# Pluggable Authentication Modules

Dag-Erling Smørgrav

$FreeBSD: doc/en_US.ISO8859-1/articles/pam/article.sgml,v 1.29 2006/02/07 21:32:42 joel Exp $

This article was written for the FreeBSD Project by ThinkSec AS and Network Associates Laboratories, the Security Research Division of Network Associates, Inc. under DARPA/SPAWAR contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS research program.

This article describes the underlying principles and mechanisms of the Pluggable Authentication Modules (PAM) library, and explains how to configure PAM, how to integrate PAM into applications, and how to write PAM modules.

## Table of Contents

## 1. Introduction

The Pluggable Authentication Modules (PAM) library is a generalized API for authentication-related services which allows a system administrator to add new authentication methods simply by installing new PAM modules, and to modify authentication policies by editing configuration files.

PAM was defined and developed in 1995 by Vipin Samar and Charlie Lai of Sun Microsystems, and has not changed much since. In 1997, the Open Group published the X/Open Single Sign-on (XSSO) preliminary specification, which standardized the PAM API and added extensions for single (or rather integrated) sign-on. At the time of this writing, this specification has not yet been adopted as a standard.

Although this article focuses primarily on FreeBSD 5.x, which uses OpenPAM, it should be equally applicable to FreeBSD 4.x, which uses Linux-PAM, and other operating systems such as Linux and Solaris™.

## 2. Terms and conventions

### 2.1. Definitions

The terminology surrounding PAM is rather confused. Neither Samar and Lai's original paper nor the XSSO specification made any attempt at formally defining terms for the various actors and entities involved in PAM, and the terms that they do use (but do not define) are sometimes misleading and ambiguous. The first attempt at establishing a consistent and unambiguous terminology was a whitepaper written by Andrew G. Morgan (author of Linux-PAM) in 1999. While Morgan's choice of terminology was a huge leap forward, it is in this author's opinion by no means perfect. What follows is an attempt, heavily inspired by Morgan, to define precise and unambiguous terms for all actors and entities involved in PAM.

**account**

The set of credentials the applicant is requesting from the arbitrator.

**applicant**

The user or entity requesting authentication.

**arbitrator**

The user or entity who has the privileges necessary to verify the applicant's credentials and the authority to grant or deny the request.

**chain**

A sequence of modules that will be invoked in response to a PAM request. The chain includes information about the order in which to invoke the modules, what arguments to pass to them, and how to interpret the results.

### client

The application responsible for initiating an authentication request on behalf of the applicant and for obtaining the necessary authentication information from him.

### facility

One of the four basic groups of functionality provided by PAM: authentication, account management, session management and authentication token update.

### module

A collection of one or more related functions implementing a particular authentication facility, gathered into a single (normally dynamically loadable) binary file and identified by a single name.

### policy

The complete set of configuration statements describing how to handle PAM requests for a particular service. A policy normally consists of four chains, one for each facility, though some services do not use all four facilities.

### server

The application acting on behalf of the arbitrator to converse with the client, retrieve authentication information, verify the applicant's credentials and grant or deny requests.

### service

A class of servers providing similar or related functionality and requiring similar authentication. PAM policies are defined on a per-service basis, so all servers that claim the same service name will be subject to the same policy.

### session

The context within which service is rendered to the applicant by the server. One of PAM's four facilities, session management, is concerned exclusively with setting up and tearing down this context.

### token

A chunk of information associated with the account, such as a password or passphrase, which the applicant must provide to prove his identity.

**transaction**

A sequence of requests from the same applicant to the same instance of the same server, beginning with authentication and session set-up and ending with session tear-down.

## 2.2. Usage examples

This section aims to illustrate the meanings of some of the terms defined above by way of a handful of simple examples.

### 2.2.1. Client and server are one

This simple example shows `alice` su(1)'ing to `root`.

```
% whoami
alice
% ls -l `which su`
-r-sr-xr-x  1 root  wheel  10744 Dec  6 19:06 /usr/bin/su
% su -
Password: xi3kiune
# whoami
root
```

- The applicant is `alice`.
- The account is `root`.
- The su(1) process is both client and server.
- The authentication token is `xi3kiune`.
- The arbitrator is `root`, which is why su(1) is setuid `root`.

### 2.2.2. Client and server are separate

The example below shows `eve` try to initiate an ssh(1) connection to `login.example.com`, ask to log in as `bob`, and succeed. Bob should have chosen a better password!

```
% whoami
eve
% ssh bob@login.example.com
bob@login.example.com's password: god
Last login: Thu Oct 11 09:52:57 2001 from 192.168.0.1
Copyright (c) 1980, 1983, 1986, 1988, 1990, 1991, 1993, 1994
The Regents of the University of California.  All rights reserved.
FreeBSD 4.4-STABLE (LOGIN) #4: Tue Nov 27 18:10:34 PST 2001

Welcome to FreeBSD!
```

```
%
```

- The applicant is `eve`.
- The client is Eve's ssh(1) process.
- The server is the sshd(8) process on `login.example.com`
- The account is `bob`.
- The authentication token is `god`.
- Although this is not shown in this example, the arbitrator is `root`.

#### 2.2.3. Sample policy

The following is FreeBSD's default policy for `sshd`:

```
sshd auth required pam_nologin.so no_warn
sshd auth required pam_unix.so no_warn try_first_pass
sshd account required pam_login_access.so
sshd account required pam_unix.so
sshd session required pam_lastlog.so no_fail
sshd password required pam_permit.so
```

- This policy applies to the `sshd` service (which is not necessarily restricted to the sshd(8) server.)
- `auth`, `account`, `session` and `password` are facilities.
- `pam_nologin.so`, `pam_unix.so`, `pam_login_access.so`, `pam_lastlog.so` and `pam_permit.so` are modules. It is clear from this example that `pam_unix.so` provides at least two facilities (authentication and account management.)

# 3. PAM Essentials

## 3.1. Facilities and primitives

The PAM API offers six different authentication primitives grouped in four facilities, which are described below.

`auth`

*Authentication.* This facility concerns itself with authenticating the applicant and establishing the account credentials. It provides two primitives:

- pam_authenticate(3) authenticates the applicant, usually by requesting an authentication token and comparing it with a value stored in a database or obtained from an authentication server.
- pam_setcred(3) establishes account credentials such as user ID, group membership and resource limits.

`account`

*Account management.* This facility handles non-authentication-related issues of account availability, such as access restrictions based on the time of day or the server's work load. It provides a single primitive:

- pam_acct_mgmt(3) verifies that the requested account is available.

`session`

*Session management.* This facility handles tasks associated with session set-up and tear-down, such as login accounting. It provides two primitives:

- pam_open_session(3) performs tasks associated with session set-up: add an entry in the `utmp` and `wtmp` databases, start an SSH agent, etc.
- pam_close_session(3) performs tasks associated with session tear-down: add an entry in the `utmp` and `wtmp` databases, stop the SSH agent, etc.

`password`

*Password management.* This facility is used to change the authentication token associated with an account, either because it has expired or because the user wishes to change it. It provides a single primitive:

- pam_chauthtok(3) changes the authentication token, optionally verifying that it is sufficiently hard to guess, has not been used previously, etc.

## 3.2. Modules

Modules are a very central concept in PAM; after all, they are the "M" in "PAM". A PAM module is a self-contained piece of program code that implements the primitives in one or more facilities for one particular mechanism; possible mechanisms for the authentication facility, for instance, include the UNIX® password database, NIS, LDAP and Radius.

### 3.2.1. Module Naming

FreeBSD implements each mechanism in a single module, named `pam_`*`mechanism`*`.so` (for instance, `pam_unix.so` for the UNIX mechanism.) Other implementations sometimes have separate modules for separate facilities, and include the facility name as well as the mechanism name in the module name. To name one example, Solaris has a `pam_dial_auth.so.1` module which is commonly used to authenticate dialup users.

### 3.2.2. Module Versioning

FreeBSD's original PAM implementation, based on Linux-PAM, did not use version numbers for PAM modules. This would commonly cause problems with legacy applications, which might be linked against older versions of the system libraries, as there was no way to load a matching version of the required modules.

OpenPAM, on the other hand, looks for modules that have the same version number as the PAM library (currently 2), and only falls back to an unversioned module if no versioned module could be loaded. Thus legacy modules can be provided for legacy applications, while allowing new (or newly built) applications to take advantage of the most recent modules.

Although Solaris PAM modules commonly have a version number, they are not truly versioned, because the number is a part of the module name and must be included in the configuration.

## 3.3. Chains and policies

When a server initiates a PAM transaction, the PAM library tries to load a policy for the service specified in the pam_start(3) call. The policy specifies how authentication requests should be processed, and is defined in a configuration file. This is the other central concept in PAM: the possibility for the admin to tune the system security policy (in the wider sense of the word) simply by editing a text file.

A policy consists of four chains, one for each of the four PAM facilities. Each chain is a sequence of configuration statements, each specifying a module to invoke, some (optional) parameters to pass to the module, and a control flag that describes how to interpret the return code from the module.

Understanding the control flags is essential to understanding PAM configuration files. There are four different control flags:

`binding`

> If the module succeeds and no earlier module in the chain has failed, the chain is immediately terminated and the request is granted. If the module fails, the rest of the chain is executed, but the request is ultimately denied.
>
> This control flag was introduced by Sun in Solaris 9 (SunOS™ 5.9), and is also supported by OpenPAM.

`required`

> If the module succeeds, the rest of the chain is executed, and the request is granted unless some other module fails. If the module fails, the rest of the chain is also executed, but the request is ultimately denied.

`requisite`

> If the module succeeds, the rest of the chain is executed, and the request is granted unless some other module fails. If the module fails, the chain is immediately terminated and the request is denied.

`sufficient`

> If the module succeeds and no earlier module in the chain has failed, the chain is immediately terminated and the request is granted. If the module fails, the module is ignored and the rest of the chain is executed.
>
> As the semantics of this flag may be somewhat confusing, especially when it is used for the last module in a chain, it is recommended that the `binding` control flag be used instead if the implementation supports it.

`optional`

> The module is executed, but its result is ignored. If all modules in a chain are marked `optional`, all requests will always be granted.

When a server invokes one of the six PAM primitives, PAM retrieves the chain for the facility the primitive belongs to, and invokes each of the modules listed in the chain, in the order they are listed, until it reaches the end, or determines that no further processing is necessary (either because a `binding` or `sufficient` module succeeded, or because a `requisite` module failed.) The request is granted if and only if at least one module was invoked, and all non-optional modules succeeded.

Note that it is possible, though not very common, to have the same module listed several times in the same chain. For instance, a module that looks up user names and passwords in a directory server could be invoked multiple times with different parameters specifying different directory servers to contact. PAM treat different occurrences of the same module in the same chain as different, unrelated modules.

## 3.4. Transactions

The lifecycle of a typical PAM transaction is described below. Note that if any of these steps fails, the server should report a suitable error message to the client and abort the transaction.

1. If necessary, the server obtains arbitrator credentials through a mechanism independent of PAM—most commonly by virtue of having been started by `root`, or of being setuid `root`.
2. The server calls pam_start(3) to initialize the PAM library and specify its service name and the target account, and register a suitable conversation function.
3. The server obtains various information relating to the transaction (such as the applicant's user name and the name of the host the client runs on) and submits it to PAM using pam_set_item(3).
4. The server calls pam_authenticate(3) to authenticate the applicant.
5. The server calls pam_acct_mgmt(3) to verify that the requested account is available and valid. If the password is correct but has expired, pam_acct_mgmt(3) will return `PAM_NEW_AUTHTOK_REQD` instead of `PAM_SUCCESS`.
6. If the previous step returned `PAM_NEW_AUTHTOK_REQD`, the server now calls pam_chauthtok(3) to force the client to change the authentication token for the requested account.
7. Now that the applicant has been properly authenticated, the server calls pam_setcred(3) to establish the credentials of the requested account. It is able to do this because it acts on behalf of the arbitrator, and holds the arbitrator's credentials.
8. Once the correct credentials have been established, the server calls pam_open_session(3) to set up the session.
9. The server now performs whatever service the client requested—for instance, provide the applicant with a shell.
10. Once the server is done serving the client, it calls pam_close_session(3) to tear down the session.
11. Finally, the server calls pam_end(3) to notify the PAM library that it is done and that it can release whatever resources it has allocated in the course of the transaction.

# 4. PAM Configuration

## 4.1. PAM policy files

### 4.1.1. The `/etc/pam.conf` file

The traditional PAM policy file is `/etc/pam.conf`. This file contains all the PAM policies for your system. Each line of the file describes one step in a chain, as shown below:

```
login   auth    required        pam_nologin.so  no_warn
```

The fields are, in order: service name, facility name, control flag, module name, and module arguments. Any additional fields are interpreted as additional module arguments.

A separate chain is constructed for each service / facility pair, so while the order in which lines for the same service and facility appear is significant, the order in which the individual services and facilities are listed is not. The examples in the original PAM paper grouped configuration lines by facility, and the Solaris stock `pam.conf` still does that, but FreeBSD's stock configuration groups configuration lines by service. Either way is fine; either way makes equal sense.

### 4.1.2. The `/etc/pam.d` directory

OpenPAM and Linux-PAM support an alternate configuration mechanism, which is the preferred mechanism in FreeBSD. In this scheme, each policy is contained in a separate file bearing the name of the service it applies to. These files are stored in `/etc/pam.d/`.

These per-service policy files have only four fields instead of `pam.conf`'s five: the service name field is omitted. Thus, instead of the sample `pam.conf` line from the previous section, one would have the following line in `/etc/pam.d/login`:

```
auth    required        pam_nologin.so  no_warn
```

As a consequence of this simplified syntax, it is possible to use the same policy for multiple services by linking each service name to a same policy file. For instance, to use the same policy for the `su` and `sudo` services, one could do as follows:

```
# cd /etc/pam.d
# ln -s su sudo
```

This works because the service name is determined from the file name rather than specified in the policy file, so the same file can be used for multiple differently-named services.

Since each service's policy is stored in a separate file, the `pam.d` mechanism also makes it very easy to install additional policies for third-party software packages.

### 4.1.3. The policy search order

As we have seen above, PAM policies can be found in a number of places. What happens if policies for the same service exist in multiple places?

It is essential to understand that PAM's configuration system is centered on chains.

## 4.2. Breakdown of a configuration line

As explained in the *PAM policy files* section, each line in `/etc/pam.conf` consists of four or more fields: the service name, the facility name, the control flag, the module name, and zero or more module arguments.

The service name is generally (though not always) the name of the application the statement applies to. If you are unsure, refer to the individual application's documentation to determine what service name it uses.

Note that if you use `/etc/pam.d/` instead of `/etc/pam.conf`, the service name is specified by the name of the policy file, and omitted from the actual configuration lines, which then start with the facility name.

The facility is one of the four facility keywords described in the *Facilities and primitives* section.

Likewise, the control flag is one of the four keywords described in the *Chains and policies* section, describing how to interpret the return code from the module. Linux-PAM supports an alternate syntax that lets you specify the action to associate with each possible return code, but this should be avoided as it is non-standard and closely tied in with the way Linux-PAM dispatches service calls (which differs greatly from the way Solaris and OpenPAM do it.) Unsurprisingly, OpenPAM does not support this syntax.

## 4.3. Policies

To configure PAM correctly, it is essential to understand how policies are interpreted.

When an application calls pam_start(3), the PAM library loads the policy for the specified service and constructs four module chains (one for each facility.) If one or more of these chains are empty, the corresponding chains from the policy for the `other` service are substituted.

When the application later calls one of the six PAM primitives, the PAM library retrieves the chain for the corresponding facility and calls the appropriate service function in each module listed in the chain, in the order in which they were listed in the configuration. After each call to a service function, the module type and the error code returned by the service function are used to determine what happens next. With a few exceptions, which we discuss below, the following table applies:

**Table 1. PAM chain execution summary**

| | **`PAM_SUCCESS`** | **`PAM_IGNORE`** | **`other`** |
|---|---|---|---|
| binding | if (!fail) break; | - | fail = true; |
| required | - | - | fail = true; |
| requisite | - | - | fail = true; break; |
| sufficient | if (!fail) break; | - | - |
| optional | - | - | - |

If `fail` is true at the end of a chain, or when a "break" is reached, the dispatcher returns the error code returned by the first module that failed. Otherwise, it returns `PAM_SUCCESS`.

The first exception of note is that the error code `PAM_NEW_AUTHTOK_REQD` is treated like a success, except that if no module failed, and at least one module returned `PAM_NEW_AUTHTOK_REQD`, the dispatcher will return `PAM_NEW_AUTHTOK_REQD`.

The second exception is that pam_setcred(3) treats `binding` and `sufficient` modules as if they were `required`.

The third and final exception is that pam_chauthtok(3) runs the entire chain twice (once for preliminary checks and once to actually set the password), and in the preliminary phase it treats `binding` and `sufficient` modules as if they were `required`.

## 5. FreeBSD PAM Modules

### 5.1. pam_deny(8)

The pam_deny(8) module is one of the simplest modules available; it responds to any request with `PAM_AUTH_ERR`. It is useful for quickly disabling a service (add it to the top of every chain), or for terminating chains of `sufficient` modules.

### 5.2. pam_echo(8)

The pam_echo(8) module simply passes its arguments to the conversation function as a `PAM_TEXT_INFO` message. It is mostly useful for debugging, but can also serve to display messages such as “Unauthorized access will be prosecuted” before starting the authentication procedure.

### 5.3. pam_exec(8)

The pam_exec(8) module takes its first argument to be the name of a program to execute, and the remaining arguments are passed to that program as command-line arguments. One possible application is to use it to run a program at login time which mounts the user’s home directory.

### 5.4. pam_ftpusers(8)

The pam_ftpusers(8) module

### 5.5. pam_group(8)

The pam_group(8) module accepts or rejects applicants on the basis of their membership in a particular file group (normally `wheel` for su(1)). It is primarily intended for maintaining the traditional behaviour of BSD su(1), but has many other uses, such as excluding certain groups of users from a particular service.

### 5.6. pam_guest(8)

The pam_guest(8) module allows guest logins using fixed login names. Various requirements can be placed on the password, but the default behaviour is to allow any password as long as the login name is that of a guest account. The pam_guest(8) module can easily be used to implement anonymous FTP logins.

### 5.7. pam_krb5(8)

The pam_krb5(8) module

### 5.8. pam_ksu(8)

The pam_ksu(8) module

## 5.9. pam_lastlog(8)

The pam_lastlog(8) module

## 5.10. pam_login_access(8)

The pam_login_access(8) module provides an implementation of the account management primitive which enforces the login restrictions specified in the login.access(5) table.

## 5.11. pam_nologin(8)

The pam_nologin(8) module refuses non-root logins when `/var/run/nologin` exists. This file is normally created by shutdown(8) when less than five minutes remain until the scheduled shutdown time.

## 5.12. pam_opie(8)

The pam_opie(8) module implements the opie(4) authentication method. The opie(4) system is a challenge-response mechanism where the response to each challenge is a direct function of the challenge and a passphrase, so the response can be easily computed “just in time” by anyone possessing the passphrase, eliminating the need for password lists. Moreover, since opie(4) never reuses a challenge that has been correctly answered, it is not vulnerable to replay attacks.

## 5.13. pam_opieaccess(8)

The pam_opieaccess(8) module is a companion module to pam_opie(8). Its purpose is to enforce the restrictions codified in opieaccess(5), which regulate the conditions under which a user who would normally authenticate herself using opie(4) is allowed to use alternate methods. This is most often used to prohibit the use of password authentication from untrusted hosts.

In order to be effective, the pam_opieaccess(8) module must be listed as `requisite` immediately after a `sufficient` entry for pam_opie(8), and before any other modules, in the `auth` chain.

## 5.14. pam_passwdqc(8)

The pam_passwdqc(8) module

## 5.15. pam_permit(8)

The pam_permit(8) module is one of the simplest modules available; it responds to any request with `PAM_SUCCESS`. It is useful as a placeholder for services where one or more chains would otherwise be empty.

## 5.16. pam_radius(8)

The pam_radius(8) module

### 5.17. pam_rhosts(8)

The pam_rhosts(8) module

### 5.18. pam_rootok(8)

The pam_rootok(8) module reports success if and only if the real user id of the process calling it (which is assumed to be run by the applicant) is 0. This is useful for non-networked services such as su(1) or passwd(1), to which the `root` should have automatic access.

### 5.19. pam_securetty(8)

The pam_securetty(8) module

### 5.20. pam_self(8)

The pam_self(8) module reports success if and only if the names of the applicant matches that of the target account. It is most useful for non-networked services such as su(1), where the identity of the applicant can be easily verified.

### 5.21. pam_ssh(8)

The pam_ssh(8) module provides both authentication and session services. The authentication service allows users who have passphrase-protected SSH secret keys in their `~/.ssh` directory to authenticate themselves by typing their passphrase. The session service starts ssh-agent(1) and preloads it with the keys that were decrypted in the authentication phase. This feature is particularly useful for local logins, whether in X (using xdm(1) or another PAM-aware X login manager) or at the console.

### 5.22. pam_tacplus(8)

The pam_tacplus(8) module

### 5.23. pam_unix(8)

The pam_unix(8) module implements traditional UNIX password authentication, using getpwnam(3) to obtain the target account's password and compare it with the one provided by the applicant. It also provides account management services (enforcing account and password expiration times) and password-changing services. This is probably the single most useful module, as the great majority of admins will want to maintain historical behaviour for at least some services.

## 6. PAM Application Programming

This section has not yet been written.

## 7. PAM Module Programming

This section has not yet been written.

# A. Sample PAM Application

The following is a minimal implementation of su(1) using PAM. Note that it uses the OpenPAM-specific openpam_ttyconv(3) conversation function, which is prototyped in `security/openpam.h`. If you wish build this application on a system with a different PAM library, you will have to provide your own conversation function. A robust conversation function is surprisingly difficult to implement; the one presented in the *Sample PAM Conversation Function* appendix is a good starting point, but should not be used in real-world applications.

```
#include <sys/param.h>
#include <sys/wait.h>

#include <err.h>
#include <pwd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <syslog.h>
#include <unistd.h>

#include <security/pam_appl.h>
#include <security/openpam.h> /* for openpam_ttyconv() */

extern char **environ;

static pam_handle_t *pamh;
static struct pam_conv pamc;

static void
usage(void)
{

fprintf(stderr, "Usage: su [login [args]]\n");
exit(1);
}

int
main(int argc, char *argv[])
{
char hostname[MAXHOSTNAMELEN];
const char *user, *tty;
char **args, **pam_envlist, **pam_env;
struct passwd *pwd;
int o, pam_err, status;
pid_t pid;

while ((o = getopt(argc, argv, "h")) != -1)
switch (o) {
```

```
case 'h':
default:
usage();
}

argc -= optind;
argv += optind;

if (argc > 0) {
user = *argv;
--argc;
++argv;
} else {
user = "root";
}

/* initialize PAM */
pamc.conv = &openpam_ttyconv;
pam_start("su", user, &pamc, &pamh);

/* set some items */
gethostname(hostname, sizeof(hostname));
if ((pam_err = pam_set_item(pamh, PAM_RHOST, hostname)) != PAM_SUCCESS)
goto pamerr;
user = getlogin();
if ((pam_err = pam_set_item(pamh, PAM_RUSER, user)) != PAM_SUCCESS)
goto pamerr;
tty = ttyname(STDERR_FILENO);
if ((pam_err = pam_set_item(pamh, PAM_TTY, tty)) != PAM_SUCCESS)
goto pamerr;

/* authenticate the applicant */
if ((pam_err = pam_authenticate(pamh, 0)) != PAM_SUCCESS)
goto pamerr;
if ((pam_err = pam_acct_mgmt(pamh, 0)) == PAM_NEW_AUTHTOK_REQD)
pam_err = pam_chauthtok(pamh, PAM_CHANGE_EXPIRED_AUTHTOK);
if (pam_err != PAM_SUCCESS)
goto pamerr;

/* establish the requested credentials */
if ((pam_err = pam_setcred(pamh, PAM_ESTABLISH_CRED)) != PAM_SUCCESS)
goto pamerr;

/* authentication succeeded; open a session */
if ((pam_err = pam_open_session(pamh, 0)) != PAM_SUCCESS)
goto pamerr;

/* get mapped user name; PAM may have changed it */
pam_err = pam_get_item(pamh, PAM_USER, (const void **)&user);
if (pam_err != PAM_SUCCESS || (pwd = getpwnam(user)) == NULL)
goto pamerr;

/* export PAM environment */
```

```
if ((pam_envlist = pam_getenvlist(pamh)) != NULL) {
for (pam_env = pam_envlist; *pam_env != NULL; ++pam_env) {
putenv(*pam_env);
free(*pam_env);
}
free(pam_envlist);
}

/* build argument list */
if ((args = calloc(argc + 2, sizeof *args)) == NULL) {
warn("calloc()");
goto err;
}
*args = pwd->pw_shell;
memcpy(args + 1, argv, argc * sizeof *args);

/* fork and exec */
switch ((pid = fork())) {
case -1:
warn("fork()");
goto err;
case 0:
/* child: give up privs and start a shell */

/* set uid and groups */
if (initgroups(pwd->pw_name, pwd->pw_gid) == -1) {
warn("initgroups()");
_exit(1);
}
if (setgid(pwd->pw_gid) == -1) {
warn("setgid()");
_exit(1);
}
if (setuid(pwd->pw_uid) == -1) {
warn("setuid()");
_exit(1);
}
execve(*args, args, environ);
warn("execve()");
_exit(1);
default:
/* parent: wait for child to exit */
waitpid(pid, &status, 0);

/* close the session and release PAM resources */
pam_err = pam_close_session(pamh, 0);
pam_end(pamh, pam_err);

exit(WEXITSTATUS(status));
}

pamerr:
fprintf(stderr, "Sorry\n");
```

```
err:
pam_end(pamh, pam_err);
exit(1);
}
```

## B. Sample PAM Module

The following is a minimal implementation of pam_unix(8), offering only authentication services. It should build and run with most PAM implementations, but takes advantage of OpenPAM extensions if available: note the use of pam_get_authtok(3), which enormously simplifies prompting the user for a password.

```
#include <sys/param.h>

#include <pwd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

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

#ifndef _OPENPAM
static char password_prompt[] = "Password:";
#endif

#ifndef PAM_EXTERN
#define PAM_EXTERN
#endif

PAM_EXTERN int
pam_sm_authenticate(pam_handle_t *pamh, int flags,
int argc, const char *argv[])
{
#ifndef _OPENPAM
struct pam_conv *conv;
struct pam_message msg;
const struct pam_message *msgp;
struct pam_response *resp;
#endif
struct passwd *pwd;
const char *user;
char *crypt_password, *password;
int pam_err, retry;

/* identify user */
if ((pam_err = pam_get_user(pamh, &user, NULL)) != PAM_SUCCESS)
return (pam_err);
if ((pwd = getpwnam(user)) == NULL)
return (PAM_USER_UNKNOWN);
/* get password */
#ifndef _OPENPAM
pam_err = pam_get_item(pamh, PAM_CONV, (const void **)&conv);
if (pam_err != PAM_SUCCESS)
return (PAM_SYSTEM_ERR);
msg.msg_style = PAM_PROMPT_ECHO_OFF;
msg.msg = password_prompt;
msgp = &msg;
#endif
for (retry = 0; retry < 3; ++retry) {
#ifdef _OPENPAM
pam_err = pam_get_authtok(pamh, PAM_AUTHTOK,
    (const char **)&password, NULL);
#else
resp = NULL;
pam_err = (*conv->conv)(1, &msgp, &resp, conv->appdata_ptr);
if (resp != NULL) {
if (pam_err == PAM_SUCCESS)
password = resp->resp;
else
free(resp->resp);
free(resp);
}
#endif
if (pam_err == PAM_SUCCESS)
break;
}
if (pam_err == PAM_CONV_ERR)
return (pam_err);
if (pam_err != PAM_SUCCESS)
return (PAM_AUTH_ERR);

/* compare passwords */
if ((!pwd->pw_passwd[0] && (flags & PAM_DISALLOW_NULL_AUTHTOK)) ||
    (crypt_password = crypt(password, pwd->pw_passwd)) == NULL ||
    strcmp(crypt_password, pwd->pw_passwd) != 0)
pam_err = PAM_AUTH_ERR;
else
pam_err = PAM_SUCCESS;
#ifndef _OPENPAM
free(password);
#endif
return (pam_err);
}

PAM_EXTERN int
pam_sm_setcred(pam_handle_t *pamh, int flags,
int argc, const char *argv[])
{

return (PAM_SUCCESS);
}
```

```
PAM_EXTERN int
pam_sm_acct_mgmt(pam_handle_t *pamh, int flags,
int argc, const char *argv[])
{

return (PAM_SUCCESS);
}

PAM_EXTERN int
pam_sm_open_session(pam_handle_t *pamh, int flags,
int argc, const char *argv[])
{

return (PAM_SUCCESS);
}

PAM_EXTERN int
pam_sm_close_session(pam_handle_t *pamh, int flags,
int argc, const char *argv[])
{

return (PAM_SUCCESS);
}

PAM_EXTERN int
pam_sm_chauthtok(pam_handle_t *pamh, int flags,
int argc, const char *argv[])
{

return (PAM_SERVICE_ERR);
}

#ifdef PAM_MODULE_ENTRY
PAM_MODULE_ENTRY("pam_unix");
#endif
```

## C. Sample PAM Conversation Function

The conversation function presented below is a greatly simplified version of OpenPAM's openpam_ttyconv(3). It is fully functional, and should give the reader a good idea of how a conversation function should behave, but it is far too simple for real-world use. Even if you are not using OpenPAM, feel free to download the source code and adapt openpam_ttyconv(3) to your uses; we believe it to be as robust as a tty-oriented conversation function can reasonably get.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

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

int
converse(int n, const struct pam_message **msg,
struct pam_response **resp, void *data)
{
struct pam_response *aresp;
char buf[PAM_MAX_RESP_SIZE];
int i;

data = data;
if (n <= 0 || n > PAM_MAX_NUM_MSG)
return (PAM_CONV_ERR);
if ((aresp = calloc(n, sizeof *aresp)) == NULL)
return (PAM_BUF_ERR);
for (i = 0; i < n; ++i) {
aresp[i].resp_retcode = 0;
aresp[i].resp = NULL;
switch (msg[i]->msg_style) {
case PAM_PROMPT_ECHO_OFF:
aresp[i].resp = strdup(getpass(msg[i]->msg));
if (aresp[i].resp == NULL)
goto fail;
break;
case PAM_PROMPT_ECHO_ON:
fputs(msg[i]->msg, stderr);
if (fgets(buf, sizeof buf, stdin) == NULL)
goto fail;
aresp[i].resp = strdup(buf);
if (aresp[i].resp == NULL)
goto fail;
break;
case PAM_ERROR_MSG:
fputs(msg[i]->msg, stderr);
if (strlen(msg[i]->msg) > 0 &&
    msg[i]->msg[strlen(msg[i]->msg) - 1] != '\n')
fputc('\n', stderr);
break;
case PAM_TEXT_INFO:
fputs(msg[i]->msg, stdout);
if (strlen(msg[i]->msg) > 0 &&
    msg[i]->msg[strlen(msg[i]->msg) - 1] != '\n')
fputc('\n', stdout);
break;
default:
goto fail;
}
}
*resp = aresp;
return (PAM_SUCCESS);
 fail:
        for (i = 0; i < n; ++i) {
                if (aresp[i].resp != NULL) {
                        memset(aresp[i].resp, 0, strlen(aresp[i].resp));
                        free(aresp[i].resp);
                }
        }
        memset(aresp, 0, n * sizeof *aresp);
*resp = NULL;
return (PAM_CONV_ERR);
}
```

## Further Reading

This is a list of documents relevant to PAM and related issues. It is by no means complete.

### Papers

*Making Login Services Independent of Authentication Technologies (http://www.sun.com/software/solaris/pam/pam.external.pdf)*, Vipin Samar and Charlie Lai, Sun Microsystems.

*X/Open Single Sign-on Preliminary Specification (http://www.opengroup.org/pubs/catalog/p702.htm)*, The Open Group, 1-85912-144-6, June 1997.

*Pluggable Authentication Modules (http://www.kernel.org/pub/linux/libs/pam/pre/doc/current-draft.txt)*, Andrew G. Morgan, October 6, 1999.

### User Manuals

*PAM Administration (http://www.sun.com/software/solaris/pam/pam.admin.pdf)*, Sun Microsystems.

### Related Web pages

*OpenPAM homepage (http://openpam.sourceforge.net/)*, Dag-Erling Smørgrav, ThinkSec AS.

*Linux-PAM homepage (http://www.kernel.org/pub/linux/libs/pam/)*, Andrew G. Morgan.

*Solaris PAM homepage (http://wwws.sun.com/software/solaris/pam/)*, Sun Microsystems.