The Linux-PAM Application Developers' Guide

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

DRAFT v0.76 2001/12/08

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 funtions 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.

Contents

1 Introduction

1.1 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 -ldl
```

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_misc.h>

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

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.

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*).

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 users 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.

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

Below we document those functions in the **Linux-PAM** library that may be called from an application.

3.1.1 Initialization of Linux-PAM

```
extern int pam_start(const char *service_name, const char *user,
                     const struct pam_conv *pam_conversation,
                     pam_handle_t **pamh);
```

This is the first of the **Linux-PAM** functions that must be called by an application. It initializes the interface and reads the system configuration file, `/etc/pam.conf` (see the **Linux-PAM** System Administrators' Guide). Following a successful return (`PAM_SUCCESS`) the contents of `*pamh` is a handle that provides continuity for successive calls to the **Linux-PAM** library. The arguments expected by `pam_start` are as follows: the `service_name` of the program, the `username` of the individual to be authenticated, a pointer to an application-supplied `pam_conv` structure and a pointer to a `pam_handle_t` *pointer*.

The `pam_conv` structure is discussed more fully in the section 3.2.1 (below). The `pam_handle_t` is a *blind* structure and the application should not attempt to probe it directly for information. Instead the **Linux-PAM** library provides the functions `pam_set_item` and `pam_get_item`. These functions are documented below.

3.1.2 Termination of the library

```
extern int pam_end(pam_handle_t *pamh, int pam_status);
```

This function is the last function an application should call in the **Linux-PAM** library. Upon return the handle `pamh` is no longer valid and all memory associated with it will be invalid (likely to cause a segmentation fault if accessed).

Under normal conditions the argument `pam_status` has the value PAM_SUCCESS, but in the event of an unsuccessful application for service the appropriate **Linux-PAM** error-return value should be used here. Note, `pam_end()` unconditionally shuts down the authentication stack associated with the `pamh` handle. The value taken by `pam_status` is used as an argument to the module specific callback functions, `cleanup()` (see the **Linux-PAM** *Module Developers' Guide*). 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.

3.1.3 Setting PAM items

```
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 following **item_type**s:

`PAM_SERVICE`

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

`PAM_USER`

The username of the entity under who's 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 "Please enter username: ".

`PAM_TTY`

The terminal name: prefixed by `/dev/` if it is a device file; for graphical, X-based, applications the value for this item should be the `$DISPLAY` variable.

`PAM_RUSER`

The requesting entity: user's username for a locally requesting user or 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. It should generally match the current `PAM_RHOST` value. That is, `"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. `"luser@localhost"` or `"evil@evilcom.com"` are valid `"PAM_RUSER@PAM_RHOST"` examples. In some applications, `PAM_RHOST` may be NULL. In such situations, it is unclear where the authentication request is originating from.

`PAM_CONV`

The conversation structure (see section 3.2.1 (below)).

`PAM_FAIL_DELAY`

A function pointer to redirect centrally managed failure delays (see section 3.1.6 (below)).

For all `item_types`, 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 (see section 3.2.1 (below)). In the case of `PAM_FAIL_DELAY`, `item` is a function pointer: `void (*delay_fn)(int retval, unsigned usec_delay, void *appdata_ptr)` (see section 3.1.6 (below)).

A successful call to this function returns `PAM_SUCCESS`. However, the application should expect at least one the following errors:

`PAM_SYSTEM_ERR`

The `pam_handle_t` passed as a first argument to this function was invalid.

`PAM_PERM_DENIED`

An attempt was made to replace the conversation structure with a `NULL` value.

`PAM_BUF_ERR`

The function ran out of memory making a copy of the item.

`PAM_BAD_ITEM`

The application attempted to set an undefined or inaccessible item.

3.1.4 Getting PAM items

```
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 indicated `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!

A successful call is signaled by a return value of `PAM_SUCCESS`. However, the application should expect one of the following errors:

`PAM_SYSTEM_ERR`

The `pam_handle_t` passed as a first argument to this function was invalid.

`PAM_PERM_DENIED`

The value of `item` was `NULL`.

`PAM_BAD_ITEM`

The application attempted to set an undefined or inaccessible item.

Note, in the case of an error, the contents of `item` is not modified - that is, it retains its pre-call value. One should take care to initialize this value prior to calling `pam_get_item()`. Since, if its value - despite the `pam_get_item()` function failing - is to be used the consequences are undefined.

3.1.5 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.

3.1.6 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 application 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 relevant **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* (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.

For applications written with a single thread that are event driven in nature, `libpam` 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, **Linux-PAM** supplies the `PAM_FAIL_DELAY` item. It can be queried and set with `pam_get_item()` and `pam_set_item()` 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` (`pam_handle_t`). This last value was set by the application when it called `pam_start` or explicitly with `pam_set_item(... , PAM_CONV, ...)`. Note, if `PAM_FAIL_DELAY` is unset (or set to `NULL`), then `libpam` will perform any delay.

3.1.7 Authenticating the user

```
extern int pam_authenticate(pam_handle_t *pamh, int flags);
```

This function serves as an interface to the authentication mechanisms of the loaded modules. The single *optional* flag, which may be logically OR'd with `PAM_SILENT`, takes the following value,

`PAM_DISALLOW_NULL_AUTHTOK`

Instruct the authentication modules to return `PAM_AUTH_ERR` if the user does not have a registered authorization token—it is set to `NULL` in the system database.

The value returned by this function is 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.

If one or more of the authentication modules fails to load, for whatever reason, this function will return `PAM_ABORT`.

3.1.8 Setting user credentials

```
    extern int pam_setcred(pam_handle_t *pamh, int flags);
```

This function is used to set the module-specific credentials of the user. It is usually called after the user has been authenticated, after the account management function has been called but before a session has been opened for the user.

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 (or UN*X 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.

This function simply calls the `pam_sm_setcred` functions of each of the loaded modules. Valid `flags`, any 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.

A successful return is signalled with `PAM_SUCCESS`. Errors that are especially relevant to this function are the following:

`PAM_CRED_UNAVAIL`

A 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 an authentication module.

`PAM_CRED_ERR`

A module was unable to set the credentials of the user.

3.1.9 Account management

```
    extern int pam_acct_mgmt(pam_handle_t *pamh, int flags);
```

This function is typically called after the user has been authenticated. It establishes whether the user's account is healthy. That is to say, whether the user's account is still active and whether the user is permitted to gain access to the system at this time. Valid flags, any one of which, may be logically OR'd with `PAM_SILENT`, and are the same as those applicable to the `flags` argument of `pam_authenticate`.

This function simply calls the corresponding functions of each of the loaded modules, as instructed by the configuration file, `/etc/pam.conf`.

The normal response from this function is `PAM_SUCCESS`, however, specific failures are indicated by the following error returns:

`PAM_AUTHTOKEN_REQD`

The user **is** valid but their authentication token has *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_ACCT_EXPIRED`

The user is no longer permitted to access the system.

`PAM_AUTH_ERR`

There was an authentication error.

`PAM_PERM_DENIED`

The user is not permitted to gain access at this time.

`PAM_USER_UNKNOWN`

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

3.1.10 Updating authentication tokens

```
extern int pam_chauthtok(pam_handle_t *pamh, const int flags);
```

This function is used to change the authentication token for a given user (as indicated by the state associated with the handle, `pamh`). The following is a valid but optional flag which may be logically OR'd with `PAM_SILENT`,

`PAM_CHANGE_EXPIRED_AUTHTOK`

This argument indicates to the modules that the users authentication token (password) should only be changed if it has expired.

Note, if this argument is not passed, the application requires that *all* authentication tokens are to be changed.

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

`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_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`

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

3.1.11 Session initialization

```
extern int pam_open_session(pam_handle_t *pamh, int flags);
```

This function is used to indicate that an authenticated session has begun. It is used to inform the modules that the user is currently in a session. It should be possible for the **Linux-PAM** library to open a session and close the same session (see section 3.1.12 (below)) from different applications.

Currently, this function simply calls each of the corresponding functions of the loaded modules. The only valid flag is `PAM_SILENT` and this is, of course, *optional*.

If any of the *required* loaded modules are unable to open a session for the user, this function will return `PAM_SESSION_ERR`.

3.1.12 Terminating sessions

```
extern int pam_close_session(pam_handle_t *pamh, int flags);
```

This function is used to indicate that an authenticated session has ended. It is used to inform the modules that the user is exiting a session. It should be possible for the **Linux-PAM** library to open a session and close the same session from different applications.

This function simply calls each of the corresponding functions of the loaded modules in the same order that they were invoked with `pam_open_session()`. The only valid flag is `PAM_SILENT` and this is, of course, *optional*.

If any of the *required* loaded modules are unable to close a session for the user, this function will return `PAM_SESSION_ERR`.

3.1.13 Setting PAM environment variables

The `libpam` library associates with each PAM-handle (`pamh`), a set of *PAM environment variables*. These variables are intended to hold the session environment variables that the user will inherit when the session is granted and the authenticated user obtains access to the requested service. For example, when `login` has finally given the user a shell, the environment (as viewed with the command `env`) will be what `libpam` was maintaining as the PAM environment for that service application. Note, these variables are not the environment variables of the `login` application. This is principally for two reasons: `login` may want to have an environment that cannot be seen or manipulated by a user; and `login` (or whatever the serving application is) may be maintaining a number of parallel sessions, via different `pamh` values, at the same time and a single environment may not be appropriately shared between each of these. The PAM environment may contain variables seeded by the applicant user's client program, for example, and as such it is not appropriate for one applicant to interfere with the environment of another applicant.

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

This function attempts to (re)set a **Linux-PAM** environment variable. 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 **Linux-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 **Linux-PAM** environment.

Success is indicated with a return value of `PAM_SUCCESS`. Failure is indicated by one of the following returns:

`PAM_PERM_DENIED`

name given is a `NULL` pointer

`PAM_BAD_ITEM`

variable requested (for deletion) is not currently set

`PAM_ABORT`

the **Linux-PAM** handle, `pamh`, is corrupt

`PAM_BUF_ERR`

failed to allocate memory when attempting update

3.1.14 Getting a PAM environment variable

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

Obtain the value of the indicated **Linux-PAM** environment variable. On error, internal failure or the unavailability of the given variable (unspecified), this function simply returns `NULL`.

3.1.15 Getting the PAM environment

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

The PAM environment variables (see section 3.1.13 (above)) are a complete set of enviroment variables that are associated with a PAM-handle (`pamh`). They represent the contents of the *regular* environment variables of the authenticated user when service is granted.

Th function, `pam_getenvlist()` returns a pointer to a complete, `malloc()`'d, copy of the PAM environment. It is a pointer to a duplicated list of environment variables. 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.

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 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.

3.2 What is expected of an application

3.2.1 The conversation function

An application must provide a "conversation function". It is used for direct communication between a loaded module and the application and will typically provide a means for the module to prompt the user for a password etc. . The structure, `pam_conv`, is defined by including `<security/pam_appl.h>`; to be,

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

It is initialized by the application before it is passed to the library. The *contents* of this structure are attached to the `*pamh` handle. The point of this argument is to provide a mechanism for any loaded module to interact directly with the application program. This is why it is called a *conversation* structure.

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. Note, `*resp` is an `struct pam_response` array and *not* an array of pointers.

The message (from the module to the application) passing structure is defined by `<security/pam_appl.h>` as:

```
struct pam_message {
    int msg_style;
    const char *msg;
};
```

Valid choices for `msg_style` are:

`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

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 descrepency between the way Linux-PAM handles the `const struct pam_message **msg` conversation function argument from the way that Solaris' PAM (and derivitives, 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_meg` read only 'struct pam_message' *pointers*. Solaris' PAM implementation interprets this argument as a pointer to a pointer to an array of `num_meg` `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])
```

The response (from the application to the module) passing structure is defined by including `<security/pam_appl.h>` as:

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

Currently, there are no definitions for `resp_retcode` values; the normal value is `0`.

Prior to the 0.59 release of Linux-PAM, the length of the returned `pam_response` array was equal to the number of *prompts* (types `PAM_PROMPT_ECHO_OFF` and `PAM_PROMPT_ECHO_ON`) in the `pam_message` array with which the conversation function was called. This meant that it was not always necessary for the module to `free(3)` the responses if the conversation function was only used to display some text.

Post Linux-PAM-0.59. The number of responses is always equal to the `num_msg` conversation function argument. This is slightly easier to program but 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.

The maximum length of the `pam_msg.msg` and `pam_response.resp` character strings is `PAM_MAX_MSG_SIZE`. (This is not enforced by Linux-PAM.)

`PAM_SUCCESS` is the expected return value of this function. However, should an error occur the application should not set `*resp` but simply return `PAM_CONV_ERR`.

Note, if an application wishes to use two conversation functions, it should activate the second with a call to `pam_set_item()`.

Notes: New item types are being added to the conversation protocol. Currently Linux-PAM supports: `PAM_BINARY_PROMPT` and `PAM_BINARY_MSG`. These two are intended for server-client hidden information exchange and may be used as an interface for maching-machine authentication.

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.

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 5 (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` (userid) of the running process; the identity of the privilege granting user is the `euid` (effective userid) 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: `<unset>` = 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 mannor that preserves the PAM API. Such tolerance may be especially important when attempting to track a malicious adversary.

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 functions for allocating memory (securely), a text based conversation function, and routines for enhancing the standard PAM-environment variable support.

5.1 Requirements

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 10 (the bibliography)) below.

5.2 Macros supplied

5.2.1 Safe duplication of strings

```
x_strdup(const char *s)
```

This macro is a replacement for the `xstrdup()` function that was present in earlier versions of the library and which clashed horribly with a number of applications. It returns a duplicate copy of the `NUL` terminated string, `s`. `NULL` is returned if there is insufficient memory available for the duplicate or if `s` is `NULL` to begin with.

5.3 Functions supplied

5.3.1 A text based conversation function

```
extern int misc_conv(int num_msg, const struct pam_message **msgm,
                     struct pam_response **response, void *appdata_ptr);
```

This is a function that 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 `struct pam_conv`, 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:

`extern time_t pam_misc_conv_warn_time;`

This variable contains the *time* (as returned by `time()`) 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.

`extern const char *pam_misc_conv_warn_line;`

Used in conjuction 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.Time is running out...\n", but this can be changed by the application prior to passing control to **Linux-PAM**.

`extern time_t pam_misc_conv_die_time;`

This variable contains the *time* (as returned by `time()`) that the conversation will time out. By default it has the value `0`, which indicates that the conversation function will not timeout. The application may set its value to sometime in the future, this should be done prior to passing control to the **Linux-PAM** library.

`extern const char *pam_misc_conv_die_line;`

Used in conjuction 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.Sorry, your time is up!\n", but this can be changed by the application prior to passing control to **Linux-PAM**.

`extern int pam_misc_conv_died;`

Following a return from the **Linux-PAM** libraray, 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.

`extern 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.)

`extern 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.

5.3.2 Transcribing an environment to that of Linux-PAM

```
extern int pam_misc_paste_env(pam_handle_t *pamh,
                              const char * const * user_env);
```

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

5.3.3 Liberating a locally saved environment

```
extern char **pam_misc_drop_env(char **env);
```

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

5.3.4 BSD like Linux-PAM environment variable setting

```
extern int pam_misc_setenv(pam_handle_t *pamh, const char *name,
                           const char *value, int readonly);
```

This function performs a task equivalent to `pam_putenv()`, 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 **Linux-PAM** variable is already set, the replacement will only be applied if the last argument, `readonly`, is zero.

6 Porting legacy applications

The following is extracted from an email. I'll tidy it up later.

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.

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, a user can authenticate him/herself in a variety of ways. Updating the user's authentication token thus corresponds to *refreshing* the object they use to authenticate themself 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.

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]

  You need to add the following (or equivalent) to the /etc/pam.conf file.
  # check authorization
  check_user   auth       required     /usr/lib/security/pam_unix_auth.so
  check_user   account    required     /usr/lib/security/pam_unix_acct.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 */
    }
```

9 Files

`/usr/include/security/pam_appl.h`

header file for **Linux-PAM** applications interface

`/usr/include/security/pam_misc.h`

header file for useful library functions for making applications easier to write

`/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.

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.

11 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! :)

- `pam_strerror()` should be internationalized....
- Note, the `resp_retcode` of struct `pam_message`, has no purpose at the moment. Ideas/suggestions welcome!
- more security issues are required....

12 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**.

13 Bugs/omissions

This manual is hopelessly unfinished. Only a partial list of people is credited for all the good work they have done.

14 Copyright information for this document

`$Id:  pam_appl.sgml,v 1.9 2002/05/10 05:25:52 agmorgan Exp $`