Making Login Services Independent of Authentication Technologies*

*Vipin Samar** and Charlie Lai***
SunSoft, Inc.

Abstract

As current authentication mechanisms evolve and as new authentication mechanisms are introduced, system entry services such as `login`, `rlogin`, and `telnet` must continually be customized to incorporate these changes. With the Pluggable Authentication Module (PAM) framework, multiple authentication technologies can be added without changing any of the login services, thereby preserving existing system environments. PAM can be used to integrate login services with different authentication technologies, such as RSA, DCE, Kerberos, S/Key, and smart card based authentication systems. Thus, PAM enables networked machines to exist peacefully in a heterogeneous environment, where multiple security mechanisms are in place.

1. Introduction

Modular design and pluggability have become important for users who want ease of use. In the PC hardware arena, no one wants to set the interrupt vector numbers or resolve the addressing conflict between various devices. In the software arena, people also want to be able to replace components easily, for easy customization, maintenance, and upgrades.

Authentication software deserves special attention because authentication forms a very critical component of any secure computer system. The authentication infrastructure and its components may have to be modified or replaced, either because some deficiencies have been found in the current algorithms, or because sites want to enforce a different security policy than that provided by the system vendor. The replacement and modification should be done in such a way that these changes do not affect the user.

The solution has to address not only how the applications use the new authentication mechanisms in a generic fashion, but also how the user will be authenticated by these mechanisms in a generic way. The former is addressed by GSS-API [Linn 93], while this paper addresses the later; these two efforts are complementary.

In a typical UNIX system, system-entry services (for example, `login`, `dtlogin`, `ftp`) call `getspnam`() to retrieve a user's encrypted password from the `/etc/shadow` file, NIS, or NIS+ naming service to perform the necessary authentication. This dependency between the system-entry services and the standard UNIX database makes it difficult for these services to accommodate new authentication technologies.

For any security system to succeed, it has to be easy to use. In the case of authentication, the single most important ease-of-use characteristic is that the user should not need to learn various ways of authentication or remember multiple passwords. Ideally, one authentication system would be sufficient. But for heterogeneous sites, multiple authentication mechanisms have to coexist. The problem of integrating multiple authentication mechanisms, such as Kerberos [Neuman 94], RSA [Rivest 78], and Diffie-Hellman [Diffie 76, Taylor 88], is also referred to as the *integrated login*, or *unified login* problem. Even if users have to use multiple authentication mechanisms, they should not be forced to type multiple passwords. Furthermore, users should be able to use the new network identity without taking any further actions. The key here is in modular integration of the network authentication technologies with `login` and other system-entry services.

This paper discusses the architecture and design of pluggable authentication modules. This design enables the use of field-replaceable authentication modules, along with unified login capability. It thus provides for both *pluggability* and *ease-of-use*.

The organization of the paper includes discussions of

- the need for a generic way to authenticate the user by various system-entry services within the operating system.
- the goals and constraints of the design.
- the architecture, description of the interfaces, and *stacking* of modules to get unified login functionality.
- experience with the design.
- a description of future work.

---

* An earlier version of this paper was presented at the 3rd ACM Conference on Computer and Communications Security, March, 1996.

**email: vipin@eng.sun.com, charlie@eng.sun.com

2. Design Goals

An identification and authentication (I&A) mechanism establishes a user's identity to the system (that is, to a local machine's operating system) and to other principals on the network. A typical UNIX system has various ports of entry, such as `login`, `dtlogin`, `rlogind`, `ftpd`, `rsh`, `su`, and `telnetd`. In all cases, the user must be identified and authenticated before granted appropriate access rights. The user identification and authentication for all these entry points needs to be coordinated to ensure a secure system.

In most current UNIX systems, the login mechanism is based on password verification using the modified DES algorithm. The security of the implementation assumes that the password cannot be guessed, and that the password does not go over the wire in the clear. The classical assumptions may be acceptable on a trusted network, but an open environment needs more restrictive and stronger authentication mechanisms. Examples of such mechanisms include Kerberos, RSA, Diffie-Hellman, one-time passwords [Skey 94], and challenge-response based smart card authentication systems.

Implementations typically hard-code use of such mechanisms within their login applications. This may be appropriate for a certain set of users who have the exact requirements as envisioned by the vendor. But this makes it very difficult for the remaining set of users, who have different authentication requirements. Since this list of new technologies and user requirements will continue to evolve, it is important that the system-entry services do not have hard-coded dependencies on any of these authentication mechanisms.

The goals of such a system are as follows:

- The system administrator should be able to choose the default authentication mechanism for the machine. This can range from a simple password-based mechanism to a biometric or a smart card based system.
- The administrator should be able to configure the user authentication mechanism on per application basis. For example, a site may require S/Key password authentication for `telnet` access, while allowing console `login` sessions with just UNIX password authentication.
- The framework should support the display requirements of the applications. For example, for a graphical login session, the user may have to enter the user name, and the password may have to be entered in a new window. For networking services, such as `ftp` and `telnet`, the user name and password has to be transmitted over the network to the server machine.
- The administrator should be able to configure multiple authentication protocols for each application. For example, one may want the users to get authenticated by both Kerberos and RSA authentication systems.
- The administrator should be able to configure the system so that the user is authenticated with all authentication protocols without retyping the password.
- The system-entry services should not have to change when the underlying mechanism changes. This can be very useful for third-party developers who often do not have the source code for these services.
- The architecture should provide for a *pluggable* model for system authentication, as well as for other related tasks, such as password, account, and session management.
- For backward-compatibility reasons, the architecture should support the authentication requirements of the current system-entry services.
- The API should be independent of the operating system.

Issues not specifically addressed in the PAM design include:

- The problem of how identity, once established, is communicated to other interested parties (also known as the *single-sign-on* problem). The PAM design focuses only on providing a generic scheme through which users use passwords to establish their identities to the machine. The Generic Security Services Application Programming Interface (GSS-API) can then be used by applications for secure and authenticated communication without knowing the underlying mechanism.
- The problem of sending the passwords over the network in the clear.

3. Overview of the PAM Framework

The goals listed above can be met through a framework in which authentication modules can be *plugged* independently of the application with the *Pluggable Authentication Modules* (PAM) framework.

The core components of the PAM framework are the authentication library API (the front end) and the authentication mechanism-specific modules (the back end), connected through the Service Provider Interface (SPI). Applications write to the PAM API, while the authentication-system providers write to the PAM SPI and supply the back end modules that are independent of the application.

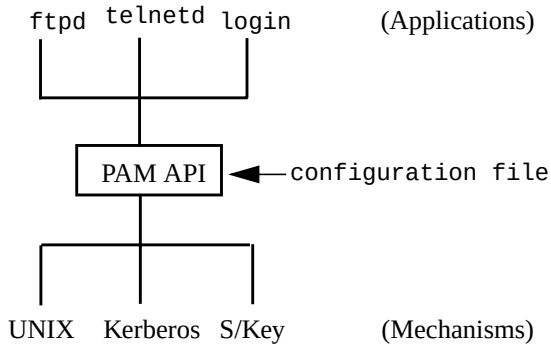

Figure 1: Basic PAM Architecture

Figure 1 illustrates the relationship between the application, the PAM library, and the authentication modules. Three applications, `login`, `telnetd` and `ftpd`, are shown that use the PAM authentication interfaces. When an application calls the PAM API, it loads the appropriate authentication module, as determined by the configuration file. The request is forwarded to the underlying authentication module, for example, UNIX password, Kerberos, S/Key, to perform the specified operation. The PAM layer then returns the response from the authentication module to the application.

PAM unifies system authentication and access control, and allows plugging of associated authentication modules through

well–defined interfaces. The plugging can be defined through various means, one of which uses a configuration file, such as the one in Table 1.

In this example, several fields have been deleted from the file to demonstrate the basic structure. For each system application, the file specifies the authentication module to be loaded. In Table 1, `login` uses the UNIX password module, while `ftp` and `telnet` use the S/Key module.

Table 1. Simplified View of a Sample PAM Configuration File

Service	Module_path
`login`	pam_unix.so
`ftp`	pam_skey.so
`telnet`	pam_skey.so

Authentication configuration is only one aspect of this interface. Other critical components include account management, session management, and password management. For example, the `login` program may be required to verify not only the password, but also whether the account has aged or expired. Generic interfaces also need to be provided so that the password can be changed according to the requirements of the module. Furthermore, the application may be required to log information about the current session, as determined by the module.

Not all applications or services may need all of the above components. For example, while `login` may need access to all four components, `su` may need access to just the authentication component. Some applications may use some specific authentication and password management modules but share the account and session management modules with others.

This reasoning led to a partitioning of the entire set of interfaces into four areas of functionality:

(1) Authentication

(2) Account

(3) Session

(4) Password

The concept of PAM was extended to these functional areas by implementing each of them as a separate, pluggable module.

Breaking the functionality into four modules helps the module providers because they can use the system-provided modules that they are not changing. For example, if suppliers want to provide a better version of Kerberos, they can provide just a new authentication and password module, and reuse the existing ones for account and session.

4. PAM Module Interfaces

Each interface listed here has a corresponding interface exported by the underlying PAM module. Appendix A contains more details on specific APIs. Appendix B describes a sample login application that uses the PAM APIs.

Briefly, the interfaces exported by the four modules are:

- **Authentication management** – includes the `pam_authenticate()` function to authenticate the user, and the `pam_setcred()` interface to set, refresh, or destroy the user credentials.
- **Account management** – includes the `pam_acct_mgmt()` function to check whether authenticated users should receive access to their accounts. This function can implement account expiration and access hour restrictions.
- **Session management** – includes the `pam_open_session()` and `pam_close_session()` functions for session management and accounting. For example, the system may store the total time for the session.
- **Password management** – includes the `pam_chauthtok()` function to change the password.

5. PAM Framework Interfaces

The PAM framework also provides a set of administrative interfaces to support the above modules and to provide for application-module communication. No corresponding service provider interface (SPI) is available for such functions.

5.1. Administrative Interfaces

Each set of PAM transactions starts with `pam_start()` and ends with the `pam_end()` function. The interfaces `pam_get_item()` and `pam_set_item()` are used to read and write the state information associated with the PAM transaction.

If an error occurs with any of the PAM interfaces, the error message can be printed with `pam_strerror()`.

5.2. Application Module Communication

During application initialization, certain data such as the user name is saved in the PAM framework layer through `pam_start()` so that it can be used by the underlying modules. The application can pass opaque data to the module which the modules hand back while communicating with the user.

The modules can communicate the environment variables associated with the given identity back to the application with the `pam_putenv()` API. The application can read the associated environment variables with the `pam_getenv()` and `pam_getenvlist()` APIs.

5.3. User Module Communication

The `pam_start()` function also passes a callback conversation function, to be used later by the underlying modules to read and write module specific authentication information. For example, these functions can prompt the user for the password in a way determined by the application. Thus, graphical, non-graphical, or networked applications can all use PAM.

5.4. Inter-Module Communication

Though the modules are independent, they can share certain common information about the authentication session, such as user name, service name, password, and conversation function through the `pam_get_item()` and `pam_set_item()` interfaces. The application can also

use these APIs to change the state information after having called `pam_start()` once.

5.5. Module State Information

The PAM service modules may want to keep certain module-specific state information about the PAM session. The service modules can use the `pam_get_data()` and `pam_set_data()` interfaces to access and update module-specific information from the PAM handle.

Since the PAM modules are loaded upon demand, there is no direct module initialization support in the PAM framework. If the PAM service modules have to do certain initialization tasks, the tasks must be done when the modules are first invoked. However, if certain clean-up tasks need to be done when the authentication session ends, the modules should use `pam_set_data()` to specify the clean-up functions, which would be called when `the` application calls `pam_end()`.

6. Module Configuration Management

Table 2 shows a sample `pam.conf` configuration file with support for authentication, session, account, and password management modules. `login` has four entries: one each for authentication processing, session management, account management and password updates. Each entry specifies the module name to be loaded for the given module type. In Table 2, the `ftp` service uses the authentication and session modules. Note that all services here share the same session management module, but have different authentication modules.

The first field, *service*, denotes the systems-entry application, for example, `login`, `passwd`, `rlogin`. The name `OTHER` indicates the module used by all other applications not specified in this file. This name can also be used if all services have the same requirements. Since all the services in Table 2 use the same session module, those lines can be replaced with a single `OTHER` line.

The second field, *module_type*, indicates the type of the PAM functional module. It can be one of `auth`, `account`, `session`, or `password` modules.

The third field, *control_flag* determines the behavior of stacking multiple modules by specifying whether any particular module is `requisite`, `required`, `sufficient`, or `optional`. The next section describes stacking in more detail.

The fourth field, *module_path*, specifies the location of the module. The PAM framework loads this module upon demand to invoke the required function. A default directory path may be defined to make administration easier.

The fifth field, *options*, is used by the PAM framework layer to pass module-specific options to the modules. It is up to the module to parse and interpret the options. The modules can use this field to turn on debugging or to pass any module specific parameters such as a time-out value. The field is also used to support password mapping. The system administrator can use the `options` field to fine-tune the PAM modules.

If any fields are invalid, or if a module is not found, that line is ignored, and the error is logged as critical. If no entries are found for the given module type, then the PAM framework returns an error to the application. If PAM is not properly configured, even the superuser may not be able to login into the system. The operating system must provide a mechanism to bypass the PAM layer during the booting process, so that the configuration file can be repaired.

7. Integrating Multiple Authentication Services with Stacking

In the world of heterogeneous systems, the system administrator often has to deal with the problem of integrating multiple authentication mechanisms. The user often has to know about the authentication command of the new authentication module (for example, `kinit`, `dce_login`) after logging into the system. This is not user-friendly because it forces people to remember to type the new command and enter the new password. This functionality should be invisible instead of burdening the user with it.

Therefore, two problems must be addressed:

(1) Supporting multiple authentication mechanisms.

(2) Providing unified login in the presence of multiple mechanisms.

The previous section described how to replace the default authentication module with any other module of choice. The

Table 2. PAM Configuration File (pam.conf) with Different Modules

Service	Module_type	Control_flag	Module_path	Options
`login`	auth	required	`pam_unix_auth.so`	nowarn
`login`	session	required	`pam_unix_session.so`	
`login`	account	required	`pam_unix_account.so`	
`login`	password	required	`pam_unix_passwd.so`	
`ftp`	auth	required	`pam_skey_auth.so`	debug
`ftp`	session	required	`pam_unix_session.so`	
`telnet`	session	required	`pam_unix_session.so`	
`passwd`	password	required	`pam_unix_passwd.so`	
`OTHER`	auth	required	`pam_unix_auth.so`	
`OTHER`	session	required	`pam_unix_session.so`	
`OTHER`	account	required	`pam_unix_account.so`	

same model can be extended to provide support for multiple modules.

7.1. Design for Stacked Modules

One possible design is to provide hard-coded rules in `login` or other applications requiring authentication services [Adamson 95]. But this becomes very specific to the particular combination of authentication protocols, and also requires the source code of the application. Digital's Security Integration Architecture [SIA 95] addresses this problem by specifying the same list of authentication modules for all applications. Since requirements for various applications can vary, it is essential that the configuration be on a per-application basis.

To support multiple authentication mechanisms, the PAM framework was extended to support *stacking*. When any API is called, the back ends for the stacked modules are invoked in the order listed, and the result returned to the caller. In Figure 2, the authentication service of `login` is stacked, and the user is authenticated by Kerberos, UNIX, and RSA authentication mechanisms respectively. Note this example has no stacking for session or account management modules.

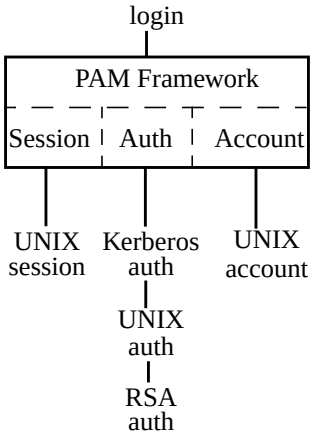

Figure 2: Stacking Within the PAM Architecture

Stacking is specified through additional entries in the configuration file shown earlier. As shown in Table 3, for each application, such as `login`, the configuration file can specify multiple mechanisms to be invoked in the specified order. When mechanisms fail, the *control_flag* decides which error should be returned to the application. Since the user should not know which authentication module failed when a bad password was typed, the PAM framework continues to call other authentication modules on the stack even on failure. The semantics of the control flag are:

- `required` – Normally set when authentication by this module is a *must*. With this flag, any module failure results in the PAM framework returning the error to the caller *after* executing all other modules on the stack. For the function to be able to return success to the application, all `required` modules have to report success.
- `requisite` – Normally set when authentication by this module is a *must* and that no subsequent modules should be invoked if this particular module fails. With this flag, any module failure results in the PAM framework returning the error to the caller *immediately* without executing any more modules on the stack. For the function to be able to return success to the application, all `requisite` modules have to report success.
- `optional` – Used when the user is allowed access, even if that particular module has failed. With this flag, the PAM framework ignores the module failure and continues processing the next module in sequence.
- `sufficient` – Used if the module succeeds the PAM framework, then returns success to the application immediately without trying any other subsequent modules even if they are `required` or `requisite`. For failure cases, the `sufficient` modules are treated as `optional`.

Table 3 shows a sample configuration file that stacks the `login` command. Here Kerberos, UNIX and RSA authentication services authenticate the user. The `required` key word for *control_flag* enforces users to log in only if they are authenticated by *both* UNIX and Kerberos services. RSA authentication is optional by virtue of the `optional` key word in the *control_flag* field. The user can still log in even if RSA authentication fails.

Table 4 illustrates the `sufficient` flag for the `rlogin` and the `requisite` flag for `su` services. The Berkeley `rlogin` authentication protocol specifies that if the remote host is trusted (as specified in the `/etc/hosts.equiv` file or in the `.rhosts` file in the home directory of the user), then the `rlogin` daemon should not require the user to type the password. If this is not the case, then the user must type the password. Instead of hard coding this policy in the `rlogin` daemon, it can be expressed with the `pam.conf` file in Table 4. The PAM module `pam_rhosts_auth.so` implements the `.rhosts` policy described above. If a site administrator requires remote login with passwords, then the first line should be deleted. If the *control_flag* is changed from `sufficient` to `required`, then `rlogin` would allow remote login sessions only with a password from the set of trusted hosts.

In the case of su, the user is authenticated by the `in-house` and `UNIX` authentication modules. Because the `in-house` and `UNIX` authentication modules are `requisite` and `required`, respectively, an error is returned back to the application if either module fails. In addition, if the `requisite` module (`in-house` authentication) fails, the `UNIX` authentication module is never invoked, and the error is returned immediately back to the application. Such a behavior might be desirable, for example,

Table 3. PAM Configuration File with Support for Multiple Authentication Modules

Service	Module_type	Control_flag	Module_path	Options
`login`	auth	required	`pam_kerb_auth.so`	debug
`login`	auth	required	`pam_unix_auth.so`	use_mapped_pass
`login`	auth	optional	`pam_rsa_auth.so`	try_first_pass

Table 4. PAM Configuration File for the `rlogin` and `su` Service

Service	Module_type	Control_flag	Module_path	Options
`rlogin`	`auth`	`sufficient`	`pam_rhosts_auth.so`	
`rlogin`	`auth`	`required`	`pam_unix_auth.so`	
`su`	`auth`	`requisite`	`pam_in-house.so`	
`su`	`auth`	`required`	`pam_unix.so`	

where the `in-house` module checks to see whether the user is coming through a network, and if such is the case, it returns failure so that the `UNIX` module is not invoked and the password is not sent in the clear.

7.2. Password-Mapping

Multiple authentication mechanisms on a machine can lead to multiple passwords for users to remember. One ease of use solution is to have the same password for all mechanisms. This, however, can also weaken security—if that password were to be compromised in any of the multiple mechanisms, all mechanisms would be compromised at the same time. Furthermore, different authentication mechanisms may have their own distinctive password requirements in regards to length, allowed characters, time interval between updates, aging, locking, and so forth. These requirements make using the same password for multiple authentication mechanisms problematic.

The following solution, while not precluding use of the same password for every mechanism, permits a different password for each mechanism through *password-mapping*. This mapping enables the user's *primary* password to be used for encrypting the user's other (*secondary*) passwords. It also permits storing these encrypted passwords in a place where they are available to the user. Once the primary password is verified, the authentication module obtains the password by decrypting the mechanism-specific encrypted password (also called a "mapped password") with the primary password, and then passing it to the authentication service. The security of this design for password-mapping assumes that the primary password is the user's strongest password; that is, it is not easily guessed, due to its length, type and mix of characters used, and so on.

If an error occurs in password mapping, or if mapping does not exist, each authentication module should prompt the user for a password.

To support password mapping, the PAM framework saves the primary password and provides it to stacked authentication modules. The password is cleared out before the `pam_authenticate` function returns.

How the password is encrypted or stored depends completely on the module implementation. If the encrypted passwords are stored in an untrusted publicly accessible place, this does provide an intruder with opportunities for potential dictionary attack.

Though password mapping is voluntary, all module providers should add support for the following four mapping options:

- `use_first_pass` – Use the same password as the first module that asked for a password. The module should not ask for the password if the user cannot be authenticated by the first password. This option is normally used when the system administrator wants to enforce the same password across multiple modules.
- `try_first_pass` – Same as `use_first_pass`, except that if the primary password is not valid, the module should prompt the user for the password.
- `use_mapped_pass` – Use the password-mapping scheme to get the actual password for this module. The module should not ask for the password if the user cannot be authenticated by the first password. The module can store the encrypted secondary password in a trusted or untrusted place, such as a smart card, a local file, or a directory service. One possible implementation is to get the mapped-password using the XFN API [XFN 95], and decrypt it with the primary password to get the module-specific password. The XFN API allows user-defined attributes, such as *mapped-password*, to be stored in the *user-context*. Using the XFN API is particularly attractive because many systems may support the XFN API in the future. Another option for DCE sites is to store such encrypted passwords in the DCE registry as an extended registry attribute (ERA). Note that the process for obtaining passwords is completely left to the module.
- `try_mapped_pass` – Same as `use_mapped_pass`, except that if the primary password is not valid, the module should prompt the user for the password.

When passwords get updated, the PAM framework stores both the old as well as the new password and makes them available to the other dependent authentication modules. Other modules can use this information to update the encrypted password without requiring the user to type the sequence of passwords again. The PAM framework clears out the passwords before returning to the application.

Table 3 on the previous page illustrates how `login` can use the same password for authenticating to the standard UNIX login, Kerberos, and RSA services. First, the user is authenticated to the primary authentication service (Kerberos `login` in this example) with the primary password. Then, the option `use_mapped_pass` indicates to the UNIX module to use the primary password to decrypt the stored UNIX password and then use it to get authenticated to local UNIX. After that succeeds, the option `try_first_pass` indicates to the RSA module that it should use the primary password typed earlier for authenticating the user. The RSA module should prompt for the RSA password, only if the password is incorrect. Note that in most cases, the user has to enter the password just once.

If a one-time password scheme (for example, S/Key) is used, password mapping does not apply.

7.3. Implications of Stacking on the PAM Design

Because PAM has stacking capability, the PAM APIs are designed to not return any data to the application, except status. If this were not the case, it would be difficult for the PAM framework to decide which module should return data to the application. When an error occurs, the application does not know which modules failed. This behavior requires the application to be completely independent from the modules.

The services that use the PAM API typically provide the user name as understood by the operating system to all the underlying PAM modules. Thus, PAM modules must convert the name to their own internal format. For example, the Kerberos module may have to convert the user name to a Kerberos principal name.

Stacking also forces the modules to be designed, so that they can occur anywhere in the stack without any side-effects.

Since the authentication and the password module are very closely related, they must be configured at the same time with compatible options.

8. Integration with Smart Cards

Many networking authentication protocols require possession of a long key to establish the user identity. For ease-of-use reasons, that long key is normally encrypted with the user's password so that the user need not memorize it. However, weak passwords can be compromised through a dictionary attack, thus undermining the stronger network authentication mechanism. Furthermore, the encrypted data is normally stored in a centrally accessible service whose availability depends upon the reliability of the associated service. Solutions have been proposed to use a pass-phrase or one-time-password, but they are much longer than the regular eight character passwords traditionally used with UNIX `login`. This makes the solution user-unfriendly because it requires longer strings to be remembered and typed.

Smart cards solve this problem by reducing password exposure by supporting a *two factor* authentication mechanism: the first with the possession of the card, and the second with the knowledge of the PIN associated with the card. Not only can the smart cards be a secure repository of multiple passwords, they can also provide encryption and authentication functions. Thus, the long private key is never exposed outside the card.

The PAM framework allows for integrating smart cards to the system by providing a smart card specific module for authentication. Furthermore, the unified login problem is simplified because the multiple passwords for various authentication mechanisms can be stored on the smart card itself. This can be enabled by adding a suitable key-word, such as `use_smart_card` in the *options* field. It should be noted that the implementation of this feature depends upon the underlying PAM modules.

9. Security Issues

It is important to understand the impact of PAM on the security of any system, so that the site-administrator can make an informed decision before implementing PAM.

- **Sharing of passwords with multiple authentication mechanisms** – For multiple authentication modules, use the same password for all of them. If the password for any of the multiple authentication system modules is compromised, the user's password in all systems would be compromised. If this is a concern, consider multiple passwords at the cost of ease-of-use.
- **Password-mapping** – Encrypting all other passwords with the primary password assumes that it is lot more difficult to crack the primary password. It also assumes that reasonable steps have been taken to ensure limited availability of the encrypted primary password. If this is not done, an intruder could target the primary password as the first point of dictionary attack. If one of the other modules provides stronger security than the password based security, the site would negate the strong security by using password-mapping. If this is a concern, consider multiple passwords at the cost of ease-of-use. If smart cards are used, they obviate the need for password-mapping.
- **Security of the configuration file** – Since the policy file dictates how the user is authenticated, this file should be protected from unauthorized modifications.
- **Stacking various PAM modules** – The system administrator should fully understand the implications of stacking various modules, their respective orders, and interactions. The composition of various authentication modules should be carefully examined. The trusted computing base of the machine now includes the PAM modules.

10. Relationship with GSS-API

PAM provides complementary functionality to GSS-API, in that it provides mechanisms through which the user gets authenticated to any new system-level authentication service on the machine. GSS-API then uses the credentials for authenticated and secure communications with other application-level service entities on the network.

11. Experience with PAM

The PAM framework was first added in the Solaris 2.3 release as a private internal interface. PAM is currently being used by several system entry applications such as `login`, `passwd`, `su`, `dtlogin`, `rlogind`, `rshd`, `telnetd`, `ftpd`, `in.rexecd`, and `uucpd`. PAM provides an excellent framework to encapsulate the authentication-related tasks for the entire system. The Solaris 2.3 PAM APIs were enhanced and simplified to support stacking. Though the examples in this paper refer to UNIX, PAM has no dependencies on UNIX and can easily be ported to other platforms.

Sample implementations of PAM modules have been developed for UNIX, DCE, Kerberos, S/Key, `rlogin`, and dialpass authentication. In addition, a sample module has been developed to assist module providers. Some third parties have used the PAM interface to extend the security mechanisms offered by the Solaris environment.

The Common Desktop Environment (CDE) vendors have accepted the PAM API as the API to be used for integrating the graphical interface for `login`, `dtlogin`, with multiple authentication mechanisms. The Open Group has approved the PAM APIs as preliminary specification for Single-Sign-On (SSO).

The new PAM framework, applications, and modules are available with the Solaris 2.6 release.

12. Conclusion

The PAM framework and the module interfaces provide pluggability for user authentication, as well as for account, session, and password management. `login` and other system-entry services can use the PAM architecture, and thus ensure that all entry

points for the system have been secured. This architecture enables replacement and modification of authentication modules in the field to secure the system against the newly found weaknesses without changing any of the system services.

The PAM framework can be used to integrate `login` and `dtlogin` with different authentication mechanisms such as RSA and Kerberos. Multiple authentication systems can be accessed with the same password. The PAM framework also enables easy integration of smart cards into the system.

13. Acknowledgments

PAM development has spanned several release cycles at SunSoft. P. Rajaram originated the earlier ideas behind PAM. Shau-Ping Lo, Chuck Hickey, and Alex Choy did the first design and implementation. Bill Shannon and Don Stephenson helped with the PAM architecture. Rocky Wu prototyped stacking of multiple modules. Paul Fronberg, and Roland Schemers made significant enhancements to the PAM interfaces. Kathy Slattery wrote the PAM documentation. John Perry integrated PAM within the CDE framework.

APPENDIX A. PAM APIs

This appendix gives an informal description of the various interfaces of PAM. Since the goal is to introduce the PAM interfaces, not all flags and options have been fully defined and explained.

The PAM Service Provider Interface is very similar to the PAM API, except for extra parameters to pass module-specific options to the underlying modules.

A.1. Framework Layer APIs

```
pam_start(char *service_name,
    char *user_name,
    struct pam_conv *pam_conversation,
    pam_handle_t **pamh);
```

`pam_start()` is called to initiate an authentication transaction. `pam_start()` takes as arguments the name of the service, the name of the user to be authenticated, and the conversation structure. `pamh` is later used as a handle for subsequent calls to the PAM library.

The PAM modules do not communicate directly with the user; instead they rely on the application to perform all such interaction. The application needs to provide the conversation functions, `conv()`, and associated application data pointers through a `pam_conv` structure when it initiates an authentication transaction. The module uses the `conv()` function to prompt the user for data, display error messages, or display text information.

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

`pam_end()` is called to terminate the PAM transaction as specified by `pamh`, and to free any storage area allocated by the PAM modules with `pam_set_item()`.

```
pam_set_item(pam_handle_t *pamh,
    int item_type,
    void *item );

pam_get_item(pam_handle_t *pamh,
    int item_type,
    void **item);
```

`pam_get_item()` and `pam_set_item()` allow the parameters specified in the initial call to `pam_start()` to be read and updated. `pam_set_item()` is used when a particular parameter is not available when `pam_start()` is called or must be modified after the initial call to `pam_start()`. `pam_set_item()` is passed the `item`, and its type, `item_type`. `pam_get_item()` returns the item associated with the `item_type`. The values for `item_type` are listed in Table 5.

Table 5. Possible Values for `item_type`

Item Name	Description
PAM_SERVICE	Service name
PAM_USER	User name
PAM_RUSER	Remote user name
PAM_TTY	tty name
PAM_RHOST	Remote host name
PAM_CONV	`pam_conv` structure
PAM_AUTHTOK	Authentication token
PAM_OLDAUTHTOK	Old authentication token
PAM_USER_PROMPT	Default prompt used by `pam_get_user()`

Note that the values of `PAM_AUTHTOK` and `PAM_OLDAUTHTOK` are only available to PAM modules and not to the applications. They are explicitly cleared out by the framework before returning to the application.

```
char *
pam_strerror(
    pam_handle_t *pamh,
    int errnum );
```

The `pam_strerror()` function maps the error number to a PAM error message string, and returns a pointer to that string.

```
pam_get_user(pam_handle_t *pamh,
    char **user,
    const char *prompt);
```

The `pam_get_user()` function is used by PAM service modules to retrieve the current user name from the PAM handle. If the user name has not been set via `pam_start()` or `pam_get_item()`, then the PAM conversation function will be used to prompt the user for the name.

```
pam_set_data(pam_handle_t *pamh,
    const char *module_data_name,
    const char *data,
    (*cleanup)(pam_handle_t *pamh, char *data, int error));
```

The `pam_set_data()` function stores module-specific data within the PAM handle. The `module_data_name` uniquely specifies the name to which some data and cleanup callback function can be attached. The cleanup function is called when `pam_end()` is invoked.

```
pam_get_data(pam_handle_t *pamh,
    const char *module_data_name,
```

```
    void **datap);
```

The `pam_get_data()` function obtains module-specific data from the PAM handle stored previously by the `pam_set_data()` function. The `module_data_name` uniquely specifies the name for which data has to be obtained.

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

The `pam_putenv()` routine sets the PAM environment variable name to value. This is typically called by modules to set an environment variable.

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

The `pam_getenv()` function returns the value attached with the given environment variable.

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

The `pam_getenvlist()` function returns the list of all environment variables attached with the PAM handle.

A.2. Authentication APIs

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

The `pam_authenticate()` function is called to verify the identity of the current user. The user is usually required to enter a password or similar authentication token, depending upon the authentication module configured with the system. The user in question is specified by a prior call to `pam_start()`.

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

The `pam_setcred()` function is called to set the credentials of the current process associated with the authentication handle, `pamh`. The actions that can be denoted through `flags` include credential initialization, refresh, reinitialization, and deletion.

A.3. Account Management API

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

The function `pam_acct_mgmt()` is called to determine whether the current user's account and password are valid. This typically includes checking for password and account expiration, valid login times, etc. The user in question is specified by a prior call to `pam_start()`.

A.4. Session Management APIs

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

`pam_open_session()` is called to inform the session modules that a new session has been initialized. All programs that use PAM should invoke `pam_open_session()` when beginning a new session.

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

Upon termination of this session, the `pam_close_session()` function should be invoked to inform the underlying modules that the session has terminated.

A.5. Password Management API

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

`pam_chauthtok()` is called to change the authentication token associated with the user referenced by the authentication handle `pamh`.

APPENDIX B. SAMPLE PAM APPLICATION

This appendix shows a sample `login` application which uses the PAM APIs. It is not meant to be a fully functional login program, as some functionality has been left out in order to emphasize the use of PAM APIs.

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

static int login_conv(int num_msg, struct pam_message **msg,
    struct pam_response **response, void *appdata_ptr);
struct pam_conv pam_conv = {login_conv, NULL};
pam_handle_t *pamh;             /* Authentication handle */

void
main(int argc, char *argv[], char **renvp)
{

    /* Insert code to collect PAM parameters */

     /* Call pam_start to initiate PAM operations */
    if ((pam_start("login", user_name, &pam_conv, &pamh))
        != PAM_SUCCESS)
        login_exit(1);

    pam_set_item(pamh, PAM_TTY, ttyn);
    pam_set_item(pamh, PAM_RHOST, remote_host);

    while (!authenticated && retry < MAX_RETRIES) {
        status = pam_authenticate(pamh, 0);
        authenticated = (status == PAM_SUCCESS);
    }

    if (status != PAM_SUCCESS) {
        fprintf(stderr,"error: %s\n", pam_strerror(pamh,
            status));
        login_exit(1);
    }

    /* now check if the authenticated user is allowed to login. */
    if ((status = pam_acct_mgmt(pamh, 0)) !=
                    PAM_SUCCESS) {
        if (status == PAM_AUTHTOK_EXPIRED) {
            status = pam_chauthtok(pamh, 0);
            if (status != PAM_SUCCESS)
                login_exit(1);
        }
    }

    /*
     * call pam_open_session to open the authenticated session
     * pam_close_session gets called by the process that
     * cleans up the utmp entry (i.e., init)
```

```
        */
        if (status = pam_open_session(pamh, 0) !=
                PAM_SUCCESS)
            login_exit(status);

        /* set up the process credentials */
        setgid(pwd->pw_gid);

        /*
         * Initialize the supplementary group access list before
         * pam_setcred because PAM modules might add groups
         * during the pam_setcred call
         */
        initgroups(user_name, pwd->pw_gid);

        status = pam_setcred(pamh, PAM_ESTABLISH_CRED);
        if (status != PAM_SUCCESS)
            login_exit(status);

        /* set the real (and effective) UID */
        setuid(pwd->pw_uid);

        pam_end(pamh, PAM_SUCCESS);

        /*
         * Done using PAM
         * Now implement all login related other tasks
         */
}

/*
 * login_exit - Call exit() and terminate.
 * This function is here for PAM so cleanup can
 * be done before the process exits.
 */

static void
login_exit(int exit_code)
{
        if (pamh)
            pam_end(pamh, PAM_ABORT);
        exit(exit_code);
}

 /*
 * login_conv():
 * This is the conv (conversation) function called from
 * a PAM authentication module to print error messages
 * or garner information from the user.
 */

int
login_conv(int num_msg, struct pam_message **msg,
        struct pam_response **response, void *appdata_ptr)
{
        while (num_msg--) {
            switch (m->msg_style) {
            case PAM_PROMPT_ECHO_OFF:
                r->resp = strdup(getpass(m->msg));
                break;
            case PAM_PROMPT_ECHO_ON:
                (void) fputs(m->msg, stdout);
                r->resp = malloc(PAM_MAX_RESP_SIZE);
                fgets(r->resp, PAM_MAX_RESP_SIZE, stdin);
                /* add code here to remove \n from fputs */
                break;
            case PAM_ERROR_MSG:
                (void) fputs(m->msg, stderr);
                break;
            case PAM_TEXT_INFO:
                (void) fputs(m->msg, stdout);
                break;
            default:
                log_error();
                break;
            }
        }
        return (PAM_SUCCESS);
}
```

REFERENCES

[Adamson 95] W. A. Adamson, J. Rees, and P. Honeyman, “Joining Security Realms: A Single Login for Netware and Kerberos,” in Proceedings of the 5th USENIX Security Symposium, Salt Lake City, June 1995.

[Diffie 76] W. Diffie and M. E. Hellman, “New Directions in Cryptography,” IEEE Transactions on Information Theory, November 1976.

[Linn 93] J. Linn, “Generic Security Service Application Programming Interface,” Internet RFC 1508, 1509, 1993.

[Neuman 94] B. C. Neuman and T. Ts’o, “Kerberos, An Authentication Service for Computer Networks,” IEEE Communications, 32(9):33-38, September 1994. http://nii.isi.edu/publications/kerberos-neuman-tso.html

[Rivest 78] R. L. Rivest, A. Shamir, and L. Adleman., “A Method for Obtaining Digital Signatures and Public-key Cryptosystems,” Communications of the ACM, 21(2), 1978.

[SIA 95] “Digital UNIX Security,” Digital Equipment Corporation, Order Number AA-Q0R2C-TE, July 1995.

[Skey 94] N. M. Haller, “The S/Key One-Time Password System,” ISOC Symposium on Network and Distributed Security, 1994.

[Taylor 88] B. Taylor and D. Goldberg, “Secure Networking in the Sun Environment,” Sun Microsystems Technical Paper, 1988.

[XFN 95] “Federated Naming: The XFN Specification,” X/Open Company, Ltd., U.K., X/Open Document #C403, July 1995.