The PWDB Library Guide

Cristian Gafton and Andrew G. Morgan DRAFT v0.54 1996/12/4

This manual documents the *Password database* library (`libpwdb`). This is a library implementing a generic user information database. It was specifically designed to satisfy the needs of **Linux-PAM**.

Contents

1 Introduction

This is a document describing the libpwdb functions (or how they will function). Cristian and Andrew will be modifying this file as we go along. Hopefully it will describe the library when it is finished.

1.1 Motivation

Linux, at the level of the kernel, identifies a user by a 16-bit integer–their *UID*. At the level of a functioning system, the user has other characteristics; a name, group memberships, a home, a number of aliases (email address etc.) and a password.

The mapping of a UID to each of these pieces of personal data is with respect to a database file. Unfortunately, depending on circumstances there is no single location for this database. The most basic Linux system might have a single `/etc/passwd` file, with a single entry for each user, and an `/etc/group` file containing a list of the groups defined on the local system. In shadow aware systems a user might additionally have an `/etc/shadow` entry. In a networked environment, all of the users might be identified by entries in a file located on a remote computer.

While it is possible for a local system to adopt one scheme to satisfy the demands of the POSIX `getpw..` commands, it is problematic for an individual system to simultaneously support a variety of databases to define its users and groups.

It is desirable for applications to be able to obtain user-information in a transparent manner. At the same time certain applications might like to obtain information from a specific database for a given user - these applications might perform the task of maintaining and updating databases for example. The functions of libc, do not currently offer this flexibility. It is for this reason that *libpwdb* has been created. It is intended to provide a flexible and yet simple database management primarily for user and group information under Linux systems. An API that all applications can use.

Should this library become popular, it is anticipated that the database will be extended to include other types of database files.

2 Configuration

2.1 The configuration file

Use of `libpwdb` requires the presence of a correctly formatted configuration file: `/etc/pwdb.conf`. It has the following syntax:

```
#
```

```
# first the list of user databases
#
    user:
            list1
            list2
            list3
            ...
#
# next, the list of group databases
#
    group:
            list1
            list2
            ...
#
# end of file
#
```

Here, `listN` has the following form `name1+name2+name3`. It indicates a collection of databases that are merged to form the record for the user/group. When selecting the default database (PWDB_DEFAULT) the library chooses the list whose first `named` database contains an entry for the requested `user` (`id`). Currently, the `nameN` items are from the following selection: `nis`, `unix`, `radius` and `shadow`.

Newlines are ignored except where they terminate comments; comments are preceded with '`#`' characters.

A simple example `/etc/pwdb.conf` file would be:

```
# This is an example /etc/pwdb.conf file. It defines the
# database information sources for users in this system

# First, we define where user information is stored
# (here, users are listed in /etc/passwd and may have
# supplementary information in the /etc/shadow file)
# Should the user have a shadow entry the combination is
# used

    user:
            shadow + unix
            unix

# Second, we define where the users' groups are listed
# (here the groups are listed in the /etc/group file
# in addition, there is group information to be found
# in the /etc/sgroup file)

    group:
            unix + shadow

#
```

```
# end of file
```

3 Implementation

This will get completed as we develop the library

3.1 The core (public) functions

`libpwdb` offers the following generic interface:

3.1.1 Initialization and termination

- `int pwdb_start(void)`

  Initialize the library for use by the current application Will read the configuration file and publicize the database policies (as they are listed in `/etc/pwdb.conf`) in `const int *pwdb_policy` and `const int *pwdb_group_policy`. The library maintains a count of the number of times it has been `pwdb_start()`ed.

- `int pwdb_end(void)`

  Once the `pwdb_start()`ed count returns to zero, this function closes down the library and free()'s all memory allocated by it. Any attempt to `pwdb_end()` the library more times than it has been `pwdb_start()`ed will cause `PWDB_ABORT` to be returned. *This feature can be used by an application to guarantee the library is shutdown..*

  ```
  while (pwdb_end() == PWDB_SUCCESS);
  ```

3.1.2 Structure management

- `int pwdb_entry_delete(const struct pwdb_entry **e)`

  `free()` the memory associated with the pointer `*e`. Its value will be overwritten with `'`
  `0'`s before the memory is `free()`d. This is reassuring from the point of view of minimizing security problems. This function should be used to liberate a `pwdb_entry` returned by `pwdb_get_entry()`.

- `int pwdb_get_entry(const struct pwdb *p, const char *entry, const struct pwdb_entry **e);`

  Read (and duplicate) an entry from the argument pwdb structure. Should the requested entry not prove to be present, `PWDB_NOT_FOUND` is returned.

- `int pwdb_set_entry(const struct pwdb *p, const char *entry, const void *datum, const int length, int (*compare)(const void *, const void *, int), int (*strvalue)(const void *, char *, int), int max_strval_size))`

  Set an entry in the argument pwdb structure. One can delete an entry from a struct pwdb by calling this function with `NULL` for `datum` and `-1` for `length`.

The two functions passed as arguments in this call are as follows:

`int compare(const void *value1, const void *value2, int length)`
: compare the two entry `values`. They are both of given `length` in `chars`.

`int strvalue(const void *value, char *buffer, int length)`
: Produce a text version of the given entry `value`. The `buffer` is the destination of the output text and the `length` is that of the `*value` in bytes. Note, the buffer is guaranteed by the calling process to be long enough for the output text.

- `int pwdb_delete(const struct pwdb **old)`

  Applications use this to liberate the memory associated with a `pwdb` structure. Following a successful `free()`, `*old` is set to `NULL`.

- `int pwdb_new(const struct pwdb **new, int life_sec)`

  Applications can request a new (blank) `pwdb` structure with this call. Note that it is returned as a `const` structure. This is to prevent the application from altering the structure directly. An application should use the library functions to modify this structure. The `life_sec` time is used to limit the length of time this pwdb structure will be valid for. It is some number of seconds from the present. If `life_sec` is non-zero, the `pwdb` structure will expire in that many seconds. Zero indicates the `pwdb` structure will non expire.

  ```
  int pwdb_merge(const struct pwdb *target, const struct pwdb *source,
                 int overwrite)
  ```

  this function copies the elements of `source` to `target`. If `overwrite` is `PWDB_TRUE` then all elements of `source` that are also in `target` will be overwritten by this call.

- `int pwdb_expire(const struct pwdb *p, int life_sec)`

  This function can shorten the lifetime of a the referenced `struct pwdb`. It computes the expiry time from the present with respect to `life_sec` and if this is *before* the expiration time currently associated with the `*p` structure, it shortens the life of the structure accordingly.

  Note, it is not possible to extend the life of a `pwdb` structure, only to *shorten* it. An argument of `0` or less will result in the immediate expiry of the `pwdb` structure.

- `int pwdb_source(const struct pwdb *old, const pwdb_type *src, const char *class, const char *name, const int id)`

  This function is used to set the source of the indicate `pwdb` structure. The argument `src` is a pointer to a list of `pwdb_type` entries. This list is terminated with a `_PWDB_MAX_TYPES` item. Valid types are listed in the `<security/pwdb_public.h>` file. The remaining arguments are used to initialize the caching facilities.

3.1.3 Database query functions

- `int pwdb_locate(const char *class, const pwdb_type *src, const char *name, const int id, struct pwdb **p)`

  Obtain the entry for a given name and/or id in the indicated database. If `*p` is not `NULL` the database-module may choose to use the information it contains. It is intended that information obtained with this function is merged with that of the original `*p` structure. If `*p` is `NULL`, then a struct pwdb is allocated by the `pwdb_locate()` function in the first module used (and eventually merged with the subsequent modules, depending on the local setup). The `class` is the kind of database we are searching, examples include `user` and `group`.

- `int pwdb_request(const char *class, const pwdb_type *src, const char *entry, struct pwdb **p)`

  This function will further query the database, for an entry that may depend on the entries already present in the `*p` structure. The `entry`s that can be appended to the existing `*p` structure are defined for the `class` of database. For example, one may request the *"groupids"* `entry` from `"group"` class, which will search the group database for the list of groups to which a given user belongs. The name of the user is passed as an entry in the preexisting `pwdb` structure.

- `int pwdb_replace(const char *class, const pwdb_type *src, const char *name, const int id, struct pwdb **p)`

  Add/replace the entry for a name and/or id in the indicated list of databases. The fields for the new database entries are taken from the `p` argument. The `src` argument is the list of entries that will be updated.

- `int pwdb_remove(const char *class, const pwdb_type *src, const char *name, const int id, struct pwdb **p)`

  Remove the entry for the indicated `name` and/or `id` in the indicated database. If not `NULL` the remove function may obtain access information from the `p` argument.

  Note, this function only acts on the specified `class` of database. If reference to a `name` or `id` is present in another `class` of database, then it is the responsibility of the application to purge these databases too. The `pwdb_remove()` function is not sufficiently powerful to follow up on cross-references.

  As an example of the above concern, consider the removal of `name` "joe" from the `"user"` database. The `"group"` database entries that refer to "joe" as a group-member are unaffected by this `pwdb_remove()` request. Instead, it is the responsibility of the calling application to search for such entries and systematically `pwdb_remove()` them.

- `int pwdb_support(const char *class, const pwdb_type *src, const char *entry_name)`

  Indicate whether the given entry name is supported by the given database(s). `PWDB_SUCCESS` indicates yes. Anything else is NO (or something more specific, could be "yes, if you supply a pass_phrase" for example).

- `int pwdb_flags(const char *class, const pwdb_type *db, pwdb_flag *flag_p)`

In order to know in advance whether a process is able to read/modify a specified database, this command is provided by each module. The input arguments are the `class` of database (`user`, `group` etc.), `db` (the database(s) we are going to use) and some storage space for the returned flags.

Valid return flags which can be logically OR'd together are:

`PWDB_F_NOACCESS`
: insufficient privilege to access database

`PWDB_F_NOUPDATE`
: insufficient privilege to alter an entry

`PWDB_F_PASS_PHRASE`
: to access the database, the process must supply a "pass_phrase" entry with a preallocated pwdb structure (use pwdb_new() call)

..and..
: *more flags are likely to be added*

To establish if a given flag is set the following macro is supplied:

- `pwdb_on(flag, PWDB_F_XXX)`

  it returns TRUE(1) or FALSE(0).

3.1.4 Diagnostics

- `const char *pwdb_strerror(int pwdb_error)`

  return a textual description of the indicated return value.

  Valid return codes are:

`PWDB_SUCCESS`
: task completed successfully

`PWDB_BAD_REQUEST`
: request not recognized

`PWDB_TOO_WEAK`
: insufficient privilege for operation

`PWDB_ABORT`
: internal failure - seek help

`PWDB_BLOCKED`
: another process has locked resource

`PWDB_MALLOC`
: insufficient memory for operation

`PWDB_NOT_FOUND`
: requested item was not found

`PWDB_PASS_PHRASE_REQD`

pass_phrase needed to satisfy request *here the application should supply a pwdb structure with a "pass_phrase" entry and call the function again*

`PWDB_CONF_ERR`

there is a problem with the PWDB_CONF file.

`PWDB_EXPIRED`

the referenced pwdb structure has expired it is no longer valid and should be deleted.

`PWDB_UNSUPPORTED`

this function is not supported by some module (not supported means also unimplemented, for a while...)

`PWDB_TIMEOUT`

a timeout occured while performing the function. Presently could show up only when using RADIUS interface.

3.1.5 Types of database

The pwdb_type of database a request is associated with is given by one of the following values:

`PWDB_DEFAULT`

no database indicated, use configured list

`PWDB_UNIX`

generic /etc/passwd and /etc/group files

`PWDB_SHADOW`

/etc/shadow and /etc/gshadow *Intended to supplement other databases*

`PWDB_NIS`

Use NIS server

`PWDB_RADIUS`

Use RADIUS server

- `const char *pwdb_db_name(pwdb_type src)`

  return a character representation of the database functions

3.2 Functions supplied by database modules

Each module must provide 7 functions to the generic pwdb interface. They are registered with the generic interface via a structure of the following form:

```
struct _pwdb_module {
    pwdb_type type;                 /* type of database (code) */
    const char *name;               /* type of database (text) */
    const char *class;              /* class of database (user/group) */

    /* FUNCTIONS: used to access the relevant database */

    int (*locate)(const char *name, const int id, const struct pwdb **p);
    int (*request)(const char *entry_name, const struct pwdb **p);
    int (*replace)(const char *name, const int id, const struct pwdb **p);
    int (*delete)(const char *name, const int id, const struct pwdb **p);
    int (*support)(const char *entry_name);
    int (*flags)(pwdb_flag *flags);
    int (*cleanup)(int code);
};
```

For the functions above taking a `name` and an `id` entry, the application may choose to leave one unspecified with the following defaults:

`PWDB_NAME_UNKNOWN`

just look at the id field

`PWDB_ID_UNKNOWN`

just look at the name field

In the case that the application supplies neither the `name` or the `id`, the module functions will try to obtain the relevant information from the argument `pwdb` structure.

It is legal for both the `name` and `id` to be specified. In this case they must both match an entry in the database to satisfy one of the above function calls. If both values are supplied and there is no entry in the database which matches them, `PWDB_BAD_REQUEST` is returned.

The structure is registered via an entry in the modules list (see pwdb_module.c).

3.3 Standard entries in pwdb structures

The following are standard entries in the pwdb structure. They can be read/written with calls to `pwdb_g/set_entry`.

First, we consider the `"user"` class of databases. For these, two entries are mandatory. They correspond to the name of the user and the user's uid.

`user`

character string; the user's login id.

`uid`

uid_t; the user's *user-id*.

The next entries are named by convention. Where possible new database functions should map these entries into their corresponding fields. These entries correspond to the entries in the `/etc/passwd` file.

`passwd`

character string; the *encrypted* password for the user.

`defer_pass`

This entry is intended to take care of situations that the normal `passwd` field is not used for the password. The `defer_pass` entry contains a character string that has typically two functions:

- it indicates that the password is to be stored in the database that sets this entry.
- it serves as a substitution string for databases that would normally contain the password entry.

For example, for a `unix+shadow` setup, `defer_pass` would have the value "`x`". The `unix` (no `shadow`) value for this entry is "U" which implies that the `passwd` field came from the user's entry in the `/etc/passwd` file.

`gid`

gid_t; the user's principal *group-id*.

`group`

character string; naming the user's principal group.

`gecos`

character string; giving a more complete name for the user. It is conventional for this field to contain office and other information concerning the real-world identity of the user.

`dir`

character string; the home directory of the user.

`shell`

character string; the shell that the user prefers to use.

These entries correspond to the entries in the `/etc/group` file *in addition to the user and gid entries above*. They can be `pwdb_request()`d from the "group" class of databases.

`groups`

character string; listing the group memberships of the user. The field separators are commas – no spaces.

`groupids`

array of `gid_t`; an array containing the group id's of the user in numerical form.

The following are intended to correspond to `/etc/shadow` entries.

`last_change`

long integer; day of last change of password

`min_change`

long integer; minimum number of days between password changes

`max_change`

integer; maximum number of days between password changes

`warn_change`

long integer; number of days to warn user to change their password before it expires

`defer_change`

long integer; number of days after a user's password has expired before the user is denied access

`expire`

long integer; day the user's account expires

The following is the entry used to supply a *clear-text* password for access to the database.

`pass_phrase`

character string; this is the password required to access the user's record in a database

When integrating another database format the implementor is strongly encouraged to try to reuse the entries above to the extent they are appropriate. Should there be an absent entry in any database, the database management functions should be able to supply a reasonable default but only when updating its database.

3.4 Helper functions

3.4.1 Public functions

- `char *_pwdb_delete_string(char *s)`

  overwrite the string 's' and return `NULL`. usage:

  `old_ptr = _pwdb_delete(old_ptr);`

- `char *_pwdb_dup_string(const char *s)`

  `malloc()` a copy of the string 's'. Return its address or `NULL` if `s == NULL` or on error.

  this memory will *not* be `free()`'d by a call to `pwdb_end()`.

- `void pwdb_print_pwdb_struct(const struct pwdb *p)`

  Dump the contents of `*p` to the `stderr`. Useful for debugging.

3.4.2 Private functions

- `static struct pwdb *_pwd_check(const struct pwdb *p)`

  Establish if the pwdb structure was allocated by the library This library should not honor requests from elsewhere. return the local version (non-const) of this structure or NULL on error.

  this function is strictly designed for the use of the generic code. Both modules and applications should never need to call it.

3.5 Typical usage

Here is a skeleton usage for a login type program.

```
pwdb_start();
pwdb_locate("user", PWDB_DEFAULT, username, PWDB_ID_UNKNOWN, &pw);
pwdb_request_group("group", PWDB_DEFAULT, "groupids", &pw);
pwdb_get_entry(pw, "uid", &e1);
pwdb_get_entry(pw, "gid", &e2);
pwdb_get_entry(pw, "groupids", &e3);
pwdb_end();
```

4 Supported modules

PLEASE NOTE. Currently few group functions have been implemented.

4.1 UNIX module

This section documents the current state of the UNIX module. From the point of view of the application, it is named "unix" and has the `pwdb_type` `PWDB_UNIX`.

Entries supported by the "user" class of the UNIX module are as follows:

`user`

- username

`uid`

- user-id

`gid`

- group-id

`passwd`

- encrypted password

`defer_pass`

- "U" unless set by other database

`gecos`

- user information

`dir`

- home directory

`shell`

- shell executable

Entries supported by the "group" class of the UNIX module are as follows:

`group`

- username

`gid`

- group-id

`passwd`

- encrypted password

`users`

- text list of user names separated by commas

The `pwdb_request()` function call is only supported for the "group" class. The two entries that may be requested are:

`groups`

- text list of group names separated by commas

`groupids`

- array of `gid_t` values that contain the numerical form of the "groups" entry.

Note, for such requests to be honored the name of the user should be contained in the `pwdb **` argument prior to the `pwdb_request()` call.

4.2 Shadow module

This section documents the current implementation of the *shadow* database module.

The *shadow* module does not make any use of the `id` parameter. Since the shadow database does not contain such an entry, the only way of identifying a user is with a `name` argument. However, an `id` based lookup is possible, if the `shadow` database is searched after a database that contains a user-uid mapping. In this case the “`user`” entry in the partially built `pwdb` structure is used to locate the appropriate entry in the shadow file.

The shadow module is designed to work in conjunction with a database that provides the standard `user-uid` mapping. It should be noted that it does not provide sufficient information to support a user login session.

Entries supported by the ”user” class of the shadow module are:

`user`

- username

`passwd`

- encrypted password

`last_change`

- date password was last changed

`min_change`

- minimum period before password can be changed

`max_change`

- lifetime of current password

`warn_change`

- number of days prior to expiry that the user should be warned

`defer_change`

- grace period before password is finally invalid

`expire`

- date account expires

Entries supported by the ”group” class of the shadow module are:

`group`

- groupname

`passwd`

- encrypted group password

`users`

- text list of user names separated by commas (members of the group)

`admins`

- text list of user names separated by commas (administrators of the group)

4.3 NIS module

- user removal is not possible. (lack of documentation)
- user creation is not possible (lack of documentation)

4.4 RADIUS module

The RADIUS module is acting just as a user validation mechanism. The official Livingston radiusd 2.0 is supported, but in order to take advantage of the all information and auth tokens the radius server can provide, a session PAM module should be written, and be stacked over pam_unix.

The following should be taken into consideration when writing applications to authenticate to a radius server:

- the pwdb group functions are not supported (RADIUS does not have this concept)
- Other than checking for username/password pairs, the radius module can not be used alone with the stock radiusd server to handle the user login without a suitable RADIUS client (for example there is no way to get an UID for a user). However, with few hacks to the radius server and proper modification of the `/etc/raddb/dictionary` file this module uses, a NIS-like environment could be achieved. Full documentation on how to achieve this will be added later.
- All radius module functions that access the remote server require the presence of the "pass_phrase" pwdb entry which contains the user password in *clear text*. The radius module will destroy this information as soon as it can dispose of it, so an application can assume that after a call to a function in the radius module which returned with `PWDB_SUCCESS`, the pass_phrase entry is wiped out.
- When updating a RADIUS user password, one should supply **both** `pass_phrase` and `passwd` entries in *clear text*. By convention, the `pass_phrase` contains the old password (which is required by the RADIUS server to authenticate the user) and the `passwd` entry contains the new password. One should be very careful about this issue, as some other modules used before RADIUS authentication may set the value of `passwd` entry, and the application should make sure that the clear text password is passed to the RADIUS module for changing password. The RADIUS module will wipe out both `pass_phrase` and `password` entries - thus the application can not rely on validity of any of those entries in the pwdb structure after a call to the update function of the RADIUS module.

  Note that the RADIUS server must permit changing of the passwords user passwords. IF the RADIUS does not accept changing the user passwords, a PWDB_TIMEOUT will occur.
- The password update is the **only** function supported by the RADIUS update function.

Entries supported by this database are set according to the definitions from `/etc/raddb/dictionary` file. Three entries have a special meaning when calling the RADIUS functions: `user`, `passwd` and `pass_phrase`. The `passwd` and `pass_phrase` will be wiped out by the RADIUS functions as soon as the module can dispose them. The reponse from the RADIUS server is processed and entries are set in the `pwdb` structure according to the names from the dictionary file.

A sample `RADIUS dictionary` entry list is provided here - valid for Livingston RADIUSD 2.0:

```
#--------------------------------------------------------------------------
#
# @(#)dictionary        1.3 10/1/96  Copyright 1991 Livingston Enterprises Inc
#
#--------------------------------------------------------------------------
#
#       This file contains dictionary translations for parsing
#       requests and generating responses.  All transactions are
#       composed of Attribute/Value Pairs.  The value of each attribute
#       is specified as one of 4 data types.  Valid data types are:
#
#       string - 0-253 octets
#       ipaddr - 4 octets in network byte order
#       integer - 32 bit value in big endian order (high byte first)
#       date - 32 bit value in big endian order - seconds since
#                                       00:00:00 GMT,  Jan.  1,  1970
#

ATTRIBUTE       User-Name               1       string
ATTRIBUTE       Password                2       string
ATTRIBUTE       CHAP-Password           3       string
ATTRIBUTE       NAS-IP-Address          4       ipaddr
ATTRIBUTE       NAS-Port                5       integer
ATTRIBUTE       Service-Type            6       integer
ATTRIBUTE       Framed-Protocol         7       integer
ATTRIBUTE       Framed-IP-Address       8       ipaddr
ATTRIBUTE       Framed-IP-Netmask       9       ipaddr
ATTRIBUTE       Framed-Routing          10      integer
ATTRIBUTE       Filter-Id               11      string
ATTRIBUTE       Framed-MTU              12      integer
ATTRIBUTE       Framed-Compression      13      integer
ATTRIBUTE       Login-IP-Host           14      ipaddr
ATTRIBUTE       Login-Service           15      integer
ATTRIBUTE       Login-TCP-Port          16      integer
ATTRIBUTE       Reply-Message           18      string
ATTRIBUTE       Callback-Number         19      string
ATTRIBUTE       Callback-Id             20      string
ATTRIBUTE       Framed-Route            22      string
ATTRIBUTE       Framed-IPX-Network      23      ipaddr
ATTRIBUTE       State                   24      string
ATTRIBUTE       Session-Timeout         27      integer
ATTRIBUTE       Idle-Timeout            28      integer
ATTRIBUTE       Termination-Action      29      integer
ATTRIBUTE       Called-Station-Id       30      string
ATTRIBUTE       Calling-Station-Id      31      string
ATTRIBUTE       Acct-Status-Type        40      integer
ATTRIBUTE       Acct-Delay-Time         41      integer
```

```
ATTRIBUTE       Acct-Input-Octets       42      integer
ATTRIBUTE       Acct-Output-Octets      43      integer
ATTRIBUTE       Acct-Session-Id         44      string
ATTRIBUTE       Acct-Authentic          45      integer
ATTRIBUTE       Acct-Session-Time       46      integer
ATTRIBUTE       Acct-Terminate-Cause    49      integer
ATTRIBUTE       NAS-Port-Type           61      integer
ATTRIBUTE       Port-Limit              62      integer
```

5 Proposals/unresolved issues

There have been proposals to offer other flavors of database access through this library. It is hoped that the library will prove flexible enough to support this need. But such support is not currently the primary goal. In other words Cristian and Andrew will not be doing it in the short term.. so unless you want to volunteer, please don't hold your breath! :)

6 References

7 Acknowledgments

This document was written by Cristian Gafton and Andrew G. Morgan.

Thanks are also due to the following for their many helpful comments and suggestions: Nalin Dahyabhai, Torsten Duwe, Elliot Lee, Marek Michalkiewicz, Aleph One and Roland Schemers.

More names to be added..