Open Software Foundation
Request For Comments: 86.0
October 1995

V. Samar (SunSoft)
R. Schemers (SunSoft)

UNIFIED LOGIN WITH PLUGGABLE AUTHENTICATION MODULES (PAM)

1. INTRODUCTION

Since low-level authentication mechanisms constantly evolve, it is important to shield the high-level consumers of these mechanisms (system-entry services and users) from such low-level changes. With the Pluggable Authentication Module (PAM) framework, we can provide pluggability for a variety of system-entry services — not just system authentication *per se*, but also for account, session and password management. PAM's ability to *stack* authentication modules can be used to integrate `login` with different authentication mechanisms such as RSA, DCE, and Kerberos, and thus unify login mechanisms. The PAM framework can also provide easy integration of smart cards into the system.

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 what was provided by the system vendor. The replacement and modification should be done in such a way that the user is not affected by these changes.

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

Since most system-entry services (for example, `login`, `dtlogin`, `rlogin`, `ftp`, `rsh`) may want to be independent of the specific authentication mechanisms used by the machine, it is important that there be a framework for *plugging* in various mechanisms. This requires that the system applications use a standard API to interact with the authentication services. If these system-entry services remain independent of the actual mechanism used on that machine, the system administrator can install suitable authentication modules without requiring changes to these applications.

For any security system to be successful, 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 be required to learn about various ways of authentication and remember multiple passwords. Ideally, there should be one all-encompassing authentication system where there is only one password, but for heterogeneous sites, multiple authentication mechanisms have to co-exist. The problem of integrating multiple authentication mechanisms such as Kerberos [Steiner 88], RSA [Rivest 78], and Diffie-Hellman [Diffie 76, Taylor 88], is also referred to as *integrated login*, or *unified login* problem. Even if the user has to use multiple authentication mechanisms, the user should not be forced to type multiple passwords. Furthermore, the user 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.

In this RFC we discuss the architecture and design of pluggable authentication modules. This design gives the capability to use field-replaceable authentication modules along with unified login capability. It thus provides for both *pluggability* and *ease-of-use*.

The RFC is organized as follows. We first motivate the need for a generic way to authenticate the user by various system-entry services within the operating system. We describe the goals and constraints of the design. This leads to the architecture, description of the interfaces, and *stacking* of modules to get unified login functionality. We then describe our experience with the design, and end with a description of future work.

2. OVERVIEW OF IDENTIFICATION AND AUTHENTICATION MECHANISMS

An identification and authentication (''I&A'') mechanism is used to establish a user's identity the system (i.e., to a local machine's operating system) and to other principals on the network. On a typical UNIX system, there are various ports of entry into the system, such as `login`, `dtlogin`, `rlogin`, `ftp`, `rsh`, `su`, and `telnet`. In all cases, the user has to be identified and authenticated before granting appropriate access rights to the user. The user identification and authentication for all these entry points needs to be coordinated to ensure a secure system.

In most of the current UNIX systems, the login mechanism is based upon verification of the password 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. These assumptions, however, are not universally valid. Various programs are now available freely on the Internet that can run dictionary attack against the encrypted password. Further, some of the network services (for example, `rlogin`, `ftp`, `telnet`) send the password over in clear, and there are ''sniffer'' programs freely available to steal these passwords. The classical assumptions may be acceptable on a trusted network, but in an open environment there is a need to use more restrictive and stronger authentication mechanisms. Examples of such mechanisms include Kerberos, RSA, Diffie-Hellman, one-time password [Skey 94], and challenge-response based smart card authentication systems. Since this list will continue to evolve, it is important that the system-entry services do not have hard-coded dependencies on any of these authentication mechanisms.

3. DESIGN GOALS

The goals of the PAM framework are as follows:

(a) 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.

(b) It should be possible to configure the user authentication mechanism on a per application basis. For example, a site may require S/Key password authentication for `telnet` access, while allowing machine `login` sessions with just UNIX password authentication.

(c) The framework should support the display requirements of the applications. For example, for a graphical login session such as `dtlogin`, the user name and the password may have to be entered in a new window. For networking system-entry applications such as `ftp` and `telnet`, the user name and password has to be transmitted over the network to the client machine.

(d) It should be possible to configure multiple authentication protocols for each of those applications. For example, one may want the users to get authenticated by both Kerberos and RSA authentication systems.

(e) The system administrator should be able to *stack* multiple user authentication mechanisms such that the user is authenticated with all authentication protocols without retyping the password.

(f) The architecture should allow for multiple passwords if necessary to achieve higher security for users with specific security requirements.

(g) The system-entry services should not be required to change when the underlying mechanism changes. This can be very useful for third-party developers because they often do not have the source code for these services.

(h) 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.

(i) For backward-compatibility reasons, the PAM API should support the authentication requirements of the current system-entry services.

There are certain issues that the PAM framework does not specifically address:

(a) We focus only on providing a generic scheme through which users use passwords to establish their identities to the machine. Once the identity is established, how the identity is communicated to other interested parties is outside the scope of this design. There are efforts underway at IETF [Linn 93] to develop a Generic Security Services Application Interface (GSSAPI) that can be used by applications for secure and authenticated communication without knowing the underlying mechanism.

(b) The *single-signon* problem of securely transferring the identity of the caller to a remote site is not addressed. For example, the problem of delegating credentials from the `rlogin` client to the other machine without typing the password is not addressed by our work. We also do not address the problem of sending the passwords over the network in the clear.

(c) We do not address the source of information obtained from the ‘‘`getXbyY()`’’ family of calls (e.g., `getpwnam()`). Different operating systems address this problem differently. For example, Solaris uses the name service switch (NSS) to determine the source of information for the ‘‘`getXbyY()`’’ calls. It is expected that data which is stored in multiple sources (such as passwd entries in NIS+ and the DCE registry) is kept in sync using the appropriate commands (such as `passwd_export`).

4. OVERVIEW OF THE PAM FRAMEWORK

We propose that the goals listed above can be met through a framework in which authentication modules can be *plugged* independently of the application. We call this 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.

```
  ftp     telnet   login   (Applications)
   |        |        |
   |        |        |
   +--------+--------+
            |
      +-----+-----+
      |  PAM API  |   <-- pam.conf file
      +-----+-----+
            |
   +--------+--------+
 UNIX   Kerberos  Smart Cards   (Mechanisms)
```

Figure 1: The Basic PAM Architecture

Figure 1 illustrates the relationship between the application, the PAM library, and the authentication modules. Three applications (`login`, `telnet` and `ftp`) are shown which use the PAM authentication interfaces. When an application makes a call to the PAM API, it loads the appropriate authentication module as determined by the configuration file, `pam.conf`. The request is forwarded to the underlying authentication module (for example, UNIX password, Kerberos, smart cards) 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 for the system, 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. For each of the system applications, the file specifies the authentication module that should be loaded. In the example below, `login` uses the UNIX password module, while `ftp` and `telnet` use the S/Key module.

Table 1: A 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 want 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 want to log information about the current session as determined by the module.

Not all applications or services may need all of the above components, and not each authentication module may need to provide support for all of the interfaces. 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 leads to a partitioning of the entire set of interfaces into four areas of functionality: (1) authentication, (2) account, (3) session, and (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 libraries for the modules that they are not changing. For example, if a supplier wants to provide a better version of Kerberos, they can just provide that new authentication and password module, and reuse the existing ones for account and session.

4.1. Module Description

More details on specific API's are described in Appendix A. A brief description of four modules follows:

(a) Authentication management: This set includes the `pam_authenticate()` function to authenticate the user, and the `pam_setcred()` interface to set, refresh or destroy the user credentials.

(b) Account management: This set includes the `pam_acct_mgmt()` function to check whether the authenticated user should be given access to his/her account. This function can implement account expiration and access hour restrictions.

(c) Session management: This set includes the `pam_open_session()` and `pam_close_session()` functions for session management and accounting. For example, the system may want to store the total time for the session.

(d) Password management: This set includes a function, `pam_chauthtok()`, to change the password.

5. FRAMEWORK INTERFACES

The PAM framework further provides a set of administrative interfaces to support the above modules and to provide for application-module communication. There is no corresponding service provider interface (SPI) 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 there is any error 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 also pass opaque data to the module which the modules will pass back while communicating with the user.

5.3. User-Module Communication

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

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. These API's can also be used by the application 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 session. The interfaces `pam_get_data()` and `pam_set_data()` can be used by the service modules to access and update module-specific information as needed from the PAM handle. The modules can also attach a cleanup function with the data. The cleanup function is executed when `pam_end()` is called to indicate the end of the current authentication activity.

Since the PAM modules are loaded upon demand, there is no direct module initialization support in the PAM framework. If there are certain initialization tasks that the PAM service modules have to do, they should be done upon the first invocation. However, if there are certain clean-up tasks 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 `pam_end()` is called by the application.

6. MODULE CONFIGURATION MANAGEMENT

Table 2 shows an example of a configuration file `pam.conf` with support for authentication, session, account, and password management modules. `login` has three entries: one each for authentication processing, session management and account management. Each entry specifies the module name that should be loaded for the given module type. In this example, the `ftp` service uses the authentication and session modules. Note that all services here share the same session management module, while having different authentication modules.

```
Table 2: Configuration File (pam.conf) with Different Modules
         and Control Flow

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
ftp     auth        required     pam_skey_auth.so     debug
ftp     session     required     pam_unix_session.so
telnet  session     required     pam_unix_session.so
login   password    required     pam_unix_passwd.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
```

The first field, *service*, denotes the service (for example, `login`, `passwd`, `rlogin`). The name `OTHER` indicates the module used by all other applications that have not been specified in this file. This name can also be used if all services have the same requirements. In the example, since all the services use the same session module, we could have replaced those lines 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 *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.

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. This field can be used by the modules to turn on debugging or to pass any module specific parameters such as a timeout value. It is also used to support unified login as described below. The options field can be used by the system administrator to fine-tune the PAM modules.

If any of the fields are invalid, or if a module is not found, that line is ignored and the error is logged as a critical error via `syslog(3)`. If no entries are found for the given module type, then the PAM framework returns an error to the application.

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 is often required 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.

There are two problems to be addressed here:

(a) Supporting multiple authentication mechanisms.

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

In the previous section, we described how one could replace the default authentication module with any other module of choice. Now we demonstrate how the same model can be extended to provide support for multiple modules.

7.1. Design for Stacked Modules

One possibility was 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 UNIX, Kerberos, and RSA authentication mechanisms. Note that in this example, there is no stacking for session or account management modules.

```
            login
              |
     +--------+--------+
     |        |        |
  session   auth    account
     |        |        |
  +--+--+  +--+--+  +--+--+
  | PAM |  | PAM |  | PAM |
  +--+--+  +--+--+  +--+--+
     |        |        |
   UNIX     UNIX     UNIX
  session   auth    account
              |
          Kerberos
            auth
              |
             RSA
            auth
```

Figure 2: Stacking With the PAM Architecture

Stacking is specified through additional entries in the configuration file shown earlier. As shown in Table 2, for each application (such as `login`) the configuration file can specify multiple mechanisms that have 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 as follows:

(a) `required`: With this flag, the 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. This flag is normally set when authentication by this module is a *must*.

(b) `optional`: With this flag, the PAM framework ignores the module failure and continues with the processing of the next module in sequence. This flag is used when the user is allowed to login even if that particular module has failed.

(c) `sufficient`: With this flag, if the module succeeds the PAM framework returns success to the application immediately without trying any other modules. For failure cases, the *sufficient* modules are treated as `optional`.

Table 3 shows a sample configuration file that stacks the `login` command. Here the user is authenticated by UNIX, Kerberos, and RSA authentication services. The `required` key word for *control_flag* enforces that the user is allowed to login only if he/she is 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 3: PAM Configuration File with Support for Stacking

service	module_type	control_flag	module_path	options
login	auth	required	pam_unix.so	debug
login	auth	required	pam_kerb.so	use_mapped_pass
login	auth	optional	pam_rsa.so	use_first_pass

Table 4 illustrates the use of the sufficient flag for the `rlogin` service. The Berkeley `rlogin` 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 is required to type the password. Instead of hard coding this policy in the `rlogin` daemon, this can be expressed with the `pam.conf` file in Table 4. The PAM module `pam_rhosts_auth.so.1` implements the `.rhosts` policy described above. If a site administrator wants to enable remote login with only passwords, then the first line should be deleted.

```
Table 4: PAM Configuration File for the rlogin service

service module_type control_flag module_path        options
------- ----------- ------------ -----------        -------
rlogin  auth        sufficient   pam_rhosts_auth.so
rlogin  auth        required     pam_unix.so
```

7.2. Password-Mapping

Multiple authentication mechanisms on a machine can lead to multiple passwords that users have to remember. One attractive solution from the ease-of-use viewpoint is to use the same password for all mechanisms. This, however, can also weaken the security because 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 its length, allowed characters, time interval between updates, aging, locking, and so forth. These requirements make it problematic to use the same password for multiple authentication mechanisms.

The solution we propose, while not precluding use of the same password for every mechanism, allows for a different password for each mechanism through what we call *password-mapping*. This basically means using the user's *primary* password to encrypt the user's other (*secondary*) passwords, and storing these encrypted passwords in a place where they are available to the user. Once the primary password is verified, the authentication modules would obtain the other passwords for their own mechanisms by decrypting the mechanism-specific encrypted password with the primary password, and 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, in terms of its unguessability (length, type and mix of characters used, etc.).

If there is any error in password-mapping, or if the mapping does not exist, the user will be prompted for the password by each authentication module.

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 depends completely on the module implementation. The encrypted secondary password (also called a ''mapped password'') can be stored in a trusted or untrusted place, such as a smart card, a local file, or a directory service. 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, it is recommended that all module providers add support for the following four mapping options:

(a) `use_first_pass`: Use the same password used by the first mechanism 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.

(b) `try_first_pass`: This is the same as `use_first_pass`, except that if the primary password is not valid, it should prompt the user for the password.

(c) `use_mapped_pass`: Use the password-mapping scheme to get the actual password for this module. One possible implementation is to get the mapped-password using the XFN API [XFN 94], and decrypt it with the primary password to get the module-specific password. The module should not ask for the password if the user cannot be authenticated by the first 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 support for the XFN may be found on many systems in the future.

(d) `try_mapped_pass`: This is the same as `use_mapped_pass`, except that if the primary password is not valid, it should prompt the user for the password.

When passwords get updated, the PAM framework stores both the old as well as the new password to be able to inform other dependent authentication modules about the change. Other modules can use this information to update the encrypted password without forcing the user to type the sequence of passwords again. The PAM framework clears out the passwords before returning to the application.

Table 3 illustrates how the same password can be used by `login` for authenticating to the standard UNIX login, Kerberos and RSA services. Once the user has been authenticated to the primary authentication service (UNIX `login` in this example) with the primary password, the option `use_mapped_pass` indicates to the Kerberos module that it should use the primary password to decrypt the stored Kerberos password and then use the Kerberos password to get the ticket for the ticket-granting-service. After that succeeds, the option `use_first_pass` indicates to the RSA module that instead of prompting the user for a password, it should use the primary password typed earlier for authenticating the user. Note that in this scenario, the user has to enter the password just once.

Note that if a one-time password scheme (e.g., S/Key) is used, password mapping cannot apply.

7.3. Implications of Stacking on the PAM Design

Because of the stacking capability of PAM, we have designed the PAM API's 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 there is any error, the application does not know which of the modules failed. This behavior enables (even requires) the application to be completely independent from the modules.

Another design decision we have made is that PAM gives only the user name to all the underlying PAM modules, hence it is the responsibility of the PAM modules to convert the name to their own internal format. For example, the Kerberos module may have to convert the UNIX user name to a Kerberos principal name.

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

Since modules such as the authentication and the password module are very closely related, it is important they be configured in the same order and 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 is not required to memorize it. However, weak passwords can be compromised through a dictionary attack and thus undermine 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 those 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.

For most authentication protocol implementations, the trust boundary is the local machine. This assumption may not be valid in cases where the user is mobile and has to use publicly available networked computers. In such cases, it is required that the clear text of the key or the password never be made available to the machine.

Smart cards solve the above problems 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 the encryption and authentication functions such that 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 keyword such as `use_smart_card` in the *options* field.

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.

(a) Sharing of passwords with multiple authentication mechanisms.

If there are multiple authentication modules, one possibility is to use the same password for all of them. If the password for any of the multiple authentication system is compromised, the user's password in all systems would be compromised. If this is a concern, then multiple passwords might be considered at the cost of ease-of-use.

(b) Password-mapping.

This technique of encrypting all other passwords with the primary password assumes that it is lot more difficult to crack the primary password and 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 provide stronger security than the password based security, the site would be negating the strong security by using password-mapping. If this is a concern, then multiple passwords might be considered at the cost of ease-of-use. If smart cards are used, they obviate the need for password-mapping completely.

(c) Security of the configuration file.

Since the policy file dictates how the user is authenticated, this file should be protected from unauthorized modifications.

(d) Stacking various PAM modules.

The system administrator should fully understand the implications of stacking various modules that will be installed on the system and 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. EXPERIENCE WITH PAM

The PAM framework was first added in 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`, `uucpd`, `init`, `sac`, and `ttymon`. We have found that PAM provides an excellent framework to encapsulate the authentication-related tasks for the entire system. The Solaris 2.3 PAM API's were hence enhanced and simplified to support stacking.

PAM modules have been developed for UNIX, DCE, Kerberos, S/Key, remote user authentication, and dialpass authentication. Other PAM modules are under development, and integration with smart cards is being planned.

Some third parties have used the PAM interface to extend the security mechanisms offered by the Solaris environment.

The PAM API has been accepted by Common Desktop Environment (CDE) vendors as the API to be used for integrating the graphical interface for login, `dtlogin` with multiple authentication mechanisms.

11. FUTURE WORK

Amongst the various components of PAM, the password component needs to be carefully examined to see whether the stacking semantics are particularly applicable, and how PAM should deal with partial failures when changing passwords.

The *control_flag* of the configuration file can be extended to include other semantics. For example, if the error is ''name service not available'', one may want to retry. It is also possible to offer semantics of ''return success if any of the modules return success''.

In an earlier section, we had mentioned integration of smart cards with PAM. Though we feel that integration should be straight forward from the PAM architecture point of view, there may be some issues with implementation because the interfaces to the smart cards have not yet been standardized.

One possible extension to PAM is to allow the passing of module-specific data between applications and PAM modules. For example, the `login` program likes to build its new environment from a select list of variables, yet the DCE module needs the `KRB5CCNAME` variable to be exported to the child process. For now we have modified the `login` program to explicitly export the `KRB5CCNAME` variable.

Administrative tools are needed to help system administrators modify `pam.conf`, and perform sanity checks on it (i.e., a `pam_check` utility).

12. CONCLUSION

The PAM framework and the module interfaces provide pluggability for user authentication, as well as for account, session and password management. The PAM architecture can be used by `login` and by all other system-entry services, 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 provides easy integration of smart cards into the system.

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.

13. ACKNOWLEDGEMENTS

PAM development has spanned several release cycles at SunSoft. 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, Charlie Lai, and Roland Schemers made very significant enhancements to the PAM interfaces and took the project to completion within a very short time. Kathy Slattery wrote the PAM documentation. John Perry integrated PAM within the CDE framework.

APPENDIX A. PAM API'S

This appendix gives an informal description of the various interfaces of PAM. Since the goal here is just for the reader to get a working knowledge about the PAM interfaces, not all flags and options have been fully defined and explained. The API's described here are subject to change.

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

A.1. Framework Layer API's

```
int
pam_start(
    char *service_name,
    char *user,
    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, the address of 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 text information.

```
int
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()`.

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

int
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. This is useful 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 a pointer to the object, `item`, and its type, `item_type`. `pam_get_item()` is passed the address of the pointer, `item`, which is assigned the address of the requested object.

The `item_type` is one of the following:

Table 5: Possible Values for Item_type

Item Name	Description
PAM_SERVICE	The service name
PAM_USER	The user name
PAM_TTY	The tty name
PAM_RHOST	The remote host name
PAM_CONV	The pam_conv structure
PAM_AUTHTOK	The authentication token (password)
PAM_OLDAUTHTOK	The old authentication token
PAM_RUSER	The remote user name

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(
    int errnum
);
```

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

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

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.

```
int
pam_get_data(
    pam_handle_t *pamh,
    char *module_data_name,
    void **datap
);
```

The `pam_get_data()` function obtains module-specific data from the PAM handle stored previously by the `pam_get_data()` function. The `module_data_name` uniquely specifies the name for which data has to be obtained. This function is normally used to retrieve module specific state information.

A.2. Authentication API's

```
int
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()`, and is referenced by the authentication handle, `pamh`.

```
int
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

```
int
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()`, and is referenced by the authentication handle, `pamh`.

A.4. Session Management API's

```
int
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 which use PAM should invoke `pam_open_session()` when beginning a new session.

```
int
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's

```
int
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`. After the call, the authentication token of the user will be changed in accordance with the authentication module configured on the system.

APPENDIX B. SAMPLE PAM APPLICATION

This appendix shows a sample `login` application which uses the PAM API's. 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 API's.

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

static int login_conv(int num_msg, struct pam_message **msg,
    struct pam_response **response, void *appdata_ptr);

static struct pam_conv pam_conv = {login_conv, NULL};

static pam_handle_t *pamh;     /* Authentication handle */

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

    /*
     * Call pam_start to initiate a PAM authentication operation
     */

    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(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);
    } else {
         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.
 * This should be done before pam_setcred because
 * the 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 */

/*
 * Add DCE/Kerberos cred name, if any.
```

```
     * XXX - The module specific stuff should be removed from login
     * program eventually.  This is better placed in DCE module and
     * will be once PAM has routines for "exporting" environment
     * variables.
     */
    krb5p = getenv("KRB5CCNAME");
    if (krb5p != NULL) {
        ENVSTRNCAT(krb5ccname, krb5p);
        envinit[basicenv++] = krb5ccname;
    }
    environ = envinit; /* Switch to the new environment. */
    exec_the_shell();

    /* All done */
}

/*
 * 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);
    /*NOTREACHED*/
}

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

static 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:
                /* add code here to log error message, etc */
            break;
        }
    }
    return (PAM_SUCCESS);
}
```

APPENDIX C. DCE MODULE

This appendix describes a sample implementation of a DCE PAM module. In order to simplify the description, we do not address the issues raised by password-mapping or stacking. The intent is to show which DCE calls are being made by the DCE module.

The `pam_sm_*()` functions implement the PAM SPI functions which are called from the PAM API functions.

C.1. DCE Authentication Management

The algorithm for authenticating with DCE (not including error checking, prompting for passwords, etc.) is as follows:

```
pam_sm_authenticate()
{
    sec_login_setup_identity(...);
    pam_set_data(...);
    sec_login_valid_and_cert_ident(...);
}

pam_sm_setcred()
{
    pam_get_data(...);
    sec_login_set_context(...);
}
```

The `pam_sm_authenticate()` function for DCE uses the `pam_set_data()` and `pam_get_data()` functions to keep state (like the `sec_login_handle_t` context) between calls. The following cleanup function is also registered and gets called when `pam_end()` is called:

```
dce_cleanup()
{
    if (/* PAM_SUCCESS and
            sec_login_valid_and_cert_ident success */) {
       sec_login_release_context(...);
    } else {
       sec_login_purge_context(...);
    }
}
```

If everything was successful we release the login context, but leave the credentials file intact. If the status passed to `pam_end()` was not `PAM_SUCCESS` (i.e., a required module failed) we purge the login context which also removes the credentials file.

C.2. DCE Account Management

The algorithm for DCE account management is as follows:

```
pam_sm_acct_mgmt()
{
    pam_get_data(...);
    sec_login_inquire_net_info(...);
    /* check for expired password and account */
    sec_login_free_net_info(...);
}
```

The `sec_login_inquire_net_info()` function is called to obtain information about when the user's account and/or password are going to expire. A warning message is displayed (using the conversation function) if the user's account or password is going to expire in the near future, or has expired. These warning messages can be disabled using the `nowarn` option in the `pam.conf` file.

C.3. DCE Session Management

The DCE session management functions are currently empty. They could be modified to optionally remove the DCE credentials file upon logout, etc.

C.4. DCE Password Management

The algorithm for DCE password management is as follows:

```
pam_sm_chauthtok
{
    sec_rgy_site_open(...);
    sec_rgy_acct_lookup(...);
    sec_rgy_acct_passwd(...);
    sec_rgy_site_close(...);
}
```

The `sec_rgy_acct_passwd()` function is called to change the user's password in the DCE registry.

REFERENCES

[Adamson 95] W. A. Adamson, J. Rees, and P. Honeyman, ''Joining Security Realms: A Single Login for Netware and Kerberos'', CITI Technical Report 95-1, Center for Information Technology Integration, University of Michigan, Ann Arbor, MI, February 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.

[Rivest 78] R. L. Rivest, A. Shamir, and L. Adleman., ''A Method for Obtaining Digital Signatures and Pubic-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.

[Steiner 88] J.G. Steiner, B. C. Neuman, and J. I. Schiller, ''Kerberos, An Authentication Service for Open Network Systems'', in Proceedings of the Winter USENIX Conference, Dallas, Jan 1988.

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

[XFN 94] ‘‘Federated Naming: the XFN Specifications’’, X/Open Preliminary Specification, X/Open Document #P403, ISBN:1-85912-045-8, X/Open Co. Ltd., July 1994.

AUTHOR’S ADDRESS

Vipin Samar
SunSoft, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
USA
Internet email: vipin@eng.sun.com
Telephone: +1-415-336-1002

Roland J. Schemers III
SunSoft, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
USA
Internet email: schemers@eng.sun.com
Telephone: +1-415-336-1035