KADM5 Library and Server Implementation Design*

Barry Jaspan

September 10, 1996

Contents

*api-server-design.tex,v 1.20 1996/08/21 20:00:20 bjaspan Exp

1 Overview

The KADM5 administration system is designed around the KADM5 API. The “server-side” library libkadm5srv.a implements the KADM5 API by operating directly on the underlying KDC and admin databases. The “client-side” library libkadm5clnt.a implements the KADM5 API via an RPC mechanism. The administration server kadmind accepts RPC requests from the client-side library and translates them into calls to the server-side library, performing authentication, authorization, and logging along the way.

The two libraries, libkadm5clnt.a and libkadm5srv.a, export the identical kadm5 interface; for example, both contain definitions for kadm5_get_principal, and all other kadm5 functions. In most cases, the client library function just marshalls arguments and results into and out of an RPC call, whereas the server library function performs the actual operation on the database file. kadm5_init_*, however, are substantially different even though they export the same interface: on the client, they establish the RPC connection and GSS-API context, whereas on the server side the open the database files, read in the password dictionary, and the like. Also, the kadm5_free functions operate on local process memory in both libraries.

The admin server is implemented as a nearly-stateless transaction server, where each admin API function represents a single transaction. No per-client or per-connection information is stored; only local database handles are maintained between requests. The RPC mechanism provides access to remote callers' authentication credentials for authorization purposes.

The admin API is exported via an RPC interface that hides all details about network encoding, authentication, and encryption of data on the wire. The RPC mechanism does, however, allow the server to access the underlying authentication credentials for authorization purposes.

The admin system maintains two databases:

- The master Kerberos (KDC) database is used to store all the information that the Kerberos server understands, thus allowing the greatest functionality with no modifications to a standard KDC.
- The KDC database also stores kadm5-specific per-principal information in each principal's krb5_tl_data list. In a prior version, this data was stored in a separate admin principal database; thus, when this document refers to "the admin principal database," it now refers to the appropriate krb5_tl_data entries in the KDC database.
- The policy database stores kadm5 policy information.

The per-principal information stored in the admin principal database consists of the principal's policy name and an array of the principal's previous keys. The old keys are stored encrypted in the key of the special principal "kadmin/history" that is created by the server library when it is first needed. Since a change in kadmin/history's key renders every principal's key history array useless, it can only be changed using the ovsec_adm_edit utility; that program will reencrypt every principal's key history in the new key.[1] The server library refuses all requests to change kadmin/history's key.

2 API Handles

Each call to kadm5_init_* on the client or server creates a new API handle. The handles encapsulate the API and structure versions specified by kadm5_init_*'s caller and all other internal data needed by the library. A process can have multiple open API handles simultaneously by calling kadm5_init_* multiple times, and call can specify a different version, client or service principal, and so forth.

Each kadm5 function verifies the handle it is given with the CHECK_HANDLE or _KADM5_CHECK_HANDLE macros. The CHECK_HANDLE macro differs for the client and server library because the handle types used by those libraries differ, so it is defined in both <client_internal.h> and <server_internal.h> in the library source directory. In each header file, CHECK_HANDLE first calls GENERIC_CHECK_HANDLE, defined

[1] ovsec_adm_edit has not yet been implemented, and there are currently no plans to implement it; thus, the history cannot currently be changed.

in <admin_internal.h>, which verifies the magic number, API version, and structure version that is contained in both client and server handles. CHECK_HANDLE then calls either CLIENT_CHECK_HANDLE or SERVER_CHECK_HANDLE respectively to verify the client- or server-library specific handle fields.

The CHECK_HANDLE macro is useful because it inlines the handle check instead of requiring a separate function call. However, using CHECK_HANDLE means that a source file cannot be compiled once and included into both the client and server library, because CHECK_HANDLE is always either specific to either the client or server library, not both. There are a number of functions that can be implemented with the same code in both the client and server libraries, however, including all of the kadm5_free functions and kadm5_chpass_principal_util. The _KADM5_CHECK_HANDLE macro solves this problem; instead of inlining the handle check, it calls the function _kadm5_check_handle which is defined separately in both the client and server library, in client_init.c and server_init.c. Since these two files are only compiled once and put in a single library, they simply verify the handle they are passed with CHECK_HANDLE and return the result.

3 API Versioning

The KADM5 system is designed to support multiple versions of the KADM5 API. Presently, two versions exist: KADM5_API_VERSION_1 and KADM5_API_VERSION_2. The former is equivalant to the initial OpenVision API, OVSEC_KADM_API_VERSION_1; the latter was created during the initial integration of the OpenVision system into the MIT release.

Implementing a versioned API in C via with both local and RPC access presents a number of design issues, some of them quite subtle. The contexts in which versioning considerations must be made include:

1. Typedefs, function declarations, and defined constants depend on the API version a client is written to and must be correct at compile time.
2. Each function in the server library must behave according to the API version specified by the caller at runtime to kadm5_init_*.
3. The XDR functions used by the RPC layer to transmit function arguments and results must encode data structures correctly depending on the API version specified by the client at runtime.
4. Each function in the client library must behave according to the API version specified by the caller at runtime to kadm5_init_*.

5. The RPC server (kadmind) must accept calls from a client using any supported API version, and must then invoke the function in the server library corresponding to the RPC with the API version indicated by the client caller.
6. When a first API function is invoked that needs to call a second function in the API on its own behalf, and that second API function's behavior depends on the API version specified, the first API function must either be prepared to call the second API function at whatever version its caller specifies or have a means of always calling the second API function at a pre-determined version.

The following functions describe how each context is handled.

3.1 Designing for future compatibility

Any code whose behavior depends on the API version should be written so as to be compatible with future, currently unknown API versions on the grounds that any particularly piece of API behavior will most likely not change between versions. For example, in the current system, the code is not written as "if this is VERSION_1, do X, else if this is VERSION_2, do Y"; instead, it is written as "if this is VERSION_1, do X; else, do Y." The former will require additional work when VERSION_3 is defined, even if "do Y" is still the correct action, whereas the latter will work without modification in that case.

3.2 Header file declarations

Typedefs, defined constants and macros, and function declarations may change between versions. A client is always written to a single, specific API version, and thus expects the header files to define everything according to that API. Failure of a header file to define values correctly will result in either compiler warnings (e.g. if the pointer type of a function argument changes) or fatal errors (e.g. if the number of arguments to a function changes, or the fields of a structure change). For example, in VERSION_1, kadm5_get_policy took a pointer to a pointer to a structure, and in VERSION_2 it takes a pointer to a structure; that would generate a warning if not correct. In VERSION_1, kadm5_randkey_principal accepted three arguments but in VERSION_2 accepts four; that would generate a fatal error.

The header file defines everything correctly based on the value of the USE_KADM5_API_VERSION constant. The constant can be assigned to an integer corresponding to any supported API version, and defaults to the newest version. The header files then simply use an #ifdef to include the right definitions:

```
#if USE_KADM5_API_VERSION == 1
kadm5_ret_t    kadm5_get_principal(void *server_handle,
                                   krb5_principal principal,
                                   kadm5_principal_ent_t *ent);
#else
kadm5_ret_t    kadm5_get_principal(void *server_handle,
                                   krb5_principal principal,
                                   kadm5_principal_ent_t ent,
                                   long mask);
#endif
```

3.3 Server library functions

Server library functions must know how many and what type of arguments to expect, and must operate on those arguments correctly, based on the API version with which they are invoked. The API version is contained in the handle that is alwasy passed as their first argument, generated by kadm5_init_* (to which the client specified the API version to use at run-time).

In general, it is probably unsafe for a compiled function in a library to re-interpret the number and type of defined arguments at run-time since the calling conventions may not allow it; for example, a function whose first argument was a short in one version and a pointer in the next might fail if it simply typed-casted the argument. In that case, the function would have to written to take variable arguments (i.e. use <stdarg.h>) and extract them from the stack based on the API version. Alternatively, a separate function for each API version could be defined, and <kadm5/admin.h> could be written to `#define` the exported function name based on the value of USE_KADM5_API_VERSION.

In the current system, it turns out, that isn't necessary, and future implementors should take try to ensure that no version has semantics that will cause such problems in the future. All the functions in KADM5 that have different arguments or results between VERSION_1 and VERSION_2 do so simply by type-casting their arguments to the appropriate version and then have separate code paths to handle each one correctly. kadm5_get_principal, in svr_principal.c, is a good example. In VERSION_1, it took the address of a pointer to a kadm5_principal_ent_t to fill in with a pointer to allocated memory; in VERSION_2, it takes a pointer to a structure to fill in, and a mask of which fields in that structure should be filled in. Also, the contents of the kadm5_principal_ent_t changed slightly between the two versions. kadm5_get_principal handles versioning as follows (following along in the source code will be helpful):

1. If VERSION_1, it saves away its entry argument (address of a pointer to a structure)

and resets its value to contain the address of a locally stack-allocated entry structure; this allows most of the function to written once, in terms of VERSION_2 semantics. If VERSION_1, it also resets its mask argument to be KADM5_PRINCIPAL_NORMAL_MASK, because that is the equivalent to VERSION_1 behavior, which was to return all the fields of the structure.

2. The bulk of the function is implemented as expected for VERSION_2.

3. The new fields in the VERSION_2 entry structure are assigned inside a block that is only execute if the caller specified VERSION_2. This saves a little time for a VERSION_1 caller.

4. After the entry structure is filled, the function checks again if it was called as VERSION_1. If so, it allocates a new kadm5_principal_ent_t_v1 structure (which is conveniently defined in the header file) with malloc, copies the appropriate values from the entry structure into the VERSION_1 entry structure, and then writes the address of the newly allocated memory into address specified by the original entry argument which it had previously saved away.

There is another complication involved in a function re-interpreting the number of arguments it receives at compile time—it cannot assign any value to an argument for which the client did not pass a value. For example, a VERSION_1 client only passes three arguments to kadm5_get_principal. If the implementation of kadm5_get_principal notices that the caller is VERSION_1 and therefore assigns its fourth argument, mask, to a value that mimics the VERSION_1 behavior, it may inadvertently overwrite data on its caller's stack. This problem can be avoided simply by using a true local variable in such cases, instead of treating an unpassed argument as a local variable.

3.4 XDR functions

The XDR functions used to encode function arguments and results must know how to encode the data for any API version. This is important both so that all the data gets correctly transmitted and so that protocol compatibility between clients or servers using the new library but an old API version is maintained; specific, new kadmind servers should support old kadm5 clients.

The signature of all XDR functions is strictly defined: they take the address of an XDR function and the address of the data object to be encoded or decoded. It is thus impossible to provide the API version of the data object as an additional argument to an XDR function. There are two other means to convey the information, storing the API version to use as a field in the data object itself and creating separate XDR functions to handle each different version of the data object, and both of them are used in KADM5.

In the client library, each kadm5 function collects its arguments into a single structure to be passed by the RPC; similarly, it expects all of the results to come back as a single structure from the RPC that it will then decode back into its constituent pieces (these are the standard ONC RPC semantics). In order to pass versioning information to the XDR functions, each function argument and result datatype has a filed to store the API version. For example, consider kadm5_get_principal's structures:

```
struct gprinc_arg {
        krb5_ui_4 api_version;
        krb5_principal princ;
        long mask;
};
typedef struct gprinc_arg gprinc_arg;
bool_t xdr_gprinc_arg();

struct gprinc_ret {
        krb5_ui_4 api_version;
        kadm5_ret_t code;
        kadm5_principal_ent_rec rec;
};
typedef struct gprinc_ret gprinc_ret;
bool_t xdr_gprinc_ret();
```

kadm5_get_principal (in client_principal.c) assigns the api_version field of the gprinc_arg to the version specified by its caller, assigns the princ field based on its arguments, and assigns the mask field from its argument if the caller specified VERSION_2. It then calls the RPC function clnt_call, specifying the XDR functions xdr_gprinc_arg and xdr_gprinc_ret to handle the arguments and results.

xdr_gprinc_arg is invoked with a pointer to the gprinc_arg structure just described. It first encodes the api_version field; this allows the server to know what to expect. It then encodes the krb5_principal structure and, if api_version is VERSION_2, the mask. If api_version is not VERSION_2, it does not encode *anything* in place of the mask, because an old VERSION_1 server will not expect any other data to arrive on the wire there.

The server performs the kadm5_get_principal call and returns its results in an XDR encoded gprinc_ret structure. clnt_call, which has been blocking until the results arrived, invokes xdr_gprinc_ret with a pointer to the encoded data for it to decode. xdr_gprinc_ret first decodes the api_version field, and then the code field since that is present in all versions to date. The kadm5_principal_ent_rec presents a problem, however. The structure does not itself contain an api_version field, but the structure is different between the two versions. Thus, a single XDR function cannot decode both versions of the structure because it will have no way to decide which version to expect. The solution is to have two functions, kadm5_principal_ent_rec_v1 and kadm5_principal_ent_rec, which always decode according to VERSION_1 or VERSION_2, respectively. gprinc_ret knows which one to invoke because it has the api_version field returned by the server (which is always the same as that specified by the client in the gpring_arg).

In hindsight, it probably would have been better to encode the API version of all structures directly in a version field in the structure itself; then multiple XDR functions for a single data type wouldn't be necessary, and the data objects would stand complete on their own. This can be added in a future API version if desired.

3.5 Client library functions

Just as with server library functions, client library functions must be able to interpret their arguments and provide result according to the API version specified by the caller. Again, kadm5_get_principal (in client_principal.c) is a good example. The gprinc_ret structure that it gets back from clnt_call contains a kadm5_principal_ent_rec or a kadm5_principal_ent_rec_v1 (the logic is simplified somewhat because the VERSION_2 structure only has new fields added on the end). If kadm5_get_principal was invoked with VERSION_2, that structure should be copied into the pointer provided as the entry argument; if it was invoked with VERSION_1, however, the structure should be copied into allocated memory whose address is then written into the pointer provided by the entry argument. Client library functions make this determination based on the API version specified in the provided handle, just like server library functions do.

3.6 Admin server stubs

When an RPC call arrives at the server, the RPC layer authenticates the call using the GSS-API, decodes the arguments into their single-structure form (ie: a gprinc_arg) and dispatches the call to a stub function in the server (in server_stubs.c). The stub function first checks the caller's authorization to invoke the function and, if authorized, calls the kadm5 function corresponding to the RPC function with the arguments specified in the single-structure argument.

Once again, kadm5_get_principal is a good example for the issues involved. The contents of the gprinc_arg given to the stub (get_principal_1) depends on the API version the caller on the client side specified; that version is available to the server in the api_version field of the gprinc_arg. When the server calls kadm5_get_principal in the server library, it must give that function an API handle that contains the API version requested by the client; otherwise the function semantics might not be correct. One possibility would be for the server to call kadm5_init for each client request, specifing the client's API version number and thus generating an API handle with the correct version, but that would be prohibitively inefficient. Instead, the server dips down in the server library's internal abstraction barrier, using the function new_server_handle to cons up a server handle based on the server's own global_server_handle but using the API version specified by the client. The server then passes the newly generated handle to kadm5_get_principal, ensuring the right behavior, and creates the gprinc_ret structure in a manner similar to that described above.

Although new_server_handle solves the problem of providing the server with an API handle containing the right API version number, it does not solve another problem: that a single source file, server_stubs.c, needs to be able to invoke functions with arguments appropriate for multiple API versions. If the client specifies VERSION_1, for example, the server must invoke kadm5_get_principal with three arguments, but if the client specifies VERSION_2 the server must invoke kadm5_get_principal with four arguments. The compiler will not allow this inconsistency. The server defines wrapper functions in a separate source file that match the old version, and the separate source file is compiled with USE_KADM5_API_VERSION set to the old version; see kadm5_get_principal_v1 in server_glue_v1.c. The server then calls the correct variant of kadm5_get_principal_* based on the API version and puts the return values into the gprinc_ret in a manner similar to that described above.

Neither of these solutions are necessarily correct. new_server_handle violates the server library's abstraction barrier and is at best a kludge; the server library should probably export a function to provide this behavior without violating the abstraction; alternatively, the librar should be modified so that having the server call kadm5_init for each client RPC request would not be too inefficient. The glue functions in server_glue_v1.c really are not necessary, because the server stubs could always just pass dummy arguments for the extra arguments; after all, the glue functions pass *nothing* for the extra arguments, so they just end up as stack garbage anyway.

Another alternative to the new_server_handle problem is to have the server always invoke server library functions at a single API version, and then have the stubs take care of converting the function arguments and results back into the form expected by the caller. In general, however, this might require the stubs to duplicate substantial logic already present in the server library and further violate the server library's abstraction barrier.

3.7 KADM5 self-reference

Some kadm5 functions call other kadm5 functions "on their own behalf" to perform functionality that is necessary but that does not directly affect what the client sees. For example, kadm5_chpass_principal has to enforce password policies; thus, it needs to call kadm5_get_principal and, if the principal has a policy, kadm5_get_policy and kadm5_modify_principal in the process of changing a principal's password. This leads to a complication: what API handle should kadm5_chpass_principal pass to the other kadm5 functions it calls?

The "obvious," but wrong, answer is that it should pass the handle it was given by its caller. The caller may provide an API handle specifying any valid API version. Although the semantics of kadm5_chpass_principal did not change between VERSION_1 and VERSION_2, the declarations of both kadm5_get_principal and kadm5_get_policy did. Thus, to use the caller's API handle, kadm5_chpass_principal will have to have a separate code path for each API version, even though it itself did not change bewteen versions, and duplicate a lot of logic found elsewhere in the library.

Instead, each API handle contains a "local-use handle," or lhandle, that kadm5 functions should use to call other kadm5 functions. For example, the client-side library's handle structure is:

```
typedef struct _kadm5_server_handle_t {
        krb5_ui_4       magic_number;
        krb5_ui_4       struct_version;
        krb5_ui_4       api_version;
        char *          cache_name;
        int             destroy_cache;
        CLIENT *        clnt;
        krb5_context    context;
        kadm5_config_params params;
        struct _kadm5_server_handle_t *lhandle;
} kadm5_server_handle_rec, *kadm5_server_handle_t;
```

The lhandle field is allocated automatically when the handle is created. All of the fields of the API handle that are accessed outside kadm5_init are also duplicated in the lhandle; however, the api_version field of the lhandle is always set to a *constant* value, regardless of the API version specified by the caller to kadm5_init. In the current implementation, the lhandle's api_version is always VERSION_2.

By passing the caller's handle's lhandle to recursively called kadm5 functions, a kadm5 function is assured of invoking the second kadm5 function with a known API version. Additionally, the lhandle's lhandle field points back to the lhandle, in case kadm5 functions call themselves more than one level deep; handle− >lhandle always points to the same lhandle, no matter how many times the indirection is performed.

This scheme might break down if a kadm5 function has to call another kadm5 function to perform operations that they client will see and for its own benefit, since the semantics of the recursively-called kadm5 function may depend on the API version specified and the client may be depending on a particular version's behavior. Future implementators should avoid creating a situation in which this is possible.

4 Server Main

The admin server starts by trapping all fatal signals and directing them to a cleanup-and-exit function. It then creates and exports the RPC interface and enters its main loop.

The main loop dispatches all incoming requests to the RPC mechanism. In a previous version, after 15 seconds of inactivity, the server closed all open databases; each database was be automatically reopened by the API function implementations as necessary. That behavior existed to protect against loss of written data before the process exited. The current database libraries write all changes out to disk immediately, however, so this behavior is no longer required or performed.

5 Remote Procedure Calls

The RPC for the Admin system will be based on ONC RPC. ONC RPC is used because it is a well-known, portable RPC mechanism. The underlying external data representation (xdr) mechanisms for wire encapsulation are well-known and extensible. Authentication to the admin server and encryption of all RPC functional arguments and results are be handled via the AUTH_GSSAPI authentication flavor of ONC RPC.

6 Database Record Types

6.1 Admin Principal, osa_princ_ent_t

The admin principal database stores records of the type osa_princ_ent_t (declared in <kadm5/adb.h>), which is the subset of the kadm5_principal_ent_t structure that is not stored in the Kerberos database plus the necessary bookkeeping information. The records are keyed by the ASCII representation of the principal's name, including the trailing NULL.

```
typedef struct _osa_pw_hist_t {
     int n_key_data;
     krb5_key_data *key_data;
} osa_pw_hist_ent, *osa_pw_hist_t;

typedef struct _osa_princ_ent_t {
        char * policy;
        u_int32 aux_attributes;
```

```
        unsigned int old_key_len;
        unsigned int old_key_next;
        krb5_kvno admin_history_kvno;
        osa_pw_hist_ent *old_keys;


        u_int32 num_old_keys;
        u_int32 next_old_key;
        krb5_kvno admin_history_kvno;
        osa_pw_hist_ent *old_keys;
} osa_princ_ent_rec, *osa_princ_ent_t;
```

The fields that are different from kadm5_principal_ent_t are:

num_old_keys The number of previous keys in the old_keys array. This value must be $0 \leq$ num_old_keys $<$ pw_history_num.

old_key_next The index into old_keys where the next key should be inserted. This value must be $0 \leq$ old_key_next $\leq$ num_old_keys.

admin_history_kvno The key version number of the kadmin/history principal's key used to encrypt the values in old_keys. If the server library finds that kadmin/history's kvno is different from the value in this field, it returns KADM5_BAD_HIST_KEY.

old_keys The array of the principal's previous passwords, each encrypted in the kadmin/history key. There are num_old_keys elements. Each "password" in the array is itself an array of n_key_data krb5_key_data structures, one for each keysalt type the password was encoded in.

6.2 Policy, osa_policy_ent_t

The policy database stores records of the type osa_policy_ent_t (declared in <kadm5/adb.h>) , which is all of kadm5_policy_ent_t plus necessary bookkeeping information. The records are keyed by the policy name.

```
typedef struct _osa_policy_ent_t {
        char *policy;

        u_int32 pw_min_life;
```

```
        u_int32 pw_max_life;
        u_int32 pw_min_length;
        u_int32 pw_min_classes;
        u_int32 pw_history_num;

        u_int32 refcnt;
} osa_policy_ent_rec, *osa_policy_ent_t;
```

6.3 Kerberos, krb5_db_entry

The Kerberos database stores records of type krb5_db_entry, which is defined in the <k5-int.h> header file. The semantics of each field are defined in the libkdb functional specification.

7 Database Access Methods

7.1 Principal and Policy Databases

This section describes the database abstraction used for the admin policy database; the admin principal database used to be treated in the same manner but is now handled more directly as krb5_tl_data; thus, nothing in this section applies to it any more. Since both databases export equivalent functionality, the API is only described once. The character T is used to represent both "princ" and "policy". The location of the principal database is defined by the configuration parameters given to any of the kadm5_init functions in the server library.

Note that this is *only* a database abstraction. All functional intelligence, such as maintaining policy reference counts or sanity checking, must be implemented above this layer.

Prototypes for the osa functions are supplied in <kadm5/adb.h>. The routines are defined in libkadm5srv.a. They require linking with the Berkely DB library.

7.1.1 Error codes

The database routines use com_err for error codes. The error code table name is "adb" and the offsets are the same as the order presented here. The error table header file is <kadm5/adb_err.h>. Callers of the OSA routines should first call init_adb_err_tbl() to initialize the database table.

OSA_ADB_OK Operation successful.

OSA_ADB_FAILURE General failure.

OSA_ADB_DUP Operation would create a duplicate database entry.

OSA_ADB_NOENT Named entry not in database.

OSA_ADB_BAD_PRINC The krb5_principal structure is invalid.

OSA_ADB_BAD_POLICY The specified policy name is invalid.

OSA_ADB_XDR_FAILURE The principal or policy structure cannot be encoded for storage.

OSA_ADB_BADLOCKMODE Bad lock mode specified.

OSA_ADB_CANTLOCK_DB Cannot lock database, presumably because it is already locked.

OSA_ADB_NOTLOCKED Internal error, database not locked when unlock is called.

OSA_ADB_NOLOCKFILE KADM5 administration database lock file missing.

Database functions can also return system errors. Unless otherwise specified, database functions return OSA_ADB_OK.

7.1.2 Locking

All of the osa_adb functions except open and close lock and unlock the database to prevent concurrency collisions. The overall locking algorithm is as follows:

1. osa_adb_open_T calls osa_adb_init_db to allocate the osa_adb_T_t structure and open the locking file for further use.
2. Each osa_adb functions locks the locking file and opens the appropriate database with osa_adb_open_and_lock, performs its action, and then closes the database and unlocks the locking file with osa_adb_close_and_unlock.
3. osa_adb_close_T calls osa_adb_fini_db to close the locking file and deallocate the db structure.

Functions which modify the database acquire an exclusive lock, others acqure a shared lock. osa_adb_iter_T acquires an exclusive lock for safety but as stated below consequences of modifying the database in the iteration function are undefined.

7.1.3 Function descriptions

```
osa_adb_ret_t
osa_adb_open_T(osa_adb_T_t *db, char *filename);
```

Open the database named filename. Returns OSA_ADB_FAILURE if it cannot open the database.

```
osa_adb_ret_t
osa_adb_close_T(osa_adb_T_t db);
```

Release all shared or exclusive locks (on BOTH databases, since they use the same lock file) and close the database.

It is an error to exit while a permanent lock is held; OSA_ADB_NOLOCKFILE is returned in this case.

```
osa_adb_ret_t osa_adb_get_lock(osa_adb_T_t db, int mode)
```

Acquire a lock on the administration databases; note that both databases are locked simultaneously by a single call. The mode argument can be OSA_ADB_SHARED, OSA_ADB_EXCLUSIVE, or OSA_ADB_PERMANENT. The first two and the third are really disjoint locking semantics and should not be interleaved.

Shared and exclusive locks have the usual semantics, and a program can upgrade a shared lock to an exclusive lock by calling the function again. A reference count of open locks is maintained by this function and osa_adb_release_lock so the functions can be called multiple times; the actual lock is not released until the final osa_adb_release_lock. Note, however, that once a lock is upgraded from shared to exclusive, or from exclusive to permanent, it is not downgraded again until released completely. In other words, get_lock(SHARED), get_lock(EXCLUSIVE), release_lock() leaves the process with an exclusive lock with a reference count of one. An attempt to get a shared or exclusive lock that conflicts with another process results in the OSA_ADB_CANLOCK_DB error code.

This function and osa_adb_release_lock are called automatically as needed by all other osa_adb functions to acquire shared and exclusive locks and so are not normally needed. They can be used explicitly by a program that wants to perform multiple osa_adb functions within the context of a single lock.

Acquiring an OSA_ADB_PERMANENT lock is different. A permanent lock consists of first acquiring an exclusive lock and then *deleting the lock file*. Any subsequent attempt to acquire a lock by a different process will fail with OSA_ADB_NOLOCKFILE instead of OSA_ADB_CANTLOCK_DB (attempts in the same process will "succeed" because only the reference count gets incremented). The lock file is recreated by osa_adb_release_lock when the last pending lock is released.

The purpose of a permanent lock is to absolutely ensure that the database remain locked during non-atomic operations. If the locking process dies while holding a permanent lock, all subsequent osa_adb operations will fail, even through a system reboot. This is useful, for example, for ovsec_adm_import which creates both new database files in a temporary location and renames them into place. If both renames do not fully complete the database will probably be inconsistent and everything should stop working until an administrator can clean it up.

```
osa_adb_ret_t osa_adb_release_lock(osa_adb_T_t db)
```

Releases a shared, exclusive, or permanent lock acquired with osa_adb_get_lock, or just decrements the reference count if multiple locks are held. When a permanent lock is released, the lock file is re-created.

All of a process' shared or exclusive database locks are released when the process terminates. A permanent lock is *not* released when the process exits (although the exclusive lock it begins with obviously is).

```
osa_adb_ret_t
osa_adb_create_T(osa_adb_T_t db, osa_T_ent_t entry);
```

Adds the entry to the database. All fields are defined. Returns OSA_ADB_DUP if it already exists.

```
osa_adb_ret_t
osa_adb_destroy_T(osa_adb_T_t db, osa_T_t name);
```

Removes the named entry from the database. Returns OSA_ADB_NOENT if it does not exist.

```
osa_adb_ret_t
osa_adb_get_T(osa_adb_T_t db, osa_T_t name,
        osa_princ_ent_t *entry);
```

Looks up the named entry in the db, and returns it in *entry in allocated storage that must be freed with osa_adb_free_T. Returns OSA_ADB_NOENT if name does not exist, OSA_ADB_MEM if memory cannot be allocated.

```
osa_adb_ret_t
osadb_adb_put_T(osa_adb_T_t db, osa_T_ent_t entry);
```

Modifies the existing entry named in entry. All fields must be filled in. Returns OSA_DB_NOENT if the named entry does not exist. Note that this cannot be used to rename an entry; rename is implemented by deleting the old name and creating the new one (NOT ATOMIC!).

```
void osa_adb_free_T(osa_T_ent_t);
```

Frees the memory associated with an osa_T_ent_t allocated by osa_adb_get_T.

```
typedef osa_adb_ret_t (*osa_adb_iter_T_func)(void *data,
                                    osa_T_ent_t entry);

osa_adb_ret_t osa_adb_iter_T(osa_adb_T_t db, osa_adb_iter_T_func func,
                    void *data);
```

Iterates over every entry in the database. For each entry ent in the database db, the function (*func)(data, ent) is called. If func returns an error code, osa_adb_iter_T returns an error code. If all invokations of func return OSA_ADB_OK, osa_adb_iter_T returns OSA_ADB_OK. The function func is permitted to access the database, but the consequences of modifying the database during the iteration are undefined.

7.2 Kerberos Database

Kerberos uses the libkdb interface to store krb5_db_entry records. It can be accessed and modified in parallel with the Kerberos server, using functions that are defined inside the KDC and the libkdb.a. The libkdb interface is defined in the libkdb functional specifications.

7.2.1 Initialization and Key Access

Keys stored in the Kerberos database are encrypted in the Kerberos master key. The admin server will therefore have to acquire the key before it can perform any key-changing operations, and will have to decrypt and encrypt the keys retrieved from and placed into the database via krb5_db_get_principal and _put_principal. This section describes the internal admin server API that will be used to perform these functions.

```
krb5_principal master_princ;
krb5_encrypt_block master_encblock;
krb5_keyblock master_keyblock;

void kdc_init_master()
```

kdc_init_master opens the database and acquires the master key. It also sets the global variables master_princ, master_encblock, and master_keyblock:

- master_princ is set to the name of the Kerberos master principal (`K/M@REALM`).
- master_encblock is something I have no idea about.
- master_keyblock is the Kerberos master key

```
krb5_error_code kdb_get_entry_and_key(krb5_principal principal,
                                      krb5_db_entry *entry,
                                      krb5_keyblock *key)
```

kdb_get_entry_and_key retrieves the named principal's entry from the database in entry, and decrypts its key into key. The caller must free entry with krb5_dbm_db_free_principal and free key->contents with free.[2]

```
krb5_error_code kdb_put_entry_pw(krb5_db_entry *entry, char *pw)
```

kdb_put_entry_pw stores entry in the database. All the entry values must already be set; this function does not change any of them except the key. pw, the NULL-terminated password string, is converted to a key using string-to-key with the salt type specified in entry->salt_type.[3]

[2]The caller should also `memset(key->contents, 0, key->length)`. There should be a function krb5_free_keyblock_contents for this, but there is not.

[3]The salt_type should be set based on the command line arguments to the kadmin server (see the "Command Line" section of the functional specification).

8 Admin Principal and Policy Database Implementation

The admin principal and policy databases will each be stored in a single hash table, implemented by the Berkeley 4.4BSD db library. Each record will consist of an entire osa_T_ent_t. The key into the hash table is the entry name (for principals, the ASCII representation of the name). The value is the T entry structure. Since the key and data must be self-contained, with no pointers, the Sun xdr mechanisms will be used to marshal and unmarshal data in the database.

The server in the first release will be single-threaded in that a request will run to completion (or error) before the next will run, but multiple connections will be allowed simultaneously.

9 ACLs, acl_check

The ACL mechanism described in the "Authorization ACLs" section of the functional specifications will be implemented by the acl_check function.

```
enum access_t {
        ACCESS_DENIED = 0,
        ACCESS_OK = 1,
};

enum access_t acl_check(krb5_principal princ, char *priv);
```

The priv argument must be one of "get", "add", "delete", or "modify". acl_check returns 1 if the principal princ has the named privilege, 0 if it does not.

10 Function Details

This section discusses specific design issues for Admin API functions that are not addresed by the functional specifications.

10.1 kadm5_create_principal

If the named principal exists in either the Kerberos or admin principal database, but not both, return KADM5_BAD_DB.

The principal's initial key is not stored in the key history array at creation time.

10.2 kadm5_delete_principal

If the named principal exists in either the Kerberos or admin principal database, but not both, return KADM5_BAD_DB.

10.3 kadm5_modify_principal

If the named principal exists in either the Kerberos or admin principal database, but not both, return KADM5_BAD_DB.

If pw_history_num changes and the new value n is smaller than the current value of num_old_keys, old_keys should end up with the n most recent keys; these are found by counting backwards n elements in old_keys from old_key_next. old_key_nexts should then be reset to 0, the oldest of the saved keys, and num_old_keys set to n, the new actual number of old keys in the array.

10.4 kadm5_chpass_principal, randkey_principal

The algorithm for determining whether a password is in the principal's key history is complicated by the use of the kadmin/history K_h encrypting key.

1. For kadm5_chpass_principal, convert the password to a key using string-to-key and the salt method specified by the command line arguments.
2. If the POLICY bit is set and pw_history_num is not zero, check if the new key is in the history.
   (a) Retrieve the principal's current key and decrypt it with K_M. If it is the same as the new key, return KADM5_PASS_REUSE.
   (b) Retrieve the kadmin/history key K_h and decrypt it with K_M.
   (c) Encrypt the principal's new key in K_h.

(d) If the principal's new key encrypted in K_h is in old_keys, return KADM5_PASS_REUSE.

(e) Encrypt the principal's current key in K_h and store it in old_keys.

(f) Erase the memory containing K_h.

3. Encrypt the principal's new key in K_M and store it in the database.

4. Erase the memory containing K_M.

To store the an encrypted key in old_keys, insert it as the old_key_next element of old_keys, and increment old_key_next by one modulo pw_history_num.

10.5 kadm5_get_principal

If the named principal exists in either the Kerberos or admin principal database, but not both, return KADM5_BAD_DB.