# The Linux-PAM Module Writers' Guide

Andrew G. Morgan, `morgan@kernel.org`

DRAFT v0.76 2002/05/09

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.

## Contents

# 1 Introduction

## 1.1 Synopsis

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

gcc -fPIC -c pam_module-name.c
ld -x --shared -o pam_module-name.so pam_module-name.o
```

## 1.2 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 `/usr/lib/security` 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 the two forms can be used together. 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.

# 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. Essesntially this is the `libpam.*` library.

### 2.1.1 Setting data

Synopsis:

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

The modules may be dynamically loadable objects. In general such files should not contain `static` variables. This and the subsequent function 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.

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()` (see the Linux-PAM Application Developers' Guide).

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()` is called by the module, the `error_status` carries the return value of the `pam_authenticate()` 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()`.

`PAM_DATA_SILENT`

Which indicates that the process would prefer to perform the `cleanup()` quietly. That is, discourages logging/messages to the user.

### 2.1.2 Getting data

Synopsis:

```
extern int pam_get_data(const pam_handle_t *pamh,
                        const char *module_data_name,
                        const void **data);
```

This function together with the previous one provides a method of associating module-specific data with the handle `pamh`. A successful call to `pam_get_data` will result in `*data` pointing to the data associated with the `module_data_name`. Note, this data is *not* a copy and should be treated as *constant* by the module.

Note, if there is an entry but it has the value `NULL`, then this call returns `PAM_NO_MODULE_DATA`.

### 2.1.3 Setting items

Synopsis:

```
extern int pam_set_item(pam_handle_t *pamh,
                        int item_type,
                        const void *item);
```

This function is used to (re)set the value of one of the `item_types`. The reader is urged to read the entry for this function in the **Linux-PAM** application developers' manual.

In addition to the `items` listed there, the module can set the following two `item_types`:

`PAM_AUTHTOK`

The authentication token (often a password). This token should be ignored by all module functions besides `pam_sm_authenticate()` and `pam_sm_chauthtok()`. 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()`.

Both of these items are reset before returning to the application. When resetting these items, the **Linux-PAM** library first writes `0`'s to the current tokens and then `free()`'s the associated memory.

The return values for this function are listed in the **Linux-PAM** Application Developers' Guide.

### 2.1.4 Getting items

Synopsis:

```
extern int pam_get_item(const pam_handle_t *pamh,
                        int item_type,
                        const void **item);
```

This function is used to obtain the value of the specified `item_type`. It is better documented in the **Linux-PAM** Application Developers' Guide. However, there are three things worth stressing here:

- Generally, if the module wishes to obtain the name of the user, it should not use this function, but instead perform a call to `pam_get_user()` (see section 2.1.6 (below)).
- The module is additionally privileged to read the authentication tokens, `PAM_AUTHTOK` and `PAM_OLDAUTHTOK` (see the section above on `pam_set_data()`).
- The module should *not* `free()` or alter the data pointed to by `*item` after a successful return from `pam_get_item()`. This pointer points directly at the data contained within the `*pamh` structure. Should a module require that a change is made to the this `ITEM` it should make the appropriate call to `pam_set_item()`.

### 2.1.5 The *conversation* mechanism

Following the call `pam_get_item(pamh,PAM_CONV,&item)`, the pointer `item` points to a structure containing an a pointer to a *conversation*-function that provides limited but direct access to the application. The purpose of this function is to allow the module to prompt the user for their password and pass other information in a manner consistent with the application. For example, an X-windows based program might pop up a dialog box to report a login failure. Just as the application should not be concerned with the method of authentication, so the module should not dictate the manner in which input (output) is obtained from (presented to) to the user.

**The reader is strongly urged to read the more complete description of the `pam_conv` structure, written from the perspective of the application developer, in the Linux-PAM Application Developers' Guide.**

The return values for this function are listed in the **Linux-PAM** Application Developers' Guide.

The `pam_response` structure returned after a call to the `pam_conv` function must be `free()`'d by the module. Since the call to the conversation function originates from the module, it is clear that this `pam_response` structure could be either statically or dynamically (using `malloc()` etc.) allocated within the application. Repeated calls to the conversation function would likely overwrite static memory, so it is required that for a successful return from the conversation function the memory for the response structure is dynamically allocated by the application with one of the `malloc()` family of commands and *must* be `free()`'d by the module.

If the `pam_conv` mechanism is used to enter authentication tokens, the module should either pass the result to the `pam_set_item()` library function, or copy it itself. In such a case, once the token has been stored (by one of these methods or another one), the memory returned by the application should be overwritten with `0`'s, and then `free()`'d.

There is a handy macro `_pam_drop_reply()` to be found in `<security/_pam_macros.h>` that can be used to conveniently cleanup a `pam_response` structure. (Note, this include file is specific to the Linux-PAM sources, and whilst it will work with Sun derived PAM implementations, it is not generally distributed by Sun.)

### 2.1.6 Getting the name of a user

Synopsis:

```
extern int pam_get_user(pam_handle_t *pamh,
                        const char **user,
                        const char *prompt);
```

This is a **Linux-PAM** library function that returns the (prospective) name of the user. To determine the username it does the following things, in this order:

- checks what `pam_get_item(pamh, PAM_USER, ... );` would have returned. If this is not `NULL` this is what it returns. Otherwise,
- obtains a username from the application via the `pam_conv` 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: "Please enter username: "

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 by the module. Instead, it will be liberated on the next call to `pam_get_user()`, or by `pam_end()` when the application ends its interaction with **Linux-PAM**.

Also, in addition, it should be noted that this function sets the `PAM_USER` item that is associated with the `pam_[gs]et_item()` function.

The return value of this function is one of the following:

- `PAM_SUCCESS` - username obtained.
- `PAM_CONV_AGAIN` - converstation did not complete and the caller is required to return control to the application, until such time as the application has completed the conversation process. A module calling `pam_get_user()` that obtains this return code, should return `PAM_INCOMPLETE` and be prepared (when invoked the next time) to recall `pam_get_user()` to fill in the user's name, and then pick up where it left off as if nothing had happened. This procedure is needed to support an event-driven application programming model.
- `PAM_CONV_ERR` - the conversation method supplied by the application failed to obtain the username.

### 2.1.7 Setting a Linux-PAM environment variable

Synopsis:

```
extern int pam_putenv(pam_handle_t *pamh, const char *name_value);
```

**Linux-PAM** comes equipped with a series of functions for maintaining a set of *environment* variables. The environment is initialized by the call to `pam_start()` and is **erased** with a call to `pam_end()`. This *environment* is associated with the `pam_handle_t` pointer returned by the former call.

The default environment is all but empty. It contains a single `NULL` pointer, which is always required to terminate the variable-list. The `pam_putenv()` function can be used to add a new environment variable, replace an existing one, or delete an old one.

- Adding/replacing a variable

  To add or overwrite a **Linux-PAM** environment variable the value of the argument `name_value`, should be of the following form:

  ```
  name_value="VARIABLE=VALUE OF VARIABLE"
  ```

  Here, `VARIABLE` is the environment variable's name and what follows the '=' is its (new) value. (Note, that `"VARIABLE="` is a valid value for `name_value`, indicating that the variable is set to `""`.)

- Deleting a variable

  To delete a **Linux-PAM** environment variable the value of the argument `name_value`, should be of the following form:

  ```
      name_value="VARIABLE"
  ```

  Here, `VARIABLE` is the environment variable's name and the absence of an '=' indicates that the variable should be removed.

In all cases `PAM_SUCCESS` indicates success.

### 2.1.8 Getting a Linux-PAM environment variable

Synopsis:

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

This function can be used to return the value of the given variable. If the returned value is `NULL`, the variable is not known.

### 2.1.9 Listing the Linux-PAM environment

Synopsis:

```
extern char * const *pam_getenvlist(pam_handle_t *pamh);
```

This function returns a pointer to the entire **Linux-PAM** environment array. At first sight the *type* of the returned data may appear a little confusing. It is basically a *read-only* array of character pointers, that lists the `NULL` terminated list of environment variables set so far.

Although, this is not a concern for the module programmer, we mention here that an application should be careful to copy this entire array before executing `pam_end()` otherwise all the variable information will be lost. (There are functions in `libpam_misc` for this purpose: `pam_misc_copy_env()` and `pam_misc_drop_env()`.)

## 2.2 Other functions provided by `libpam`

### 2.2.1 Understanding errors

- `extern const char *pam_strerror(pam_handle_t *pamh, int errnum);`

  This function returns some text describing the **Linux-PAM** error associated with the argument `errnum`. If the error is not recognized "`Unknown Linux-PAM error`" is returned.

### 2.2.2 Planning for delays

- `extern int pam_fail_delay(pam_handle_t *pamh, unsigned int micro_sec)`

  This function is offered by **Linux-PAM** to facilitate time delays following a failed call to `pam_authenticate()` and before control is returned to the application. When using this function the module programmer should check if it is available with,

```
#ifdef PAM_FAIL_DELAY
    ....
#endif /* PAM_FAIL_DELAY */
```

Generally, an application requests that a user is authenticated by **Linux-PAM** through a call to `pam_authenticate()` or `pam_chauthtok()`. These functions call each of the *stacked* authentication modules listed in the **Linux-PAM** configuration file. As directed by this file, one of more of the modules may fail causing the `pam_...()` call to return an error. It is desirable for there to also be a pause before the application continues. The principal reason for such a delay is security: a delay acts to discourage *brute force* dictionary attacks primarily, but also helps hinder *timed* (cf. covert channel) attacks.

The `pam_fail_delay()` function provides the mechanism by which an application or module can suggest a minimum delay (of `micro_sec` *micro-seconds*). **Linux-PAM** keeps a record of the longest time requested with this function. Should `pam_authenticate()` fail, the failing return to the application is delayed by an amount of time randomly distributed (by up to 25%) about this longest value.

Independent of success, the delay time is reset to its zero default value when **Linux-PAM** returns control to the application.

# 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) to the 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

To be correctly initialized, `PAM_SM_AUTH` must be `#define`'d prior to including `<security/pam_modules.h>`. This will ensure that the prototypes for static modules are properly declared.

- `PAM_EXTERN int pam_sm_authenticate(pam_handle_t *pamh, int flags, int argc, const char **argv);`

  This function performs the task of authenticating the user.

  The `flags` argument can be a logically OR'd with `PAM_SILENT` and optionally take the following value:

  `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.

  Besides `PAM_SUCCESS` return values that can be sent by this function are one of the following:

  `PAM_AUTH_ERR`
  : The user was not authenticated

  `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_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.

- `PAM_EXTERN int pam_sm_setcred(pam_handle_t *pamh, int flags, int argc, const char **argv);`

  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.

Permitted flags, one of which, may be logically OR'd with `PAM_SILENT` are,

`PAM_ESTABLISH_CRED`
: Set the credentials for the authentication service,

`PAM_DELETE_CRED`
: Delete the credentials associated with the authentication service,

`PAM_REINITIALIZE_CRED`
: Reinitialize the user credentials, and

`PAM_REFRESH_CRED`
: Extend the lifetime of the user credentials.

Prior to **Linux-PAM-0.75**, and due to a deficiency with the way the `auth` stack was handled in the case of the setcred stack being processed, the module was required to attempt to return the same error code as `pam_sm_authenticate` did. This was necessary to preserve the logic followed by libpam as it executes the stack of *authentication* modules, when the application called either `pam_authenticate()` or `pam_setcred()`. Failing to do this, led to confusion on the part of the System Administrator.

For **Linux-PAM-0.75** and later, libpam handles the credential stack much more sanely. 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`.

Besides `PAM_SUCCESS`, the module may return one of the following errors:

`PAM_CRED_UNAVAIL`
: This module cannot retrieve the user's credentials.

`PAM_CRED_EXPIRED`
: The user's credentials have expired.

`PAM_USER_UNKNOWN`
: The user is not known to this authentication module.

`PAM_CRED_ERR`
: This module was unable to set the credentials of the user.

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

To be correctly initialized, `PAM_SM_ACCOUNT` must be `#define`'d prior to including `<security/pam_modules.h>`. This will ensure that the prototype for a static module is properly declared.

- `PAM_EXTERN int pam_sm_acct_mgmt(pam_handle_t *pamh, int flags, int argc, const char **argv);`

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 the same as those applicable to the `flags` argument of `pam_sm_authenticate`.

This function may return one of the following errors,

`PAM_ACCT_EXPIRED`

The user is no longer permitted access to the system.

`PAM_AUTH_ERR`

There was an authentication error.

`PAM_AUTHTOKEN_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_USER_UNKNOWN`

The user is not known to the module's account management component.

## 3.4 Session management

To be correctly initialized, `PAM_SM_SESSION` must be `#define`'d prior to including `<security/pam_modules.h>`. This will ensure that the prototypes for static modules are properly declared.

The following two functions are defined to handle the initialization/termination of a session. For example, at the beginning of a session the module may wish to log a message with the system regarding the user. Similarly, at the end of the session the module would inform the system that the user's session has ended.

It should be possible for sessions to be opened by one application and closed by another. This either requires that the module uses only information obtained from `pam_get_item()`, or that information regarding the session is stored in some way by the operating system (in a file for example).

- `PAM_EXTERN int pam_sm_open_session(pam_handle_t *pamh, int flags, int argc, const char **argv);`

  This function is called to commence a session. The only valid, but optional, flag is `PAM_SILENT`.

  As a return value, `PAM_SUCCESS` signals success and `PAM_SESSION_ERR` failure.

- `PAM_EXTERN int pam_sm_close_session(pam_handle_t *pamh, int flags, int argc, const char **argv);`

  This function is called to terminate a session. The only valid, but optional, flag is `PAM_SILENT`.

  As a return value, `PAM_SUCCESS` signals success and `PAM_SESSION_ERR` failure.

## 3.5 Password management

To be correctly initialized, `PAM_SM_PASSWORD` must be `#define`'d prior to including `<security/pam_modules.h>`. This will ensure that the prototype for a static module is properly declared.

- `PAM_EXTERN int pam_sm_chauthtok(pam_handle_t *pamh, int flags, int argc, const char **argv);`

  This function is used to (re-)set the authentication token of the user. A valid flag, which may be logically OR'd with `PAM_SILENT`, can be built from the following list,

  `PAM_CHANGE_EXPIRED_AUTHTOK`
  : This argument indicates to the module that the users 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.

  `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.

  Note, the **Linux-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.

  `PAM_SUCCESS` is the only successful return value, valid error-returns are:

  `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_USER_UNKNOWN`
  : The user is not known to the authentication token changing service.

## 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 `syslog(3)` call to log debugging information to the system log files.

`no_warn`

Instruct module to not give warning messages to the application.

`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).

`try_first_pass`

The module should attempt authentication with the previously typed password (by a call to `pam_get_item()` for the `PAM_AUTHTOK` item). If that doesn't work, then the user is prompted for a password. (This option is intended for `auth` modules only).

`use_mapped_pass`

**WARNING:** coding this functionality may cause the module writer to break *local* encryption laws. For example, in the U.S. there are restrictions on the export computer code that is capable of strong encryption. It has not been established whether this option is affected by this law, but one might reasonably assume that it does until told otherwise. For this reason, this option is not supported by any of the modules distributed with **Linux-PAM**.

The intended function of this argument, however, is that the module should take the existing authentication token from a previously invoked module and use it as a key to retrieve the authentication token for this module. For example, the module might create a strong hash of the `PAM_AUTHTOK` item (established by a previously executed module). Then, with logical-exclusive-or, use the result as a *key* to safely store/retrieve the authentication token for this module in/from a local file *etc.* .

`expose_account`

In general the leakage of some information about user accounts is not a secure policy for modules to adopt. Sometimes information such as users names or home directories, or preferred shell, can be used to attack a user's account. In some circumstances, however, this sort of information is not deemed a threat: displaying a user's full name when asking them for a password in a secured environment could also be called being 'friendly'. The `expose_account` argument is a standard module argument to encourage a module to be less discrete about account information as it is deemed appropriate by the local administrator.

# 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(2)` 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 `lib`***`whatever`***`.so` library when it is `dlopen()`ed, try:

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

## 5.4 Added requirements for *statically* loaded modules.

Modules may be statically linked into libpam. This should be true of all the modules distributed with the basic **Linux-PAM** distribution. To be statically linked, a module needs to export information about the functions it contains in a manner that does not clash with other modules.

The extra code necessary to build a static module should be delimited with `#ifdef PAM_STATIC` and `#endif`. The static code should do the following:

- Define a single structure, `struct pam_module`, called `_pam_`*`modname`*`_modstruct`, where *`modname`* is the name of the module **as used in the filesystem** but without the leading directory name (generally `/usr/lib/security/` or the suffix (generally `.so`).

As a simple example, consider the following module code which defines a module that can be compiled to be *static* or *dynamic*:

```
#include <stdio.h>                                   /* for NULL define */

#define PAM_SM_PASSWORD         /* the only pam_sm_... function declared */
#include <security/pam_modules.h>

PAM_EXTERN int pam_sm_chauthtok(pam_handle_t *pamh, int flags,
                                int argc, const char **argv)
{
     return PAM_SUCCESS;
}

#ifdef PAM_STATIC              /* for the case that this module is static */

struct pam_module _pam_modname_modstruct = {        /* static module data */
     "pam_modname",
     NULL,
     NULL,
     NULL,
     NULL,
     NULL,
     pam_sm_chauthtok,
};

#endif                                                /* end PAM_STATIC */
```

To be linked with *libpam*, staticly-linked modules must be built from within the `Linux-PAM-X.YY/modules/` subdirectory of the **Linux-PAM** source directory as part of a normal build of the **Linux-PAM** system.

The *Makefile*, for the module in question, must execute the `register_static` shell script that is located in the `Linux-PAM-X.YY/modules/` subdirectory. This is to ensure that the module is properly registered with *libpam*.

The **two** manditory arguments to `register_static` are the title, and the pathname of the object file containing the module's code. The pathname is specified relative to the `Linux-PAM-X.YY/modules` directory. The pathname may be an empty string—this is for the case that a single object file needs to register more than one `struct pam_module`. In such a case, exactly one call to `register_static` must indicate the object file.

Here is an example; a line in the *Makefile* might look like this:

```
register:
ifdef STATIC
        (cd ..; ./register_static pam_modname pam_modname/pam_modname.o)
endif
```

For some further examples, see the `modules` subdirectory of the current **Linux-PAM** distribution.

## 6 An example module file

At some point, we may include a fully commented example of a module in this document. For now, we point the reader to these two locations in the public CVS repository:

- A module that always succeeds: *http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/pam/Linux-PAM/modules/*
- A module that always fails: *http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/pam/Linux-PAM/modules/pam_*

## 7 Files

`/usr/lib/libpam.so.*`

the shared library providing applications with access to **Linux-PAM**.

`/etc/pam.conf`

the **Linux-PAM** configuration file.

`/usr/lib/security/pam_*.so`

the primary location for **Linux-PAM** dynamically loadable object files; the modules.

## 8 See also

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

## 9 Notes

I intend to put development comments here... like "at the moment this isn't actually supported". At release time what ever is in this section will be placed in the Bugs section below! :)

- Perhaps we should keep a registry of data-names as used by `pam_[gs]et_data()` so there are no unintentional problems due to conflicts?
- `pam_strerror()` should be internationalized....
- There has been some debate about whether `initgroups()` should be in an application or in a module. It was settled by Sun who stated that initgroups is an action of the *application*. The modules are permitted to add additional groups, however.
- Refinements/futher suggestions to `syslog(3)` usage by modules are needed.

## 10 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, 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**.

## 11 Bugs/omissions

Few PAM modules currently exist. Few PAM-aware applications exist. This document is hopelessly unfinished. Only a partial list of people is credited for all the good work they have done.

## 12 Copyright information for this document

```
$Id: pam_modules.sgml,v 1.9 2002/05/10 06:00:12 agmorgan Exp $
```