Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

DS2OS - Deliverable B
Operating System Technologies

Giovanni Schmid, Alessandra Rossi

RT-ICAR-NA-2010-03 **Dicembre 2010**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR) – Sede di Napoli, Via P. Castellino 111, I-80131 Napoli, Tel: +39-0816139508, Fax: +39-0816139531, e-mail: napoli@icar.cnr.it, URL: *www.na.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

DS2OS - Deliverable B
Operating System Technologies

Giovanni Schmid[1], Alessandra Rossi

Rapporto Tecnico N.: **RT-ICAR-NA-2010-03**

Data: **Dicembre 2010**

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sede di Napoli, Via P. Castellino 111, 80131 Napoli

DS2OS - Deliverable B
Operating System Technologies

Giovanni Schmid, Alessandra Rossi

December 1, 2010

1 Introduction

The Distributed Security-Oriented Operating Systems (DS2OS) Project aims to integrate Dynamic Delegation facilities into open-source operating systems technologies, in order to obtain the building blocks of fully scalable and interoperable distributed environments.

Dynamic Delegation (DD) is an access control facility which allows users or applications on a given host to grant specific authorizations to remote principals about resources they own as a result of their accounting profiles on such host. Thus, DD represents a fully distributed, authorization-based access control that, as argued in Deliverable A, could constitute the basis for next generation collaborative environments.

As we extensively explained and justified again in Deliverable A, one main concern of DS2OS Project is to enforce DD directly at the operating system level, instead of implementing it as middleware, like for direct delegation [4]. That is also because DD is intended as the application of the direct delegation approach not only to (human) users, but to applications and processes, too [1]

In this Deliverable, we focus on the operating system technologies withstanding our implementation. One main requirement of DD would be its portability to the main OS platforms; however, that is practically unattainable today, because of the absence of standards - or at least some convergence - on frameworks and APIs concerning OS's access control technologies. Section 2 gives some general information on the open source operating system we choose as development platform, illustrating the main factors which motivated our choice.

Another main requirement for DD would be its "neutrality" w.r.t. system services and applications; that is, DD should be neither application specific nor depend on any specific system service. Although our design is general enough to encompass that neutrality, the implementation efforts in the DS2OS project have been focused to the implementation of DD for system entry services (specifically, the Secure Shell service) due to resource's and time limits. In Section 3 we review the general framework for Unix-like entry services to point out the main ingredients which are of relevance for DD. Unix-like entry services rely upon Pluggable Authentication Modules (PAM) framework [18], so our implementation heavily concerns PAM extensions in order to accomodate the functionalities of DD in terms of authentication and authorization, as described in Deliverable D. Section 5 introduces PAM related technologies, and Appendix A constitutes a PAM library reference.

A core aspect of the implementation of DD at the OS level relates to the enforcement of an appropriate *security context* for principals having the *guest* and *sponsor* profiles. Thus, a

[1]We refer collectively to these entities as *principals*, in some analogy with the FIPS definition [32]; see Deliverable A

consistent part of this Deliverable is Section 4, concerning the illustration of the file system and process protection mechanisms upon which we built our design.

2 The platform of choice and its current state

With the introduction of the release 2.10 of Solaris in 2005, Sun Microsystems (now an Oracle Corporation subsidiary) implemented many innovative features and technologies. Moreover, radical changes were done to the Solaris software development architecture itself. Many of the Solaris related software technologies were made open source, and an open source project and software repository (OpenSolaris) [6] was put as the innovation engine for the enterprise, mission critical oriented platform (Solaris). In the following five years, that has resulted in a boosting environment for interesting advancements in Solaris, and - more generally - in the theory and practice of operating systems.

After the acquisition of Sun Microsystems by Oracle Corporation in January 2010 many "thinking heads" of the (Open)Solaris Project have moved away from the company, mainly because of their disappointing about Oracle's vision and strategy concerning truly open-source projects, which has resulted in substantial changes in the management of the OpenSolaris project and in the resignation of the OpenSolaris Governing Board on August 23, 2010. This detrimental process at the expense of "openess" in the new roadmap for Solaris technologies, has however fostered very new interesting OpenSolaris-based projects such as Nexenta [31], Reliant Security [15], Joyent [3] and, last but not least, gave rise to Illumos [30], a community maintained derivative of the OpenSolaris ON source, providing a continuation of the OpenSolaris project named OpenIndiana [14].

At the time of writing, two main descendants of the Sun OS platform are available on the market: the Solaris Express 11, released on November 2010 by Oracle, and the OpenIndiana distribution based on build 143 of OpenSolaris. Moreover, various minor distros exist, and porting projects of the most innovative OpenSolaris technologies to other operating systems have been started over time. Thus, the heritage of Opensolaris is well giving its fruits and it will probably have a long breed also in the open source community. That roots its main cause in a bunch of new OS technologies offering unique properties around file and data management, security and namespace isolation, fault management, and observability.

For these reasons, we decided to carry out the DS2OS Project on OpenSolaris. In the sequel, for the sake of simplicity, we will use the term OpenSolaris to refer in general to Solaris-related technologies and issues.

The main innovative aspects and/or subsystems of OpenSolaris are the following:

- **Predictive Self-Healing** [13] is a fault management architecture that enables a simplified management and fault recovery model for systems and services, wherein traditional error messages intended for humans are replaced by binary telemetry events consumed by software components that automatically diagnose the underlying fault or defect;

- **Service Management Facility** (SMF) [9] is part of the Predictive Self-Healing technology, and defines a programming model for providing persistently running applications called services;

- **Dtrace** is the acronym for **Dynamic Tracing Framework** [7, 22], an advanced observability technology for troubleshooting kernel and application problems on production systems in real time. DTrace can be used to get a global overview of a running system, or much more fine-grained information, such as the kernel code path that was executed as a result of a given application function call, or a list of the processes accessing a specific file. DTrace has its own scripting language; it resembles to the ”C” language and was called ”D”.

- **ZFS** or **Zettabyte Filesystem** [10, 24] is an high performance, high capacity filesystem that actually realizes an integration of the concepts of filesystem and volume management, so allowing the flexibility of the pooled storage model with a much easier administration, and at a fraction of its costs. ZFS uses a transactional, copy-on-write update model providing for snapshots and copy-on-write clones, continuous integrity checking and automatic repair. ZFS also provides built-in compression and encryption, fast and easy data replication, and a logical administrative model.

- **Resource Management** has a long tradition in Solaris; indeed, it was first introduced in 1998. However, its importance and functionalities have been greatly expanded in OpenSolaris, when it got a really important part of the Zone's technology (see below). Resource Management refers to the collection of facilities used to configure, monitor, and control the system's resource allocation to running processes. In OpenSolaris, the grouping of processes onto which to impose limits can be specified at the *task*, *project* or *zone* level. Tasks are simply groups of processes, as those constituting a shell job. A project is a group of tasks; e.g. the conjunction of the tasks ”apache” and ”mysql”. Finally, a zone - from the perspective of the resource management - is just a group of one or more projects. A limit on a zone would limit all projects in that zone, one on a project would limit all tasks grouped by that project, and a limit on a task would limit all processes grouped by the task.

- **Zones** [11, 23] are used to create virtualized environments for running software in a secure and isolated way. As FreeBSD *jails* [2] OpenSolaris zones shares only one istance of the operating system, and multiple runtime environments can coexist. Within a zone, it appears to users and applications that they are running on a standalone system. Users and processes outside the zone cannot be seen or affected, there are no name conflicts on files or ports, and the behavior of software within the zone is contained to that zone. Together with the resource management features previously seen, zones give rise to **Containers**, which allow for the splitting of physical resources among different virtual environments.

- **Process Rights Management** [26] is a very fine-grained and flexible protection framework for administrative tasks that previously required superuser rights. Process rights management uses **process privileges** [21, 5] to restrict processes at the command, user, role, or system level. A privilege is a discrete right that a process requires to perform an operation. The system restricts processes to only those privileges that are required to perform the current task.

Process privileges, containers and ZFS were particularly useful for our scopes. Nowdays, no other operating system offers these technologies natively, and at the maturity level of OpenSolaris.
Containers are particular useful to provide insulation among multiple running environments on the same physical host, and the approach we have followed to build the “guests' views” of the system was partially inspired by this technology, as shown in Deliverable D.
Again in Deliverable D, we will show how the ability to create effortless user-tailored ZFSes is used to build the home directories of guest users, and how the advanced ZFS access control lists (ACL) allow for the enforcement of fine-grain, multilevel filesystem security policies.

Process privileges are not a novelty in the Unix arena; on the contrary, they can even be traced back to more than twenty years ago, when the IEEE/PASC/SEC working group began its work on the denition of the POSIX 1003.1e standard which, however, was never completed. Actually, both Linux, Sgi IRIX and some FreeBSD distributions provide process privilege implementations based, at some extent, on POSIX 1003.1e draft [1]. Nevertheless, such implementations are mutually incompatible and, worse, they cannot coexist with the traditional Unix superuser security model. The approach chosen quite recently for OpenSolaris is more comprehensive; indeed, it is fully compatible with the superuser model and introduces the so-called basic privilege set, which is a very useful feature from our viewpoint. Moreover, process privileges in Opensolaris are strictly integrated with Role-based access control (RBAC), as we explain in Section 4. Basic privileges are those granted to ordinary user processes; thus, restricting such set allows to represent on the system principals (like our *guests*) which must have less rights than ordinary users. One more step toward our research targets is represented by the *Fine Grained Access Policy* (FGAP) Project, a project endorsed by the OpenSolaris Community that plans to provide a mechanism to "sandbox" applications running under user accounts, by first removing basic privileges and then granting them on a case-by- case basis [12]. To this end, the set of *basic* privileges (4.2) need to grow to include fine grained capabilities, and we are going to work in synergy with this project to include a set of privileges suitable to express the capabilities to be granted or denied to a guest profile [2]. Process privileges are first outlined in Section 4.2, and reviewed in Deliverable D expecially with the goal of expanding process rights.

3 Entry system services workflow

An **Entry system service** is an application intended to allow an authorized user to log into a computer system and to perform various tasks using the system resources. Access to these resources must be granted or denied with respect to the *access control policy* in force onto the system for that user. Thus, an entry system service needs to match a user identity with the set of authorizations which specifies what resources the user is allowed to access and what kind of use he is granted for such resources. General-purpose, modern operating systems provide the **Access control framework**, a (logical) subsystem intended to control the access to a resource (*target*) by an entity making an access request (*initiator*). It implements various mechanisms and levels of abstraction to accomplish the above tasks, and in Unix-like OSs is realized as a three-stages pipeline as follows.

1. The first stage, a.k.a. **Identification stage**, is intended to verify the existence of an identifier for the initiator with respect to a userland namespace (e.g. the *login- names* namespace, the *NIS/NIS+* or *LDAP* namespaces). If the above verification succeded, then this stage is responsible of retrieving some basic information for the kernel to operate with the initiator. This information include the *User IDentifier (UID)* of the initiator, a non negative integer which uniquely identify such user for the kernel.

   The workflow for this stage is as follows. Using a suitable service interface (eventually realized throught a client application), the initiator submit their access request, providing an identifier I and an authentication token A_p. The identification subsystem searches a repository of information for I. If it successes, then the name R of the repository is returned for being processed by the second stage.

[2]A first enlargement of this set appeared in Solaris 10 9/10 (update 9), where was introduced a privilege for creating sockets and network end points.

2. The **Authentication stage** is intended to prove the identity of the initiator as declared by the user in the first stage. This is achieved throught the verification of an authentication token (usually a password), which is supposed to be only known to such user.

   In this stage, the authentication subsystem checks the matching (usually throught a cryptographic hash function) of the authentication token A_s related to I in R with the one A_p provided by the initiator. If A_p and A_s match, then the initiator is allowed to entry into the system, otherwise a suitable message is displayed to the user informing that loggin in failed.

3. Finally, it is demanded to the **Authorization stage** to grant or deny the initiatior, as identified by I , access to resources during their activities on the system. This is achieved by matching I, and eventually an authorization profile for I, against the access constrains enforced onto system resources by the local access control policy; as detailed in Section 4 for the case of OpenSolaris.

As for any Unix-like system, entry services in OpenSolaris are tipically implemented as *PAM consumers*; that is, they make use - partially or integrally - of PAM services for their authentication related tasks. Moreover, the naming services required in the identification stage are implemented throught the Name Service Switch (NSS) framework. These aspects are of some relevance for our design, and will be treated in Section 5 and in the Appendix A.

4 The Security Context

In the traditional Unix security model, the actions that a process, a user or an application can perform within the system are checked in a simple way. The root user, also referred as superuser, is all-powerful. A process has all privileges if and only if its effective UID (EUID) is equal to the UID of the root user, that is UID=0. Instead, if a process EUID is different from 0, then the process runs under the context of an ordinary user, considered unprivileged by the operating system. A not-root process can perform only actions on objects with explicit grant access rights for the user and the groups of users to which she belong on the basis of the OS mechanisms enforcing security at the file system level (see Section 4.3).

This simplistic security context derives, as explained in Deliverable A, from the fact that the user profile on such systems for a principal p is defined by

$$p \longrightarrow (u, a) \longrightarrow \{UID, GID_1, ..., GID_n\} \ ;$$

that is, the only authorizations granted to p are those derived by matching the credentials (EUID/EGID) of the processes spawned by the user u (as inherited by its profile) towards file ownership and permissions.

The traditional Unix security model, therefore, enforces a UID-based privilege mechanism with just one privilege: the root privilege. This is not a flexible mechanism: it is not just that root is too much powerful, but that regular users or processes are not powerful enough to perform some required actions. Indeed, this model does allow neither to restrict a process to a limited set of granted actions, nor to add or drop actions at occurency. For example, in order to perfom its tasks, a mail server needs rights to access a privileged network port, and in a such kind of security model the above can be accomplished only by granting to the mail server the root credentials.

The lack of fine-grained control over process and user credentials turns out in a poor implementation of the *least privilege principle* [17] and, ultimately, in scarce security. Most Unix-like operating systems, because of the limitations imposed by their security models, have to run a large number of their applications and user processes with root privileges, giving them the capability to read and modify other processes, memory, I/O devices, etc. Many exploits rely on *privileges escalation* to gain superuser access to a system via bugs like buffer overflows or incomplete mediations.

At its outmost level, system security in OpenSolaris in enforced through a **Role-based access control (RBAC)** policy. This kind of access control model - introduced in 1992 by D. Ferraiolo and R. Kuhn [16], and lately adopted as the ANSI Standard 359-2004 - allows system administrators to delegate the administrative control of parts of the system to users. Moreover, it provides the decoupling of administrative tasks from actual users.

OpenSolaris RCBAC traces back to *Trusted Solaris* (SunOS 4.1.3 based versions); the implementation was slighly different but the basic concepts are the same. In OpenSolaris, RBAC was integrated with process privileges and the service management facility, obtaining a very flexible security model which allows for a very strict enforcement of the least privilege principle [19].

The OpenSolaris RBAC implementation builds upon the two main concepts of *role*, and *user security attributes*.

A **Role** is similar to a normal user in that it has its own UID, GID, home directory, shell and password. Likewise users, roles are specified through the `password` database. However, a role cannot be used to log directly into a system either at the console or by any remote access service. Moreover, a role can only be accessed by a user who has previously been authorized to assume that role. Finally, roles are not permitted to assume other roles. To enable users to assume a role, their accounts must be modified to reflect this permission.

User security attributes are defined for each account in the `user_attr` database. Each record of `user_attr` composes of five (eventually void) colon-separated fields: `user`, `qualifier`, `res1`, `res2` and `attr`. The only actually used fields are `user` and `attr`, since the others are reserved for a future use. The `user` field defines the name of the user as specified in the `passwd` database, whilst the `attr` field is an optional list of semicolon-separated key-value pairs that describe the security attributes for `user`. Zero or more keys may be specified. OpenSolaris provides actually many different kind of user security attributes. The followings are those provided by a box without *trusted extensions* [21, 29]:

- **Authorizations** are permissions that can be granted to a standard user or role for performing a class of actions otherwise prohibited by security policy. Very often, authorizations are used in concert with privileged programs or services for the purpose of access control; see Section 4.1.

- A **Privilege** is an attribute of a user, a program or a process that is enforced by the kernel and is used to override a standard security policy. Privileges are assigned to a user through the `defaultpriv` and `limitpriv` sets: `defaultpriv` represents the default set of privileges assigned to a user's inheritable set upon login, whereas `limitpriv` is the maximum set of privileges a user or any process started by the user, whether through the `su` command or any other means, can obtain. See Section 4.2.

- **Execution attributes** are security attributes that enable a command to perform an operation. Execution attributes can be represented by standard process credentials (e.g. EUID, EGID, etc.) or privileges. Along with the rights profile mechanism (see below), execution attributes enable to isolate privileged commands. Instead of changing the ID on a command that anyone can access, one can place the command with execution attributes in a rights profile. A user or role with that rights profile can then run the program without having to become superuser.

OpenSolaris RBAC implementation collects superuser capabilities into *rights profiles*. A **Rights Profile** is a set of *execution attributes*, *authorizations* and/or other rights profiles used for subsequent assignment to a role or user. Multiple profiles can be assigned to the same role or user. Profiles are defined through the `prof_attr` and `exec_attr` databases. Specifically, `prof_attr` is the local database of available profiles and their related authorizations, and `exec_attr` matches such profiles with sets of commands with execution attributes. A rights profile is intended to be used in conjunction with a *profile shell*. At the time of writing, the following profile shells are available: `pfsh`, `pfcsh` and `pfksh`. These shells represent the profile-aware versions of standard shells, obtained by invoking `pfexec` rather than `exec`. Running a profile shell for a role or a user overrides their standard security attributes, since these shell load `exec_attr`. A default profile for any user is assigned in `policy.conf`. If no profiles are assigned, the profile shells do not allow the user to execute.

From the point of view of DS2OS Project, OpenSolaris RBAC implementation offers two features which are crucial for implementing the *sponsor* and the *guest* profiles (see Deliverable A), and that are unavailable in the traditional OS security models. They are, respectively:

- with RBAC and directly-assigned privileges and authorizations, a primary system administrator or supersuser can easily create (pseudo)users with administrative capabilities but without full superuser capability;
- with RBAC and removed privileges, a user with administrative capabilities but without full superuser capability can easily create (pseudo)users with fewer capabilities than ordinary users.

Thus, it takes relevance to detail about authorizations and privileges. A further important topic for our purposes is represented by the OpenSolaris mechanisms to enforce file system security. These arguments are treated in the following three sections.

4.1 Authorizations

Authorizations describe types of operations on classes of objects, and are defined in the `auth_attr` database. Authorizations enforce policy at the user application level: RBAC-compliant applications can check users' authorizations in order to grant access to specific operations within the application or to the application itself. This check replaces the check in conventional Unix applications for UID=0.

Authorizations are represented by fully qualified names like Java class names. An authorization name is a unique string that identifies the organization that created the authorization and the functionality it controls. Following the Java naming convention, the hierarchical individual components of an authorization are separated by a dot, starting with the reverse order Internet domain of the creating organization, and ending with the specific function within a class of authorizations. For example, if a user has the `solaris.jobs.admin` authorization, she is able to read or write to other users' files. Without such authorization, the user can operate on owned files only. An asterisk is used as a wildcard to indicate all authorization in a class. For example, `solaris.*` encompasses all solaris authorizations, whereas `solaris.jobs.*` covers the authorizations `solaris.jobs.admin`, `solaris.jobs.grant` and `solaris.jobs.user`. When an authorization name ends with the keyword `grant`, it is used to support fine-grained delegation. For example, a principal owning the `solaris.jobs.grant` authorization can delegate to other users the administration of `cron` and `at` facilities.

Authorization checks are provided for both C/C++ and Java programs. The C/C++ API is implemented in the `libsecdb` library, whilst the Java APIs are implemented using the Java Native Interface (JNI), which uses the same functions in `libsecdb`, making the two implementations compatible. The most important authorization function is `chkauthattr`, which verifies whether or not a principal has a given authorization. System-wide default authorizations are set-up through the file `/etc/security/policy.conf`: `chkauthattr` first reads such file, checking if the given authorization is contained in the keyword `AUTHS_GRANTED`, or in any of the user profiles listed via the key `PROFS_GRANTED`. Such keys indicate indeed the authorizations and profiles allowed by default on the system. A part from these defaults, a principal is considered to have been assigned an authorization if either of the following is true:

- The authorization name matches exactly any authorization assigned in the `user_attr` or `prof_attr` databases (authorization names are case-sensitive).
- The authorization name suffix is not the keyword `grant` and the authorization name matches any authorization up to the asterisk (*) character assigned in the `user_attr` or `prof_attr` databases.

Applications that check for authorizations include but are not limited to audit administration commands (`auditconfig`,`auditreduce`, etc.), printer administration commands (e.g. `lpadmin` and `lpfilter`), batch job-related commands (`at`, `atq`, `cron`, etc.) and device-oriented commands (`allocate`, `cdrw`, etc.).

4.2 Privileges

The simple “check if EUID is equal or not to 0” has been replaced in OpenSolaris with the ability to grant one or more specific **Privileges** that enable processes to perform otherwise restricted operations. Whilst authorizations serve to overcome the ”all-or-nothing”, UID-based approach at the application level, privileges get the same goal but at the kernel level. Without the proper privilege, a process is prevented from performing some operations by the kernel. The privilege model provides greater security than the superuser model. Privileges that have been removed from a process cannot be exploited. Process privileges prevent a program or administrative account from gaining access to all capabilities, and can provide an additional safeguard for sensitive files, where *discretionaty access control* (DAC) protections alone can be exploited to gain access (see Section 4.3).

In Solaris Developer Edition 01/08 (a.k.a. Nevada 65b) there were 68 privileges, and this set has been costantly increased over time by developers to encompass new requirements. At the time of writing, the latest OpenSolaris distribution ufficially released by Sun/Oracle (i.e. Opensolaris build 133) counts 79 privileges.

Privileges can be grouped on the basis of the subsystem they affect:

- **File privileges** operate on file system objects. For example, the `file_dac_write` privilege overrides DAC when writing to files;
- **IPC privileges** override IPC object access controls. For example, the `ipc_dac_read` privilege enables a process to read remote shared memory that is protected by DAC;
- **Network privileges** give access to specific network functionality. For example, the `net_rawaccess` privilege enables a device to connect to the network;

- **Process privileges** allow processes to modify restricted properties of the process itself. Process privileges include privileges that have a very limited effect. For example, the `proc_clock_highres` privilege enables a process to use high resolution timers;
- **System privileges** give processes unrestricted access to various system properties. For example, the `sys_linkdir` privilege enables a process to make and break hard links to directories.

The OpenSolaris privilege model provides four sets of privileges per process in addition to the legacy process credentials. These sets (plus an opaque set introduced for technical reasons) allow for the implementation of the *least privilege principle* through the *privilege bracketing* programming technique [28]. The four programmable sets and the opaque one are as follows:

- **Effective** set, the set E of currently active privileges for the running process. This set represents the counterpart in the OpenSolaris security model of the effective UID (EUID) in the Unix traditional model. Indeed, the access control for a userland process is enforced by checking whether the required privilege is a member of the set E for such a process. A process can add privileges that are in the *permitted* set (see below) to the effective set, and can also remove privileges from E.
- **Permitted** set, that is the maximum set P of privileges available to the process. The permitted set constitutes a superset of the effective set E ($E \subseteq P$). Privileges can be removed from the permitted set, but privileges cannot be added to the set. When privileges are removed from P they are automatically removed from E, in order to satisfy the previous relationship. Privileges removed from P cannot later be restored by the process.
- **Inheritable** set: this is the set I of privileges that will be inherited by a child process upon an `exec` system call. Privileges removed from I cannot be carried over to the process child processes. I represents the extent to which a child process can gain more privileges w.r.t. its parent. When privileges are removed from P, they are not removed from I. So, in general, I is not a subset of the permitted set P; however, I must be a subset of the limit set L (see below). This schema allows for the analogous in the privilege context of the traditional set-ID mechanism. After the call to `exec`, P and E are equal, except in the special case of a setuid program. For a setuid program, after the call to `exec`, the set of privileges given by $I \subseteq L$ is assigned to P and E for that process.
- The **Limit** set L constitutes the maximum possible set of privileges for any child of the current process, and the process itself. By default, L is the set of all privileges. Processes can shrink L but can never extend it. Privileges removed from L are automatically removed from E, P and I, and they can never be obtained again by any of the process's children nor the process itself. Thus, the following relationships hold $E \subseteq P \subseteq P \cup I \subseteq L$.
- The **Saved** set is the counterpart of the saved UID (SUID): this set allows for the implementation of the *least privilege principle* and it cannot be directly managed by a process.

Besides the above sets, the kernel recognizes a **Basic** privilege set, which corresponds to the set of privileges for standard users and processes. On an unmodified system, each user's initial inheritable set I equals the basic set at login. Conversely than the inheritable set, the basic set cannot be modified.

Programs that are coded to use privileges are called privilege-aware programs. A privilege-aware program turns on and off the use of privilege during program execution. Privileges are managed by a privilege-aware program through the system call `setppriv`: using it a process can add privileges from P to E and I, and remove privileges from all these three sets. Adding a privilege to E or I which does not belong to P is not permitted. Likewise, adding privileges to P and/or L is not permitted.
Whatever the performed action, `setppriv` is implemented is such a way the following conditions are always met, where X and X' denote a privilege set before and after calling `setppriv` by the running process, respectively:

$$I' \subseteq I \cup P, \quad P' \subseteq P, \quad E' \subseteq P', \quad S' \subseteq P' \ .$$

Privilege-aware programs include but are not limited to Kerberos commands (e.g. `kadmin`, `kprop`), network commands (`ifconfig`, `snoop`, etc.), file and file system commands (e.g. `chmod`, `mount`), and commands that control processes (such as `kill` and `pcred`).

The system administrator is responsible for assigning privileges. Typically, privileges are assigned as execution attributes to commands in a rights profile. The rights profile is then assigned to a role or to a user. CLIs to perform these actions are `smuser` and `smrole`. Privileges can also be assigned directly by expanding the initial inheritable set of privileges for users, roles, or systems. This is however a riskier way to assign privileges: since all privileges in the inheritable set are in the permitted and effective sets, all commands that the user or role types in a shell can use the directly assigned privileges. Directly assigned privileges enable a user or role to easily perform operations that can be outside the bounds of their administrative responsiblities.

4.3 File system security

In general purpose OSes, file system security roots in the **Discretionary Access Control (DAC)** model, that is "a means of restricting access to objects based on the identity of subjects and/or groups to which they belong" [33]. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) to any other subject. DAC policies can be represented through *access control matrices* or, equivalently, a set of *access control lists*.

An **Access control list (ACL)** is a list of permissions attached to an object. An ACL specifies which users and system processes are granted access to objects. In a typical ACL, each entry in the list specifies a subject and an operation. For example, the entry (Bob, delete) on the ACL for a file gives Bob permission to delete it.

Traditionally, Unix systems share a simple ACL model in which every file system object is associated with three sets of permissions that define access for the owner, the owning group, and the rest of users (simply called "others"). Each set may contain Read (r), Write (w), and Execute (x) permissions. This scheme is implemented using only nine bits for each object. In addition to these nine bits, the Set User Id, Set Group Id, and Sticky bits are used for a number of special cases [27].

Besides the above basic ACL model, most of the Unix-like file systems support the so called POSIX ACLs, based on POSIX drafts never completed. POSIX ACLs are additional sets of read/write/execute triplets (rwx) that can be added on to files, directories, devices, or any other file system objects on per user or per group basis.

OpenSolaris goes further, supporting NFSv4 ACLs [25] in its new Zettabyte filesystem (ZFS). NFSv4 ACLs address several limitations with the POSIX ACLs, such as not being able to revoke permissions and being coarse-grained. NFSv4 ACLs are made up of three parts: the part for the owner, the part for the group of the owner, and one part for everyone (other). These parts correspond to the traditional Unix classification of subjects, but now each part has an allow list and a deny list, making permissions much more granular. In addition to the the default three parts (and analogously than in POSIX ACLs), one can add NFSv4 ACLs for individual users and for specific groups.

5 PAM and its OpenSolaris implementation

The Pluggable Authentication Modules (PAM) framework [20] offers advantages both to application developers and system administrators.
Developers can use PAM services without having to know the semantics of the authentication policy, since algorithms are centrally supplied. Moreover, the algorithms can be modified independently of the individual applications, thus decoupling the authentication mechanisms from the application layer.
Administrators can instead tailor the authentication process to the needs of a particular system without having to change any applications. Adjustments are indeed made through the PAM configuration file, which allows for an easy configuration of the authentication flow for each service.

The PAM framework composes of a set of shared libraries, called *services modules*, an Application Programming Interface (API), a Service Programming Interface (SPI), and the configuration file `pam.conf`. Service modules implement one or more PAM services. PAM actually offers four types of services:

- **Authentication**, for granting users access to an account or an application service. Modules that provide this service authenticate users and set up user credentials.
- **Account management**, for determining whether the current user's account is valid. Modules that provide this service can check password or account expiration and time-restricted access.
- **Session management**, for setting up and terminating login sessions.
- **Password management**, for enforcing password strength rules and performing authentication token updates.

PAM services should be implemented in separate modules, since the use of simple modules with well-defined tasks increases configuration flexibility. In the sequel, we will refer to MPI as the programming interface to build services modules. As we shall see, MPI composes of SPI plus some interfaces which belong to the API.

The API plus the SPI constitute the PAM library, which in OpenSolaris is referenced as `libpam` and implemented as the shared object library file `/usr/lib/libpam.so.1` The interfaces in `libpam` are stable, and each of them is multi-threading-safe only if each thread within the multithreaded application uses its own *PAM handle* (s. below). The PAM API is composed of fiftheen functions subdivided into six categories. These categories correspond to the four types of PAM security services discussed above, plus two more management-oriented categories: *PAM session* and *PAM environment*. They are shown in Table 1.

Any AP interface is named as `pam_`*task*, where *task* indicates the kind of action performed by the interface. For each of the four service categories listed Table 1, dynamically loadable shared modules exist that provides the appropriate service layer functionality upon demand. When an application calls on any of the AP interfaces belonging to these categories, `libpam`

Category	API	MPI	Module type

PAM Session	`pam_start`
	`pam_get_user`
	`pam_set_item`
	`pam_get_item`
	`pam_strerror`
	`pam_end`
		`pam_set_data`
		`pam_get_data`
		`pam_set_item`
		`pam_get_item`
			none

Authentication	`pam_authenticate` `pam_setcred`	`pam_sm_authenticate` `pam_sm_setcred`	`auth`
Account management	`pam_acct_mgmt`	`pam_sm_acct_mgmt`	`account`
Session management	`pam_open_session` `pam_close_session`	`pam_sm_open_session` `pam_sm_close_session`	`session`
Password management	`pam_chauthtok`	`pam_sm_chauthtok`	`password`
PAM Environment	`pam_getenv` `pam_getenvlist`	`pam_putenv`	none

Table 1: PAM application and module interfaces subdivided by categories

reads the configuration file `pam.conf` to determine which modules participate in the operation for the application. Then, in response to the API call `pam_`*task*, the PAM framework calls the corresponding `pam_sm_`*task* SP-interface from the previously retrieved modules. The only difference between the `pam_*` interfaces and their corresponding `pam_sm_*` interfaces is that all the latter interfaces require the two extra parameters `argc` and `argv` to pass service-specific options to the shared modules. They are otherwise identical (see Appendix A). The SPI composes of just six interfaces, and constitutes a subset of the MPI. In turn, MPI composes of the SPI, the majority of APIs, and some functions used to access and update module-specific information from the pam handle.

A sequence of calls to PAM API sharing a common set of state information is referred to as a **PAM session**. PAM sessions are initiated by calling `pam_start`, and end with the `pam_end` function. The `pam_start` function returns a PAM handle which uniquely identifies the session and is used by all the subsequent calls to the library that pertain to that session. A PAM handle is a structure containing all the information related to a PAM session, among which are an array of *items* (see Table 14 in Appendix A), module-specific data (e.g. module state information), and a list of *environment variables* desumed from the environment of the calling application. As a consequence of a call to `pam_start`, some of the above information is initializated and can be passed as opaque data to the underlying modules through the PAM handle.

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, local or networked applications can all use PAM.

After a call to `pam_start`, the types of information conveyed through the handle can be managed as follows:

- The interfaces `pam_get_item` and `pam_set_item` are used by both PAM consumers and modules to read and write a PAM information item, respectively.

- The modules can communicate the environment variables associated with the given identity back to the application with the `pam_putenv` MPI. The application can read

SSH Userauth	PAM Service Name	Description
`none`	`sshd-none`	no authentication
`password`	`sshd-password`	plaintext password-based authentication
`keyboard-interactive`	`sshd-kbdint`	a generic prompt/reply protocol
`pubkey`	`sshd-pubkey`	a public key system where keys are associated with users
`hostbased`	`sshd-hostbased`	a public key system where keys are associated with hosts
`gssapi-with-mic`	`sshd-gssapi`	a GSS-API based method
`gssapi-keyex`	`sshd-gssapi`	a GSS-API based method

Table 2: SSH user authentication types and related PAM service names.

the associated environment variables with the `pam_getenv` and `pam_getenvlist` APIs.

- 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`.

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

5.1 How SunSSH uses PAM

A **PAM consumer** is an application wich makes uses of PAM services. PAM consumers are tipically entry system services such as `login`, `rlogin` and `ssh`; although some different example exist (e.g. `cron` and `su`). As an example of how consumers use the PAM library for user authentication, consider how SunSSH authenticates a user in OpenSolaris.
At the time of writing, the version 2 of SSH protocol (SSHv2) [34] provides the first four user authentication types indicated in Table 2. The two last, GSS-based authentication types are specific of SunSSH [8]. Each type has its own PAM service name.

The daemon sshd uses PAM for the three initial authentication methods as well as for account management, session management, and password management for all authentication methods.

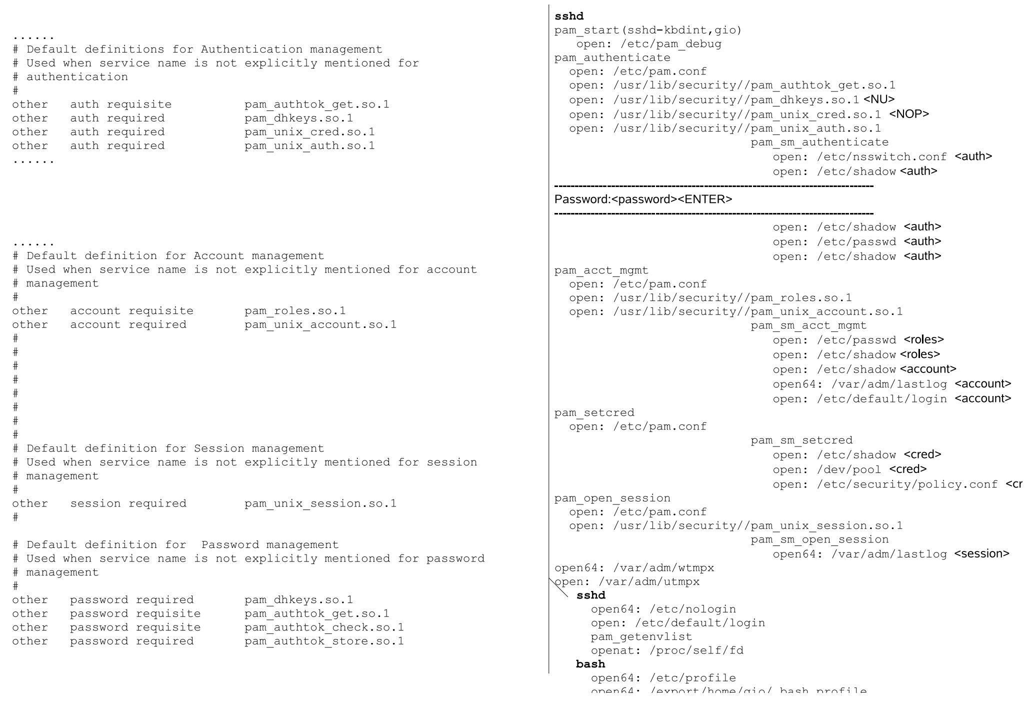
Specifically, sshd calls `pam_authenticate` for the `none`, `password` and `keyboard-interactive` SSHv2 userauth types. Other SSHv2 authentication methods do not call `pam_authenticate`. `pam_acct_mgmt` is called for each authentication method that succeeds. `pam_setcred` and `pam_open_session` are called when authentication succeeds and `pam_close_session` is called when connections are closed. `pam_open_session` and `pam_close_session` are also called when SSHv2 channels with ptys are opened and closed.

If `pam_acct_mgmt` returns `PAM_NEW_AUTHTOK_REQD` (indicating that the user's authentication tokens have expired), and if version 2 of the protocol is in use, then sshd forces the use of "keyboard-interactive" userauth. The "keyboard-interactive" userauth will call `pam_chauthtok` if `pam_acct_mgmt` once again returns `PAM_NEW_AUTHTOK_REQD`. By this means, administrators are able to control what authentication methods are allowed for SSHv2 on a per-user basis.

In what follows, we describe the interactions btw. SunSSH and PAM for the `keyboard-interactive` method. In such a case, in response to a request from a remote client, the `sshd` network daemon forks a new `sshd` process which, after some processing in order to acquire the locale settings, opens the access-control related files and libraries as illustrated in Figure 1, left side. PAM services modules are loaded according to what defined in the `pam.conf` file (see right side of Figure 1). The example refers to a user who has the following entries in the `passwd` and `user_attr` databases:

```
# passwd DB entry
gio:x:100:10:Giovanni Schmid:/export/home/gio:/bin/bash
# user_attr DB entry
gio::::defaultpriv=basic,priv_sys_ip_config;type=normal
```

These entries state that the account refers to a not-privileged, normal user (not a role) with UID=100, GID=10 and `/bin/bash` as start-up program at login. Moreover, the inheritable set of privileges upon login for the user is the basic set plus `priv_sys_ip_config`.



```
......
# Default definitions for Authentication management
# Used when service name is not explicitly mentioned for
# authentication
#
other   auth requisite          pam_authtok_get.so.1
other   auth required           pam_dhkeys.so.1
other   auth required           pam_unix_cred.so.1
other   auth required           pam_unix_auth.so.1
......



......
# Default definition for Account management
# Used when service name is not explicitly mentioned for account
# management
#
other   account requisite       pam_roles.so.1
other   account required        pam_unix_account.so.1
#
#
#
#
#
#
#
#
# Default definition for Session management
# Used when service name is not explicitly mentioned for session
# management
#
other   session required        pam_unix_session.so.1
#

# Default definition for  Password management
# Used when service name is not explicitly mentioned for password
# management
#
other   password required       pam_dhkeys.so.1
other   password requisite      pam_authtok_get.so.1
other   password requisite      pam_authtok_check.so.1
other   password required       pam_authtok_store.so.1
```

```
sshd
pam_start(sshd-kbdint,gio)
   open: /etc/pam_debug
pam_authenticate
  open: /etc/pam.conf
  open: /usr/lib/security//pam_authtok_get.so.1
  open: /usr/lib/security//pam_dhkeys.so.1 <NU>
  open: /usr/lib/security//pam_unix_cred.so.1 <NOP>
  open: /usr/lib/security//pam_unix_auth.so.1
                          pam_sm_authenticate
                             open: /etc/nsswitch.conf <auth>
                             open: /etc/shadow <auth>
--------------------------------------------------------------
Password:<password><ENTER>
--------------------------------------------------------------
                             open: /etc/shadow <auth>
                             open: /etc/passwd <auth>
                             open: /etc/shadow <auth>
pam_acct_mgmt
  open: /etc/pam.conf
  open: /usr/lib/security//pam_roles.so.1
  open: /usr/lib/security//pam_unix_account.so.1
                          pam_sm_acct_mgmt
                             open: /etc/passwd <roles>
                             open: /etc/shadow <roles>
                             open: /etc/shadow <account>
                             open64: /var/adm/lastlog <account>
                             open: /etc/default/login <account>
pam_setcred
  open: /etc/pam.conf
                          pam_sm_setcred
                             open: /etc/shadow <cred>
                             open: /dev/pool <cred>
                             open: /etc/security/policy.conf <cr
pam_open_session
  open: /etc/pam.conf
  open: /usr/lib/security//pam_unix_session.so.1
                          pam_sm_open_session
                             open64: /var/adm/lastlog <session>
open64: /var/adm/wtmpx
open: /var/adm/utmpx
   sshd
     open64: /etc/nologin
     open: /etc/default/login
     pam_getenvlist
     openat: /proc/self/fd
   bash
     open64: /etc/profile
     open64: /export/home/gio/.bash_profile
```

Figure 1: PAM stacks for the SunSSH service as defined in the `pam.conf` file (right side), and PAM modules loaded during a SunSSH run (left side).

Files and libraries involved in the the process (see right side of Figure 1) are as follows:

- `/etc/pam.conf`, the configuration file for PAM. This file defines, for each PAM service category, what stacks of PAM modules must be loaded for each of the PAM-aware system services. Since the SSH service is not explictly named, it follows the stacks related to the `other` flag (see Figure 1).

- `pam_authtok_get`, a PAM module implementing `pam_sm_authenticate` and `pam_sm_chauthtok`, in order to provide password prompting functionality to both the Authentication service and the Password Management service.
- `pam_dhkeys`, which provides functionality to two PAM services: Secure RPC authentication and Secure RPC authentication token management. Secure RPC authentication differs from regular unix authentication because NIS+ and other ONC RPCs use Secure RPC as the underlying security mechanism. This module is opened because of the `pam.conf` file settings for SSH, but it is not actually used for the `kbdint` SSH method.
- `pam_unix_cred` is the PAM user credential authentication module for UNIX. It implements `pam_sm_setcred`, providing Solaris Unix style credential setting. That includes initializing the audit characteristics if not already initialized and setting the user's default and limit privileges.
- `pam_unix_auth` implements `pam_sm_authenticate`, which verifies that the password contained in the PAM item `PAM_AUTHTOK` is the correct password for the user specified in the item `PAM_USER`. If `PAM_REPOSITORY` is specified, then user's password is fetched from that repository. Otherwise, the default `nsswitch.conf` repository is searched for that user.
- `/etc/nsswitch.conf` is the configuration file for the name service switch. It establish which repositories of information have to be searched for each naming service.
- `/etc/shadow` is an access-restricted ASCII system file that stores users' encrypted passwords and related information.
- `/etc/passwd` is a local source of information about users' accounts.
- `pam_roles` implements `pam_sm_acct_mgmt`, providing functionality to verify that a user is authorized to assume a role. It also prevents direct logins to a role. The `user_attr` database is used to determine which users can assume which roles.
- `pam_unix_account` implements `pam_sm_acct_mgmt`, providing functions to validate that the user's account is not locked or expired and that the user's password does not need to be changed. The module retrieves account information from the configured databases in `nsswitch.conf`. In our case, the local database `/etc/shadow` is used.
- `/var/adm/lastlog` is a file contains information about users' last login times. This information is printed out for the user by some entry services (e.g. SSH, Login) and other services (e.g. Finger).
- `/etc/default/login` is a file containing values for various environment variables, traditionally used by the login command and, starting from Solaris Express 10/04 release, by the sshd daemon too. The `/etc/default/login` variables can be overridden by values in the `sshd_config` file.
- `/etc/security/policy.conf` is the file that provides the security policy configuration for user-level attributes.

- pam_unix_session is the session management PAM module for UNIX. It implements pam_sm_open_session and pam_sm_close_session. pam_sm_open_session updates the /var/adm/lastlog file with the information contained in the PAM_USER, PAM_TTY, and PAM_RHOSTS items. pam_unix_account uses this account to determine the previous time the user logged in. pam_sm_close_session is a null function.

- /var/adm/wtmpx contains the history of user access and accounting information for the utmpx database.

- /var/adm/utmpx contains user access and accounting information.

- /etc/nologin contains the message displayed to users attempting to log on to a machine in the process of being shutdown. After displaying the contents of the nologin file, the login procedure terminates, preventing the user from logging onto the machine.

As clearly shown by Figure 1, sshd initiates a PAM session by calling pam_start and by specifying the generic svc service name, which was previously instantiated to ones of the values in column 2 of Table 2. In our case, this value is sshd-kbdint. The call to pam_start returns in pamh the address of the handle which uniquely identifies the session.
Using the previous handle , sshd calls pam_authenticate, pam_acct_mgmt, pam_setcred, and pam_open_session. For conciseness, we will detail in the sequel just about the call to pam_authenticate, the others being similar. See Appendix A for more details about these other APIs.
In response to the call to pam_authenticate, the PAM framework loads all the modules of type auth for the other service entries in the pam.conf file (see Figure 1, left side). Since each of the above modules has its control flag equal to requisite or required, then the pam_sm_authenticate function therein is executed for all of them. However, the code in pam_dhkeys.so is related to RPC authentication, so it adds nothing in our case; whilst the function in pam_unix_cred.so does not make any operation at all. Thus, the only modules which take effect in this operation are pam_authtok_get and pam_unix_auth. Since the user name has been passed in the variable throught the handle, pam_authtok_get only prompts the user for a password. The string provided by the user trought the conversation function is stored in the PAM_USER variable . Finally, the pam_unix_auth module checks the validity of the password in by comparing it with the hash value stored in the shadow database for the user PAM_USER.

References

[1] IEEE Draft P1003.1e
http://wt.xpilot.org/publications/posix.1e/download/Posix_1003.1e-990310.pdf.gz

[2] FreeBSD Documentation Project : Chapter 15 Jails
http://www.freebsd.org/doc/en/books/handbook/jails.html

[3] Joyent Smart Computing http://www.joyent.com/

[4] Lorch M. , D. Kafura: Supporting Secure ad hoc User Collaborations in Grid Environments, *Grid Computing - GRID 2002 3rd International Workshop, LNCS 2536* (2002)

[5] Mauro J., McDougall R.: *Solaris Internals* (2nd ed.), Sun Microsystem Press (2005)

[6] OpenSolaris Community http://hub.opensolaris.org/bin/view/Main/

[7] OpenSolaris Community Group Dtrace
http://hub.opensolaris.org/bin/view/Community+Group+dtrace/

[8] OpenSolaris Community Group Security: SSH
http://hub.opensolaris.org/bin/view/Community+Group+security/SSH

[9] OpenSolaris Community Group SMF
http://hub.opensolaris.org/bin/view/Community+Group+smf/

[10] OpenSolaris Community Group ZFS
http://hub.opensolaris.org/bin/view/Community+Group+zfs/

[11] OpenSolaris Community Group Zones
http://hub.opensolaris.org/bin/view/Community+Group+zones/

[12] OpenSolaris Project: Fine Grained Access Policy
(FGAP) http://hub.opensolaris.org/bin/view/Project+fgap/

[13] Oracle Technology Network: Predictive Self-Healing
http://www.oracle.com/technetwork/systems/dtrace/self-healing/index.html

[14] Project OpenIndiana http://openindiana.org/

[15] RELIANT Security http://www.reliantsec.net/

[16] Role Based Access Control http://csrc.nist.gov/groups/SNS/rbac/

[17] Saltzer J. H., Schroeder M. D. The Protection of Information in Computer Systems, *Proceedings of the IEEE* (1975)

[18] Samar V., C. Lai: Making Login Services Independent of Authentication Technologies, *Proceedings of the SunSoft Developers Conference* (1996)

[19] Solaris 10 System Administrator Collection: Solaris Security Services
http://docs.sun.com/app/docs/doc/816-4557

[20] Solaris 10 System Administrator Collection: Solaris Security Services Chapter 17 Using PAM
http://docs.sun.com/app/docs/doc/816-4557/pam-1?l=all&a=view

[21] Solaris 10 System Administrator Collection: Solaris Security Services: Part III Roles, Rights Profiles, and Privileges
`http://docs.sun.com/app/docs/doc/816-4557/prbactm-1?l=en&a=view`

[22] Solaris 10 Software Developer Collection: Solaris Dynamic Tracing Guide
`http://docs.sun.com/app/docs/doc/817-6223`

[23] Solaris 10 System Administration Guide: Solaris Containers: Resource Management and Solaris Zones: Zones
`http://docs.sun.com/app/docs/doc/817-1592`

[24] Solaris 10 System Administrator Collection: Solaris ZFS Administration Guide
`http://docs.sun.com/app/docs/doc/819-5461`

[25] Solaris 10 System Administrator Collection: Solaris ZFS Administration Guide : Access Control List (ACL)
`http://docs.sun.com/app/docs/doc/819-5461/ftyxi?a=view`

[26] Solaris 10 What's New Collection: Process Rights Management
`http://docs.sun.com/app/docs/doc/817-0547/whatsnew_503-1?a=view`

[27] Solaris man pages section 2: System Calls
`http://docs.sun.com/app/docs/doc/819-2241/chmod-2?l=it&a=view&q=chmod`

[28] Solaris Security for Developers Guide
`http://docs.sun.com/app/docs/doc/816-4863`

[29] Sun Microsystems Security Engineers: *Solaris 10 Security Essentials*, Sun Microsystem Press (2009).

[30] The Illumos Foundation `http://www.illumos.org/`

[31] The Nexenta Project `http://www.nexenta.org/`

[32] U.S. Dept. of Commerce: FIPS Pubs 196 - *Entity Authentication Using Publi Key Cryptography* (1997)

[33] U.S. Dept. of Defence: *Trusted Computer System Evaluation Criteria* (1985)

[34] Ylonen T.: The Secure Shell (SSH) Authentication Protocol. *RFC 4252* (2006)

Appendix A - PAM library reference

Starting a PAM session

The function pam_start allocates space, performs various initialization activities, and assigns a PAM *authentication handle* to be used for subsequent calls to the library.

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

int pam_start(const char *service, const char *user,
   const struct pam_conv *pam_conv, pam_handle_t **pamh);
```

Parameter	Description
service	name of the current service
user	name of the user to be authenticated
pam_conv	address of the conversation structure
pamh	address of a variable to be assigned the PAM authentication handle

Table 3: Parameters for the pam_start function

The calling parameters of this function are illustrated in Table 3. The service name service is used by the PAM framework to determine which rules in the configuration file, pam.conf, are applicable. The service name is generally used for logging and error-reporting.
The PAM handle pamh is an opaque handle that is used by the PAM framework to store information about the current operation. Within one authentication transaction, all calls to the PAM interface should be made with the same authentication handle returned by pam_start. This is necessary because certain service modules may store module-specific data in a handle that is intended for use by other modules. For example, during the call to pam_authenticate, service modules may store data in the handle that is intended for use by pam_setcred.

The authentication service in PAM does not communicate directly with the user; instead it relies on the application to perform all such interactions. The application passes a pointer to the function, conv, along with any associated application data pointers, through a pam_conv structure to the authentication service when it initiates an authentication transaction, via a call to pam_start. The service will then use the function, conv, to prompt the user for data, output error messages, and display text information. The conversation function is provided by the application: the underlying PAM service module(s) invokes this function to output information to and retrieve input from the user. The pam_conv structure is as follows:

```
struct pam_conv {
   int   (*conv)();     /* Conversation function */
   void  *appdata_ptr;  /* Application data */
};
```

and the conversation function has the prototype

```
int conv(int num_msg, const struct pam_message **msg,
       struct pam_response **resp, void *appdata_ptr);
```

Parameter	Description
`num_msg`	number of messages associated with the call
`msg`	pointer to an array of the `pam_message` structure
`num_msg`	length of the array of the `pam_message` structure
`resp`	pointer to an array the `pam_response` structure
`num_msg`	length of the array of the `pam_response` structure
`appdata_ptr`	application data pointer passed by the application to the PAM service modules

Table 4: Conversation function parameters and their meaning.

Flag	Description
`PAM_PROMPT_ECHO_OFF`	Prompt user, disabling echoing of response
`PAM_PROMPT_ECHO_ON`	Prompt user, enabling echoing of response
`PAM_ERROR_MSG`	Print error message
`PAM_TEXT_INFO`	Print general text information

Table 5: Styles of a PAM message.

Table 4 describes the input parameters of the conversation function. Since the PAM modules pass `appdata_ptr` back through the conversation function, the applications can use this pointer to point to any application-specific data. The `pam_message` structure is used to pass prompt, error message, or any text information from the authentication service to the application or user. It is the responsibility of the PAM service modules to localize the messages. The memory used by `pam_message` has to be allocated and freed by the PAM modules. The `pam_message` structure has the following entries:

```
struct pam_message{
   int     msg_style;
   char    *msg;
};
```

The field message style, `msg_style`, can be set to one of the values reported in Table 5: The maximum size of the message and the response string is `PAM_MAX_MSG_SIZE` as defined in `/security/pam.appl.h`. The structure `pam_response` is used by the authentication service to get the user's response back from the application or user. The storage used by `pam_response` has to be allocated by the application and freed by the PAM modules. The `pam_response` structure has the following entries:

```
struct pam_response{
   char *resp;
   int  resp_retcode;  /* currently not used, */
                       /* should be set to 0 */
};
```

It is the responsibility of the conversation function to strip off NEWLINE characters for `PAM_PROMPT_ECHO_OFF` and `PAM_PROMPT_ECHO_ON` message styles, and to add NEWLINE characters (if appropriate) for `PAM_ERROR_MSG` and `PAM_TEXT_INFO` message styles.

Flag	Description
PAM_SILENT	The authentication service should not generate any messages
PAM_DISALLOW_NULL_AUTHTOK	The authentication service should return PAM_AUTH_ERR if the user has a null authentication token

Table 6: Values for the flags parameter in pam_*_authenticate.

Authentication

After initiating an authentication transaction with pam_start, applications can invoke pam_authenticate to authenticate a particular user:

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

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

In response to a such call, the PAM framework calls pam_sm_authenticate:

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

int pam_sm_authenticate(pam_handle_t *pamh, int flags,
    int argc, const char **argv);
```

from the modules listed in the pam.conf file. The user is usually required to enter a password or similar authentication token depending upon the authentication service configured within the system. The main task of pam_sm_authenticate is then to verify:

- the existence of the user in a suitable username repository;
- the matching of the supplied authentication token with a the one for that user stored in a suitable token repository.

The user in question should have been specified by a prior call to pam_start or pam_set_item, and is referenced by pam_*_authenticate through the pam handle pamh. Table 6 reports the flags that may be set in the flags field:

These functions return PAM_SUCCESS in case of successfull authentication, whilst in case of error the codes are those reported in Table 7:

In the case of authentication failures due to an incorrect username or password, it is the responsibility of the application to retry pam_authenticate and to maintain the retry count. An authentication service module may implement an internal retry count and return an error PAM_MAXTRIES if the module does not want the application to retry.
If the PAM framework cannot load the authentication module, then it will return PAM_ABORT. This indicates a serious failure, and the application should not attempt to retry the authentication.
The argc and argv argument in pam_sm_authenticate represent the number of module options passed in from the configuration file pam.conf, and their values, respectively. These options are interpreted and processed by the authentication service. If any unknown option is passed in, the module should log the error and ignore the option.
Before returning, pam_sm_authenticate should call pam_get_item and retrieve the authentication token, PAM_AUTHTOK. If it has not been set before and the value is NULL,

Error Code	Description
PAM_AUTH_ERR	Authentication failure
PAM_CRED_INSUFFICIENT	Cannot access authentication data due to insufficient credentials
PAM_AUTHINFO_UNAVAIL	Underlying authentication service cannot retrieve authentication information
PAM_USER_UNKNOWN	User not known to the underlying authentication module
PAM_MAXTRIES	An authentication service has maintained a retry count which has been reached. No further retries should be attempted

Table 7: Error codes for the `pam_*_authenticate` functions.

`pam_sm_authenticate` should set it to the authentication token entered by the user using `pam_set_item`.
If the user is unknown to the authentication service, the service module should mask this error and continue to prompt the user for a password. It should then return the error, `PAM_USER_UNKNOWN`. This strategy avoids the decoupling of username guessing from password guessing for an attacker.
An authentication module may save the authentication status (success or reason for failure) as state in the authentication handle using `pam_set_data`. This information is intended for use by `pam_setcred`.

Account management

Despite of the success of `pam_authenticate`, in some cases a user must be denied to successfully complete an authentication transaction. Indeed, the user's account could be expired, or it could be subjected to access hour restrictions. Thus, after the user has been authenticated with `pam_authenticate`, an application typically calls `pam_acct_mgmt`:

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

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

in order to determine if the current user's account is valid. In response to a call to `pam_acct_mgmt`, the PAM framework calls `pam_sm_acct_mgmt`:

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

int pam_sm_acct_mgmt(pam_handle_t *pamh, int flags,
    int argc, const char **argv);
```

from the modules listed in configuration file. The account management provider supplies the back-end functionality for this interface function. Applications should not call this function directly.
The `pam_sm_acct_mgmt` function determines whether or not the current user's account and password are valid. This includes checking for password and account expiration, and valid login times.
The user in question is specified by a prior call to `pam_start`, and is referenced by the authentication handle, `pamh`.

Flag	Description
PAM_SILENT	The service should not generate any messages
PAM_DISALLOW_NULL_AUTHTOK	The account management service should return PAM_NEW_AUTHTOK_REQD if the user has a null authentication token

Table 8: Flags for the account management functions.

Error Code	Description	Intf.
PAM_AUTH_ERR	Authentication failure	A
PAM_NEW_AUTHTOK_REQD	New authentication token required	*
PAM_ACCT_EXPIRED	User account has expired	*
PAM_USER_UNKNOWN	User not known to the underlying authentication module	*
PAM_PERM_DENIED	User denied access to account at this time	S
PAM_IGNORE	Ignore underlying account module regardless of whether the control flag is required, optional or sufficient	S

Table 9: Error codes for `pam_acct_mgmt` and `pam_sm_acct_mgmt`.

Flags that may be set in the `flags` field are reported in Table 8: The `argc` and `argv` argument in `pam_sm_acct_mgmt` represent the number of module options passed in from the configuration file and their values, respectively. These options are interpreted and processed by the account management service. If an unknown option is passed to the module, an error should be logged through `syslog` and the option ignored.
If there are no restrictions to logging in, then `pam*_acct_mgmt` returns `PAM_SUCCESS`. Otherwise, Table 9 shows the errors returned by `pam_acct_mgmt` only (A), `pam_sm_acct_mgmt` only (S), and both (*). `PAM_NEW_AUTHTOK_REQD` is normally returned if the machine security policies require that the authentication token should be changed because the password is NULL or has aged. If an account management module determines that the user authentication token has aged or expired, it should save this information as state in the authentication handle, `pamh`, using `pam_set_data`. `pam_chauthok` uses this information to determine which authentication tokens have expired.
If the `PAM_REPOSITORY` item_type is set and a service module does not recognize the type, the service module does not process any information, and returns `PAM_IGNORE`. If the `PAM_REPOSITORY` item_type is not set, a service module performs its default action.

Session management

The PAM framework supplies the two APIs `pam_open_session` and `pam_close_session`:

```
#include <security/pam\_appl.h>

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

for opening and closing a user session, respectively. An application calls `pam_open_session` after a user has been successfully authenticated with `pam_authenticate` and `pam_acct_mgmt`. This function is used to notify the session modules that a new session has been initiated.

Error Code	Description	Intf.
`PAM_SESSION_ERR`	making or removing an entry for the specified session was impossible	*
`PAM_IGNORE`	Ignore underlying session module regardless of whether the control flag is required, optional or sufficient	S

Table 10: Error returned by the session management functions.

All programs that use the PAM library should invoke `pam_open_session` when beginning a new session. Upon termination of this activity, `pam_close_session` should be invoked to inform PAM that the session has terminated. The use of these functions is typically related to the management of user session information, e.g. updating information about the last login.
In response to a call to `pam_open_session` and `pam_close_session` the PAM framework calls `pam_sm_open_session` and `pam_sm_close_session` respectively, from the (session) modules listed in the configuration file for the caller. The session management provider supplies the back-end functionality for this interface function.
In the standard UNIX session management module, `pam_unix_session`, this back-end functionality is implemented as follows:

- `pam_sm_open_session` updates the /var/adm/lastlog file with the information contained in the `PAM_USER`, `PAM_TTY`, and `PAM_RHOSTS` items. `pam_unix_account` uses this account to determine the previous time the user logged in.
- `pam_sm_close_session` is a null function.

The only flag that can be set in the `flags` field for all these functions is `PAM_SILENT`, and all of them return `PAM_SUCCESS` upon successful completion. The `argc` argument in `pam_sm_close_session` represents the number of module options passed in from the configuration file `pam.conf`. Instead, `argv` specifies the module options, which are interpreted and processed by the session management service. If an unknown option is passed in, an error should be logged through `syslog` and the option ignored.

Table 10 shows the errors returned by all these functions (*), and `pam_sm_*_session` only (S). In many instances, the `pam_open_session` and `pam_close_session` calls may be made by different processes. For example, in UNIX the login process opens a session, while the init process closes the session. In this case, UTMPX/WTMPX[3] entries may be used to link the call to `pam_close_session` with an earlier call to `pam_open_session`. This is possible because UTMPX/WTMPX entries are uniquely identified by a combination of attributes, including the user login name and device name, which are accessible through the PAM handle, `pamh`. The call to `pam_open_session` should precede UTMPX/WTMPX entry management, and the call to `pam_close_session` should follow UTMPX/WTMPX exit management.

Setting user's credentials

If the user has been successfully authenticated, the application calls `pam_setcred`

[3] The utmpx and wtmpx files are extended database files that have superseded the obsolete utmp and wtmp database files. The utmpx database contains user access and accounting information for commands such as who, write, and login. The wtmpx database contains the history of user access and accounting information for the utmpx database.

Flag	Description
PAM_ESTABLISH_CRED	Set user credentials for an authentication service
PAM_DELETE_CRED	Delete user credentials associated with an authentication service
PAM_REINITIALIZE_CRED	Reinitialize user credentials
PAM_REFRESH_CRED	Extend lifetime of user credentials
PAM_SILENT	Authentication service should not generate any messages

Table 11: Flags for pam_setcred and pam_sm_setcred.

Error Code	Description
PAM_CRED_UNAVAIL	Underlying authentication service cannot retrieve user credentials unavailable
PAM_CRED_EXPIRED	User credentials expired
PAM_CRED_ERR	Failure setting user credentials
PAM_USER_UNKNOWN	User not known to the underlying authentication module

Table 12: Values that may be returned upon error by pam*_setcred functions.

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

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

to set any user credential associated with the authentication service. It is typically called after a session has been opened with pam_open_session.
In response to a call to pam_setcred, the PAM framework calls pam_sm_setcred():

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

int pam_sm_setcred(pam_handle_t *pamh, int flags, int argc,
    const char **argv);
```

The user is specified by a prior call to pam_start or pam_set_item, and is referenced by the authentication handle, pamh. The flags reported in Table 11 may be set in the flags field. Note that the first four flags are mutually exclusive. If no flag is set, PAM_ESTABLISH_CRED is used as the default. Upon success, pam*_setcred return PAM_SUCCESS, whilst the values that may be returned upon error are summarized in Table 12.

Information management

The pam_get_item and pam_set_item functions

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

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

Parameter	Description
pamh	authentication handle returned by pam_start
item	a pointer to the object to set or to get
item_type	type of object to set or to get

Table 13: Parameters of pam_*_item functions.

Item Type	Description
PAM_AUSER	The authenticated user name.
PAM_AUTHTOK	The user authentication token.
PAM_CONV	The pam_conv structure.
PAM_OLDAUTHTOK	The old user authentication token.
PAM_RESOURCE	A semicolon-separated list of key=value pairs that represent the set of resource controls for application by pam_setcred or pam_open_session
PAM_RHOST	The remote host name
PAM_RUSER	The rlogin/rsh untrusted remote user name
PAM_SERVICE	The service name.
PAM_TTY	The tty name.
PAM_USER	The user name.
PAM_USER_PROMPT	The default prompt used by pam_get_user
PAM_REPOSITORY	The repository that contains the authentication token information.

Table 14: PAM session information types

allow applications and PAM service modules to access and to update PAM information as needed. Parameters for such functions are reported in Table 13. If successful, the pam_set_item copies the item to an internal storage area allocated by the authentication module and returns PAM_SUCCESS. An item that had been previously set will be overwritten by the new value. The object data retrieved by pam_get_item is valid until modified by a subsequent call to pam_set_item for the same item_type, or unless it is modified by any of the underlying service modules. If the item has not been previously set, pam_get_item returns a null pointer. An item retrieved by pam_get_item should not be modified or freed. The item will be released by pam_end. Upon success, pam_get_item returns PAM_SUCCESS; otherwise it returns an error code. The type of information to get or to set is specified by item_type, and are summarized in Table 14

The pam_repository structure is defined as:

```
struct pam_repository {
   char   *type;       /* Repository type, e.g., files, */
                       /* nis, ldap */
   void   *scope;      /* Optional scope information */
   size_t  scope_len;  /* length of scope information */
};
```

The item_type PAM_SERVICE can be set only by pam_start and is read-only to both applications and service modules. PAM_AUSER is not intended as a replacement for PAM_USER. It is expected to be used to supplement PAM_USER when there is an authenticated user from a source other than pam_authenticate. Such sources could be sshd host-based authentica-

tion, kerberized `rlogin`, and `su`.
If the `PAM_REPOSITORY` item_type is set and a service module does not recognize the type, the service module does not process any information, and returns `PAM_IGNORE`. If the `PAM_REPOSITORY` item_type/ is not set, a service module performs its default action.

Environment management

The `pam_getenv` function

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

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

searches the PAM handle pamh for a value associated with `name`. If a value is present, `pam_getenv` makes a copy of the value and returns a pointer to the copy back to the calling application. If no such entry exists, `pam_getenv` returns NULL. It is the responsibility of the calling application to free the memory returned by `pam_getenv`.

The `pam_getenvlist` function

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

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

returns a list of all the PAM environment variables stored in the PAM handle `pamh`. The list is returned as a null-terminated array of pointers to strings. Each string contains a single PAM environment variable of the form name=value. The list returned is a duplicate copy of all the environment variables stored in pamh. It is the responsibility of the calling application to free the memory returned by `pam_getenvlist`.

The `pam_putenv` function

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

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

sets the value of the PAM environment variable name equal to value either by altering an existing PAM variable or by creating a new one. The `name_value` argument points to a string of the form name=value. A call to `pam_putenv` does not immediately change the environment. All `name_value` pairs are stored in the PAM handle pamh. In case of success `pam_putenv` returns `PAM_SUCCESS`; whereas, the errors returned by the funcion in case of failure are described in Table 15.

Module information

The `pam_set_data` and `pam_get_data` functions

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

int pam_set_data(pam_handle_t *pamh,
    const char *module_data_name, void *data,
    void  (*cleanup) (pam_handle_t *pamh, void *data,
    int pam_end_status));
int pam_get_data(const pam_handle_t *pamh,
    const char *module_data_name, const void **data);
```

Error Type	Description
`PAM_OPEN_ERR`	dlopen() failed when dynamically loading a service module.
`PAM_SYMBOL_ERR`	Symbol not found.
`PAM_SERVICE_ERR`	Error in service module.
`PAM_SYSTEM_ERR`	System error.
`PAM_BUF_ERR`	Memory buffer error.
`PAM_CONV_ERR`	Conversation failure.
`PAM_PERM_DENIED`	Permission denied.

Table 15: Error types returned by `pam_putenv`.

allow PAM service modules to access and update module specific information as needed. These functions should not be used by applications.

Error management

The `pam_strerror` function

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

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

maps the PAM error number in `errnum` to a PAM error message string, and returns a pointer to that string. The application should not free or modify the string returned. The `pamh` argument is the PAM handle obtained by a prior call to `pam_start`. If `pam_start` returns an error, a null PAM handle should be passed.

Ending a PAM session

The `pam_end` function:

```
#include <security/pam\_appl.h>

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

is called to terminate the authentication transaction identified by `pamh` and to free any storage area allocated by the authentication module. The argument, status, is passed to the *cleanup* function stored within the pam handle, and is used to determine what module-specific state must be purged. A cleanup function is attached to the handle by the underlying PAM modules through a call to `pam_set_data` to free module-specific data.

Appendix B - Security attributes database library reference

The **Security attributes database library** `secdb` is a set of API introduced in OpenSolaris to manage `user_attr`, the extended user attributes database. The internal representation of a `user_attr` entry is a `userattr_t` structure defined in `<user_attr.h>` with the following members:

```
char  *name;       /* name of the user */
char  *qualifier;  /* reserved for future use */
char  *res1;       /* reserved for future use */
char  *res2;       /* reserved for future use */
kva_t *attr;       /* list of attributes */
```

The library `secdb` composes of the following functions: `getuserattr`, `getusernam`, `getuseruid`, `fgetuserattr`, `free_userattr`, `setuserattr` and `enduserattr`.
The `getuserattr`, `getusernam` and `getuseruid` functions return one or more entries from `user_attr`, whose source is specified through the `nsswitch.conf` file.

```
#include <user_attr.h>

userattr_t *getuserattr(void);
userattr_t *getusernam(const char* name);
userattr_t *getuseruid(uid_t uid);
```

Precisely, `getuserattr` enumerates `user_attr` entries, `getusernam` searches for a `user_attr` entry with a given user name `name`, and `getuserid` searches for a `user_attr` entry with a user ID given by `uid`.
The `fgetuserattr` function:

```
#include <user_attr.h>

userattr_t *fgetuserattr(FILE *f);
```

does not use `nsswitch.conf` but reads and parses the next line from the stream `f`, which is assumed to have the format of the `user_attr` file.
The `free_userattr` function:

```
#include <user_attr.h>

void free_userattr(userattr_t *userattr);
```

releases memory allocated by the `getusernam`, `getuserattr`, and `fgetuserattr` functions.
Finally, the `setuserattr` and `enduserattr` functions:

```
#include <user_attr.h>

void setuserattr(void);
void enduserattr(void);
```

serve to get back to the beginning of the enumeration of `user_attr` entries, and to indicate that `user_attr` processing is complete, respectively.

Appendix C - Standard C library APIs of interest

Password management

The following functions are used to to obtain shadow password entries:

```
#include <shadow.h>

struct spwd *getspnam(const char *name);
struct spwd *getspnam_r(const char *name, struct spwd *result,
    char *buffer, int buflen);
struct spwd *getspent(void);
struct spwd *getspent_r(struct spwd *result, char *buffer,
    int buflen);
void setspent(void);
void endspent(void);
struct spwd *fgetspent(FILE *fp);
struct spwd *fgetspent_r(FILE *fp, struct spwd *result,
    char *buffer, int buflen);
```

The `getspnam*` fuctions get the single shadow password record related to username `nam`, whilst `*spent*` functions are used to retrieve sequentially a set of contiguous records (eventually all the records composing `/etc/shadow`). In this case, the caller must provide a call to `setspent`, followed by one or more calls to `getspent*` or `fgetspent*`, and closing with `endspent`. The `getsp*` functions store the retrieved records in suitable buffers of the caller, whereas `fgetsp*` store them into a file.

Each record of the `shadow` file is represented by the `spwd` structure, defined in `shadow.h`

```
struct spwd{
         char           *sp_namp;     /* login name */
         char           *sp_pwdp;     /* encrypted passwd */
         long           sp_lstchg;    /* date of last change */
         long           sp_min;       /* min days to passwd change */
         long           sp_max;       /* max days to passwd change*/
         long           sp_warn;      /* warning period */
         long           sp_inact;     /* max days inactive */
         long           sp_expire;    /* account expiry date */
         unsigned long  sp_flag;      /* not used */
      };
```

Privilege management

The function to operate on privilege sets are `setppriv` and `getppriv`, along with a group of “service” functions to query privilege sets and perform set operations on them.

```
#include <priv.h>

int getppriv(priv_ptype_t which, priv_set_t *set);
int setppriv(priv_op_t op, priv_ptype_t which, priv_set_t *set);
```

The `getppriv` function returns the process privilege set specified by `which` in the set pointed to by `set`. The `setppriv` function sets or changes the process privilege set. Its `which` argument specifies the name of the privilege set, whilst the set argument specifies the set. The op argument specifies the operation and can be one of PRIV_OFF, PRIV_ON or PRIV_SET. If op is PRIV_OFF, the privileges in set are removed from the set specified by which. There are no restrictions on removing privileges, but the following apply:

- Privileges removed from PRIV_PERMITTED are silently removed from PRIV_EFFECTIVE.
- If privileges are removed from PRIV_LIMIT, they are not removed from the other sets until one of exec functions has successfully completed.

If op is PRIV_ON, the privileges in set are added to the set specified by which, with the following rules:

- Privileges in PRIV_PERMITTED can be added to PRIV_EFFECTIVE without restriction.
- Privileges in PRIV_PERMITTED can be added to PRIV_INHERITABLE without restriction.
- All operations that attempt to add privileges that are already present are permitted.
- Adding privileges to PRIV_LIMIT or PRIV_PERMITTED is not permitted.
- Adding privileges which are not in PRIV_PERMITTED to PRIV_INHERITABLE or PRIV_EFFECTIVE is not permitted.

The service functions all have the prefix priv_ in their names, followed by the name of the operation they perform on the sets privileges they act upon.
The functions priv_addset and priv_delset:

```
#include <priv.h>

int priv_addset(priv_set_t *sp, const char *priv);
int priv_delset(priv_set_t *sp, const char *priv);
```

add/remove the privilege priv to/from sp, respectively.
The function priv_allocset

```
#include <priv.h>

priv_set_t *priv_allocset(void);
```

allocates sufficient memory to contain a privilege set.
The functions priv_emptyset, priv_fillset, priv_freeset and priv_inverse:

```
#include <priv.h>

void priv_emptyset(priv_set_t *sp);
void priv_fillset(priv_set_t *sp);
void priv_freeset(priv_set_t *sp);
void priv_inverse(priv_set_t *sp);
```

all operate on the single privilege set sp. The priv_emptyset function clears all privileges from sp. The priv_fillset function asserts all privileges in sp. The function priv_freeset frees the storage allocated by priv_allocset. The function priv_inverse inverts the privilege set sp.
The two functions priv_intersect and priv_union

```
#include <priv.h>

void priv_intersect(const priv_set_t *src, priv_set_t *dst);
void priv_union(const priv_set_t *src, priv_set_t *dst);
```

perform the set operations of intersection and union with the privilege sets `src` and `dst`, putting the result in `dst`.
Lastly, the functions `priv_isemptyset`, `priv_isequalset`, `priv_isfullset`, `priv_ismember` and `priv_issubset`

```
#include <priv.h>

boolean_t priv_isemptyset(const priv_set_t *sp);
boolean_t priv_isequalset(const priv_set_t *src, const priv_set_t *dst);
boolean_t priv_isfullset(const priv_set_t *sp);
boolean_t priv_ismember(const priv_set_t *sp, const char *priv);
boolean_t priv_issubset(const priv_set_t *src, const priv_set_t *dst);
```

serve to perform logical operations such as testing if a privilege set is void or if it contains a given privilege.

Name Service Switch aware functions

Authentication-related repositories of naming information and their related NSS-aware functions are listed in Table 16

Database	Used By	Name Service(s)
auth_attr	getauthnam	files
group	getgrnam	files, NIS,
hosts, ipnodes	gethostbyname, getaddrinfo	
passwd	getpwnam, getauusernam, getauusernam	files, NIS,
shadow	getspnam	files, NIS,
prof_attr	getprofnam, getexecprof	files
publickey	getpublickey	files
user_attr	getuserattr	files

Table 16: Authentication-related repositories of naming information and their related NSS-aware functions.