The Linux-PAM Application Developers' Guide

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

The Linux-PAM Application Developers' Guide

by Andrew G. Morgan and Thorsten Kukuk

Version 1.7.3, September 2026

Abstract

This manual documents what an application developer needs to know about the *Linux-PAM* library. It describes how an application might use the *Linux-PAM* library to authenticate users. In addition it contains a description of the functions to be found in `libpam_misc` library, that can be used in general applications. Finally, it contains some comments on PAM related security issues for the application developer.

1. Introduction 1
1.1. Description 1
1.2. Synopsis 1
2. Overview 2
3. The public interface to *Linux-PAM* 4
3.1. What can be expected by the application 4
3.1.1. Initialization of PAM transaction 4
3.1.2. Termination of PAM transaction 5
3.1.3. Setting PAM items 5
3.1.4. Getting PAM items 7
3.1.5. Strings describing PAM error codes 9
3.1.6. Request a delay on failure 9
3.1.7. Authenticating the user 10
3.1.8. Setting user credentials 11
3.1.9. Account validation management 12
3.1.10. Updating authentication tokens 13
3.1.11. Start PAM session management 14
3.1.12. terminating PAM session management 14
3.1.13. Set or change PAM environment variable 15
3.1.14. Get a PAM environment variable 15
3.1.15. Getting the PAM environment 16
3.2. What is expected of an application 16
3.2.1. The conversation function 16
3.3. Programming notes 18
4. Security issues of *Linux-PAM* 19
4.1. Care about standard library calls 19
4.2. Choice of a service name 19
4.3. The conversation function 20
4.4. The identity of the user 20
4.5. Sufficient resources 20
5. A library of miscellaneous helper functions 21
5.1. Functions supplied 21
5.1.1. Text based conversation function 21
5.1.2. Transcribing an environment to that of PAM 22
5.1.3. Liberating a locally saved environment 22
5.1.4. BSD like PAM environment variable setting 23
6. Porting legacy applications 24
7. Glossary of PAM related terms 25
8. An example application 26
9. Files 28
10. See also 29
11. Author/acknowledgments 30
12. Copyright information for this document 31

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*.

It is the purpose of the *Linux-PAM* project to liberate the development of privilege granting software from the development of secure and appropriate authentication schemes. This is accomplished by providing a documented library of functions that an application may use for all forms of user authentication management. This library dynamically loads locally configured authentication modules that actually perform the authentication tasks.

From the perspective of an application developer the information contained in the local configuration of the PAM library should not be important. Indeed it is intended that an application treat the functions documented here as a 'black box' that will deal with all aspects of user authentication. 'All aspects' includes user verification, account management, session initialization/termination and also the resetting of passwords (*authentication tokens*).

1.2. Synopsis

For general applications that wish to use the services provided by *Linux-PAM* the following is a summary of the relevant linking information:

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

cc -o application .... -lpam
```

In addition to **libpam**, there is a library of miscellaneous functions that make the job of writing *PAM-aware* applications easier (this library is not covered in the DCE-RFC for PAM and is specific to the Linux-PAM distribution):

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

cc -o application .... -lpam -lpam_misc
```

Chapter 2. Overview

Most service-giving applications are restricted. In other words, their service is not available to all and every prospective client. Instead, the applying client must jump through a number of hoops to convince the serving application that they are authorized to obtain service.

The process of *authenticating* a client is what PAM is designed to manage. In addition to authentication, PAM provides account management, credential management, session management and authentication-token (password changing) management services. It is important to realize when writing a PAM based application that these services are provided in a manner that is *transparent* to the application. That is to say, when the application is written, no assumptions can be made about *how* the client will be authenticated.

The process of authentication is performed by the PAM library via a call to `pam_authenticate()`. The return value of this function will indicate whether a named client (the *user*) has been authenticated. If the PAM library needs to prompt the user for any information, such as their *name* or a *password* then it will do so. If the PAM library is configured to authenticate the user using some silent protocol, it will do this too. (This latter case might be via some hardware interface for example.)

It is important to note that the application must leave all decisions about when to prompt the user at the discretion of the PAM library.

The PAM library, however, must work equally well for different styles of application. Some applications, like the familiar **login** and **passwd** are terminal based applications, exchanges of information with the client in these cases is as plain text messages. Graphically based applications, however, have a more sophisticated interface. They generally interact with the user via specially constructed dialogue boxes. Additionally, network based services require that text messages exchanged with the client are specially formatted for automated processing: one such example is **ftpd** which prefixes each exchanged message with a numeric identifier.

The presentation of simple requests to a client is thus something very dependent on the protocol that the serving application will use. In spite of the fact that PAM demands that it drives the whole authentication process, it is not possible to leave such protocol subtleties up to the PAM library. To overcome this potential problem, the application provides the PAM library with a *conversation* function. This function is called from *within* the PAM library and enables the PAM to directly interact with the client. The sorts of things that this conversation function must be able to do are prompt the user with text and/or obtain textual input from the user for processing by the PAM library. The details of this function are provided in a later section.

For example, the conversation function may be called by the PAM library with a request to prompt the user for a password. Its job is to reformat the prompt request into a form that the client will understand. In the case of **ftpd**, this might involve prefixing the string with the number **331** and sending the request over the network to a connected client. The conversation function will then obtain any reply and, after extracting the typed password, will return this string of text to the PAM library. Similar concerns need to be addressed in the case of an X-based graphical server.

There are a number of issues that need to be addressed when one is porting an existing application to become PAM compliant. A section below has been devoted to this: Porting legacy applications.

Besides authentication, PAM provides other forms of management. Session management is provided with calls to `pam_open_session()` and `pam_close_session()`. What these functions actually do is up to the local administrator. But typically, they could be used to log entry and exit from the system or for mounting and unmounting the user's home directory. If an application provides continuous service for a period of time, it should probably call these functions, first open after the user is authenticated and then close when the service is terminated.

Account management is another area that an application developer should include with a call to `pam_acct_mgmt()`. This call will perform checks on the good health of the user's account (has it expired etc.). One of the things this function may check is whether the user's authentication token has expired - in such a case the application may choose to attempt to update it with a call to `pam_chauthtok()`, although some applications are not suited to this task (**ftp** for example) and in this case the application should deny access to the user.

PAM is also capable of setting and deleting the user's credentials with the call `pam_setcred()`. This function should always be called after the user is authenticated and before service is offered to the user. By convention, this should be the last call to the PAM library before the PAM session is opened. What exactly a credential is, is not well defined. However, some examples are given in the glossary below.

Chapter 3. The public interface to *Linux-PAM*

Firstly, the relevant include file for the *Linux-PAM* library is `<security/pam_appl.h>`. It contains the definitions for a number of functions. After listing these functions, we collect some guiding remarks for programmers.

3.1. What can be expected by the application

3.1.1. Initialization of PAM transaction

#include <security/pam_appl.h>

```
int pam_start(service_name, user, pam_conversation, pamh);

const char *service_name;
const char *user;
const struct pam_conv *pam_conversation;
pam_handle_t **pamh;

int pam_start_confdir(service_name, user, pam_conversation, confdir,
pamh);

const char *service_name;
const char *user;
const struct pam_conv *pam_conversation;
const char *confdir;
pam_handle_t **pamh;
```

3.1.1.1. DESCRIPTION

The `pam_start` function creates the PAM context and initiates the PAM transaction. It is the first of the PAM functions that needs to be called by an application. The transaction state is contained entirely within the structure identified by this handle, so it is possible to have multiple transactions in parallel. But it is not possible to use the same handle for different transactions, a new one is needed for every new context.

The *service_name* argument specifies the name of the service to apply and will be stored as PAM_SERVICE item in the new context. The policy for the service will be read from the file `/etc/pam.d/service_name` or, if that file does not exist, from `/etc/pam.conf`.

The *user* argument can specify the name of the target user and will be stored as PAM_USER item. If the argument is NULL, the module has to ask for this item if necessary.

The *pam_conversation* argument points to a *struct pam_conv* describing the conversation function to use. An application must provide this for direct communication between a loaded module and the application.

Following a successful return (PAM_SUCCESS) the contents of *pamh* is a handle that contains the PAM context for successive calls to the PAM functions. In an error case is the content of *pamh* undefined.

The *pam_handle_t* is a blind structure and the application should not attempt to probe it directly for information. Instead the PAM library provides the functions pam_set_item(3) and pam_get_item(3). The PAM handle cannot be used for multiple authentications at the same time as long as `pam_end` was not called on it before.

The `pam_start_confdir` function behaves like the `pam_start` function but it also allows setting *confdir* argument with a path to a directory to override the default (`/etc/pam.d`) path for service policy files. If the *confdir* is NULL, the function works exactly the same as `pam_start`.

3.1.1.2. RETURN VALUES

PAM_ABORT General failure.

PAM_BUF_ERR Memory buffer error.

PAM_SUCCESS Transaction was successfully started.

PAM_SYSTEM_ERR System error, for example a NULL pointer was submitted instead of a pointer to data.

3.1.2. Termination of PAM transaction

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

int pam_end(pamh, pam_status);

pam_handle_t *pamh;
int pam_status;
```

3.1.2.1. DESCRIPTION

The `pam_end` function terminates the PAM transaction and is the last function an application should call in the PAM context. Upon return the handle *pamh* is no longer valid and all memory associated with it will be invalid.

The *pam_status* argument should be set to the value returned to the application by the last PAM library call.

The value taken by *pam_status* is used as an argument to the module specific callback function, `cleanup()` (See pam_set_data(3) and pam_get_data(3)). In this way the module can be given notification of the pass/fail nature of the tear-down process, and perform any last minute tasks that are appropriate to the module before it is unlinked. This argument can be logically OR'd with *PAM_DATA_SILENT* to indicate that the module should not treat the call too seriously. 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.).

This function *free*'s all memory for items associated with the pam_set_item(3) and pam_get_item(3) functions. Pointers associated with such objects are not valid anymore after `pam_end` was called.

3.1.2.2. RETURN VALUES

PAM_SUCCESS Transaction was successfully terminated.

PAM_SYSTEM_ERR System error, for example a NULL pointer was submitted as PAM handle or the function was called by a module.

3.1.3. 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;
```

3.1.3.1. 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.

3.1.3.2. 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 successfully updated.

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

3.1.4. 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;
```

3.1.4.1. 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.

3.1.4.2. 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 successfully updated.
PAM_SYSTEM_ERR	The *pam_handle_t* passed as first argument was invalid.

3.1.5. Strings describing PAM error codes

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

const char *pam_strerror(pamh, errnum);

pam_handle_t *pamh;
int errnum;
```

3.1.5.1. 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.

3.1.5.2. RETURN VALUES

This function returns always a pointer to a string.

3.1.6. Request a delay on failure

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

int pam_fail_delay(pamh, usec);

pam_handle_t *pamh;
unsigned int usec;
```

3.1.6.1. 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.

3.1.6.2. RETURN VALUES

PAM_SUCCESS Delay was successfully adjusted.

PAM_SYSTEM_ERR A NULL pointer was submitted as PAM handle.

3.1.7. Authenticating the user

#include <security/pam_appl.h>

```
int pam_authenticate(pamh, flags);

pam_handle_t *pamh;
int flags;
```

3.1.7.1. DESCRIPTION

The `pam_authenticate` function is used to authenticate the user. The user is required to provide an authentication token depending upon the authentication service, usually this is a password, but could also be a finger print.

The PAM service module may request that the user enter their username via the conversation mechanism (see pam_start(3) and pam_conv(3)). The name of the authenticated user will be present in the PAM item PAM_USER. This item may be recovered with a call to pam_get_item(3).

The *pamh* argument is an authentication handle obtained by a prior call to pam_start(). The flags argument is the binary or of zero or more of the following values:

PAM_SILENT	Do not emit any messages.
PAM_DISALLOW_NULL_AUTHTOK	The PAM module service should return PAM_AUTH_ERR if the user does not have a registered authentication token.

3.1.7.2. RETURN VALUES

PAM_ABORT	The application should exit immediately after calling pam_end(3) first.
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_MAXTRIES	One or more of the authentication modules has reached its limit of tries authenticating the user. Do not try again.
PAM_SUCCESS	The user was successfully authenticated.
PAM_USER_UNKNOWN	User unknown to authentication service.

3.1.8. Setting user credentials

#include <security/pam_appl.h>

```
int pam_setcred(pamh, flags);

pam_handle_t *pamh;
int flags;
```

3.1.8.1. DESCRIPTION

The `pam_setcred` function is used to establish, maintain and delete the credentials of a user. It should be called to set the credentials after a user has been authenticated and before a session is opened for the user (with pam_open_session(3)). The credentials should be deleted after the session has been closed (with pam_close_session(3)).

A credential is something that the user possesses. It is some property, such as a *Kerberos* ticket, or a supplementary group membership that make up the uniqueness of a given user. On a Linux system the user's *UID* and *GID*'s are credentials too. However, it has been decided that these properties (along with the default supplementary groups of which the user is a member) are credentials that should be set directly by the application and not by PAM. Such credentials should be established, by the application, prior to a call to this function. For example, initgroups(2) (or equivalent) should have been performed.

Valid *flags*, any one of which, may be logically OR'd with `PAM_SILENT`, are:

PAM_ESTABLISH_CRED	Initialize the credentials for the user.
PAM_DELETE_CRED	Delete the user's credentials.
PAM_REINITIALIZE_CRED	Fully reinitialize the user's credentials.
PAM_REFRESH_CRED	Extend the lifetime of the existing credentials.

3.1.8.2. RETURN VALUES

PAM_BUF_ERR	Memory buffer error.
PAM_CRED_ERR	Failed to set user credentials.
PAM_CRED_EXPIRED	User credentials are expired.
PAM_CRED_UNAVAIL	Failed to retrieve user credentials.
PAM_SUCCESS	Data was successfully stored.
PAM_SYSTEM_ERR	A NULL pointer was submitted as PAM handle, the function was called by a module or another system error occurred.
PAM_USER_UNKNOWN	User is not known to an authentication module.

3.1.9. Account validation management

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

int pam_acct_mgmt(pamh, flags);

pam_handle_t *pamh;
int flags;
```

3.1.9.1. DESCRIPTION

The `pam_acct_mgmt` function is used to determine if the user's account is valid. It checks for authentication token and account expiration and verifies access restrictions. It is typically called after the user has been authenticated.

The *pamh* argument is an authentication handle obtained by a prior call to pam_start(). The flags argument is the binary or of zero or more of the following values:

PAM_SILENT	Do not emit any messages.
PAM_DISALLOW_NULL_AUTHTOK	The PAM module service should return PAM_NEW_AUTHTOK_REQD if the user has a null authentication token.

3.1.9.2. RETURN VALUES

PAM_ACCT_EXPIRED	User account has expired.
PAM_AUTH_ERR	Authentication failure.

PAM_NEW_AUTHTOK_REQD	The user account is valid but their authentication token is *expired*. The correct response to this return-value is to require that the user satisfies the `pam_chauthtok()` function before obtaining service. It may not be possible for some applications to do this. In such cases, the user should be denied access until such time as they can update their password.
PAM_PERM_DENIED	Permission denied.
PAM_SUCCESS	The authentication token was successfully updated.
PAM_USER_UNKNOWN	User unknown to password service.

3.1.10. Updating authentication tokens

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

int pam_chauthtok(pamh, flags);

pam_handle_t *pamh;
int flags;
```

3.1.10.1. DESCRIPTION

The `pam_chauthtok` function is used to change the authentication token for a given user (as indicated by the state associated with the handle *pamh*).

The *pamh* argument is an authentication handle obtained by a prior call to pam_start(). The flags argument is the binary or of zero or more of the following values:

PAM_SILENT	Do not emit any messages.
PAM_CHANGE_EXPIRED_AUTHTOK	This argument indicates to the modules that the user's authentication token (password) should only be changed if it has expired. If this argument is not passed, the application requires that all authentication tokens are to be changed.

3.1.10.2. RETURN VALUES

PAM_AUTHTOK_ERR	A module was unable to obtain the new authentication token.
PAM_AUTHTOK_RECOVERY_ERR	A module was unable to obtain the old authentication token.
PAM_AUTHTOK_LOCK_BUSY	One or more of the modules was unable to change the authentication token since it is currently locked.
PAM_AUTHTOK_DISABLE_AGING	Authentication token aging has been disabled for at least one of the modules.
PAM_PERM_DENIED	Permission denied.
PAM_SUCCESS	The authentication token was successfully updated.
PAM_TRY_AGAIN	Not all of the modules were in a position to update the authentication token(s). In such a case none of the user's authentication tokens are updated.

PAM_USER_UNKNOWN User unknown to password service.

3.1.11. Start PAM session management

#include <security/pam_appl.h>

```
int pam_open_session(pamh, flags);

pam_handle_t *pamh;
int flags;
```

3.1.11.1. DESCRIPTION

The `pam_open_session` function sets up a user session for a previously successfully authenticated user. The session should later be terminated with a call to pam_close_session(3).

It should be noted that the effective uid, geteuid(2), of the application should be of sufficient privilege to perform such tasks as creating or mounting the user's home directory for example.

The flags argument is the binary or of zero or more of the following values:

PAM_SILENT Do not emit any messages.

3.1.11.2. RETURN VALUES

PAM_ABORT General failure.

PAM_BUF_ERR Memory buffer error.

PAM_SESSION_ERR Session failure.

PAM_SUCCESS Session was successfully created.

3.1.12. terminating PAM session management

#include <security/pam_appl.h>

```
int pam_close_session(pamh, flags);

pam_handle_t *pamh;
int flags;
```

3.1.12.1. DESCRIPTION

The `pam_close_session` function is used to indicate that an authenticated session has ended. The session should have been created with a call to pam_open_session(3).

It should be noted that the effective uid, geteuid(2). of the application should be of sufficient privilege to perform such tasks as unmounting the user's home directory for example.

The flags argument is the binary or of zero or more of the following values:

PAM_SILENT Do not emit any messages.

3.1.12.2. RETURN VALUES

PAM_ABORT	General failure.
PAM_BUF_ERR	Memory buffer error.
PAM_SESSION_ERR	Session failure.
PAM_SUCCESS	Session was successfully terminated.

3.1.13. 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;
```

3.1.13.1. 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.

3.1.13.2. 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.

3.1.14. Get a PAM environment variable

#include <security/pam_appl.h>

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

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

3.1.14.1. 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.

3.1.14.2. RETURN VALUES

The `pam_getenv` function returns NULL on failure.

3.1.15. Getting the PAM environment

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

char **pam_getenvlist(pamh);

pam_handle_t *pamh;
```

3.1.15.1. 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.

3.1.15.2. RETURN VALUES

The `pam_getenvlist` function returns NULL on failure.

3.2. What is expected of an application

3.2.1. 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;
};
```

3.2.1.1. 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 deriva-

tives, 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])
```

3.2.1.2. RETURN VALUES

PAM_BUF_ERR	Memory buffer error.
PAM_CONV_ERR	Conversation failure. The application should not set **resp*.
PAM_SUCCESS	Success.

3.3. Programming notes

Note, all of the authentication service function calls accept the token *PAM_SILENT*, which instructs the modules to not send messages to the application. This token can be logically OR'd with any one of the permitted tokens specific to the individual function calls. *PAM_SILENT* does not override the prompting of the user for passwords etc., it only stops informative messages from being generated.

Chapter 4. Security issues of *Linux-PAM*

PAM, from the perspective of an application, is a convenient API for authenticating users. PAM modules generally have no increased privilege over that possessed by the application that is making use of it. For this reason, the application must take ultimate responsibility for protecting the environment in which PAM operates.

A poorly (or maliciously) written application can defeat any *Linux-PAM* module's authentication mechanisms by simply ignoring it's return values. It is the applications task and responsibility to grant privileges and access to services. The *Linux-PAM* library simply assumes the responsibility of *authenticating* the user; ascertaining that the user *is* who they say they are. Care should be taken to anticipate all of the documented behavior of the *Linux-PAM* library functions. A failure to do this will most certainly lead to a future security breach.

4.1. Care about standard library calls

In general, writers of authorization-granting applications should assume that each module is likely to call any or *all* 'libc' functions. For 'libc' functions that return pointers to static/dynamically allocated structures (ie. the library allocates the memory and the user is not expected to '`free()`' it) any module call to this function is likely to corrupt a pointer previously obtained by the application. The application programmer should either re-call such a 'libc' function after a call to the *Linux-PAM* library, or copy the structure contents to some safe area of memory before passing control to the *Linux-PAM* library.

Two important function classes that fall into this category are getpwnam(3) and syslog(3).

4.2. Choice of a service name

When picking the *service-name* that corresponds to the first entry in the *Linux-PAM* configuration file, the application programmer should *avoid* the temptation of choosing something related to `argv[0]`. It is a trivial matter for any user to invoke any application on a system under a different name and this should not be permitted to cause a security breach.

In general, this is always the right advice if the program is setuid, or otherwise more privileged than the user that invokes it. In some cases, avoiding this advice is convenient, but as an author of such an application, you should consider well the ways in which your program will be installed and used. (Its often the case that programs are not intended to be setuid, but end up being installed that way for convenience. If your program falls into this category, don't fall into the trap of making this mistake.)

To invoke some *target* application by another name, the user may symbolically link the target application with the desired name. To be precise all the user need do is, **ln -s /target/application ./preferred_name** and then run **./preferred_name**.

By studying the *Linux-PAM* configuration file(s), an attacker can choose the **preferred_name** to be that of a service enjoying minimal protection; for example a game which uses *Linux-PAM* to restrict access to certain hours of the day. If the service-name were to be linked to the filename under which the service was invoked, it is clear that the user is effectively in the position of dictating which authentication scheme the service uses. Needless to say, this is not a secure situation.

The conclusion is that the application developer should carefully define the service-name of an application. The safest thing is to make it a single hard-wired name.

4.3. The conversation function

Care should be taken to ensure that the `conv()` function is robust. Such a function is provided in the library **libpam_misc** (see below).

4.4. The identity of the user

The *Linux-PAM* modules will need to determine the identity of the user who requests a service, and the identity of the user who grants the service. These two users will seldom be the same. Indeed there is generally a third user identity to be considered, the new (assumed) identity of the user once the service is granted.

The need for keeping tabs on these identities is clearly an issue of security. One convention that is actively used by some modules is that the identity of the user requesting a service should be the current *UID* (user ID) of the running process; the identity of the privilege granting user is the *EUID* (effective user ID) of the running process; the identity of the user, under whose name the service will be executed, is given by the contents of the *PAM_USER* pam_get_item(3). Note, modules can change the values of *PAM_USER* and *PAM_RUSER* during any of the `pam_*()` library calls. For this reason, the application should take care to use the `pam_get_item()` every time it wishes to establish who the authenticated user is (or will currently be).

For network-serving databases and other applications that provide their own security model (independent of the OS kernel) the above scheme is insufficient to identify the requesting user.

A more portable solution to storing the identity of the requesting user is to use the *PAM_RUSER* pam_get_item(3). The application should supply this value before attempting to authenticate the user with `pam_authenticate()`. How well this name can be trusted will ultimately be at the discretion of the local administrator (who configures PAM for your application) and a selected module may attempt to override the value where it can obtain more reliable data. If an application is unable to determine the identity of the requesting entity/user, it should not call pam_set_item(3) to set *PAM_RUSER*.

In addition to the *PAM_RUSER* item, the application should supply the *PAM_RHOST* (*requesting host*) item. As a general rule, the following convention for its value can be assumed: NULL = unknown; localhost = invoked directly from the local system; *other.place.xyz* = some component of the user's connection originates from this remote/requesting host. At present, PAM has no established convention for indicating whether the application supports a trusted path to communication from this host.

4.5. Sufficient resources

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

This is also true of conversation prompts. The application should not accept prompts of arbitrary length with out checking for resource allocation failure and dealing with such extreme conditions gracefully and in a manner that preserves the PAM API. Such tolerance may be especially important when attempting to track a malicious adversary.

Chapter 5. A library of miscellaneous helper functions

To aid the work of the application developer a library of miscellaneous functions is provided. It is called **libpam_misc**, and contains a text based conversation function, and routines for enhancing the standard PAM-environment variable support.

The functions, structures and macros, made available by this library can be defined by including `<security/pam_misc.h>`. It should be noted that this library is specific to *Linux-PAM* and is not referred to in the defining DCE-RFC (see See also) below.

5.1. Functions supplied

5.1.1. Text based conversation function

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

int misc_conv(num_msg, msgm, response, appdata_ptr);

int num_msg;
const struct pam_message **msgm;
struct pam_response **response;
void *appdata_ptr;
```

5.1.1.1. DESCRIPTION

The `misc_conv` function is part of **libpam_misc** and not of the standard **libpam** library. This function will prompt the user with the appropriate comments and obtain the appropriate inputs as directed by authentication modules.

In addition to simply slotting into the appropriate pam_conv(3), this function provides some time-out facilities. The function exports five variables that can be used by an application programmer to limit the amount of time this conversation function will spend waiting for the user to type something. The five variables are as follows:

time_t pam_misc_conv_warn_time;
: This variable contains the *time* (as returned by time(2)) that the user should be first warned that the clock is ticking. By default it has the value 0, which indicates that no such warning will be given. The application may set its value to sometime in the future, but this should be done prior to passing control to the *Linux-PAM* library.

const char *pam_misc_conv_warn_line;
: Used in conjunction with `pam_misc_conv_warn_time`, this variable is a pointer to the string that will be displayed when it becomes time to warn the user that the timeout is approaching. Its default value is a translated version of "...Time is running out...", but this can be changed by the application prior to passing control to *Linux-PAM*.

time_t pam_misc_conv_die_time;
: This variable contains the *time* (as returned by time(2)) that the will time out. By default it has the value 0, which indicates that the con-

versation function will not timeout. The application may set its value to sometime in the future, but this should be done prior to passing control to the *Linux-PAM* library.

const char *pam_misc_conv_die_line;
: Used in conjunction with `pam_misc_conv_die_time`, this variable is a pointer to the string that will be displayed when the conversation times out. Its default value is a translated version of "...Sorry, your time is up!", but this can be changed by the application prior to passing control to *Linux-PAM*.

int pam_misc_conv_died;
: Following a return from the *Linux-PAM* library, the value of this variable indicates whether the conversation has timed out. A value of 1 indicates the time-out occurred.

The following two function pointers are available for supporting binary prompts in the conversation function. They are optimized for the current incarnation of the **libpamc** library and are subject to change.

int (*pam_binary_handler_fn)(void *appdata, pamc_bp_t *prompt_p);
: This function pointer is initialized to NULL but can be filled with a function that provides machine-machine (hidden) message exchange. It is intended for use with hidden authentication protocols such as RSA or Diffie-Hellman key exchanges. (This is still under development.)

int (*pam_binary_handler_free)(void *appdata, pamc_bp_t *delete_me);
: This function pointer is initialized to `PAM_BP_RENEW(delete_me, 0, 0)`, but can be redefined as desired by the application.

5.1.2. Transcribing an environment to that of PAM

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

int pam_misc_paste_env(pamh, user);

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

5.1.2.1. DESCRIPTION

This function takes the supplied list of environment pointers and *uploads* its contents to the PAM environment. Success is indicated by PAM_SUCCESS.

5.1.3. Liberating a locally saved environment

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

int pam_misc_drop_env(env);

char **env;
```

5.1.3.1. DESCRIPTION

This function is defined to complement the pam_getenvlist(3) function. It liberates the memory associated with `env`, *overwriting* with *0* all memory before `free()`ing it.

5.1.4. BSD like PAM environment variable setting

#include <security/pam_misc.h>

```
int pam_misc_setenv(pamh, name, value, readonly);

pam_handle_t *pamh;
const char *name;
const char *value;
int readonly;
```

5.1.4.1. DESCRIPTION

This function performs a task equivalent to pam_putenv(3), its syntax is, however, more like the BSD style function; `setenv()`. The *`name`* and *`value`* are concatenated with an '=' to form a name=value and passed to `pam_putenv()`. If, however, the PAM variable is already set, the replacement will only be applied if the last argument, *`readonly`*, is zero.

Chapter 6. Porting legacy applications

The point of PAM is that the application is not supposed to have any idea how the attached authentication modules will choose to authenticate the user. So all they can do is provide a conversation function that will talk directly to the user(client) on the modules' behalf.

Consider the case that you plug a retinal scanner into the login program. In this situation the user would be prompted: "please look into the scanner". No username or password would be needed - all this information could be deduced from the scan and a database lookup. The point is that the retinal scanner is an ideal task for a "module".

While it is true that a pop-daemon program is designed with the POP protocol in mind and no-one ever considered attaching a retinal scanner to it, it is also the case that the "clean" PAM'ification of such a daemon would allow for the possibility of a scanner module being be attached to it. The point being that the "standard" pop-authentication protocol(s) [which will be needed to satisfy inflexible/legacy clients] would be supported by inserting an appropriate pam_qpopper module(s). However, having rewritten **popd** once in this way any new protocols can be implemented in-situ.

One simple test of a ported application would be to insert the **pam_permit** module and see if the application demands you type a password... In such a case, **xlock** would fail to lock the terminal - or would at best be a screen-saver, ftp would give password free access to all etc.. Neither of these is a very secure thing to do, but they do illustrate how much flexibility PAM puts in the hands of the local admin.

The key issue, in doing things correctly, is identifying what is part of the authentication procedure (how many passwords etc..) the exchange protocol (prefixes to prompts etc., numbers like 331 in the case of ftpd) and what is part of the service that the application delivers. PAM really needs to have total control in the authentication "procedure", the conversation function should only deal with reformatting user prompts and extracting responses from raw input.

Chapter 7. Glossary of PAM related terms

The following are a list of terms used within this document.

Authentication token	Generally, this is a password. However, users can authenticate themselves in a variety of ways. Updating the user's authentication token thus corresponds to *refreshing* the object they use to authenticate themselves with the system. The word password is avoided to keep open the possibility that the authentication involves a retinal scan or other non-textual mode of challenge/response.
Credentials	Having successfully authenticated the user, PAM is able to establish certain characteristics/attributes of the user. These are termed *credentials*. Examples of which are group memberships to perform privileged tasks with, and *tickets* in the form of environment variables etc. . Some user-credentials, such as the user's UID and GID (plus default group memberships) are not deemed to be PAM-credentials. It is the responsibility of the application to grant these directly.

Chapter 8. An example application

To get a flavor of the way a *Linux-PAM* application is written we include the following example. It prompts the user for their password and indicates whether their account is valid on the standard output, its return code also indicates the success (0 for success; 1 for failure).

```
/*
  This program was contributed by Shane Watts
  [modifications by AGM and kukuk]

  You need to add the following (or equivalent) to the
  /etc/pam.d/check_user file:
  # check authorization
  auth       required     pam_unix.so
  account    required     pam_unix.so
 */

#include <security/pam_appl.h>
#include <security/pam_misc.h>
#include <stdio.h>

static struct pam_conv conv = {
    misc_conv,
    NULL
};

int main(int argc, char *argv[])
{
    pam_handle_t *pamh=NULL;
    int retval;
    const char *user="nobody";

    if(argc == 2) {
        user = argv[1];
    }

    if(argc > 2) {
        fprintf(stderr, "Usage: check_user [username]\n");
        exit(1);
    }

    retval = pam_start("check_user", user, &conv, &pamh);

    if (retval == PAM_SUCCESS)
        retval = pam_authenticate(pamh, 0);    /* is user really user? */

    if (retval == PAM_SUCCESS)
        retval = pam_acct_mgmt(pamh, 0);       /* permitted access? */

    /* This is where we have been authorized or not. */

    if (retval == PAM_SUCCESS) {
```

```
            fprintf(stdout, "Authenticated\n");
        } else {
            fprintf(stdout, "Not Authenticated\n");
        }

        if (pam_end(pamh,retval) != PAM_SUCCESS) {     /* close Linux-PAM */
            pamh = NULL;
            fprintf(stderr, "check_user: failed to release authenticator\n");
            exit(1);
        }

        return ( retval == PAM_SUCCESS ? 0:1 );       /* indicate success */
    }
```

Chapter 9. Files

/usr/include/security/pam_appl.h	Header file with interfaces for *Linux-PAM* applications.
/usr/include/security/pam_misc.h	Header file for useful library functions for making applications easier to write.

Chapter 10. See also

- The Linux-PAM System Administrators' Guide.
- The Linux-PAM Module Writers' 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 11. 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 12. Copyright information for this document

Copyright (c) 2006 Thorsten Kukuk <kukuk@thkukuk.de>
Copyright (c) 1996-2002 Andrew G. Morgan <morgan@kernel.org>

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, and the entire permission notice in its entirety, including the disclaimer of warranties.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

Alternatively, this product may be distributed under the terms of the GNU General Public License (GPL), in which case the provisions of the GNU GPL are required instead of the above restrictions. (This clause is necessary due to a potential bad interaction between the GNU GPL and the restrictions contained in a BSD-style copyright.)

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH