Kerberos V5 Implementer's Guide

MIT Information Systems

June 15, 1994

Contents

1 Introduction

This document is designed to aide the programmer who plans to change MIT's implementation of Kerberos significantly. It is geared towards programmers who are already familiar with Kerberos and how MIT's implementation is structured.

Some of the more basic information needed can be found in the API document, although many functions have been repeated where additional information has been included for the implementer. The function descriptions included are up to date, even if the description of the functions may not be very verbose.

2 Cache and Key table functions

2.1 Credentials cache functions

The credentials cache functions (some of which are macros which call to specific types of credentials caches) deal with storing credentials (tickets, session keys, and other identifying information) in a semi-permanent store for later use by different programs.

2.1.1 The krb5_cc_ops structure

In order to implement a new credentials cache type, the programmer should declare a **krb5_cc_ops** structure, and fill in the elements of the structure appropriately, by implementing each of the credential cache functions for the new credentials cache type.

The prefix element specifies the prefix name of the the new credential cache type. For example, if the prefix name is "FILE", then if the program calls **krb5_cc_resolve**() with a credential cache name such as "FILE:/tmp/krb5_cc_15806", then **krb5_cc_resolve**() will call the resolve function (as defined by the **krb5_cc_ops** structure where the prefix element is "FILE") and pass it the argument "/tmp/krb5_cc_15806".

Before a new credentials cache type can be recognized by **krb5_cc_resolve**(), it must be registered with the Kerberos library by calling **krb5_cc_register**().

```
typedef struct _krb5_cc_ops {
char *prefix;
char *(*get_name)((krb5_ccache));
krb5_error_code (*resolve)((krb5_ccache *, char *));
krb5_error_code (*gen_new)((krb5_ccache *));
krb5_error_code (*init)((krb5_ccache, krb5_principal));
krb5_error_code (*destroy)((krb5_ccache));
krb5_error_code (*close)((krb5_ccache));
krb5_error_code (*store)((krb5_ccache, krb5_creds *));
```

```
krb5_error_code (*retrieve)((krb5_ccache, krb5_flags,
   krb5_creds *, krb5_creds *));
krb5_error_code (*get_princ)((krb5_ccache,
krb5_principal *));
krb5_error_code (*get_first)((krb5_ccache,
krb5_cc_cursor *));
krb5_error_code (*get_next)((krb5_ccache, krb5_cc_cursor *,
   krb5_creds *));
krb5_error_code (*end_get)((krb5_ccache, krb5_cc_cursor *));
krb5_error_code (*remove_cred)((krb5_ccache, krb5_flags,
      krb5_creds *));
krb5_error_code (*set_flags)((krb5_ccache, krb5_flags));
} krb5_cc_ops;
```

2.1.2 Per-type functions

The following entry points must be implemented for each type of credentials cache. However, **resolve**() and **gen_new**() are only called by the credentials cache glue code. They are not called directly by the application.

krb5_error_code
resolve(*/* OUT */*
    krb5_ccache * `id`,
    / IN */*
    char * `residual`) resolve

Creates a credentials cache named by `residual` (which may be interpreted differently by each type of ccache). The cache is not opened, but the cache name is held in reserve.

krb5_error_code
gen_new(*/* OUT */*
    krb5_ccache * `id`) gen_new

Creates a new credentials cache whose name is guaranteed to be unique. The cache is not opened. `*id` is filled in with a **krb5_ccache** which may be used in subsequent calls to ccache functions.

krb5_error_code
init(*/* IN/OUT */*
    krb5_ccache `id`,
    / IN */*
    krb5_principal `primary_principal`) init

Creates/refreshes a credentials cache identified by `id` with primary principal set to `primary_principal`. If the credentials cache already exists, its contents are destroyed.

Modifies: cache identified by `id`.

destroy

```
krb5_error_code
destroy(/* IN */
        krb5_ccache id)
```

Destroys the credentials cache identified by `id`, invalidates `id`, and releases any other resources acquired during use of the credentials cache. Requires that `id` identifies a valid credentials cache. After return, `id` must not be used unless it is first reinitialized.

close

```
krb5_error_code
close(/* IN/OUT */
      krb5_ccache id)
```

Closes the credentials cache `id`, invalidates `id`, and releases `id` and any other resources acquired during use of the credentials cache. Requires that `id` identifies a valid credentials cache. After return, `id` must not be used unless it is first reinitialized.

store

```
krb5_error_code
store(/* IN */
      krb5_ccache id,
      krb5_creds * creds)
```

Stores `creds` in the cache `id`, tagged with `creds->client`. Requires that `id` identifies a valid credentials cache.

retrieve

```
krb5_error_code
retrieve(/* IN */
         krb5_ccache id,
         krb5_flags whichfields,
         krb5_creds * mcreds,
         /* OUT */
         krb5_creds * creds)
```

Searches the cache `id` for credentials matching `mcreds`. The fields which are to be matched are specified by set bits in `whichfields`, and always include the principal name `mcreds->server`. Requires that `id` identifies a valid credentials cache.

If at least one match is found, one of the matching credentials is returned in `*creds`. The credentials should be freed using **krb5_free_credentials**().

get_princ

```
krb5_error_code
get_princ(/* IN */
          krb5_ccache id,
          krb5_principal * principal)
```

Retrieves the primary principal of the credentials cache (as set by the **init**() request) The primary principal is filled into `*principal`; the caller should release this memory by calling **krb5_free_principal**() on `*principal` when finished.

Requires that `id` identifies a valid credentials cache.

```
krb5_error_code
get_first(/* IN */
          krb5_ccache id,
          /* OUT */
          krb5_cc_cursor * cursor)
```
get_first

Prepares to sequentially read every set of cached credentials. Requires that `id` identifies a valid credentials cache opened by **krb5_cc_open**(). `cursor` is filled in with a cursor to be used in calls to **get_next**().

```
krb5_error_code
get_next(/* IN */
         krb5_ccache id,
         /* OUT */
         krb5_creds * creds,
         /* IN/OUT */
         krb5_cc_cursor * cursor)
```
get_next

Fetches the next entry from `id`, returning its values in `*creds`, and updates `*cursor` for the next request. Requires that `id` identifies a valid credentials cache and `*cursor` be a cursor returned by **get_first**() or a subsequent call to **get_next**().

```
krb5_error_code
end_get(/* IN */
        krb5_ccache id,
        krb5_cc_cursor * cursor)
```
end_get

Finishes sequential processing mode and invalidates `*cursor`. `*cursor` must never be re-used after this call.

Requires that `id` identifies a valid credentials cache and `*cursor` be a cursor returned by **get_first**() or a subsequent call to **get_next**().

```
krb5_error_code
remove_cred(/* IN */
            krb5_ccache id,
            krb5_flags which,
            krb5_creds * cred)
```
remove_cred

Removes any credentials from `id` which match the principal name cred-¿ server and the fields in `cred` masked by `which`. Requires that `id` identifies a valid credentials cache.

couldn't delete.

```
krb5_error_code
set_flags(/* IN */
          krb5_ccache id,
          krb5_flags flags)
```
set_flags

Sets the flags on the cache `id` to `flags`. Useful flags are defined in `<krb5/ccache.h>`.

2.2 Replay cache functions

The replay cache functions deal with verifying that AP_REQ's do not contain duplicate authenticators; the storage must be non-volatile for the site-determined validity period of authenticators.

Each replay cache has a string **name** associated with it. The use of this name is dependent on the underlying caching strategy (for file-based things, it would be a cache file name). The caching strategy should use non-volatile storage so that replay integrity can be maintained across system failures.

2.2.1 The krb5_rc_ops structure

In order to implement a new replay cache type, the programmer should declare a **krb5_rc_ops** structure, and fill in the elements of the structure appropriately, by implementing each of the replay cache functions for the new replay cache type.

The prefix element specifies the prefix bf name of the the new replay cache type. For example, if the prefix **name** is "dfl", then if the program calls **krb5_rc_resolve**() with a credential cache name such as "dfl:host", then **krb5_rc_resolve**() will call the resolve function (as defined by the **krb5_rc_ops** structure where the prefix element is "dfl") and pass it the argument "host".

Before a new replay cache type can be recognized by **krb5_rc_resolve**(), it must be registered with the Kerberos library by calling **krb5_rc_register**().

```
typedef struct _krb5_rc_ops {
    char *type;
    krb5_error_code (*init)((krb5_rcache,krb5_deltat));
    krb5_error_code (*recover)((krb5_rcache));
    krb5_error_code (*destroy)((krb5_rcache));
    krb5_error_code (*close)((krb5_rcache));
    krb5_error_code (*store)((krb5_rcache,krb5_donot_replay *));
    krb5_error_code (*expunge)((krb5_rcache));
    krb5_error_code (*get_span)((krb5_rcache,krb5_deltat *));
    char *(*get_name)((krb5_rcache));
    krb5_error_code (*resolve)((krb5_rcache, char *));
} krb5_rc_ops;
```

2.2.2 Per-type functions

The following entry points must be implemented for each type of replay cache.

init

krb5_error_code
init(*/* IN */*
    krb5_rcache `id`,
    krb5_deltat `auth_lifespan`)

Creates/refreshes the replay cache identified by `id` and sets its authenticator lifespan to `auth_lifespan`. If the replay cache already exists, its contents are destroyed.

krb5_error_code recover
recover(/* *IN* */
krb5_rcache `id`)

Attempts to recover the replay cache `id`, (presumably after a system crash or server restart).

krb5_error_code destroy
destroy(/* *IN* */
krb5_rcache `id`)

Destroys the replay cache `id`. Requires that `id` identifies a valid replay cache.

krb5_error_code close
close(/* *IN* */
krb5_rcache `id`)

Closes the replay cache `id`, invalidates `id`, and releases any other resources acquired during use of the replay cache. Requires that `id` identifies a valid replay cache.

krb5_error_code store
store(/* *IN* */
krb5_rcache `id`,
krb5_donot_replay * `rep`)

Stores `rep` in the replay cache `id`. Requires that `id` identifies a valid replay cache.

Returns KRB5KRB_AP_ERR_REPEAT if `rep` is already in the cache. May also return permission errors, storage failure errors.

krb5_error_code expunge
expunge(/* *IN* */
krb5_rcache `id`)

Removes all expired replay information (i.e. those entries which are older than then authenticator lifespan of the cache) from the cache `id`. Requires that `id` identifies a valid replay cache.

krb5_error_code get_span
get_span(/* *IN* */
krb5_rcache `id`,
/* *OUT* */
krb5_deltat * `auth_lifespan`)

Fills in `auth_lifespan` with the lifespan of the cache `id`. Requires that `id` identifies a valid replay cache.

resolve

krb5_error_code
resolve(*/* IN/OUT */*
krb5_rcache `id`,
/ IN */*
char * `name`)

Initializes private data attached to `id`. This function MUST be called before the other per-replay cache functions.

Requires that `id` points to allocated space, with an initialized `id->ops` field.

Since **resolve**() allocates memory, **close**() must be called to free the allocated memory, even if neither **init**() or **recover**() were successfully called by the application.

rc_get_name

char *
krb5_rc_get_name(*/* IN */*
krb5_rcache `id`)

Returns the name (excluding the type) of the rcache `id`. Requires that `id` identifies a valid replay cache.

2.3 Key table functions

The key table functions deal with storing and retrieving service keys for use by unattended services which participate in authentication exchanges.

Keytab routines should all be atomic. Before a routine returns it must make sure that all non-sharable resources it acquires are released and in a consistent state. For example, an implementation is not allowed to leave a file open for writing or to have a lock on a file.

Note that all keytab implementations must support multiple concurrent sequential scans. Another detail to note is that the order of values returned from **get_next**() is unspecified and may be sorted to the implementor's convenience.

2.3.1 The krb5_kt_ops structure

In order to implement a new key table type, the programmer should declare a **krb5_kt_ops** structure, and fill in the elements of the structure appropriately, by implementing each of the key table functions for the new key table type.

In order to reduce the size of binary programs when static linking is used, it is common to provide two **krb5_kt_ops** structures for each key table type, one for reading only in which the pointers to the add and delete functions are zero, and one for reading and writing.

```
typedef struct _krb5_kt_ops {
char *prefix;
        /* routines always present */
krb5_error_code (*resolve)((char *,
  krb5_keytab *));
```

```
krb5_error_code (*get_name)((krb5_keytab,
  char *,
  int));
krb5_error_code (*close)((krb5_keytab));
krb5_error_code (*get)((krb5_keytab,
  krb5_principal,
  krb5_kvno,
  krb5_keytab_entry *));
krb5_error_code (*start_seq_get)((krb5_keytab,
  krb5_kt_cursor *));
krb5_error_code (*get_next)((krb5_keytab,
  krb5_keytab_entry *,
  krb5_kt_cursor *));
krb5_error_code (*end_get)((krb5_keytab,
  krb5_kt_cursor *));
/* routines to be included on extended version (write routines) */
krb5_error_code (*add)((krb5_keytab,
  krb5_keytab_entry *));
krb5_error_code (*remove)((krb5_keytab,
  krb5_keytab_entry *));
} krb5_kt_ops;
```

2.3.2 Per-type functions that are always present

The following entry points must be implemented for each type of key table. However, **resolve**(), **remove**() and **add**() are only called by the key table glue code. They are not called directly by the application.

however, application programs are not expected to call **resolve**(), **remove**(), or **add**() directly.

krb5_error_code resolve
resolve(*/* IN */*
char * `residual`,
/ OUT */*
krb5_keytab * `id`)

Fills in `*id` with a handle identifying the keytab with name "residual". The interpretation of "residual" is dependent on the type of keytab.

krb5_error_code get_name
get_name(*/* IN */*
krb5_keytab `id`,
/ OUT */*
char * `name`,
/ IN */*
int `namesize`)

name is filled in with the first `namesize` bytes of the name of the keytab identified by **id**(). If the name is shorter than `namesize`, then ,
name will be null-terminated.

close

```
krb5_error_code
close(/* IN */
      krb5_keytab id)
```

Closes the keytab identified by `id` and invalidates `id`, and releases any other resources acquired during use of the key table.

Requires that `id` identifies a valid credentials cache.

get

```
krb5_error_code
get(/* IN */
    krb5_keytab id,
    krb5_principal principal,
    krb5_kvno vno,
    /* OUT */
    krb5_keytab_entry * entry)
```

Searches the keytab identified by `id` for an entry whose principal matches `principal` and whose key version number matches `vno`. If `vno` is zero, the first entry whose principal matches is returned.

This routine should return an error code if no suitable entry is found. If an entry is found, the entry is returned in `*entry`; its contents should be deallocated by calling **close**() when no longer needed.

close

```
krb5_error_code
close(/* IN/OUT */
      krb5_keytab_entry * entry)
```

Releases all storage allocated for `entry`, which must point to a structure previously filled in by **get**() or **get_next**().

start_seq_get

```
krb5_error_code
start_seq_get(/* IN */
              krb5_keytab id,
              /* OUT */
              krb5_kt_cursor * cursor)
```

Prepares to read sequentially every key in the keytab identified by `id`. `cursor` is filled in with a cursor to be used in calls to **get_next**().

get_next

```
krb5_error_code
get_next(/* IN */
         krb5_keytab id,
         /* OUT */
         krb5_keytab_entry * entry,
         /* IN/OUT */
         krb5_kt_cursor cursor)
```

Fetches the "next" entry in the keytab, returning it in `*entry`, and updates `*cursor` for the next request. If the keytab changes during the sequential get, an error must be guaranteed. `*entry` should be freed after use by calling **close**().

Requires that `id` identifies a valid credentials cache. and `*cursor` be a cursor returned by **start_seq_get**() or a subsequent call to **get_next**().

krb5_error_code
end_get(/* *IN* */
        krb5_keytab `id`,
        krb5_kt_cursor * `cursor`) end_get

Finishes sequential processing mode and invalidates `cursor`, which must never be re-used after this call.

Requires that `id` identifies a valid credentials cache. and `*cursor` be a cursor returned by **start_seq_get**() or a subsequent call to **get_next**().

2.3.3 Per-type functions to be included for write routines

krb5_error_code
add(/* *IN* */
    krb5_keytab `id`,
    krb5_keytab_entry * `entry`) add

Stores `entry` in the keytab `id`. Fails if the entry already exists.

This operation must, within the constraints of the operating system, not return until it can verify that the write has completed succesfully. For example, in a UNIX file-based implementation, this routine (or the part of the underlying implementation that it calls) would be responsible for doing an **fsync**() on the file before returning.

This routine should return an error code if `entry` is already present in the keytab or if the key could not be stored (quota problem, etc).

krb5_error_code
remove(/* *IN* */
        krb5_keytab `id`,
        krb5_keytab_entry * `entry`) remove

Searches the keytab `id` for an entry that exactly matches `entry`. If one is found, it is removed from the keytab.

3 Operating-system specific functions

The operating-system specific functions provide an interface between the other parts of the `libkrb5.a` libraries and the operating system.

Beware! Any of the functions below are allowed to be implemented as macros. Prototypes for functions can be found in `<krb5/libos-proto.h>`; other definitions (including macros, if used) are in `<krb5/libos.h>`.

The following global symbols are provided in `libos.a`. If you wish to substitute for any of them, you must substitute for all of them (they are all declared and initialized in the same object file):

extern char *`krb5_config_file`**:** name of configuration file

extern char *`krb5_trans_file`**:** name of hostname/realm name translation file

extern char *`krb5_defkeyname`**:** default name of key table file

extern char *`krb5_lname_file`**:** name of aname/lname translation database

extern int `krb5_max_dgram_size`**:** maximum allowable datagram size

extern int `krb5_max_skdc_timeout`**:** maximum per-message KDC reply timeout

extern int `krb5_skdc_timeout_shift`**:** shift factor (bits) to exponentially back-off the KDC timeouts

extern int `krb5_skdc_timeout_1`**:** initial KDC timeout

extern char *`krb5_kdc_udp_portname`**:** name of KDC UDP port

extern char *`krb5_default_pwd_prompt1`**:** first prompt for password reading.

extern char *`krb5_default_pwd_prompt2`**:** second prompt

4 Principal database functions

The `libkdb.a` library provides a principal database interface to be used by the Key Distribution center and other database manipulation tools.

db_set_name

krb5_error_code
krb5_db_set_name(/* *IN* */
char * `name`)

Set the name of the database to `name`.

Must be called before **krb5_db_init**() or after **krb5_db_fini**(); must not be called while db functions are active.

db_set_nonblocking

krb5_error_code
krb5_db_set_nonblocking(/* *IN* */
krb5_boolean `newmode`,
/* *OUT* */
krb5_boolean * `oldmode`)

Changes the locking mode of the database functions, returning the previous mode in `*oldmode`.

If `newmode` is TRUE, then the database is put into non-blocking mode, which may result in “database busy” error codes from the get, put, and iterate routines.

If `newmode` is FALSE, then the database is put into blocking mode, which may result in delays from the get, put and iterate routines.

The default database mode is blocking mode.

krb5_error_code db_init
krb5_db_init()

Called before using **krb5_db_get_principal**(), **krb5_db_put_principal**(), **krb5_db_iterate**(), and **krb5_db_set_nonblocking**().

Does any required initialization.

krb5_error_code db_fini
krb5_db_fini()

Called after all database operations are complete, to perform any required clean-up.

krb5_error_code db_get_age
krb5_db_get_age(*/* IN */*
char * db_name,
/ OUT */*
time_t * age)

Retrieves the age of the database **db_name**() (or the current default database if **db_name**() is NULL).

*age is filled in in local system time units, and represents the last modification time of the database.

krb5_error_code db_create
krb5_db_create(*/* IN */*
char * db_name)

Creates a new database named **db_name**(). Will not create a database by that name if it already exists. The database must be populated by the caller by using **krb5_db_put_principal**().

krb5_error_code db_rename
krb5_db_rename(*/* IN */*
char * source,
char * dest)

Renames the database ,
source to ,
dest

Any database named ,
dest is destroyed.

db_get_principal

```
krb5_error_code
krb5_db_get_principal(/* IN */
                      krb5_principal searchfor,
                      /* OUT */
                      krb5_db_entry * entries,
                      /* IN/OUT */
                      int * nentries,
                      /* OUT */
                      krb5_boolean * more)
```

Retrieves the principal records named by `searchfor`.

`entries` must point to an array of `*nentries` krb5_db_entry structures. At most `*nentries` structures are filled in, and `*nentries` is modified to reflect the number actually returned.

`*nentries` must be at least one (1) when this function is called.

`*more` is set to TRUE if there are more records that wouldn't fit in the available space, and FALSE otherwise.

The principal structures filled in have pointers to allocated storage; **krb5_db_free_principal**() should be called with `entries` and `*nentries` in order to free this storage when no longer needed.

db_free_principal

```
void
krb5_db_free_principal(/* IN */
                       krb5_db_entry * entries,
                       int nentries)
```

Frees allocated storage held by `entries` as filled in by **krb5_db_get_principal**().

db_put_principal

```
krb5_error_code
krb5_db_put_principal(/* IN */
                      krb5_db_entry * entries,
                      int * nentries)
```

Stores the `*nentries` principal structures pointed to by `entries` in the database.

`*nentries` is updated upon return to reflect the number of records acutally stored; the first `*nentries` records will have been stored in the database.

db_iterate

```
krb5_error_code
krb5_db_iterate(/* IN */
                krb5_error_code (*func)(krb5_pointer ,
                                        krb5_db_entry * ),
                krb5_pointer iterate_arg)
```

Iterates over the database, fetching every entry in an unspecified order and calling `(*func)(iterate_arg, principal)` where `principal` points to a record from the database.

If `(*func)()` ever returns an error code, the iteration should be aborted and that error code is returned by this function.

krb5_error_code **krb5_db_store_mkey**(/* *IN* */
char * `keyfile`,
krb5_principal `mname`,
krb5_keyblock * `key`)

db_store_mkey

Put the KDC database master key into the file `keyfile`. If `keyfile` is NULL, then a default file name derived from the principal name `mname` is used.

krb5_error_code **krb5_db_fetch_mkey**(/* *IN* */
krb5_principal `mname`,
krb5_encrypt_block * `eblock`,
krb5_boolean `fromkeyboard`,
krb5_boolean `twice`,
krb5_data `salt`,
/* *IN/OUT* */
krb5_keyblock * `key`)

db_fetch_mkey

Get the KDC database master key from somewhere, filling it into `*key`. `key->keytype` should be set to the desired key type.

If `fromkeyboard` is TRUE, then the master key is read as a password from the user's terminal. In this case: `eblock` should point to a block with an appropriate **string_to_key**() function; if `twice` is TRUE, the password is read twice for verification; and if `salt` is non-NULL, it is used as the salt when converting the typed password to the master key.

If `fromkeyboard` is false, then the key is read from a file whose name is derived from the principal name `mname`. Therefore, `eblock`, `twice` and `salt` are ignored.

`mname` is the name of the key sought; this is often used by **string_to_key**() to aid in conversion of the password to a key.

krb5_error_code **krb5_kdb_encrypt_key**(/* *IN* */
krb5_encrypt_block * `eblock`,
const krb5_keyblock * `in`,
/* *IN/OUT* */
krb5_encrypted_keyblock * `out`)

kdb_encrypt_key

Encrypt a key for storage in the database. `eblock` is used to encrypt the key in `in` into `out`; the storage pointed to by `*out` is allocated before use and should be freed when the caller is finished with it.

kdb_decrypt_key

```
krb5_error_code
krb5_kdb_decrypt_key(/* IN */
                     krb5_encrypt_block * eblock,
                     const krb5_encrypted_keyblock * in,
                     /* IN/OUT */
                     krb5_keyblock * out)
```

Decrypt a key from storage in the database. `eblock` is used to decrypt the key in `in` into `out`; the storage pointed to by `*out` is allocated before use and should be freed when the caller is finished with it.

db_setup_mkey_name

```
krb5_error_code
krb5_db_setup_mkey_name(/* IN */
                        const char *keyname,
                        const char *realm,
                        /* OUT */
                        char ** fullname,
                        krb5_principal * principal)
```

Given a key name `keyname` and a realm name `realm`, construct a principal which can be used to fetch the master key from the database. This principal is filled into `*principal`; `*principal` should be freed by **krb5_free_principal**() when the caller is finished.

If `keyname` is NULL, the default key name will be used.

If `fullname` is not NULL, it is set to point to a string representation of the complete principal name; its storage may be freed by calling **free**() on `*fullname`.

5 Encryption system interface

Kerberos v5 has the ability to use multiple encryption systems. Any encryption system which desires to link with and be usable from the MIT Kerberos v5 implementation must implement at least this interface:

5.1 Functional interface

encrypt_func

```
krb5_error_code
encrypt_func(krb5_const_pointer in,
             krb5_pointer out,
             const size_t size,
             krb5_encrypt_block * eblock,
             krb5_pointer ivec)
```

Encrypts `size` bytes at `in`, storing result in `out`. `eblock` points to an encrypt block which has been initialized by **process_key**().

`in` must include sufficient space beyond the `size` bytes of input data to hold pad and redundancy check bytes; the macro **krb5_encrypt_size**() can be used to compute this size.

out must be preallocated by the caller to contain sufficient storage to hold the output; the macro **krb5_encrypt_size**() can be used to compute this size.

ivec points to an initial vector/seed to be used in the encryption. If null, the cryptosystem may choose an appropriate initialization vector.

decrypt_func

krb5_error_code
decrypt_func(krb5_const_pointer in,
krb5_pointer out,
const size_t size,
krb5_encrypt_block * eblock,
krb5_pointer ivec)

Decrypts size bytes at in, storing result in out. eblock points to an encrypt block which has been initialized by **process_key**().

size must be a multiple of the encryption block size.

out must be preallocated by the caller to contain sufficient storage to hold the output; this is guaranteed to be no more than the input size.

ivec points to an initial vector/seed to be used in the decryption. If null, the cryptosystem may choose an appropriate ivec.

process_key

krb5_error_code
process_key(krb5_encrypt_block * eblock,
const krb5_keyblock * keyblock)

Does any necessary key preprocessing (such as computing key schedules for DES). eblock->crypto_entry must be set by the caller; the other elements of eblock are to be assigned by this function. [In particular, eblock->key must be set by this function if the key is needed in raw form by the encryption routine.]

The caller may not move or reallocate keyblock before calling **finish_key**() on eblock.

finish_key

krb5_error_code
finish_key(krb5_encrypt_block * eblock)

Does any necessary clean-up on eblock (such as releasing resources held by eblock->priv.

string_to_key

krb5_error_code
string_to_key(const krb5_keytype keytype,
krb5_keyblock * keyblock,
const krb5_data * data,
const krb5_data salt)

Converts the string pointed to by data into an encryption key of type keytype. *keyblock is filled in with the key info; in particular, keyblock->contents is to be set to allocated storage. It is the responsibility of the caller to release this storage when the generated key no longer needed.

The routine may use salt to seed or alter the conversion algorithm.

If the particular function called does not know how to make a key of type `keytype`, an error may be returned.

init_random_key

```
krb5_error_code
init_random_key(const krb5_keyblock * seedblock,
                krb5_pointer * seed)
```

Initialize the random key generator using the encryption key `seedblock` and allocating private sequence information, filling in `*seed` with the address of such information. `*seed` is to be passed to **random_key**() to provide sequence information.

finish_random_key

```
krb5_error_code
finish_random_key(krb5_pointer * seed)
```

Free any resources held by `seed` and assigned by **init_random_key**().

random_key

```
krb5_error_code
random_key(krb5_pointer * seed,
           krb5_keyblock ** keyblock)
```

Generate a random encryption key, allocating storage for it and filling in the keyblock address in `*keyblock`. When the caller has finished using the keyblock, he should call **krb5_free_keyblock**() to release its storage.

combine_keys

```
krb5_error_code
combine_keys(/* IN */
             const krb5_keyblock * key1,
             const krb5_keyblock * key2,
             /* OUT */
             krb5_keyblock ** outkey)
```

Combine the two encryption keys `key1` and `key2` to generate a new output key `outkey`. `outkey` is filled in to point to the freshly-allocated key. When the caller is finished using the `*outkey`, it should be freed with **krb5_free_keyblock**().

5.2 Other data elements

In addition to the above listed function entry points, each encryption system should have an entry in `krb5_csarray` and a **krb5_cryptosystem_entry** structure describing the entry points and key and padding sizes for the encryption system.

5.3 DES functions

The DES functions conform to the encryption interface required by the Kerberos version 5 library, and provide an encryption mechanism based on the DES Cipher-block chaining mode (CBC), with the addition of a cyclical redundancy check (CRC-32) for integrity checking upon decryption.

The functions have the same signatures as those described by the main library document; the names are:

mit_des_encrypt_func()

mit_des_decrypt_func()

mit_des_process_key()

mit_des_finish_key()

mit_des_string_to_key()

mit_des_init_random_key()

mit_des_finish_random_key()

mit_des_random_key()

The **krb5_cryptosystem_entry** for this cryptosystem is `mit_des_cryptosystem_entry`.

6 Checksum interface

Kerberos v5 has the ability to use multiple checksum algorithms. Any checksum implementation which desires to link with and be usable from the MIT Kerberos v5 implementation must implement this interface:

6.1 Functional interface

krb5_error_code sum_func
sum_func(*/* IN */*
krb5_pointer `in`,
size_t `in_length`,
krb5_pointer `seed`,
size_t `seed_length`,
/ OUT */*
krb5_checksum * `outcksum`)

This routine computes the desired checksum over `in_length` bytes at `in`. `seed_length` bytes of a seed (usually an encryption key) are pointed to by `seed`. Some checksum algorithms may choose to ignore `seed`. If `seed_length` is zero, then there is no seed available. The routine places the resulting value into `outcksum->contents`.

`outcksum->contents` must be set by the caller to point to enough storage to contain the checksum; the size necessary is an element of the **krb5_checksum_entry** structure.

6.2 Other data elements

In addition to the above listed function entry point, each checksum algorithm should have an entry in `krb5_cksumarray` and a **krb5_checksum_entry** structure describing the entry points and checksum size for the algorithm.

7 CRC-32 checksum functions

The `libcrc32.a` library provides an implementation of the CRC-32 checksum algorithm which conforms to the interface required by the Kerberos library.

crc32_sum_func

```
static krb5_error_code
crc32_sum_func(/* IN */
               krb5_pointer in,
               size_t in_length,
               krb5_pointer seed,
               size_t seed_length,
               /* OUT */
               krb5_checksum * outcksum)
```

This routine computes a CRC-32 checksum over `in_length` bytes at `in`, and places the resulting value into `outcksum->contents`. `seed` is ignored.

`outcksum->contents` must be set by the caller to point to at least 4 bytes of storage.

Index