Linux Authentication: Part 1

PLUGGABLE AUTHENTICATION MODULES

The traditional Unix security model is reliable but crude compared to the complexities of NDS or Active Directory. In this series of articles, Bruce Richardson will show how you can build security systems that put proprietary systems to shame

Plug in, swap out

Traditionally, any Unix application that needed to authenticate users was compiled against a specific authentication library. Many Linux apps still do just that. If you want to use a different authentication method with such an application, you have to rewrite and recompile it using the new library.

PAM (Pluggable Authentication Modules) changes this: once an application has been compiled against the PAM libraries[1], the authentication method it uses can be reconfigured or replaced without any alteration to the app itself. It does this by making the authentication process transparent to the application. When an application needs to perform an authentication-related task (changing a user password, for example) it calls a generic PAM function. PAM then selects the actual module to perform the function (based on settings in the PAM config files) and returns the result to the application. The application knows nothing of the actual method used and simply acts on the result.

This flexibility offers a range of advantages over traditional Unix authentication:

- You can reconfigure and replace authentication methods at will.
- Authentication methods can be chained together in complex ways.
- New methods can be added by simply installing the appropriate module, enabling you to add new methods as they are developed and to upgrade existing modules as they are improved or as security patches become necessary.
- Authentication policies can be created that control the way a whole range of applications (or your entire system) is secured.

These advantages have proved so compelling that almost all Linux distributions have integrated PAM into their setup – Slackware being the notable exception.

In one aspect PAM is a thoroughly traditional Unix system: all of this can be achieved by editing a few text files.

Installing PAM

The libraries

My advice: if they aren't already installed, don't bother. PAM is an integral part of almost all Linux distributions and as such is installed as part of the core libraries. If your distribution does not use PAM then not only will you have to install the PAM libraries but you will also have to recompile all the core utilities (login, su, passwd etc), which should use PAM if you are to make full use of the PAM security model.

If you are determined to install PAM on a distribution that doesn't use it by default, consult the documentation for that distribution carefully. Changing a system's security model is by no means a trivial task.

The modules

The most commonly used modules will have been installed with the PAM libraries. Most other modules should be available as standard packages for your distribution. If you download source tarballs for PAM modules, be sure to install the development files for PAM (which will also be available as a standard package) before compiling them.

PAM module types

PAM provides functions to cover a wide range of authentication tasks (account verification, password checking etc) These tasks and the modules that service them are divided into four types:

Authentication

Auth modules do two things:

1 They establish the identity of the user. This will usually involve a name/password challenge but

could use any alternative method (smart card or retinal scanning, for example). Some auth modules (such as kerberos) may additionally grant the user an authentication token, which they can use as proof of identity to request certain services.

2 They can grant additional privileges (such as group membership) based on the established identity.

When identifying a user, any challenge or message to the user is initiated by the PAM module not the application. The application simply provides a means of passing messages to the user and returning their response to the module (by registering a conversation function). Therefore the user may not be challenged for a password if the auth module(s) can be satisfied by other means.

Account

Account modules grant or deny access based on factors other than the user's authenticated identity and perform other account management functions. The pam_time module, for instance, grants or denies access according to the time of day or the day of the week.

Session

Session modules perform any tasks that are needed before or after a user accesses a service. This may include setting environment variables, mounting drives and so on.

Password

These modules change the user's authentication token (password, retinal print, smart-card ID number or whatever).

Some modules contain components for more than one type of task. The pam_unix module, which emulates the standard Unix security model, contains components for all four task types.

Not every application requires all four task types: some utilities (such as sudo) only need an auth task, though most require at least an auth and an account setting (thus allowing the administrator to limit the service to those with valid Unix accounts on the machine).

Configuring PAM

To configure a PAM-enabled application you must associate at least one module with each task-type that the application requires. Each line in the PAM configuration file(s) associates a module with an application and a task and provides configuration arguments.

There are two formats for configuring PAM applications: the */etc/pam.conf* file (deprecated) or the */etc/pam.d/* directory.

In the first method there is one configuration file and each line has five columns (except for comments and empty lines, of course), thus:

```
#
# /etc/pam.conf
#

service-name module-type control-flag module
arguments
```

In the second method, seperate files for each application are placed in */etc/pam.d/*. The contents are almost identical to those of */etc/pam.conf:*

```
#
# /etc/pam.d/service-name
#

module-type control-flag module arguments
```

These fields function as follows:

service-name

Each PAM-enabled application identifies itself to PAM by this name, which is usually compiled into the program. In the second configuration method the service-name becomes the name of the application's config file in */etc/pam.d*.

module-type

This identifies the task for which the module is used. It can have the values auth, account, session or password.

Control flags

The control flag determines how the success or failure of an individual module affects subsequent modules in the stack and the overall result. The allowed flags and their meanings are as follows:

- **required:** The module must succeed for the task to succeed. Regardless of the result, the execution of the stack continues.
- **requisite:** The module must succeed for the task to succeed. If it fails control returns immediately to the application and no more modules in the stack are called.
- **sufficient:** If the module succeeds the task succeeds and the rest of the stack is ignored. Otherwise the stack continues to execute.
- **optional:** Has no effect on the task's success or failure unless no other module returns a positive or negative result. Stack execution continues.

Note: there is an alternative, more complex syntax for control flags. This enables you to specify how a module affects the stack depending on the exit code returned by the module and also allows you to create complex stacks with alternate paths of execution. There isn't space to discuss this method in this article, however. The simple syntax is amply sufficient for most needs while the complex syntax requires a good knowledge of the internals of PAM and PAM modules.

Allowing the administrator to limit the service to those with valid Unix accounts

Example 1: Default config (old)

```
#
# /etc/pam.conf
#
other auth required pam_unix_auth.so
other account required pam_unix_acct.so
other password required pam_unix_passwd.so
other session required pam_unix_session.so
```

control-flag

Keyword(s) which determine how the module affects the overall success or failure of the authentication task. See the sidebar Control flags.

module

The pathname/filename of the module. If it starts with / then it is an absolute path, otherwise it is a relative path, starting from the default location for PAM modules[2].

arguments

The arguments that are passed to the module. Some are generic, others are specific to individual modules.

Consult the application's documentation or source to find out which tasks it requires and what service-name it uses. Most applications come with a sample configuration file or install a default one.

Mixing config methods

Depending on how PAM was compiled on your system, you may be able to mix both */etc/pam.conf* and */etc/pam.d* configuration formats. In one compilation mode the *pam.conf* file is ignored if */etc/pam.d* is present, even if the directory is empty (DANGER! DANGER!). In the other mode both configuration sources are read but settings from */etc/pam.d* override settings in *pam.conf*. Which mode was used depends on your distribution but there is no gain in using the older, deprecated method so I recommend you use only the newer */etc/pam.d* system.

Configuring settings

If PAM has no configuration settings for a specific service it uses the settings for "other". A typical configuration for "other" is shown in the older style in Example 1 and the newer style in Example 2.

Example 2: Default config (new)

```
#
# /etc/pam.d/other
#
auth required pam_unix.so
account required pam_unix.so
password
required pam_unix.so
session required pam_unix.so
```

Example 3: Deny by default

```
#
# /etc/pam.d/other
#
auth required pam_deny.so
account required pam_deny.so
password required pam_deny.so
session required pam_deny.so
```

Both setups use the pam_unix module[3], which emulates the traditional Unix authentication methods. So now any unconfigured service will use a traditional login process (password from */etc/passwd*, group membership from */etc/groups* etc).

Substituting modules

The simplest way to alter a PAM setup is to replace the specified modules. For example, a more secure default configuration is shown in Example 3. Here the pam_unix module has been replaced with the pam_deny module, which always returns failure. Any unconfigured service will now refuse all access.

Stacking modules

More subtle variations can be created by stacking modules. Each task type can have more than one entry for each service, the set of entries for any one task type forming a stack. The modules within a stack are invoked in the order in which they are listed. In Example 4 the pam_warn module, which logs details about the authenticating user, has been added to the auth and password stacks.

Example 4: Stacking

```
#
# /etc/pam.d/other
#
auth required pam_deny.so
auth required pam_warn.so
account required pam_deny.so
password required pam_deny.so
password required pam_warn.so
session required pam_deny.so
```

Each module in a stack may affect the execution of the stack and its ultimate result, according to the control flag. The Control flags sidebar explains the different flags and Table 1 lists some of the more unusual modules you might use in your stacks.

Setting policies

An easy way to create a consistent security policy across a range of applications is to create a configuration file in */etc/pam.d* and create symlinks to

Example 5: Login configuration

```
#
# /etc/pam.d/login
#
auth required pam_issue.so issue=/etc/issue
auth requisite pam_securetty.so
auth required pam_nologin.so
auth required pam_env.so
auth required pam_unix.so nullok
account required pam_unix.so
session required pam_unix.so
session optional pam_lastlog.so
session optional pam_motd.so
session optional pam_mail.so standard noenv
password required pam_unix.so nullok obscure
min=4 max=8 md5
```

it matching the service-name of each app. This can be done even if they do not all require the same range of task types: it does no harm to configure a task that an app will never use: an app that only requires auth settings can share a config file with an app that uses all four task types.

Things to watch

Cover yourself

Always set a default (other) configuration and make it a secure one. This protects you if you forget to configure an application.

Don't trip over the extra modules

Modules like pam_env, which perform supplementary tasks that should not affect the success or failure of a stack, are supposed to return neutral result codes. To be extra safe, always use the optional control flag with such modules.

Shadow security

If you are using shadow passwords (and if not, why not?) then applications which do not run with superuser priveleges will not be able to use modules such as pam_unix that authenticate against the standard password system. So Apache cannot use modules which authenticate against shadow passwords unless you either run it as root (not a great idea) or weaken the file security on /etc/shadow (a terrible idea).

Locking yourself out

If you make a mistake configuring PAM then you may find that you cannot login to your system at all. If that happens then you will have to reboot the machine in single-user mode and fix what you broke. With this in mind it is a good idea to make a copy of your original PAM configuration files before beginning to tinker.

Practical examples

Logging in with PAM

Example 5 shows a typical PAM configuration for the login. First, a walkthrough of the auth stack:

1. The pam_issue module prints the greeting in */etc/issue*[4].
2. The pam_securetty module checks to see if the authenticating user is root: if so the module will abort the stack unless they are logging in on a tty listed in */etc/securetty*.
3. The pam_nologin module checks for the existence of */etc/nologin:* if found it returns failure (and so prevents login) unless the authenticating user is root.
4. The pam_env module creates any environment variables listed in */etc/environment*[4] or defined by the rules in */etc/security/pam_env.conf*[4].
5. Finally, the pam_unix module performs standard Unix password authentication. The *nullok* argument means that passwordless accounts are allowed.

The account component of the pam_unix module checks the user settings in */etc/shadow* to see whether the account is disabled, the password is due for changing etc.

Finally, if both the auth and account stacks have returned success then the session stack executes:

It does no harm to configure an app we will never use

Table 1: Interesting modules

Module	Components	Description
pam_cracklib	password	Include this in your password stack and it checks any proposed new password for weaknesses. Has an extensive set of arguments to allow you to specify a password policy.
pam_permit	auth, account, session, password	The reverse of pam_deny: does nothing and returns success. So the auth component waves the user through without prompting for a password, the password component falsely reports that the password has been changed etc.
pam_listfile	auth	Grants access to services by consulting a text file to see if the user matches a list. Depending on the arguments passed the file may list usernames, groups, ttys, remote hosts, remote usernames or login shells and the module may either reject those listed and accept all others or vice versa.
pam_rootok	auth	Returns success if the user is root. To give root password-free access to a service, place this before any password-requesting modules and flag it sufficient. Used in the standard config for su).

Example 6: Restricted Apache access

```
#
# /etc/pam.d/httpd
#
auth requisite pam_listfile.so item=user \
sense=allow onerr=fail file=/etc/Apache-ssl/users
auth required pam_ncp_auth.so server=AZURE
account required pam_permit.so
```

Example 7: NT or Netware

```
#
# /etc/pam.d/httpd
#
auth sufficient pam_smb_auth.so debug nolocal
auth sufficient pam_ncp_auth.so server=AZURE \
use_first_pass
auth optional pam_warn.so
account required pam_permit.so
```

1 The pam_unix module logs the username and service-name to syslog.
2 The pam_lastlog module prints information about the previous login.
3 The pam_motd module prints the contents of */etc/motd*[4].
4 The pam_mail module prints the status of the user's mailbox.

Apache and PAM

These examples are from my workplace. Both involve Apache and authentication across a network (we run both Netware 4 and an NT domain). To enable Apache to use PAM I installed the mod_auth_pam Apache module.

In the first example the machine uses apache-ssl to serve up network administration pages. Only certain IT staff should access them and I wanted them to be able to use their Netware passwords. To achieve this I created */etc/pam.d/httpd* as shown in Example 6. Here the listfile module checks to see if the user is listed in */etc/apache-ssl/users:* only if this returns success does the pam_ncp_auth module authenticate against a Netware server.

The second example, shown in Example 7, involves an Intranet server hosting applications for our users, some of whom are on the Netware network while others are on an NT domain. First the pam_smb_auth module checks the username and password against the domain. If this succeeds, then execution halts there (because of the sufficient control flag). If there is no match, however, the pam_ncp_auth module tries Netware authentication. If that also returns failure then the failed attempt is logged.

Two points of interest:

1 The *debug* keyword is a generic option that may be passed to any PAM module, causing it to write verbose information to syslog. I had some problems with the NT domain and this helped me.
2 The *use_first_pass* keyword is another generic option. It tells a module to use the password given to the previous module. If it were not used here, the user would have to retype their password before being authenticated against the Netware server. It wasn't needed in the previous example because the listfile module doesn't use a password.

Info

Primary site	*http://www.kernel.org/pub/linux/libs/pam/*
Mailing list	*http://www.redhat.com/mailing-lists/pam-list/index.html*
mod_auth_pam	*http://pam.sourceforge.net/mod_auth_pam/*

Notes

[1] For an application to use PAM it must be explicitly (re)written to use the PAM libraries. There is no magic wand that can make a non-PAM application PAM-aware.
[2] */lib/security* on most current systems
[3] The pam_unix module combines the functions of four older modules, which in newer set-ups have been replaced by symlinks pointing to *pam_unix.so*
[4]This value can be changed by passing the module an appropriate argument.

In conclusion

PAM allows you to configure the authenticating services on your machine in an extremely flexible way. It enables you to upgrade or replace authentication methods painlessly and to set general policies.

The wide range of modules available make it possible for you to integrate your Linux machines into a varied networking environment. This is no trivial thing: certain proprietary software companies would like to own all authentication methods on your network, because then they own you.

The other articles in this series will deal with specific authentication methods – and there's a PAM module for all of them.