The Linux-PAM Module Writers' Guide

Andrew G. Morgan <morgan@kernel.org>
Thorsten Kukuk <kukuk@thkukuk.de>

The Linux-PAM Module Writers' Guide

by Andrew G. Morgan and Thorsten Kukuk

Version 1.1.2, 31. August 2010

Abstract

This manual documents what a programmer needs to know in order to write a module that conforms to the *Linux-PAM* standard.It also discusses some security issues from the point of view of the module programmer.

1. Introduction 1
1.1. Description 1
1.2. Synopsis 1
2. What can be expected by the module 2
2.1. Getting and setting *PAM_ITEM*s and *data* 2
2.1.1. Set module internal data 2
2.1.2. Get module internal data 3
2.1.3. Setting PAM items 3
2.1.4. Getting PAM items 5
2.1.5. Get user name 7
2.1.6. The conversation function 7
2.1.7. Set or change PAM environment variable 9
2.1.8. Get a PAM environment variable 10
2.1.9. Getting the PAM environment 10
2.2. Other functions provided by `libpam` 11
2.2.1. Strings describing PAM error codes 11
2.2.2. Request a delay on failure 11
3. What is expected of a module 13
3.1. Overview 13
3.1.1. Functional independence 13
3.1.2. Minimizing administration problems 13
3.1.3. Arguments supplied to the module 13
3.2. Authentication management 14
3.2.1. Service function for user authentication 14
3.2.2. Service function to alter credentials 14
3.3. Account management 15
3.3.1. Service function for account management 15
3.4. Session management 16
3.4.1. Service function to start session management 16
3.4.2. Service function to terminate session management 17
3.5. Authentication token management 17
3.5.1. Service function to alter authentication token 17
4. Generic optional arguments 19
5. Programming notes 20
5.1. Security issues for module creation 20
5.1.1. Sufficient resources 20
5.1.2. Who´s who? 20
5.1.3. Using the conversation function 20
5.1.4. Authentication tokens 21
5.2. Use of syslog(3) 21
5.3. Modules that require system libraries 22
6. An example module 23
7. See also 24
8. Author/acknowledgments 25
9. Copyright information for this document 26

Chapter 1. Introduction

1.1. Description

Linux-PAM (Pluggable Authentication Modules for Linux) is a library that enables the local system administrator to choose how individual applications authenticate users. For an overview of the *Linux-PAM* library see the *Linux-PAM System Administrators' Guide*.

A *Linux-PAM* module is a single executable binary file that can be loaded by the *Linux-PAM* interface library. This PAM library is configured locally with a system file, `/etc/pam.conf`, to authenticate a user request via the locally available authentication modules. The modules themselves will usually be located in the directory `/lib/security` (or `/lib64/security`, depending on the architecture) and take the form of dynamically loadable object files (see dlopen(3). Alternatively, the modules can be statically linked into the *Linux-PAM* library; this is mostly to allow *Linux-PAM* to be used on platforms without dynamic linking available, but this is a *deprecated* functionality. It is the *Linux-PAM* interface that is called by an application and it is the responsibility of the library to locate, load and call the appropriate functions in a *Linux-PAM*-module.

Except for the immediate purpose of interacting with the user (entering a password etc..) the module should never call the application directly. This exception requires a "conversation mechanism" which is documented below.

1.2. Synopsis

```
#include <security/pam_modules.h>

gcc -fPIC -c pam_module.c
gcc -shared -o pam_module.so pam_module.o -lpam
```

Chapter 2. What can be expected by the module

Here we list the interface that the conventions that all *Linux-PAM* modules must adhere to.

2.1. Getting and setting *PAM_ITEM*s and *data*

First, we cover what the module should expect from the *Linux-PAM* library and a *Linux-PAM* aware application. Essentially this is the `libpam.*` library.

2.1.1. Set module internal data

#include <security/pam_modules.h>

```
int pam_set_data(pamh, module_data_name, data, (*cleanup)(pam_handle_t
*pamh, void *data, int error_status));

pam_handle_t *pamh;
const char *module_data_name;
void *data;
void (*cleanup)(pam_handle_t *pamh, void *data, int error_status);
```

2.1.1.1. DESCRIPTION

The `pam_set_data` function associates a pointer to an object with the (hopefully) unique string *module_data_name* in the PAM context specified by the *pamh* argument.

PAM modules may be dynamically loadable objects. In general such files should not contain *static* variables. This function and its counterpart pam_get_data(3), provide a mechanism for a module to associate some data with the handle *pamh*. Typically a module will call the `pam_set_data` function to register some data under a (hopefully) unique *module_data_name*. The data is available for use by other modules too but *not* by an application. Since this functions stores only a pointer to the *data*, the module should not modify or free the content of it.

The function `cleanup()` is associated with the *data* and, if non-NULL, it is called when this data is over-written or following a call to pam_end(3).

The *error_status* argument is used to indicate to the module the sort of action it is to take in cleaning this data item. As an example, Kerberos creates a ticket file during the authentication phase, this file might be associated with a data item. When pam_end(3) is called by the module, the *error_status* carries the return value of the pam_authenticate(3) or other *libpam* function as appropriate. Based on this value the Kerberos module may choose to delete the ticket file (*authentication failure*) or leave it in place.

The *error_status* may have been logically OR'd with either of the following two values:

PAM_DATA_REPLACE When a data item is being replaced (through a second call to `pam_set_data`) this mask is used. Otherwise, the call is assumed to be from pam_end(3).

PAM_DATA_SILENT Which indicates that the process would prefer to perform the `cleanup()` quietly. That is, discourages logging/messages to the user. It is generally used to indicate that the current closing of the library is in a fork(2)ed process, and that the parent will take care of cleaning up things that exist outside of the current process space (files etc.).

2.1.1.2. RETURN VALUES

PAM_BUF_ERR Memory buffer error.

PAM_SUCCESS Data was successful stored.

PAM_SYSTEM_ERR A NULL pointer was submitted as PAM handle or the function was called by an application.

2.1.2. Get module internal data

#include <security/pam_modules.h>

```
int pam_get_data(pamh, module_data_name, data);

const pam_handle_t *pamh;
const char *module_data_name;
const void **data;
```

2.1.2.1. DESCRIPTION

This function together with the pam_set_data(3) function is useful to manage module-specific data meaningful only to the calling PAM module.

The `pam_get_data` function looks up the object associated with the (hopefully) unique string *module_data_name* in the PAM context specified by the *pamh* argument. A successful call to `pam_get_data` will result in *data* pointing to the object. Note, this data is *not* a copy and should be treated as *constant* by the module.

2.1.2.2. RETURN VALUES

PAM_SUCCESS Data was successful retrieved.

PAM_SYSTEM_ERR A NULL pointer was submitted as PAM handle or the function was called by an application.

PAM_NO_MODULE_DATA No module specific data is present.

2.1.3. Setting PAM items

#include <security/pam_modules.h>

```
int pam_set_item(pamh, item_type, item);

pam_handle_t *pamh;
int item_type;
const void *item;
```

2.1.3.1. DESCRIPTION

The `pam_set_item` function allows applications and PAM service modules to access and to update PAM information of *item_type*. For this a copy of the object pointed to by the *item* argument is created. The following *item_type*s are supported:

PAM_SERVICE The service name (which identifies that PAM stack that the PAM functions will use to authenticate the program).

PAM_USER The username of the entity under whose identity service will be given. That is, following authentication, *PAM_USER* identifies the local entity that gets to use the service. Note, this value can be mapped from something (eg., "anonymous") to something else (eg. "guest119") by any module in the PAM stack. As such an application should consult the value of *PAM_USER* after each call to a PAM function.

PAM_USER_PROMPT The string used when prompting for a user's name. The default value for this string is a localized version of "login: ".

PAM_TTY The terminal name prefixed by `/dev/` for device files. In the past, graphical X-based applications used to store the *$DISPLAY* variable here, but with the introduction of *PAM_XDISPLAY* this usage is deprecated.

PAM_RUSER The requesting user name: local name for a locally requesting user or a remote user name for a remote requesting user.

Generally an application or module will attempt to supply the value that is most strongly authenticated (a local account before a remote one. The level of trust in this value is embodied in the actual authentication stack associated with the application, so it is ultimately at the discretion of the system administrator.

PAM_RUSER@PAM_RHOST should always identify the requesting user. In some cases, *PAM_RUSER* may be NULL. In such situations, it is unclear who the requesting entity is.

PAM_RHOST The requesting hostname (the hostname of the machine from which the *PAM_RUSER* entity is requesting service). That is *PAM_RUSER@PAM_RHOST* does identify the requesting user. In some applications, *PAM_RHOST* may be NULL. In such situations, it is unclear where the authentication request is originating from.

PAM_AUTHTOK The authentication token (often a password). This token should be ignored by all module functions besides pam_sm_authenticate(3) and pam_sm_chauthtok(3). In the former function it is used to pass the most recent authentication token from one stacked module to another. In the latter function the token is used for another purpose. It contains the currently active authentication token.

PAM_OLDAUTHTOK The old authentication token. This token should be ignored by all module functions except pam_sm_chauthtok(3).

PAM_CONV The pam_conv structure. See pam_conv(3).

The following additional items are specific to Linux-PAM and should not be used in portable applications:

PAM_FAIL_DELAY A function pointer to redirect centrally managed failure delays. See pam_fail_delay(3).

PAM_XDISPLAY The name of the X display. For graphical, X-based applications the value for this item should be the *$DISPLAY* variable. This value may be used independently of *PAM_TTY* for passing the name of the display.

PAM_XAUTHDATA A pointer to a structure containing the X authentication data required to make a connection to the display specified by *PAM_XDISPLAY*, if such information is necessary. See pam_xauth_data(3).

PAM_AUTHTOK_TYPE The default action is for the module to use the following prompts when requesting passwords: "New UNIX password: " and "Retype UNIX password: ". The example word *UNIX* can be replaced with this item, by default it is empty. This item is used by pam_get_authtok(3).

For all *item_type*s, other than PAM_CONV and PAM_FAIL_DELAY, *item* is a pointer to a <NUL> terminated character string. In the case of PAM_CONV, *item* points to an initialized *pam_conv* structure. In the case of PAM_FAIL_DELAY, *item* is a function pointer: `void (*delay_fn)(int retval, unsigned usec_delay, void *appdata_ptr)`

Both, PAM_AUTHTOK and PAM_OLDAUTHTOK, will be reset before returning to the application. Which means an application is not able to access the authentication tokens.

2.1.3.2. RETURN VALUES

PAM_BAD_ITEM The application attempted to set an undefined or inaccessible item.

PAM_BUF_ERR Memory buffer error.

PAM_SUCCESS Data was successful updated.

PAM_SYSTEM_ERR The *pam_handle_t* passed as first argument was invalid.

2.1.4. Getting PAM items

#include <security/pam_modules.h>

```
int pam_get_item(pamh, item_type, item);

const pam_handle_t *pamh;
int item_type;
const void **item;
```

2.1.4.1. DESCRIPTION

The `pam_get_item` function allows applications and PAM service modules to access and retrieve PAM information of *item_type*. Upon successful return, *item* contains a pointer to the value of the corresponding item. Note, this is a pointer to the *actual* data and should *not* be *free()*'ed or over-written! The following values are supported for *item_type*:

PAM_SERVICE The service name (which identifies that PAM stack that the PAM functions will use to authenticate the program).

PAM_USER The username of the entity under whose identity service will be given. That is, following authentication, *PAM_USER* identifies the local entity that gets to use the service. Note, this value can be mapped from something (eg., "anonymous") to something else (eg. "guest119") by any module in the PAM stack. As such an application should consult the value of *PAM_USER* after each call to a PAM function.

PAM_USER_PROMPT The string used when prompting for a user's name. The default value for this string is a localized version of "login: ".

PAM_TTY The terminal name prefixed by `/dev/` for device files. In the past, graphical X-based applications used to store the *$DISPLAY* variable here, but with the introduction of *PAM_XDISPLAY* this usage is deprecated.

PAM_RUSER — The requesting user name: local name for a locally requesting user or a remote user name for a remote requesting user.

Generally an application or module will attempt to supply the value that is most strongly authenticated (a local account before a remote one. The level of trust in this value is embodied in the actual authentication stack associated with the application, so it is ultimately at the discretion of the system administrator.

PAM_RUSER@PAM_RHOST should always identify the requesting user. In some cases, *PAM_RUSER* may be NULL. In such situations, it is unclear who the requesting entity is.

PAM_RHOST — The requesting hostname (the hostname of the machine from which the *PAM_RUSER* entity is requesting service). That is *PAM_RUSER@PAM_RHOST* does identify the requesting user. In some applications, *PAM_RHOST* may be NULL. In such situations, it is unclear where the authentication request is originating from.

PAM_AUTHTOK — The authentication token (often a password). This token should be ignored by all module functions besides pam_sm_authenticate(3) and pam_sm_chauthtok(3). In the former function it is used to pass the most recent authentication token from one stacked module to another. In the latter function the token is used for another purpose. It contains the currently active authentication token.

PAM_OLDAUTHTOK — The old authentication token. This token should be ignored by all module functions except pam_sm_chauthtok(3).

PAM_CONV — The pam_conv structure. See pam_conv(3).

The following additional items are specific to Linux-PAM and should not be used in portable applications:

PAM_FAIL_DELAY — A function pointer to redirect centrally managed failure delays. See pam_fail_delay(3).

PAM_XDISPLAY — The name of the X display. For graphical, X-based applications the value for this item should be the *$DISPLAY* variable. This value may be used independently of *PAM_TTY* for passing the name of the display.

PAM_XAUTHDATA — A pointer to a structure containing the X authentication data required to make a connection to the display specified by *PAM_XDISPLAY*, if such information is necessary. See pam_xauth_data(3).

PAM_AUTHTOK_TYPE — The default action is for the module to use the following prompts when requesting passwords: "New UNIX password: " and "Retype UNIX password: ". The example word *UNIX* can be replaced with this item, by default it is empty. This item is used by pam_get_authtok(3).

If a service module wishes to obtain the name of the user, it should not use this function, but instead perform a call to pam_get_user(3).

Only a service module is privileged to read the authentication tokens, PAM_AUTHTOK and PAM_OLDAUTHTOK.

2.1.4.2. RETURN VALUES

PAM_BAD_ITEM — The application attempted to set an undefined or inaccessible item.

PAM_BUF_ERR Memory buffer error.

PAM_PERM_DENIED The value of *item* was NULL.

PAM_SUCCESS Data was successful updated.

PAM_SYSTEM_ERR The *pam_handle_t* passed as first argument was invalid.

2.1.5. Get user name

#include <security/pam_modules.h>

```
int pam_get_user(pamh, user, prompt);

const pam_handle_t *pamh;
const char **user;
const char *prompt;
```

2.1.5.1. DESCRIPTION

The `pam_get_user` function returns the name of the user specified by pam_start(3). If no user was specified it returns what `pam_get_item (pamh, PAM_USER, ... );` would have returned. If this is NULL it obtains the username via the pam_conv(3) mechanism, it prompts the user with the first non-NULL string in the following list:

- The *prompt* argument passed to the function.
- What is returned by pam_get_item (pamh, PAM_USER_PROMPT, ...);
- The default prompt: "login: "

By whatever means the username is obtained, a pointer to it is returned as the contents of **user*. Note, this memory should *not* be *free()*'d or *modified* by the module.

This function sets the *PAM_USER* item associated with the pam_set_item(3) and pam_get_item(3) functions.

2.1.5.2. RETURN VALUES

PAM_SUCCESS User name was successful retrieved.

PAM_SYSTEM_ERR A NULL pointer was submitted.

PAM_CONV_ERR The conversation method supplied by the application failed to obtain the username.

PAM_BUF_ERR Memory buffer error.

PAM_ABORT Error resuming an old conversation.

PAM_CONV_AGAIN The conversation method supplied by the application is waiting for an event.

2.1.6. The conversation function

#include <security/pam_appl.h>

```
struct pam_message {
```

```
    int msg_style;
    const char *msg;
};

struct pam_response {
    char *resp;
    int resp_retcode;
};

struct pam_conv {
    int (*conv)(int num_msg, const struct pam_message **msg,
                struct pam_response **resp, void *appdata_ptr);
    void *appdata_ptr;
};
```

2.1.6.1. DESCRIPTION

The PAM library uses an application-defined callback to allow a direct communication between a loaded module and the application. This callback is specified by the *struct pam_conv* passed to pam_start(3) at the start of the transaction.

When a module calls the referenced conv() function, the argument *appdata_ptr* is set to the second element of this structure.

The other arguments of a call to conv() concern the information exchanged by module and application. That is to say, *num_msg* holds the length of the array of pointers, *msg*. After a successful return, the pointer *resp* points to an array of pam_response structures, holding the application supplied text. The *resp_retcode* member of this struct is unused and should be set to zero. It is the caller's responsibility to release both, this array and the responses themselves, using free(3). Note, ***resp* is a *struct pam_response* array and not an array of pointers.

The number of responses is always equal to the *num_msg* conversation function argument. This does require that the response array is free(3)'d after every call to the conversation function. The index of the responses corresponds directly to the prompt index in the pam_message array.

On failure, the conversation function should release any resources it has allocated, and return one of the predefined PAM error codes.

Each message can have one of four types, specified by the *msg_style* member of *struct pam_message*:

PAM_PROMPT_ECHO_OFF Obtain a string without echoing any text.

PAM_PROMPT_ECHO_ON Obtain a string whilst echoing text.

PAM_ERROR_MSG Display an error message.

PAM_TEXT_INFO Display some text.

The point of having an array of messages is that it becomes possible to pass a number of things to the application in a single call from the module. It can also be convenient for the application that related things come at once: a windows based application can then present a single form with many messages/prompts on at once.

In passing, it is worth noting that there is a discrepancy between the way Linux-PAM handles the const struct pam_message **msg conversation function argument and the way that Solaris' PAM (and

derivatives, known to include HP/UX, are there others?) does. Linux-PAM interprets the msg argument as entirely equivalent to the following prototype const struct pam_message *msg[] (which, in spirit, is consistent with the commonly used prototypes for argv argument to the familiar main() function: char **argv; and char *argv[]). Said another way Linux-PAM interprets the msg argument as a pointer to an array of num_msg read only 'struct pam_message' pointers. Solaris' PAM implementation interprets this argument as a pointer to a pointer to an array of num_msg pam_message structures. Fortunately, perhaps, for most module/application developers when num_msg has a value of one these two definitions are entirely equivalent. Unfortunately, casually raising this number to two has led to unanticipated compatibility problems.

For what its worth the two known module writer work-arounds for trying to maintain source level compatibility with both PAM implementations are:

- never call the conversation function with num_msg greater than one.
- set up msg as doubly referenced so both types of conversation function can find the messages. That is, make

```
msg[n] = & (( *msg )[n])
```

2.1.6.2. RETURN VALUES

PAM_BUF_ERR Memory buffer error.

PAM_CONV_ERR Conversation failure. The application should not set **resp*.

PAM_SUCCESS Success.

2.1.7. Set or change PAM environment variable

#include <security/pam_appl.h>

```
int pam_putenv(pamh, name_value);

pam_handle_t *pamh;
const char *name_value;
```

2.1.7.1. DESCRIPTION

The `pam_putenv` function is used to add or change the value of PAM environment variables as associated with the *pamh* handle.

The *pamh* argument is an authentication handle obtained by a prior call to pam_start(). The *name_value* argument is a single NUL terminated string of one of the following forms:

NAME=value of variable	In this case the environment variable of the given NAME is set to the indicated value: *value of variable*. If this variable is already known, it is overwritten. Otherwise it is added to the PAM environment.
NAME=	This function sets the variable to an empty value. It is listed separately to indicate that this is the correct way to achieve such a setting.
NAME	Without an '=' the `pam_putenv`() function will delete the corresponding variable from the PAM environment.

pam_putenv() operates on a copy of *name_value*, which means in contrast to putenv(3), the application is responsible for freeing the data.

2.1.7.2. RETURN VALUES

PAM_PERM_DENIED	Argument *name_value* given is a NULL pointer.
PAM_BAD_ITEM	Variable requested (for deletion) is not currently set.
PAM_ABORT	The *pamh* handle is corrupt.
PAM_BUF_ERR	Memory buffer error.
PAM_SUCCESS	The environment variable was successfully updated.

2.1.8. Get a PAM environment variable

#include <security/pam_appl.h>

```
const char *pam_getenv(pamh, name);

pam_handle_t *pamh;
const char *name;
```

2.1.8.1. DESCRIPTION

The pam_getenv function searches the PAM environment list as associated with the handle *pamh* for an item that matches the string pointed to by *name* and returns a pointer to the value of the environment variable. The application is not allowed to free the data.

2.1.8.2. RETURN VALUES

The pam_getenv function returns NULL on failure.

2.1.9. Getting the PAM environment

#include <security/pam_appl.h>

```
char **pam_getenvlist(pamh);

pam_handle_t *pamh;
```

2.1.9.1. DESCRIPTION

The pam_getenvlist function returns a complete copy of the PAM environment as associated with the handle *pamh*. The PAM environment variables represent the contents of the regular environment variables of the authenticated user when service is granted.

The format of the memory is a malloc()'d array of char pointers, the last element of which is set to NULL. Each of the non-NULL entries in this array point to a NUL terminated and malloc()'d char string of the form: "*name=value*".

It should be noted that this memory will never be free()'d by libpam. Once obtained by a call to pam_getenvlist, it is the responsibility of the calling application to free() this memory.

It is by design, and not a coincidence, that the format and contents of the returned array matches that required for the third argument of the execle(3) function call.

2.1.9.2. RETURN VALUES

The `pam_getenvlist` function returns NULL on failure.

2.2. Other functions provided by `libpam`

2.2.1. Strings describing PAM error codes

```
#include <security/pam_appl.h>

const char *pam_strerror(pamh, errnum);

pam_handle_t *pamh;
int errnum;
```

2.2.1.1. DESCRIPTION

The `pam_strerror` function returns a pointer to a string describing the error code passed in the argument *errnum*, possibly using the LC_MESSAGES part of the current locale to select the appropriate language. This string must not be modified by the application. No library function will modify this string.

2.2.1.2. RETURN VALUES

This function returns always a pointer to a string.

2.2.2. Request a delay on failure

```
#include <security/pam_appl.h>

int pam_fail_delay(pamh, usec);

pam_handle_t *pamh;
unsigned int usec;
```

2.2.2.1. DESCRIPTION

The `pam_fail_delay` function provides a mechanism by which an application or module can suggest a minimum delay of *usec* micro-seconds. The function keeps a record of the longest time requested with this function. Should pam_authenticate(3) fail, the failing return to the application is delayed by an amount of time randomly distributed (by up to 50%) about this longest value.

Independent of success, the delay time is reset to its zero default value when the PAM service module returns control to the application. The delay occurs *after* all authentication modules have been called, but *before* control is returned to the service application.

When using this function the programmer should check if it is available with:

```
#ifdef HAVE_PAM_FAIL_DELAY
    ....
```

```
#endif /* HAVE_PAM_FAIL_DELAY */
```

For applications written with a single thread that are event driven in nature, generating this delay may be undesirable. Instead, the application may want to register the delay in some other way. For example, in a single threaded server that serves multiple authentication requests from a single event loop, the application might want to simply mark a given connection as blocked until an application timer expires. For this reason the delay function can be changed with the *PAM_FAIL_DELAY* item. It can be queried and set with pam_get_item(3) and pam_set_item(3) respectively. The value used to set it should be a function pointer of the following prototype:

```
void (*delay_fn)(int retval, unsigned usec_delay, void *appdata_ptr);
```

The arguments being the *retval* return code of the module stack, the *usec_delay* micro-second delay that libpam is requesting and the *appdata_ptr* that the application has associated with the current *pamh*. This last value was set by the application when it called pam_start(3) or explicitly with pam_set_item(3).

Note that the PAM_FAIL_DELAY item is set to NULL by default. This indicates that PAM should perform a random delay as described above when authentication fails and a delay has been suggested. If an application does not want the PAM library to perform any delay on authentication failure, then the application must define a custom delay function that executes no statements and set the PAM_FAIL_DELAY item to point to this function.

2.2.2.2. RETURN VALUES

PAM_SUCCESS Delay was successful adjusted.

PAM_SYSTEM_ERR A NULL pointer was submitted as PAM handle.

Chapter 3. What is expected of a module

The module must supply a sub-set of the six functions listed below. Together they define the function of a *Linux-PAM module*. Module developers are strongly urged to read the comments on security that follow this list.

3.1. Overview

The six module functions are grouped into four independent management groups. These groups are as follows: *authentication*, *account*, *session* and *password*. To be properly defined, a module must define all functions within at least one of these groups. A single module may contain the necessary functions for *all* four groups.

3.1.1. Functional independence

The independence of the four groups of service a module can offer means that the module should allow for the possibility that any one of these four services may legitimately be called in any order. Thus, the module writer should consider the appropriateness of performing a service without the prior success of some other part of the module.

As an informative example, consider the possibility that an application applies to change a user's authentication token, without having first requested that *Linux-PAM* authenticate the user. In some cases this may be deemed appropriate: when **root** wants to change the authentication token of some lesser user. In other cases it may not be appropriate: when **joe** maliciously wants to reset **alice**'s password; or when anyone other than the user themself wishes to reset their *KERBEROS* authentication token. A policy for this action should be defined by any reasonable authentication scheme, the module writer should consider this when implementing a given module.

3.1.2. Minimizing administration problems

To avoid system administration problems and the poor construction of a `/etc/pam.conf` file, the module developer may define all six of the following functions. For those functions that would not be called, the module should return PAM_SERVICE_ERR and write an appropriate message to the system log. When this action is deemed inappropriate, the function would simply return PAM_IGNORE.

3.1.3. Arguments supplied to the module

The *`flags`* argument of each of the following functions can be logically OR'd with *`PAM_SILENT`*, which is used to inform the module to not pass any *text* (errors or warnings) application.

The *`argc`* and *`argv`* arguments are taken from the line appropriate to this module---that is, with the *service_name* matching that of the application---in the configuration file (see the *Linux-PAM* System Administrators' Guide). Together these two parameters provide the number of arguments and an array of pointers to the individual argument tokens. This will be familiar to C programmers as the ubiquitous method of passing command arguments to the function `main()`. Note, however, that the first argument (*`argv[0]`*) is a true argument and *not* the name of the module.

3.2. Authentication management

3.2.1. Service function for user authentication

#include <security/pam_modules.h>

```
int pam_sm_authenticate(pamh, flags, argc, argv);

pam_handle_t *pamh;
int flags;
int argc;
const char **argv;
```

3.2.1.1. DESCRIPTION

The `pam_sm_authenticate` function is the service module's implementation of the pam_authenticate(3) interface.

This function performs the task of authenticating the user.

Valid flags, which may be logically OR'd with *PAM_SILENT*, are:

PAM_SILENT — Do not emit any messages.

PAM_DISALLOW_NULL_AUTHTOK — Return *PAM_AUTH_ERR* if the database of authentication tokens for this authentication mechanism has a *NULL* entry for the user. Without this flag, such a *NULL* token will lead to a success without the user being prompted.

3.2.1.2. RETURN VALUES

PAM_AUTH_ERR	Authentication failure.
PAM_CRED_INSUFFICIENT	For some reason the application does not have sufficient credentials to authenticate the user.
PAM_AUTHINFO_UNAVAIL	The modules were not able to access the authentication information. This might be due to a network or hardware failure etc.
PAM_SUCCESS	The authentication token was successfully updated.
PAM_USER_UNKNOWN	The supplied username is not known to the authentication service.
PAM_MAXTRIES	One or more of the authentication modules has reached its limit of tries authenticating the user. Do not try again.

3.2.2. Service function to alter credentials

#include <security/pam_modules.h>

```
int pam_sm_setcred(pamh, flags, argc, argv);

pam_handle_t *pamh;
int flags;
```

```
int argc;
const char **argv;
```

3.2.2.1. DESCRIPTION

The pam_sm_setcred function is the service module's implementation of the pam_setcred(3) interface.

This function performs the task of altering the credentials of the user with respect to the corresponding authorization scheme. Generally, an authentication module may have access to more information about a user than their authentication token. This function is used to make such information available to the application. It should only be called *after* the user has been authenticated but before a session has been established.

Valid flags, which may be logically OR'd with *PAM_SILENT*, are:

PAM_SILENT	Do not emit any messages.
PAM_ESTABLISH_CRED	Initialize the credentials for the user.
PAM_DELETE_CRED	Delete the credentials associated with the authentication service.
PAM_REINITIALIZE_CRED	Reinitialize the user credentials.
PAM_REFRESH_CRED	Extend the lifetime of the user credentials.

The way the *auth* stack is navigated in order to evaluate the pam_setcred() function call, independent of the pam_sm_setcred() return codes, is exactly the same way that it was navigated when evaluating the pam_authenticate() library call. Typically, if a stack entry was ignored in evaluating pam_authenticate(), it will be ignored when libpam evaluates the pam_setcred() function call. Otherwise, the return codes from each module specific pam_sm_setcred() call are treated as *required*.

3.2.2.2. RETURN VALUES

PAM_CRED_UNAVAIL	This module cannot retrieve the user's credentials.
PAM_CRED_EXPIRED	The user's credentials have expired.
PAM_CRED_ERR	This module was unable to set the credentials of the user.
PAM_SUCCESS	The user credential was successfully set.
PAM_USER_UNKNOWN	The user is not known to this authentication module.

These, non-*PAM_SUCCESS*, return values will typically lead to the credential stack *failing*. The first such error will dominate in the return value of pam_setcred().

3.3. Account management

3.3.1. Service function for account management

#include <security/pam_modules.h>

```
int pam_sm_acct_mgmt(pamh, flags, argc, argv);

pam_handle_t *pamh;
```

```
int flags;
int argc;
const char **argv;
```

3.3.1.1. DESCRIPTION

The `pam_sm_acct_mgmt` function is the service module's implementation of the pam_acct_mgmt(3) interface.

This function performs the task of establishing whether the user is permitted to gain access at this time. It should be understood that the user has previously been validated by an authentication module. This function checks for other things. Such things might be: the time of day or the date, the terminal line, remote hostname, etc. This function may also determine things like the expiration on passwords, and respond that the user change it before continuing.

Valid flags, which may be logically OR'd with *PAM_SILENT*, are:

PAM_SILENT	Do not emit any messages.
PAM_DISALLOW_NULL_AUTHTOK	Return *PAM_AUTH_ERR* if the database of authentication tokens for this authentication mechanism has a *NULL* entry for the user.

3.3.1.2. RETURN VALUES

PAM_ACCT_EXPIRED	User account has expired.
PAM_AUTH_ERR	Authentication failure.
PAM_NEW_AUTHTOK_REQD	The user's authentication token has expired. Before calling this function again the application will arrange for a new one to be given. This will likely result in a call to `pam_sm_chauthtok()`.
PAM_PERM_DENIED	Permission denied.
PAM_SUCCESS	The authentication token was successfully updated.
PAM_USER_UNKNOWN	User unknown to password service.

3.4. Session management

3.4.1. Service function to start session management

#include <security/pam_modules.h>

```
int pam_sm_open_session(pamh, flags, argc, argv);

pam_handle_t *pamh;
int flags;
int argc;
const char **argv;
```

3.4.1.1. DESCRIPTION

The `pam_sm_open_session` function is the service module's implementation of the pam_open_session(3) interface.

This function is called to commence a session. The only valid value for `flags` is zero or:

PAM_SILENT Do not emit any messages.

3.4.1.2. RETURN VALUES

PAM_SESSION_ERR Cannot make/remove an entry for the specified session.

PAM_SUCCESS The session was successfully started.

3.4.2. Service function to terminate session management

```
#include <security/pam_modules.h>

int pam_sm_close_session(pamh, flags, argc, argv);

pam_handle_t *pamh;
int flags;
int argc;
const char **argv;
```

3.4.2.1. DESCRIPTION

The `pam_sm_close_session` function is the service module's implementation of the pam_close_session(3) interface.

This function is called to terminate a session. The only valid value for `flags` is zero or:

PAM_SILENT Do not emit any messages.

3.4.2.2. RETURN VALUES

PAM_SESSION_ERR Cannot make/remove an entry for the specified session.

PAM_SUCCESS The session was successfully terminated.

3.5. Authentication token management

3.5.1. Service function to alter authentication token

```
#include <security/pam_modules.h>

int pam_sm_chauthtok(pamh, flags, argc, argv);

pam_handle_t *pamh;
int flags;
int argc;
const char **argv;
```

3.5.1.1. DESCRIPTION

The `pam_sm_chauthtok` function is the service module's implementation of the pam_chauthtok(3) interface.

This function is used to (re-)set the authentication token of the user.

Valid flags, which may be logically OR'd with *PAM_SILENT*, are:

PAM_SILENT — Do not emit any messages.

PAM_CHANGE_EXPIRED_AUTHTOK — This argument indicates to the module that the user's authentication token (password) should only be changed if it has expired. This flag is optional and *must* be combined with one of the following two flags. Note, however, the following two options are *mutually exclusive*.

PAM_PRELIM_CHECK — This indicates that the modules are being probed as to their ready status for altering the user's authentication token. If the module requires access to another system over some network it should attempt to verify it can connect to this system on receiving this flag. If a module cannot establish it is ready to update the user's authentication token it should return *PAM_TRY_AGAIN*, this information will be passed back to the application.

If the control value *sufficient* is used in the password stack, the *PAM_PRELIM_CHECK* section of the modules following that control value is not always executed.

PAM_UPDATE_AUTHTOK — This informs the module that this is the call it should change the authorization tokens. If the flag is logically OR'd with *PAM_CHANGE_EXPIRED_AUTHTOK*, the token is only changed if it has actually expired.

The PAM library calls this function twice in succession. The first time with *PAM_PRELIM_CHECK* and then, if the module does not return *PAM_TRY_AGAIN*, subsequently with *PAM_UPDATE_AUTHTOK*. It is only on the second call that the authorization token is (possibly) changed.

3.5.1.2. RETURN VALUES

PAM_AUTHTOK_ERR — The module was unable to obtain the new authentication token.

PAM_AUTHTOK_RECOVERY_ERR — The module was unable to obtain the old authentication token.

PAM_AUTHTOK_LOCK_BUSY — Cannot change the authentication token since it is currently locked.

PAM_AUTHTOK_DISABLE_AGING — Authentication token aging has been disabled.

PAM_PERM_DENIED — Permission denied.

PAM_TRY_AGAIN — Preliminary check was unsuccessful. Signals an immediate return to the application is desired.

PAM_SUCCESS — The authentication token was successfully updated.

PAM_USER_UNKNOWN — User unknown to password service.

Chapter 4. Generic optional arguments

Here we list the generic arguments that all modules can expect to be passed. They are not mandatory, and their absence should be accepted without comment by the module.

debug — Use the pam_syslog(3) call to log debugging information to the system log files.

use_first_pass — The module should not prompt the user for a password. Instead, it should obtain the previously typed password (by a call to `pam_get_item()` for the *PAM_AUTHTOK* item), and use that. If that doesn't work, then the user will not be authenticated. (This option is intended for **auth** and **passwd** modules only).

Chapter 5. Programming notes

Here we collect some pointers for the module writer to bear in mind when writing/developing a *Linux-PAM* compatible module.

5.1. Security issues for module creation

5.1.1. Sufficient resources

Care should be taken to ensure that the proper execution of a module is not compromised by a lack of system resources. If a module is unable to open sufficient files to perform its task, it should fail gracefully, or request additional resources. Specifically, the quantities manipulated by the setrlimit(2) family of commands should be taken into consideration.

5.1.2. Who´s who?

Generally, the module may wish to establish the identity of the user requesting a service. This may not be the same as the username returned by `pam_get_user()`. Indeed, that is only going to be the name of the user under whose identity the service will be given. This is not necessarily the user that requests the service.

In other words, user X runs a program that is setuid-Y, it grants the user to have the permissions of Z. A specific example of this sort of service request is the **su** program: user **joe** executes **su** to become the user **jane**. In this situation X=**joe**, Y=**root** and Z=**jane**. Clearly, it is important that the module does not confuse these different users and grant an inappropriate level of privilege.

The following is the convention to be adhered to when juggling user-identities.

- X, the identity of the user invoking the service request. This is the user identifier; returned by the function getuid(2).
- Y, the privileged identity of the application used to grant the requested service. This is the *effective* user identifier; returned by the function geteuid(2).
- Z, the user under whose identity the service will be granted. This is the username returned by `pam_get_user()` and also stored in the *Linux-PAM* item, *PAM_USER*.
- *Linux-PAM* has a place for an additional user identity that a module may care to make use of. This is the *PAM_RUSER* item. Generally, network sensitive modules/applications may wish to set/read this item to establish the identity of the user requesting a service from a remote location.

Note, if a module wishes to modify the identity of either the *uid* or *euid* of the running process, it should take care to restore the original values prior to returning control to the *Linux-PAM* library.

5.1.3. Using the conversation function

Prior to calling the conversation function, the module should reset the contents of the pointer that will return the applications response. This is a good idea since the application may fail to fill the pointer and the module should be in a position to notice!

The module should be prepared for a failure from the conversation. The generic error would be *PAM_CONV_ERR*, but anything other than *PAM_SUCCESS* should be treated as indicating failure.

5.1.4. Authentication tokens

To ensure that the authentication tokens are not left lying around the items, *PAM_AUTHTOK* and *PAM_OLDAUTHTOK*, are not available to the application: they are defined in `<security/pam_modules.h>`. This is ostensibly for security reasons, but a maliciously programmed application will always have access to all memory of the process, so it is only superficially enforced. As a general rule the module should overwrite authentication tokens as soon as they are no longer needed. Especially before `free()`'ing them. The *Linux-PAM* library is required to do this when either of these authentication token items are (re)set.

Not to dwell too little on this concern; should the module store the authentication tokens either as (automatic) function variables or using `pam_[gs]et_data()` the associated memory should be over-written explicitly before it is released. In the case of the latter storage mechanism, the associated `cleanup()` function should explicitly overwrite the `*data` before `free()`'ing it: for example,

```
/*
 * An example cleanup() function for releasing memory that was used to
 * store a password.
 */

int cleanup(pam_handle_t *pamh, void *data, int error_status)
{
    char *xx;

    if ((xx = data)) {
        while (*xx)
            *xx++ = '\0';
        free(data);
    }
    return PAM_SUCCESS;
}
```

5.2. Use of syslog(3)

Only rarely should error information be directed to the user. Usually, this is to be limited to "*sorry you cannot login now*" type messages. Information concerning errors in the configuration file, `/etc/pam.conf`, or due to some system failure encountered by the module, should be written to syslog(3) with *facility-type LOG_AUTHPRIV*.

With a few exceptions, the level of logging is, at the discretion of the module developer. Here is the recommended usage of different logging levels:

- As a general rule, errors encountered by a module should be logged at the *LOG_ERR* level. However, information regarding an unrecognized argument, passed to a module from an entry in the `/etc/pam.conf` file, is *required* to be logged at the *LOG_ERR* level.

- Debugging information, as activated by the **debug** argument to the module in `/etc/pam.conf`, should be logged at the *LOG_DEBUG* level.

- If a module discovers that its personal configuration file or some system file it uses for information is corrupted or somehow unusable, it should indicate this by logging messages at level, *LOG_ALERT*.

- Shortages of system resources, such as a failure to manipulate a file or `malloc()` failures should be logged at level *LOG_CRIT*.
- Authentication failures, associated with an incorrectly typed password should be logged at level, *LOG_NOTICE*.

5.3. Modules that require system libraries

Writing a module is much like writing an application. You have to provide the "conventional hooks" for it to work correctly, like `pam_sm_authenticate()` etc., which would correspond to the `main()` function in a normal function.

Typically, the author may want to link against some standard system libraries. As when one compiles a normal program, this can be done for modules too: you simply append the `-l`*XXX* arguments for the desired libraries when you create the shared module object. To make sure a module is linked to the **libwhatever.so** library when it is `dlopen()`ed, try:

```
% gcc -shared -o pam_module.so pam_module.o -lwhatever
```

Chapter 6. An example module

At some point, we may include a fully commented example of a module in this document. For now, please look at the modules directory of the *Linux-PAM* sources.

Chapter 7. See also

- The Linux-PAM System Administrators' Guide.
- The Linux-PAM Application Developers' Guide.
- The V. Samar and R. Schemers (SunSoft), ``UNIFIED LOGIN WITH PLUGGABLE AUTHENTICATION MODULES'', Open Software Foundation Request For Comments 86.0, October 1995.

Chapter 8. Author/acknowledgments

This document was written by Andrew G. Morgan (morgan@kernel.org) with many contributions from Chris Adams, Peter Allgeyer, Tim Baverstock, Tim Berger, Craig S. Bell, Derrick J. Brashear, Ben Buxton, Seth Chaiklin, Oliver Crow, Chris Dent, Marc Ewing, Cristian Gafton, Emmanuel Galanos, Brad M. Garcia, Eric Hester, Roger Hu, Eric Jacksch, Michael K. Johnson, David Kinchlea, Olaf Kirch, Marcin Korzonek, Thorsten Kukuk, Stephen Langasek, Nicolai Langfeldt, Elliot Lee, Luke Kenneth Casson Leighton, Al Longyear, Ingo Luetkebohle, Marek Michalkiewicz, Robert Milkowski, Aleph One, Martin Pool, Sean Reifschneider, Jan Rekorajski, Erik Troan, Theodore Ts'o, Jeff Uphoff, Myles Uyema, Savochkin Andrey Vladimirovich, Ronald Wahl, David Wood, John Wilmes, Joseph S. D. Yao and Alex O. Yuriev.

Thanks are also due to Sun Microsystems, especially to Vipin Samar and Charlie Lai for their advice. At an early stage in the development of *Linux-PAM*, Sun graciously made the documentation for their implementation of PAM available. This act greatly accelerated the development of *Linux-PAM*.

Chapter 9. Copyright information for this document