PAM working group
Internet Draft:
Document: draft-morgan-pam-07.txt
Expires: June 13, 2000
Obsoletes: draft-morgan-pam-06.txt

A.G. Morgan
October 6, 1999

Pluggable Authentication Modules

1 Status of this memo

This document is an draft specification. The latest version of this draft may be obtained from here:

http://linux.kernel.org/pub/linux/libs/pam/pre/doc/

As

Linux-PAM-'version'-docs.tar.gz

It is also contained in the Linux-PAM tar ball.

2 Abstract

This document is concerned with the definition of a general infrastructure for module based authentication. The infrastructure is named Pluggable Authentication Modules (PAM for short).

3 Introduction

Computers are tools. They provide services to people and other computers (collectively we shall call these _users_ entities). In order to provide convenient, reliable and individual service to different entities, it is common for entities to be labelled. Having defined a label as referring to a some specific entity, the label is used for the purpose of protecting and allocating data resources.

All modern operating systems have a notion of labelled entities and all modern operating systems face a common problem: how to authenticate the association of a predefined label with applicant entities.

There are as many authentication methods as one might care to count. None of them are perfect and none of them are invulnerable. In general, any given authentication method becomes weaker over time. It is common then for new authentication methods to be developed in response to newly discovered weaknesses in the old authentication methods.

The problem with inventing new authentication methods is the fact that old applications do not support them. This contributes to an inertia that discourages the overhaul of weakly protected systems. Another problem is that individuals (people) are frequently powerless to layer the protective authentication around their systems. They are forced to rely on single (lowest common denominator) authentication schemes even in situations where this is far from appropriate.

PAM, as discussed in this document, is a generalization of the

approach first introduced in [1]. In short, it is a general framework of interfaces that abstract the process of authentication. With PAM, a service provider can custom protect individual services to the level that they deem is appropriate.

PAM has nothing explicit to say about transport layer encryption. Within the context of this document encryption and/or compression of data exchanges are application specific (strictly between client and server) and orthogonal to the process of authentication.

4 Definitions

Here we pose the authentication problem as one of configuring defined interfaces between two entities.

4.1 Players in the authentication process

PAM reserves the following words to specify unique entities in the authentication process:

applicant
: the entity (user) initiating an application for service [PAM associates the PAM_RUSER _item_ with this requesting user].

arbitrator
: the entity (user) under whose identity the service application is negotiated and with whose authority service is granted.

user
: the entity (user) whose identity is being authenticated [PAM associates the PAM_USER _item_ with this identity].

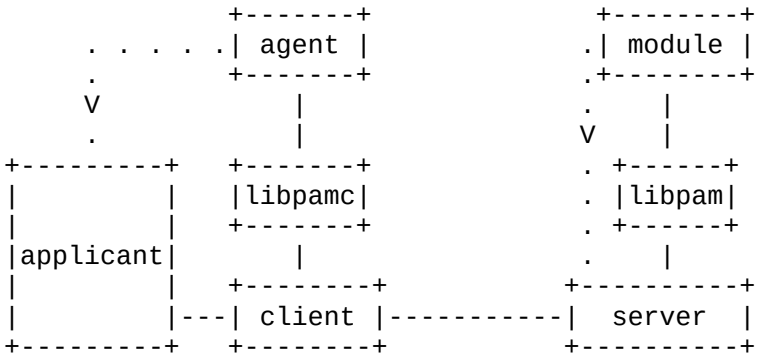
server
: the application that provides service, or acts as an authenticated gateway to the requested service. This application is completely responsible for the server end of the transport layer connecting the server to the client. PAM makes no assumptions about how data is encapsulated for exchanges between the server and the client, only that full octet sequences can be freely exchanged without corruption.

client
: application providing the direct/primary interface to applicant. This application is completely responsible for the client end of the transport layer connecting the server to the client. PAM makes no assumptions about how data is encapsulated for exchanges between the server and the client, only that full octet sequences can be freely exchanged without corruption.

module
: authentication binary that provides server-side support for some (arbitrary) authentication method.

agent
: authentication binary that provides client-side support for some (arbitrary) authentication method.

Here is a diagram to help orient the reader:



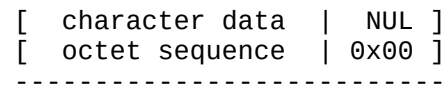
Solid lines connecting the boxes represent two-way interaction. The dotted-directed lines indicate an optional connection beteween the plugin module (agent) and the server (applicant). In the case of the module, this represents the module invoking the 'conversation' callback function provided to libpam by the server application when it inititializes the libpam library. In the case of the agent, this may be some out-of-PAM API interaction (for example directly displaying a dialog box under X).

4.2 Defined Data Types

In this draft, we define two composite data types, the text string and the binary prompt. They are the data types used to communicate authentication requests and responses.

4.2.1 text string

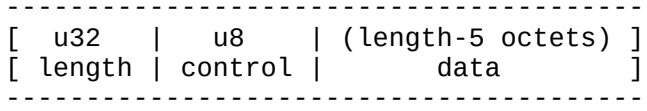
The text string is a simple sequence of non-NUL (NUL = 0x00) octets. Terminated with a single NUL (0x00) octet. The character set employed in the octet sequence may be negotiated out of band, but defaults to utf-8.



Within the rest of this text, PAM text strings are delimited with a pair of double quotes. Example, "this" = {'t';'h';'i';'s';0x00}.

4.2.2 binary prompt

A binary prompt consists of a stream of octets arranged as follows:



That is, a 32-bit unsigned integer in network byte order, a single unsigned byte of control information and a sequence of octets of

length (length-5). The composition of the _data_ is context dependent but is generally not a concern for either the server or the client. It is very much the concern of modules and agents.

For purposes of interoperability, we define the following control characters as legal.

value	symbol	description
0x01	PAM_BPC_OK	- continuation packet
0x02	PAM_BPC_SELECT	- initialization packet
0x03	PAM_BPC_DONE	- termination packet
0x04	PAM_BPC_FAIL	- unable to execute

The following control characters are only legal for exchanges between an agent and a client (it is the responsibility of the client to enforce this rule in the face of a rogue server):

0x41	PAM_BPC_GETENV	- obtain client env.var
0x42	PAM_BPC_PUTENV	- set client env.var
0x43	PAM_BPC_TEXT	- display message
0x44	PAM_BPC_ERROR	- display error message
0x45	PAM_BPC_PROMPT	- echo'd text prompt
0x46	PAM_BPC_PASS	- non-echo'd text prompt

Note, length is always equal to the total length of the binary prompt and represented by a network ordered unsigned 32 bit integer.

4.2.2.1 PAM_BPC_SELECT binary prompts

Binary prompts of control type PAM_BPC_SELECT have a defined data part. It is composed of three elements:

```
{agent_id;'/';data}
```

The agent_id is a sequence of characters satisfying the following regexp:

```
/^[a-z0-9\_]+(@[a-z0-9\_.]+)?$/
```

and has a specific form for each independent agent.

o Agent_ids that do not contain an at-sign (@) are reserved to be assigned by IANA (Internet Assigned Numbers Authority). Names of this format MUST NOT be used without first registering with IANA. Registered names MUST NOT contain an at-sign (@).

o Anyone can define additional agents by using names in the format name@domainname, e.g. "ouragent@example.com". The part following the at-sign MUST be a valid fully qualified internet domain name [RFC-1034] controlled by the person or organization defining the name. (Said another way, if you control the email address that your agent has as an identifier, they you are entitled to use this identifier.) It is up to each domain how it manages its local namespace.

The '/' character is a mandatory delimiter, indicating the end of the

agent_id. The trailing data is of a format specific to the agent with the given agent_id.

4.3 Special cases

In a previous section (4.1) we identified the most general selection of authentication participants. In the case of network authentication, it is straightforward to ascribe identities to the defined participants. However, there are also special (less general) cases that we recognize here.

The primary authentication step, when a user is directly introduced into a computer system (log’s on to a workstation) is a special case. In this situation, the client and the server are generally one application. Before authenticating such a user, the applicant is formally unknown: PAM_RUSER is NULL.

Some client-server implementations (telnet for example) provide effective full tty connections. In these cases, the four simple text string prompting cases (see below) can be handled as in the primary login step. In other words, the server absorbs most of the overhead of propagating authentication messages. In these cases, there is special client/server support for handling binary prompts.

5 Defined interfaces for information flow

Here, we discuss the information exchange interfaces between the players in the authentication process. It should be understood that the server side is responsible for driving the authentication of the applicant. Notably, every request received by the client from the server must be matched with a single response from the client to the server.

5.1 Applicant <-> client

Once the client is invoked, requests to the applicant entity are initiated by the client application. General clients are able to make the following requests directly to an applicant:

   echo text string
   echo error text string
   prompt with text string for echo’d text string input
   prompt with text string for concealed text string input

the nature of the interface provided by the client for the benefit of the applicant entity is client specific and not defined by PAM.

5.2 Client <-> agent

In general, authentication schemes require more modes of exchange than the four defined in the previous section (5.1). This provides a role for client-loadable agents. The client and agent exchange binary-messages that can have one of the following forms:

   client -> agent
    binary prompt agent expecting binary prompt reply to client

```
   agent -> client
    binary prompt reply from agent to clients binary prompt
```

Following the acceptance of a binary prompt by the agent, the agent may attempt to exchange information with the client before returning its binary prompt reply. Permitted exchanges are binary prompts of the following types:

```
   agent -> client
    set environment variable (A)
    get environment variable (B)
    echo text string (C)
    echo error text string (D)
    prompt for echo'd text string input (E)
    prompt for concealed text string input (F)
```

In response to these prompts, the client must legitimately respond with a corresponding binary prompt reply. We list a complete set of example exchanges, including each type of legitimate response (passes and a single fail):

Type	Agent request	Client response
(A)	{13;PAM_BPC_PUTENV;"FOO=BAR"}	{5;PAM_BPC_OK;}
	{10;PAM_BPC_PUTENV;"FOO="}	{5;PAM_BPC_OK;}
	{9;PAM_BPC_PUTENV;"FOO"} (*)	{5;PAM_BPC_OK;}
	{9;PAM_BPC_PUTENV;"BAR"} (*)	{5;PAM_BPC_FAIL;}
(B)	{10;PAM_BPC_GETENV;"TERM"}	{11;PAM_BPC_OK;"vt100"}
	{9;PAM_BPC_GETENV;"FOO"}	{5;PAM_BPC_FAIL;}
(C)	{12;PAM_BPC_TEXT;"hello!"}	{5;PAM_BPC_OK;}
	{12;PAM_BPC_TEXT;"hello!"}	{5;PAM_BPC_FAIL;}
(D)	{11;PAM_BPC_TEXT;"ouch!"}	{5;PAM_BPC_OK;}
	{11;PAM_BPC_TEXT;"ouch!"}	{5;PAM_BPC_FAIL;}
(E)	{13;PAM_BPC_PROMPT;"login: "}	{9;PAM_BPC_OK;"joe"}
	{13;PAM_BPC_PROMPT;"login: "}	{6;PAM_BPC_OK;""}
	{13;PAM_BPC_PROMPT;"login: "}	{5;PAM_BPC_FAIL;}
(F)	{16;PAM_BPC_PASS;"password: "}	{9;PAM_BPC_OK;"XYZ"}
	{16;PAM_BPC_PASS;"password: "}	{6;PAM_BPC_OK;""}
	{16;PAM_BPC_PASS;"password: "}	{5;PAM_BPC_FAIL;}

(*) Used to attempt the removal of a pre-existing environment variable.

5.3 Client <-> server

Once the client has established a connection with the server (the nature of the transport protocol is not specified by PAM), the server is responsible for driving the authentication process.

General servers can request the following from the client:

(to be forwarded by the client to the applicant)
 echo text string
 echo error text string
 prompt for echo'd text string response
 prompt for concealed text string response

(to be forwarded by the client to the appropriate agent)
 binary prompt for a binary prompt response

Client side agents are required to process binary prompts. The agents' binary prompt responses are returned to the server.

5.4 Server <-> module

Modules drive the authentication process. The server provides a conversation function with which it encapsulates module-generated requests and exchanges them with the client. Every message sent by a module should be acknowledged.

General conversation functions can support the following five conversation requests:

- echo text string
- echo error string
- prompt for echo'd text string response
- prompt for concealed text string response
- binary prompt for binary prompt response

The server is responsible for redirecting these requests to the client.

6 C API for application interfaces (client and server)

6.1 Applicant <-> client

No API is defined for this interface. The interface is considered to be specific to the client application. Example applications include terminal login, (X)windows login, machine file transfer applications.

All that is important is that the client application is able to present the applicant with textual output and to receive textual input from the applicant. The forms of textual exchange are listed in an earlier section (5.1). Other methods of data input/output are better suited to being handled via an authentication agent.

6.2 Client <-> agent

The client makes use of a general API for communicating with agents. The client is not required to communicate directly with available agents, instead a layer of abstraction (in the form of a library: libpamc) takes care of loading and maintaining communication with all requested agents. This layer of abstraction will choose which agents to interact with based on the content of binary prompts it receives that have the control type PAM_BPC_SELECT.

6.2.1 Client <-> libpamc

6.2.1.1 Compilation information

The C-header file provided for client-agent abstraction is included with the following source line:

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

The library providing the corresponding client-agent abstraction functions is, libpamc.

```
cc .... -lpamc
```

6.2.1.2 Initializing libpamc

The libpamc library is initialized with a call to the following function:

```
pamc_handle_t pamc_start(void);
```

This function is responsible for configuring the library and registering the location of available agents. The location of the available agents on the system is implementation specific.

pamc_start() function returns NULL on failure. Otherwise, the return value is a pointer to an opaque data type which provides a handle to the libpamc library. On systems where threading is available, the libpamc libraray is thread safe provided a single (pamc_handler_t *) is used by each thread.

6.2.1.3 Client (Applicant) selection of agents

For the purpose of applicant and client review of available agents, the following function is provided.

```
char **pamc_list_agents(pamc_handle_t pch);
```

This returns a list of pointers to the agent_id’s of the agents which are available on the system. The list is terminated by a NULL pointer. It is the clients responsibility to free this memory area by calling free() on each agent id and the block of agent_id pointers in the result.

PAM represents a server-driven authentication model, so by default any available agent may be invoked in the authentication process.

6.2.1.3.1 Client demands agent

If the client requires that a specific authentication agent is satisfied during the authentication process, then the client should call the following function, immediately after obtaining a pamc_handle_t from pamc_start().

```
int pamc_load(pamc_handle_t pch, const char *agent_id);
```

agent_id is a PAM text string (see section 4.2.2.1) and is not suffixed with a ’/’ delimiter. The return value for this function is:

```
    PAM_BPC_TRUE    - agent located and loaded.
    PAM_BPC_FALSE   - agent is not available.
```

Note, although the agent is loaded, no data is fed to it. The agent's opportunity to inform the client that it does not trust the server is when the agent is shutdown.

6.2.1.3.2 Client marks agent as unusable

The applicant might prefer that a named agent is marked as not available. To do this, the client would invoke the following function immediately after obtaining a pamc_handle_t from pam_start().

```
    int pamc_disable(pamc_handle_t pch, const char *agent_id);
```

here agent_id is a PAM text string containing an agent_id (section 4.2.2.1).

The return value for this function is:

```
    PAM_BPC_TRUE    - agent is disabled. This is the response
                      independent of whether the agent is locally
                      available.

    PAM_BPC_FALSE   - agent cannot be disabled (this may be because
                      it has already been invoked).
```

6.2.1.4 Allocating and manipulating binary prompts

All conversation between an client and an agent takes place with respect to binary prompts. A binary prompt (see section 4.2.2), is obtained, resized and deleted via the following C-macro:

CREATION of a binary prompt with control X1 and data length Y1:

```
    pamc_bp_t prompt = NULL;
    PAM_BP_RENEW(&prompt, X1, Y1);
```

REPLACEMENT of a binary prompt with a control X2 and data length Y2:

```
    PAM_BP_RENEW(&prompt, X2, Y2);
```

DELETION of a binary prompt (the referenced prompt is scrubbed):

```
    PAM_BP_RENEW(&prompt, 0, 0);
```

Note, the PAM_BP_RENEW macro always overwrites any prompt that you call it with, deleting and liberating the old contents in a secure fashion. Also note that PAM_BP_RENEW, when returning a prompt of data size Y1>0, will always append a '\0' byte to the end of the prompt (at data offset Y1). It is thus, by definition, acceptable to treat the data contents of a binary packet as a text string (see 4.2.1).

FILLING a binary prompt from a memory pointer U1 from offset O1 of length L1:

```
   PAM_BP_FILL(prompt, O1, L1, U1);
```

the CONTROL type for the packet can be obtained as follows:

```
   control = PAM_PB_CONTROL(prompt);
```

the LENGTH of a data within the prompt (_excluding_ its header information) can be obtained as follows:

```
   length = PAM_BP_LENGTH(prompt);
```

the total SIZE of the prompt (_including_ its header information) can be obtained as follows:

```
       size = PAM_BP_SIZE(prompt);
```

EXTRACTING data from a binary prompt from offset O2 of length L2 to a memory pointer U2:

```
   PAM_BP_EXTRACT(prompt, O2, L2, U2);
```

If you require direct access to the raw prompt DATA, you should use the following macro:

```
   __u8 *raw_data = PAM_BP_DATA(prompt);
```

6.2.1.5 Client<->agent conversations

All exchanges of binary prompts with agents are handled with the single function:

```
   int pamc_converse(pamc_handle_t *pch, pamc_bp_t *prompt_p);
```

The return value for pamc_converse(...) is PAM_BPC_TRUE when there is a response packet and PAM_BPC_FALSE when the client is unable to handle the request represented by the original prompt. In this latter case, *prompt_p is set to NULL.

This function takes a binary prompt and returns a replacement binary prompt that is either a request from an agent to be acted upon by the client or the 'result' which should be forwarded to the server. In the former case, the following macro will return 1 (PAM_BPC_TRUE) and in all other cases, 0 (PAM_BPC_FALSE):

```
   PAM_BPC_FOR_CLIENT(/* pamc_bp_t */ prompt)
```

Note, all non-NULL binary prompts returned by pamc_converse(...), are terminated with a '\0', even when the full length of the prompt (as returned by the agent) does not contain this delimiter. This is a defined property of the PAM_BP_RENEW macro, and can be relied upon.

Important security note: in certain implementations, agents are implemented by executable binaries, which are transparently loaded and managed by the PAM client library. To ensure there is never a leakage of elevated privilege to an unprivileged agent, the client application should go to some effort to lower its level of privilege. It remains the responsibility of the applicant and the client to ensure that it

is not compromised by a rogue agent.

6.2.1.6 Termination of agents

When closing the authentication session and severing the connection between a client and a selection of agents, the following function is used:

```
int pamc_end(pamc_handle_t *pch);
```

Following a call to pamc_end, the pamc_handle_t will be invalid.

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

PAM_BPC_TRUE - all invoked agents are content with authentication (the server is _not_ judged _un_trustworthy by any agent)

PAM_BPC_FALSE - one or more agents were unsatisfied at being terminated. In general, the client should terminate its connection to the server and indicate to the applicant that the server is untrusted.

6.2.2 libpamc <-> agents

The agents are manipulated from within libpamc. Each agent is an executable in its own right. This permits the agent to have access to sensitive data not accessible directly from the client. The mode of communication between libpamc and an agent is through a pair of pipes. The agent reads binary prompts (section 4.2.2) through its standard input file descriptor and writes response (to the server) binary prompts and instruction binary prompts (instructions for the client) through its standard output file descriptor.

6.3 Client <-> server

This interface is concerned with the exchange of text and binary prompts between the client application and the server application. No API is provided for this as it is considered specific to the transport protocol shared by the client and the server.

6.4 Server <-> modules

The server makes use of a general API for communicating with modules. The client is not required to communicate directly with available modules. By abstracting the authentication interface, it becomes possible for the local administrator to make a run time decision about the authentication method adopted by the server.

6.4.1 Functions and definitions available to servers and modules

[This section will document the following functions

```
pam_set_item()
pam_get_item()
pam_fail_delay(pam_handle_t *pamh, unsigned int micro_sec)
```

```
    pam_get_env(pam_handle_t *pamh, const char *varname)
    pam_strerror(pam_handle_t *pamh, int pam_errno)
]
```

6.4.2 Server <-> libpam

[This section will document the following pam_ calls:

```
    pam_start
    pam_end
    pam_authenticate (*)
    pam_setcred
    pam_acct_mgmt
    pam_open_session
    pam_close_session
    pam_chauthtok (*)
```

The asterisked functions may return PAM_INCOMPLETE. In such cases, the application should be aware that the conversation function was called and that it returned PAM_CONV_AGAIN to a module. The correct action for the application to take in response to receiving PAM_INCOMPLETE, is to acquire the replies so that the next time the conversation function is called it will be able to provide the desired responses. And then recall pam_authenticate (pam_chauthtok) with the same arguments. Libpam will arrange that the module stack is resumed from the module that returned before. This functionality is required for programs whose user interface is maintained by an event loop.]

6.4.3 libpam <-> modules

[This section will document the following pam_ and pam_sm_ calls:

functions provided by libpam

```
    pam_set_data
    pam_get_data
```

functions provided to libpam by each module

```
  groups:
    AUTHENTICATION
          pam_sm_authenticate
          pam_sm_setcred
    ACCOUNT
          pam_sm_acct_mgmt
    SESSION
          pam_sm_open_session
          pam_sm_close_session
    AUTHENTICATION TOKEN MANAGEMENT
          pam_sm_chauthtok
]
```

7 Security considerations

This document is devoted to standardizing authentication infrastructure: everything in this document has implications for security.

8 Contact

The email list for discussing issues related to this document is <pam-list@redhat.com>.

9 References

[1] OSF RFC 86.0, "Unified Login with Pluggable Authentication Modules (PAM)", October 1995

10 Author’s Address

Andrew G. Morgan
Email: morgan@ftp.kernel.org

$Id: draft-morgan-pam.raw,v 1.8 1999/12/14 06:31:57 morgan Exp $