesn't have a valid binding handle and there is an unauthenticated call to the secondary replica (`rs_rep_mgr_get_info_and_creds()`). The secondary replica will have its own method of authentication (`rs_rep_auth_init()`) and communicate its `auth_info` to the master through this call.

(p) The `prop_driver()` looks for a list of good candidates for the replica to ''initialize'' from (it can't take the time to send out all updates to load a new replica, unless this is the first replica other than the master to be started), and calls the new replica telling it to initialize itself (`rrs_rep_mgr_init()`).

(q) The new replica through a call to `rs_rep_mgr_init()`, authenticates the caller (doesn't want to communicate with anyone other than members of its cell) and begins its own initialization, `rs_rep_init()`. The replica reviews the list of candidates, gets a binding to one and tries to contact it. It then creates a task, `rs_rep_init_copy_all_to_me()`, and establishes a session key with the ''surrogate''. It then calls the surrogate through `rrs_rep_mgr_copy_all()` and the surrogate does a ''bulk'' transmission of the principals, groups and orgs. The replica also checkpoints the new database (`rsdb_checkpt()`), and then tries to contact the master to let it know it has successfully completed this exchange.

(r) The secondary replica will support the `rs_rep_mgr_init()` interface and will authenticate the caller, get a binding to one of the ''surrogates'' communicated through the `rs_rep_mgr_init()` call. The secondary replica establishes contact with the surrogate through a call to `rrs_rep_mgr_copy_all()` and sets up a task to receive the transmissions from the surrogate. Secondary replicas will call the `rrs_rep_mgr_init_done()` interface to contact the master to let it know it has successfully completed the exchange and to notify the master to change its state to ''initialized''.

(s) If the replica never contacts the master, the `prop_driver()` periodically checks the volatile replist for replicas in the ''initializing'' state. The master calls `init_done()` to validate that the replica has been added to the replist and updates the replist entry for the replica with the sequence number and time stamp received from the replica.

AUTHOR'S ADDRESS

Maryann Hondo
HP
300 Apollo Drive
Chelmsford, MA 02174
USA

Internet email: hondo@apollo.hp.com
Telephone: +1-508-436-4233