PAM TRICKS AND TIPS

BY MICHAEL W LUCAS

Pluggable Authentication Modules (PAM) are perhaps the least well understood, yet most broadly deployed part of FreeBSD. Every FreeBSD system uses PAM, via the OpenPAM suite. Most sysadmins don't touch it. If they have no choice but to change it, they follow cryptic online posts that look vaguely sensible. PAM is simpler than it looks, but its simplicity lets you accidentally achieve complexity. The big trick with PAM is to *not* do that.

In this article we'll look at PAM's components and configurations, help you avoid a few common mistakes, and figure out how to debug your mistakes.

What Is PAM?

Everybody agrees that usernames and passwords are not a great system of authentication. Nobody agrees on what should replace them. Different environments have different needs. Maybe you need Kerberos, or LDAP, or SSH certificates. Perhaps you authenticate sysadmins with hardware DNA scanners, or with bite guards molded to individual dental patterns. Each possible authentication method needs its own code.

You could try to compile every program to support every possible authentication method, but that leads to unsustainable code growth.

Or you could build a compatible shared library for each authentication method, and load that library only if you want to use that method. That's the "pluggable" part. Each library is an "authentication module," or just "module."

FreeBSD's OpenPAM, like primordial Solaris PAM, handles only authentication and authentication-related tasks. If you ever work with Linux, you'll see that the Linux-PAM developers decided that standard PAM was insufficiently complex and wedged other functions into their authentication stack. The features and tools found in OpenPAM work in most any PAM stack.

PAM is simpler than it looks, but its simplicity lets you accidentally achieve complexity.

PAM Configuration

Configure PAM for FreeBSD's base system in `/etc/pam.d`, and PAM for packages in `/usr/local/etc/pam.d`. Each daemon has its own configuration file, named after the program, that defines the PAM policy. Go look at the configuration for **sshd(8)**, **/etc/pam.d/sshd**. You'll find a bunch of lines like this.

```
auth            sufficient      pam_opie.so             no_warn no_fake_prompts
auth            requisite       pam_opieaccess.so       no_warn allow_local
auth            required        pam_unix.so             no_warn try_first_pass
…
```

Each line is one PAM rule. Each rule has four components: the type, the control, the module, and the module arguments. The first statement here has the type **auth**, the control **sufficient**, the module **pam_opie.so**, and the options **no_warn** and **no_fake_prompts**.

Rule Types

Authentication isn't only about the user credentials. The system must also check whether the access is permitted, provide resources for the user, and permit management of the authentication system itself. That's where the type statement comes in.

The *auth* type verifies the user's authentication information and sets resource limits. If you fat-finger your password or give a non-existent username, an auth rule kicks you out. The auth rules also establish limits like a maximum number of processes or amount of memory. We've seen auth rules above.

The *account* type controls access based on restrictions other than the user's authentication. If a user tries to log in out-of-hours, an account rule blocks that access.

The *session* type handles server-side setup. A command-line user needs a virtual terminal, a home directory, and probably a log entry saying they logged in. An anonymous FTP user should not get a virtual terminal and is limited to a particular directory. Session rules handle all this.

Finally, the *password* type handles authentication updates. Forced password changes are password rules.

Authentication isn't only about the user credentials.

Controls

You've seen access control lists in firewalls and server configurations. PAM rules are similar to access control lists. Each module can either reject, fail, or express "no opinion" on an authentication request. The control statement tells PAM how to process each module's decision. PAM has four common control statements: required, requisite, optional, and sufficient.

The *required* control means that this module must return success for the policy to permit access. If this module succeeds, the user gets access unless a later rule blocks it. PAM continues processing rules even after a required control fails.

A *requisite* control is much like a required control, but if an authentication module fails rule processing stops immediately. This can leak information about why authentication failed.

Optional controls are used to support add-on functions like SSH agents or Kerberos. They can permit or deny access if and only if no other module rejects or accepts the session. You can use optional controls for features like adding time between a failed authentication attempt and the next attempt.

The *sufficient* control means that if this module succeeds and no previous required controls failed, the user gets access immediately. Rule processing stops. If the control fails, PAM does not deny access.

Modules and Arguments

PAM modules are shared libraries that implement specific authentication functionality. Passwords are a module. Checking LDAP is a module. Poking your SSH agent is a module. To understand a PAM policy, you need to know what each module does. Every OpenPAM module, and most add-on modules, have a man page. The first time you decipher a PAM policy, you'll read a whole bunch of man pages.

PAM modules are shared libraries that implement specific authentication functionality.

Each module can also have arguments. While each module can have its own arguments to handle its particular needs, most also accept a handful of common arguments. The *debug* flag logs extra information, to hopefully give you a chance to figure out why authentication isn't working as expected. The *no_warn* flag silences any user feedback on why the authentication request was rejected.

Password handling gets two special arguments, *try_first_pass* and *use_first_pass*. The *try_first_pass* option reuses any password the user already entered, but if that doesn't work, the module may prompt for another password. The *use_first_pass* option declares that the module should use the password that was already entered, and if that doesn't work, reject the request.

Reading A Policy

Let's consider the **sshd(8)** sample rules again. These are auth rules, so they involve accepting or rejecting login credentials. Consider the first rule.

```
auth            sufficient      pam_opie.so             no_warn no_fake_prompts
```

This rule is sufficient. If the module succeeds, the authentication request is granted, and rule processing stops immediately.

The module is `pam_opie.so`. The man page `pam_opie(8)` tells us this supports OPIE. You'll need to do a little more research to see that OPIE is One-time Passwords In Everything. This rule declares that if someone authenticates with OPIE, they get access immediately.

```
auth            requisite       pam_opieaccess.so       no_warn allow_local
```

This rule is requisite. If it fails, processing stops immediately. That seems… harsh?

This rule is for another OPIE module, **`pam_opieaccess`**. If the OPIE users were immediately granted access in the previous rule, why do we have another OPIE rule? A check of **`pam_opieaccess(8)`** reveals that this module checks to see if a user is configured to require OPIE. A user that requires OPIE, who correctly enters their OPIE information, should *never* hit this rule. They *should* be booted out.

Now look at the third rule.

```
auth            required        pam_unix.so             no_warn try_first_pass
```

This is a required rule. A user who gets this far down must succeed with this module or be denied access.

The **`pam_unix(8)`** module checks the password file. It's the standard username and password authentication. So, this policy can be summarized as:

- If a user succeeds at OPIE, immediately let them in. Otherwise, keep going.
- If a user requires OPIE, reject them. A OPIE user with the right password will never get here.
- If the user enters a correct username and password, let them in.

Most accounts don't require OPIE, so we fall straight through to the password file.

That's very skimpy. What about users with invalid shells, or who have a shell of **`nologin(8)`**? What about users who aren't allowed to log in right now? Those are account rules, not auth rules. If you check **`/etc/pam.d/sshd`**, you'll see a section of account rules specifically checking user access.

Most accounts don't require OPIE, so we fall straight through to the password file.

Adding Authentication Methods

People most often stumble into PAM when their organization starts requiring two-factor authentication. Maybe that's Google Authenticator, or a Yubikey, or Cisco's Duo. Perhaps you have specialized authentication hardware than reads voiceprints or fingerprints or aromaprints. We'll use the last three to build some less common authentication rules. Here's a fairly straightforward one. I haven't read the documentation for any of these nonexistent modules, so I'm going to ignore the options.

```
auth    required  pam_voice.so
auth    required  pam_finger.so
auth    required  pam_aroma.so
```

All three policies are *required.* The user must submit a correct voiceprint and fingerprint, plus they must smell right, to authenticate.

```
auth    required  pam_voice.so
auth    requisite pam_finger.so
auth    required  pam_aroma.so
```

The middle policy, for **pam_finger.so**, is requisite. If this policy fails, checking immediately stops and the application is informed of the failure. Performing aroma analysis is expensive, and we don't want to waste resources.

Perhaps you want to allow the user a choice of what authentication method to use.

```
auth    sufficient  pam_voice.so
auth    sufficient  pam_finger.so
auth    required    pam_aroma.so
```

Here, our first two authentication methods are sufficient. The user can use a voiceprint or a fingerprint. If those fail, the user must submit their stink. I could also make the last rule sufficient, but if the policy ends here, I'd want to add another rule invoking **pam_deny.so** to explicitly reject authentication.

PAM Debugging

Your carefully tuned policy doesn't work? Too bad. PAM provides very little explicit debugging. Your three choices are the debug argument, **pam_echo**, and **pam_exec**.

Many modules support a *debug* argument. This might feed debugging to the user. It could dump debugging to a log file. It could do nothing discernible. Modules ignore unsupported arguments, so adding debug arguments throughout your policy won't break PAM any worse.

The **pam_echo(8)** module takes a string of text as an argument. The string gets passed back to the user. These rules are always of type optional. Let's add some echo debugging to one of our experimental policies.

```
auth    optional    pam_echo.so "auth policy starting, trying voice"
auth    sufficient  pam_voice.so
auth    optional    pam_echo.so "voice failed, trying finger"
auth    sufficient  pam_finger.so
auth    optional    pam_echo.so "finger failed, taking a whiff"
auth    required    pam_aroma.so
auth    optional    pam_echo.so "how did we get here?"
```

If you think this looks like scattering **printf()** calls through your code, you'd be right. It was good enough for Sun in the 1990s so it's good enough for you.

If the program feeds the output back to the user, they'll get the debugging statements in their terminal.

If the debug arguments don't provide useful information, and the program doesn't echo debugging output back to the user, then you get to get complicated with `pam_exec(8)`. The `pam_exec` module runs arbitrary commands for you. Yes, this means you could write a Perl script that checks user credentials against a Microsoft Excel spreadsheet over the network, but I hope you don't hate yourself *that* much. Most of the time, `pam_exec` is a way for intruders to mess with your authentication, but it's perfectly suited for calling a small shell script. Like this.

```
auth   optional    pam_exec.so /usr/sbin/pamdebug.sh pam_voice
auth   sufficient  pam_voice.so
auth   optional    pam_exec.so /usr/sbin/pamdebug.sh pam_finger
auth   sufficient  pam_finger.so
auth   optional    pam_exec.so /usr/sbin/pamdebug.sh pam_aroma
auth   required    pam_aroma.so
auth   optional    pam_exec.so /usr/sbin/pamdebug.sh impossible_end_of_pam_rule
```

The script itself is very simple.

```
#!/bin/sh
logger "process $PPID calling $1"
```

This logs the process ID and what stage of the authentication you're at. You wouldn't want to run this in a busy production environment where people are constantly logging on and off, but it makes debugging on a test system simple.

PAM has many more features and options than I can cover in this simple article, but hopefully you've gotten a decent idea of just how you can muck with your rules and fine-tune authentication to annoy your users in the exact manner desired.

Good luck.

MICHAEL W LUCAS is the author of *Absolute FreeBSD*, *FreeBSD Mastery: Jails*, and forty-eight other books. One of them was (drum roll please) *PAM Mastery*. Learn more at https://mwl.io.