STANDARDS PROJECT

Draft Standard for Information Technology— Portable Operating System Interface (POSIX)— Part 1: System Application Program Interface (API)— Amendment #: Protection, Audit and Control Interfaces [C Language]

Sponsor

Portable Applications Standards Committee
of the
IEEE Computer Society

Work Item Number: JTC1 22.42 %

Abstract: IEEE Std 1003.1e is part of the POSIX series of standards. It defines security interfaces to open systems for access control lists, audit, separation of privilege (capabilities), mandatory access control, and information label mechanisms. This standard is stated in terms of its C binding. + +

Keywords: auditing, access control lists, application portability, capability, information labels, mandatory access control, privilege, open systems, operating systems, portable application, POSIX, POSIX.1, security, user portability +

PSSG / D17
October 1997

ISBN-xxxx-xxxxx-x

Library of Congress Catalog Number 90-xxxxx

IEEE Standards Department
Copyright and Permissions
445 Hoes Lane, P.O. Box 1331
Piscataway, NJ 08855-1331, USA
October 1997 *XXXXXXX*

Foreword

NOTE: This foreword is not a normative part of the standard and is included for informative purposes only.

The purpose of this standard is to define a standard interface and environment for Computer Operating Systems that require certain security mechanisms. The standard is intended for system implementors and application software developers. It is an extension to IEEE Std 1003.1-1990.

Organization of the Standard

The standard is divided into several parts:

- Revisions to the General Section (Section 1)
- Revisions to Terminology and General Requirements (Section 2)
- Revisions to Process Primitives (Section 3)
- Revisions to Process Environment (Section 4)
- Revisions to Files and Directories (Section 5)
- Revisions to Input and Output Primitives (Section 6)
- Revisions to Language Specific Services for C Programming Language (Section 8)
- Access Control Lists (Section 23)
- Audit (Section 24)
- Capability (Section 25)
- Mandatory Access Control (Section 26)
- Information Labeling (Section 27)
- Annex B - Revisions to Rationale and Notes
- Annex F - Ballot Instructions

Conformance Measurement

Changes to the draft since the previous ballot are indicated by one of four marks in the right-hand margin. These change marks should aid the balloter in determining what has changed and therefore what is candidate text for comments and objections during this ballot. A bar ("|") indicates changes to the line between drafts 15 and 16. A plus ("+") indicates that text has been added in draft 16. A minus ("-") indicates that text present in that location in draft 15 has been deleted in draft 16. A percent ("%") indicates that a change was made at that location in draft 17.

In publishing this standard, both IEEE and the security working group simply intend to provide a yardstick against which various operating system implementations can be measured for conformance. It is not the intent of either IEEE or the security working group to measure or rate any products, to reward or sanction any vendors of products for conformance or lack of conformance to this standard, or to attempt to enforce this standard by these or any other means. The responsibility for determining the degree of conformance or lack thereof with this standard rests solely with the individual who is evaluating the product claiming to be in conformance with this standard.

Extensions and Supplements to This Standard

Activities to extend this standard to address additional requirements can be anticipated in the future. This is an outline of how these extensions will be incorporated, and also how users of this document can keep track of that status. Extensions are approved as "Supplements" to this document, following the IEEE Standards Procedures. Approved Supplements are published separately and are obtained from the IEEE with orders for this document until the full document is reprinted and such supplements are incorporated in their proper positions.

If you have any questions regarding this or other POSIX documents, you may contact the the IEEE Standards Office by calling IEEE at:

1 (800) 678-IEEE from within the US
1+ (908) 981-1393 from outside the US

to determine which supplements have been published. Published supplements are available for a modest fee.

Supplements are numbered in the same format as the main document with unique positions as either subsections or main sections. A supplement may include new subsections in various sections of the main document as well as new main sections. Supplements may include new sections in already approved supplements. However, the overall numbering shall be unique so that two supplements only use the same numbers when one replaces the other. Supplements may contain either required or optional facilities. Supplements may add additional conformance requirements (see POSIX.1, Implementation Conformance, 1.3) defining new classes of conforming systems or applications.

It is desirable, but perhaps unattainable, that supplements do not change the functionality of the already defined facilities. Supplements are not used to provide a general update of the standard. A general update of the standard is done through the review procedure as specified by the IEEE.

If you have interest in participating in any of the PASC working groups please send your name, address, and phone number to the Secretary, IEEE Standards Board, Institute of Electrical and Electronics Engineers, Inc., P.O. Box 1331, 445 Hoes Lane, Piscataway, NJ 08855-1331, and ask to have your request forwarded to the chairperson of the appropriate TCOS working group. If you have interest in participating in this work at the international level, contact your ISO/IEC national body.

Please report typographical errors and editorial changes for this draft standard directly to:

Casey Schaufler
Silicon Graphics
2011 North Shoreline Blvd.
P.O. Box 7311
Mountain View, CA 94039-7311
(415) 933-1634 (voice)
(415) 962-8404 (fax)
casey@sgi.com
Schaufler@DOCKMASTER.NCSC.MIL

IEEE Std 1003.1e was prepared by the security Working Group, sponsored by the Portable Applications Standards Committee of the IEEE Computer Society.

Portable Applications Standards Committee (PASC)

Chair:	Lowell Johnson
Treasurer:	Barry Needham
Secretary:	Charles Severence

Security Working Group Officials

Chair:	Lynne Ambuel
Technical Editor:	Casey Schaufler

The following people participated in the Security Working Group to develop the standard.

Lynne Ambuel	Jeanne Baccash	Lee Badger
Martin Bailey	John-Olaf Bauner	D. Elliott Bell
Lowell Bogard	Kevin Brady	Joe Brame
Matthew Brisse	Joseph Bulger	Lisa Carnahan
Mark Carson	Charisse Castagnoli	Paul Close
Roland Clouse	Peter E. Cordsen	Janet Cugini
Anthony D'Alessandro	Daniel D. Daugherty	Manilal Daya
Ana Maria De Alvare'	Terence Dowling	Jack Dwyer
Maryland R. Edwards	Ron Elliott	Lloyd English
Jeremy Epstein	Frank Fadden	Kevin Fall
David Ferbrache	Carl Freeman	Mark Funkenhauser
Morrie Gasser	Gerald B. Green	John Griffith
Henry Hall	Craig Heath	Tom Houghton
Rand Hoven	Chris Hughes	Howard Israel
Paul A. Karger	Joseph Keenan	Jerry Keselman
Yvon Klein	Andy Kochis	Steve Kramer
Steven LaFountain	Danielle Lahmani	Jason Levitt
Warren E. Loper	Jeff Mainville	Doug Mansur
Richard E. Mcnaney	Chris Milsom	Mark Modig
Jim Moseman	Kevin V. Murphy	Greg Nuss
Rose Odonnell	Gary Oing	Larry Parker
Gordon Parry	Jeff Picciotto	Michael Ressler
David Rogers	Peter L. Rosencrantz	Shawn Rovansek
Craig Rubin	Roman Saucedo	Stuart Schaeffer
Mark Schaffer	Casey Schaufler	Michael Schmitz
Larry Scott	Eric Shaffer	Olin Sibert
Rick Siebenaler	Alan Silverstein	Jon Spencer
Dennis Steinauer	Chris Steinbroner	Michael Steuerwalt
Doug Steves	Steve Sutton	W. Lee Terrell
Charlie Testa	Jeff Tofano	Brian Weis
Catherine West	Ken Witte	

Information technology—Portable operating system interface for computer environments

Section 1: Revisions to the General Section

⇒ **1.1 Scope** *This scope is to be revised and integrated appropriately into the scope when POSIX.1e is approved:*

This standard, P1003.1e/D17: October 1997 (POSIX.1e), defines five independent, optional sets of interfaces that will be used to implement protection, audit, and control mechanisms. Implementation of any or all of these interfaces does not ensure the security of the conforming system nor of conforming applications. In addition, implementation of these interfaces does not imply that a conforming system can achieve any class or level of any security evaluation criteria. These interfaces will become integrated into the ISO/IEC 9945-1: 1990 (System Application Program Interface) standard (POSIX.1) as they are approved and published. The sets of interfaces for implementation are: %

(1) Access Control Lists (ACL)

(2) Security Auditing

(3) Capability

(4) Mandatory Access Controls (MAC)

(5) Information Labeling (IL)

Each option defines new functions, as well as security-related constraints for the functions and utilities defined by other POSIX standards.

⇒ 1.2 Normative References (POSIX.1: line 39)

The following standards contain provisions that, through references in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of this standard are encouraged to investigate the possibility of applying the most recent editions of the standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards.

(1) ISO/IEC 9945-1: 1990, Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]

(2) IEEE Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shell and Utilities.

(3) P1003.2c/D17: October 1997, Draft Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shell and Utilities—Amendment #: Protection and Control Utilities

⇒ 1.3.1.3 Conforming Implementation Options (POSIX.1: line 98) *Insert the following options in alphabetic order:*

{_POSIX_ACL} Access control list option (in 2.9.3)

{_POSIX_AUD} Auditing option (in 2.9.3)

{_POSIX_CAP} Capability option (in 2.9.3)

{_POSIX_MAC} Mandatory access control option (in 2.9.3)

{_POSIX_INF} Information label option (in 2.9.3)

Section 2: Revisions to Terminology and General Requirements

⇒ 2.2.1 Terminology

⇒ 2.2.2 General Terms (POSIX.1: lines 89-397) *Add the following definitions in alphabetical order:*

2.2.2.1 access: A specific type of interaction between a process and an object that results in the flow of information from one to the other. Possible information flows include the transfer of attributes pertaining to that object, the transfer of data pertaining to that object, or the fact of existence of that object.

2.2.2.2 access acl: An access control list (ACL) which is used in making discretionary access control decisions for an object.

2.2.2.3 access control: The prevention of unauthorized access to objects by processes and, conversely, the permitting of authorized access to objects by processes.

2.2.2.4 access control list (ACL): A discretionary access control entity associated with an object, consisting of a list of entries where each entry is an identifier (e.g. user or group of users) coupled with a set of access permissions.

2.2.2.5 access control policy: A set of rules, part of a security policy, by which a user's authorization to access an object is determined.

2.2.2.6 audit: The procedure of capturing, storing, analyzing, maintaining and managing data concerning security-relevant activities.

2.2.2.7 auditable event: An activity which may cause an audit record to be reported in an audit log.

2.2.2.8 audit event type: A field within an audit record that identifies the activity reported by the record and defines the required content of the record.

2.2.2.9 audit ID: An identifier for the user accountable for an audit event.

2.2.2.10 audit record: The discrete unit of data reportable in an audit log on the occurrence of an audit event.

2.2.2.11 audit log: The destination of audit records that are generated and the source of records read by an audit post-processing application.

2.2.2.12 availability: The property of an object or subject being accessible and usable upon demand by an authorized user.

2.2.2.13 capability: An attribute of a process that is included in the determination of whether or not a process has the appropriate privilege to perform a specific POSIX.1 action where appropriate privilege is required. % –

2.2.2.14 capability flag: A per-capability attribute of a file or process that is used during *exec*() processing in computing the capability of the process executing that file. +

2.2.2.15 capability state: A grouping of all of the flags defined by an implementation for all capabilities defined for the implementation.

2.2.2.16 channel: An information transfer path within a system or a mechanism by which the path is effected.

2.2.2.17 confidentiality: The property that the existence of an object and/or its contents and/or attributes are not made available nor disclosed to unauthorized processes.

2.2.2.18 covert channel: A communications channel that allows a process to transfer information in a manner that violates the system's security policy. Covert channels are typically realized by the exploitation of mechanisms not intended to be used for communication.

2.2.2.19 data descriptor: An internal representation which uniquely identifies a data object.

2.2.2.20 default acl: An ACL which is used in determining the initial discretionary access control information for objects.

2.2.2.21 denial of service: The unauthorized prevention of authorized access to resources or the delaying of time-critical operations.

2.2.2.22 discretionary access control (DAC): A means of restricting access to objects based on the identity of the user, process, and/or groups to which the objects belong. The controls are discretionary in the sense that a subject with some access permission is capable of passing that permission (perhaps indirectly) on to other subjects.

2.2.2.23 dominate: An implementation-defined relation between the values of MAC labels or information labels.

2.2.2.24 downgrade: An operation which changes a MAC label or information label to a value that does not dominate the current label.

2.2.2.25 equivalent: An implementation-defined relation between the values of MAC labels or of information labels. Two labels are equivalent if each of the labels dominates the other.

2.2.2.26 extended ACL: An ACL that contains entries in addition to a *minimum ACL*.

2.2.2.27 exportable data: Opaque data objects for which the data is self-contained and persistent. As a result, they can be copied or stored freely.

2.2.2.28 file group class: The property of a file indicating access permissions for a process related to the process's group identification.

A process is in the file group class of a file if the process is not in the file owner class and if the effective group ID or one of the supplementary group IDs of the process matches the group ID associated with the file.

If {_POSIX_ACL} is defined, then a process is also in the file group class if the process is not in the file owner class and

(1) the effective user ID of the process matches the qualifier of one of the ACL_USER entries in the ACL associated with the file, or

(2) the effective group ID or one of the supplementary group IDs of the process matches the qualifier of one of the ACL_GROUP entries in the ACL associated with the file.

Other members of the class may be implementation defined.

2.2.2.29 formal security policy model: A precise statement of a system security policy.

2.2.2.30 information label: The representation of a security attribute of a subject or object that applies to the data contained in that subject or object and is not used for mandatory access control.

2.2.2.31 information label floating: The operation whereby one information label is combined with another information label. The specific algorithm used to define the result of a combination of two labels is implementation defined.

2.2.2.32 information label policy: The policy that determines how information labels associated with objects and subjects are automatically adjusted as data flows through the system.

2.2.2.33 MAC label: The representation of a security attribute of a subject or object which represents the sensitivity of the subject or object and is used for mandatory access control decisions. –

2.2.2.34 mandatory access control (MAC): A means of restricting and permitting access to objects based on an implementation-defined security policy using MAC labels and the use of the implementation-defined dominate operator. The restrictions are mandatory in the sense that they are always imposed by the system.

2.2.2.35 minimum ACL: An ACL that contains only the required ACL entries. –

2.2.2.36 object: A passive entity that contains or receives data. Access to an object potentially implies access to the data that it contains.

2.2.2.37 opaque data object: A data repository whose structure and representation is unspecified. Access to data contained in these objects is possible through– the use of defined programming interfaces.

2.2.2.38 persistent: A state in which data retains its original meaning as long as the system configuration remains unchanged, even across system reboots. However, any change to the system configuration (such as adding or deleting user IDs and modifying the set of valid labels) may render such data invalid. –

2.2.2.39 principle of least privilege: A security design principle that states that a process or program be granted only those privileges (e.g., capabilities) necessary to accomplish its legitimate function, and only for the time that such privileges are actually required.

2.2.2.40 query: Any operation which obtains either data or attributes from a subject or object.

2.2.2.41 read: A fundamental operation that obtains data from an object or subject.

2.2.2.42 required ACL entries: The three ACL entries that must exist in every valid ACL. These entries are exactly one entry each for the owning user, the owning group, and other users not specifically enumerated in the ACL.

2.2.2.43 security: The set of measures defined within a system as necessary to adequately protect the information to be processed by the system.

2.2.2.44 security administrator: An authority responsible for implementing the security policy for a security domain.

2.2.2.45 security attribute: An attribute associated with subjects or objects which is used to determine access rights to an object by a subject.

2.2.2.46 security domain: A set of elements, a security policy, a security authority and a set of security-relevant activities in which the set of elements are subject to the security policy, administered by the security authority, for the specified activities.

2.2.2.47 security policy: The set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information.

2.2.2.48 security policy model: A precise presentation of the security policy enforced by a system.

2.2.2.49 strictly dominate: A relation between the values of two MAC labels or information labels whereby one label dominates but is not equivalent to the other label.

2.2.2.50 subject: An active entity that causes information to flow between objects or changes the system state; e.g., a process acting on behalf of a user.

2.2.2.51 tranquillity: Property whereby the MAC label of an object can be changed only while it is not being accessed.

2.2.2.52 upgrade: An operation that changes the value of a MAC label or information label to a value that strictly dominates its previous value.

2.2.2.53 user: Any person who interacts with a computer system. Operations are performed on behalf of the user by one or more processes.

2.2.2.54 write: A fundamental operation that results only in the flow of information from a subject to an object.

⇒ 2.2.3 Abbreviations (POSIX.1: line 404)

For the purpose of this standard, the following abbreviations apply:

(1) **POSIX.1**: ISO/IEC 9845-1: 1990: Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]

(2) **POSIX.2**: ISO/IEC 9845-1: 1992: Information IEEE Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shell and Utilities

(3) **POSIX.1e**: IEEE Std 1003.1e/D17: October 1997, Draft Standard for Information Technology—Portable Operating System Interface (POSIX)—Protection, Audit and Control Interfaces

(4) **POSIX.2c**: IEEE Std 1003.2c/D17: October 1997, Draft Standard for Information Technology—Portable Operating System Interface (POSIX)—Protection and Control Utilities

⇒ 2.3 General Concepts (POSIX.1: lines 406-498)

⇒ 2.3.2 file access permissions (POSIX.1: line 413) *Change this sub-clause to "***2.3.2 file access controls***", and incorporate the concept of "file access permissions" under it along with the following new concepts:*

One standard file access control mechanism based on file permission bits and two optional file access control mechanisms based on access control lists and MAC labels are defined by this document.

⇒ 2.3.2.1 file access permissions (POSIX.1: line 414) *After the above change to section 2.3.2, create a new subsection called 2.3.2.1 and replace the previous text in POSIX.1 subsection 2.3.2 with the following.*

This standard defines discretionary file access control on the basis of file permission bits as described below. The additional provisions of section 2.3.2.2 apply only if {_POSIX_ACL} is defined.

The file permission bits of a file contain read, write, and execute/search permissions for the file owner class, file group class, and file other class.

These bits are set at file creation by *open*(), *creat*(), *mkdir*(), and *mkfifo*(). They are changed by *chmod*() and, if {_POSIX_ACL} is defined, *acl_set_file*() and *acl_set_fd*(). These bits are read by *stat*(), and *fstat*().

Implementations may provide *additional* or *alternate* file access control mechanisms, or both. An additional access control mechanism shall only further restrict the access permissions defined by the file access control mechanisms described in this section. An alternate access control mechanism shall:

(1) Specify file permission bits for the file owner class, file group class, and file other class corresponding to the access permissions, to be returned by *stat*() or *fstat*().

(2) Be enabled only by explicit user action on a per file basis by the file owner or a user with the appropriate privilege.

(3) Be disabled for a file after the file permission bits are changed for that file with *chmod*(). The disabling of the alternate mechanism need not disable any additional mechanisms defined by an implementation.

Whenever a process requests file access permission for read, write, or execute/search, if no additional mechanism denies access, access is determined as follows:

If the process possesses appropriate privilege:

— If read, write, or directory search permission is requested, access is granted.

— If execute permission is requested, access is granted if execute permission is granted to at least one user by the file access permission bits or by an alternate access control mechanism; otherwise, access is denied.

Otherwise:

— Access is granted if an alternate access control mechanism is not enabled and the requested access permission bit is set for the class (file owner class, file group class, or file other class) to which the process belongs, or if an alternate access control mechanism is enabled and it allows the requested access; otherwise, access is denied.

If {_POSIX_CAP} is defined, then appropriate privilege includes the following capabilities: CAP_DAC_WRITE for write access, CAP_DAC_EXECUTE for execute access, and CAP_DAC_READ_SEARCH for read and search access. See Table 25-5.

⇒ **2.3.2.2 access control lists:** *Add this as a new concept.*

The {_POSIX_ACL} option provides an additional access control mechanism by providing file access control based upon an access control list mechanism. The provisions of this section apply only if {_POSIX_ACL} is defined. The interaction between file permission bits and the ACL mechanism is defined such that a correspondence is maintained between them. The ACL mechanism therefore enhances access control based upon the file permission bits.

An ACL entry shall support at a minimum read, write, and execute/search permissions.

An ACL is set at file creation time by *open*(), *creat*(), *mkdir*(), and *mkfifo*(). An additional *default ACL* can be associated with a directory; the default ACL is used in setting the ACL of any object created in that directory. An ACL is changed by *acl_set_fd*() and *acl_set_file*(). A call to *acl_set_fd*() or *acl_set_file*() may also result in a change to the file's permission bits. A call to *chmod*() to change a file's permission bits will also result in a change to the corresponding entries in the ACL. The file's ACL is read by either *acl_get_fd*() or *acl_get_file*(). A process is granted discretionary access to a file only if all individual requested modes of access are granted by an ACL entry or the process possesses appropriate privileges.

Whenever a process requests file access permission for read, write, or execute/search, if no additional mechanism denies access, access is determined as follows:

If the process possesses appropriate privilege:

— If read, write or directory search permission is requested, access is granted.

— If execute permission is requested, access is granted if execute permission is specified in at least one ACL entry; otherwise, access is denied.

Otherwise:

— access is granted if an alternate access control mechanism is not enabled and the requested access permissions are granted on the basis of the evaluation of the ACL (see 23.1.5), or if an alternate access control mechanism is enabled and it allows the requested access; otherwise, access is denied.

If {_POSIX_CAP} is defined, then appropriate privileges includes the following capabilities: CAP_DAC_WRITE for write access, CAP_DAC_EXECUTE for execute access, and CAP_DAC_READ_SEARCH for read and search access. See Table 25-5.

⇒ **2.3.2.3 mandatory access control:** *Add this as a new concept.*

The {_POSIX_MAC} option provides interfaces to an additional access control mechanism based on the assignment of MAC labels to subjects and objects. The provisions of this section only apply if {_POSIX_MAC} is defined.

The MAC mechanism permits or restricts access to an object by a process based on a comparison of the MAC label of the process to the MAC label of the object. A process can read an object only if the process's MAC label dominates the object's MAC label, and write an object only if the process's MAC label is dominated by the object's MAC label. However, an implementation may impose further restrictions, permitting write access to objects only by processes with a MAC label equivalent to that of the object. The standard does not define the dominance and equivalence relationships and, thus, does not define a particular MAC policy.

MAC read access to an object by a process requires that the process's MAC label dominate the object's MAC label or that the process possess appropriate privilege. If {_POSIX_CAP} is defined, the appropriate privilege is CAP_MAC_READ. See Table 25-6.

MAC write access to an object by a process requires that the process's MAC label be dominated by the object's MAC label or that the process possess appropriate privilege. If {_POSIX_CAP} is defined, the appropriate privilege is CAP_MAC_WRITE. See Table 25-6.

Execute/search file access requires MAC read access to the file.

The MAC label of an object (including a process object) is set at creation time to dominate the MAC label of the creating process. Although this allows creation of upgraded objects, this standard provides only interfaces which will create objects with MAC labels equivalent to that of the creating process. However, interfaces are provided to allow an appropriately privileged process to upgrade existing objects.

⇒ **2.3.2.4 evaluation of file access:** *Add this as a new concept.*

Whenever a process requests file access, if an alternate access control mechanism is not enabled and all applicable POSIX.1 access control mechanisms grant the requested access and all additional access control mechanisms grant the requested access or if an alternate access control mechanism is enabled and grants the requested access, then access is granted; otherwise, access is denied.

⇒ **2.3.5 file times update: (POSIX.1: line 475)** *Add the following paragraph to the concept definition of file times update:*

When {_POSIX_MAC} is defined and the object and process MAC labels are not equivalent, then the result of marking the file time attribute *st_atime* for update shall be implementation-defined.

⇒ **2.4 Error Codes** *Add the following items to the error code definitions in alphabetic order.*

[ENOTSUP] Operation is not supported.

⇒ **2.7.2 POSIX.1 Symbols (POSIX.1: Table 2-2)** *Insert the following entries in alphabetical order in Table 2-2:*

Header	Key	Reserved Prefix	Reserved Suffix
<sys/acl.h>	1	acl_	
	2	ACL_	
<sys/audit.h>	1	aud_	
	2	AUD_	
<sys/capability.h>	1	cap_	
	2	CAP_	
<sys/inf.h>	1	inf_	
	2	INF_	
<sys/mac.h>	1	mac_	
	2	MAC_	

⇒ **2.7.3 Headers and Function Prototype (POSIX.1: line 910-927)** *Add the following entries in alphabetical order:*

<sys/acl.h> *acl_add_perm*(), *acl_calc_mask*(), *acl_clear_perms*(), *acl_copy_entry*(), *acl_copy_ext*(), *acl_copy_int*(), *acl_create_entry*(), *acl_delete_def_file*(), *acl_delete_entry*(), *acl_delete_perm*(), *acl_dup*(), *acl_free*(), *acl_from_text*(), *acl_get_entry*(), *acl_get_fd*(), *acl_get_file*(), *acl_get_permset*(), *acl_get_qualifier*(), *acl_get_tag_type*(), *acl_init*(), *acl_set_fd*(), *acl_set_file*(), *acl_set_permset*(), *acl_set_qualifier*(), *acl_set_tag_type*(), *acl_size*(), *acl_to_text*(), *acl_valid*().

<sys/audit.h> *aud_copy_ext*(), *aud_copy_int*(), *aud_delete_event*(), *aud_delete_event_info*(), *aud_delete_hdr*(), *aud_delete_hdr_info*(), *aud_delete_obj*(), *aud_delete_obj_info*(), *aud_delete_subj*(), *aud_delete_subj_info*(), *aud_dup_record*(), *aud_evid_from_text*(), *aud_evid_to_text*(), *aud_free*(), *aud_get_all_evid*(), *aud_get_event*(), *aud_get_event_info*(), *aud_get_hdr*(), *aud_get_hdr_info*(), *aud_get_id*(), *aud_get_obj*(), *aud_get_obj_info*(), *aud_get_subj*(), *aud_get_subj_info*(), *aud_id_from_text*(), *aud_id_to_text*(), *aud_init_record*(), *aud_put_event*(), *aud_put_event_info*(), *aud_put_hdr*(), *aud_put_hdr_info*(), *aud_put_obj*(), *aud_put_obj_info*(), *aud_put_subj*(), *aud_put_subj_info*(), *aud_read*(), *aud_rec_to_text*(), *aud_size*(), *aud_switch*(), *aud_valid*(), *aud_write*().

<sys/capability.h> *cap_clear*(), *cap_copy_ext*(), *cap_copy_int*(), *cap_dup*(), *cap_free*(), *cap_from_text*(), *cap_get_fd*(), *cap_get_file*(), *cap_get_flag*(), *cap_get_proc*(), *cap_init*(), *cap_set_fd*(), *cap_set_file*(), *cap_set_flag*(), *cap_set_proc*(), *cap_size*(), *cap_to_text*().

<sys/inf.h> *inf_default*(), *inf_dominate*(), *inf_equal*(), *inf_float*(), *inf_free*(), *inf_from_text*(), *inf_get_fd*(), *inf_get_file*(), *inf_get_proc*(), *inf_set_fd*(), *inf_set_file*(), *inf_set_proc*(), *inf_size*(), *inf_to_text*(), *inf_valid*().

<sys/mac.h> *mac_dominate*(), *mac_equal*(), *mac_free*(), *mac_from_text*(), *mac_get_fd*(), *mac_get_file*(), *mac_get_proc*(), *mac_glb*(), *mac_lub*(), *mac_set_fd*(), *mac_set_file*(), *mac_set_proc*(), *mac_size*(), *mac_to_text*(), *mac_valid*().

⇒ **2.8.2 Minimum Values (POSIX.1: line 983)** *Insert the following entry in Table 2-3 in alphabetical order:*

Name	Description	Value
{_POSIX_ACL_ENTRIES_MAX}	The maximum number of entries in an ACL for objects that support ACLs. Unspecified if {_POSIX_ACL} is not defined.	16

⇒ **2.8.4 Run-Time Invariant Values (Possibly Indeterminate) (POSIX.1: line 1023)** *Insert the following entry in Table 2-5 in alphabetical order:*

Name	*Description*
{_POSIX_ACL_MAX}	The maximum number of entries in an ACL for objects that support ACLs. Unspecified if {_POSIX_ACL} is not defined.–

⇒ **2.8.5 Pathname Variable Values (POSIX.1: line 1044)** *Insert the following entries in alphabetical order in Table 2-6:*

Table 2-6 - Pathname Variable Values

Name	Description	Minimum Value
{_POSIX_ACL_EXTENDED}	A value greater than zero if POSIX extended Access Control Lists are supported on the object; otherwise zero.	Zero
{_POSIX_ACL_PATH_MAX}	The maximum number of ACL entries permitted in the ACLs associated with the object. If {_POSIX_ACL_EXTENDED} is greater than zero, then this value shall be 16 or greater. If {_POSIX_ACL_EXTENDED} is zero, then this value shall be 3.	3 or 16
{_POSIX_CAP_PRESENT}	A value greater than zero if POSIX File Capability extensions are supported on the object; otherwise zero.	Zero
{_POSIX_INF_PRESENT}	A value greater than zero if POSIX Information Label functions that set the Information Label are supported on the object; otherwise zero.	Zero
{_POSIX_MAC_PRESENT}	A value greater than zero if POSIX Mandatory Access Control functions that set the MAC label are supported on the object; otherwise zero.	Zero

⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications (POSIX.1: line 1122)** *Insert the following entries in Table 2-10 in alphabetical order:*

Table 2-10 - Compile-Time Symbolic Constants

Name	Description
{_POSIX_ACL}	If this symbol is defined, it indicates that the implementation supports Access Control List extensions.
{_POSIX_AUD}	If this symbol is defined, it indicates that the implementation supports Auditing extensions.
{_POSIX_CAP}	If this symbol is defined, it indicates that the implementation supports Capability extensions.
{_POSIX_INF}	If this symbol is defined, it indicates that the implementation supports Information Label extensions.
{_POSIX_MAC}	If this symbol is defined, it indicates that the implementation supports Mandatory Access Control extensions.

Section 3: Revisions to Process Primitives

⇒ **3.1.1.2 Process Creation — Description (POSIX.1: line 36)** *Insert the following lines after line 32 in Section 3.1.1.2:*

(1) If {_POSIX_ACL} is defined, the child process shall have its own copy of any ACL pointers and ACL entry descriptors in the parent, and any ACL working storage to which they refer.

(2) If {_POSIX_AUD} is defined, the child process shall have its own copy of any audit record descriptors in the parent, and any audit working storage to which they refer. The audit state of the child, as set by *aud_switch*(), shall initially be the same as that of the parent; subsequent calls to *aud_switch*() in either process shall not affect the audit state of the other process.

⇒ **3.1.2.2 Execute a File — Description (POSIX.1: line 153)** *Insert the following at the end of the list of attributes inherited by the new process image on exec() following line 153 in Section 3.1.2.2:*

(15) If {_POSIX_MAC} is defined, the process MAC label (see 26.1.1) %

⇒ **3.1.2.2 Execute a File — Description (POSIX.1: line 168)** *Insert the following paragraphs after line 168 in section 3.1.2.2:*

If {_POSIX_ACL} is defined, the new process image created shall not inherit any ACL pointers or ACL entry descriptions or any ACL working storage from the previous process image.

If {_POSIX_AUD} is defined, the new process image shall not inherit any audit record descriptors or audit record working storage from the previous process image. Any incomplete audit records are discarded. The audit state of the process, as set by *aud_switch*() shall be the same as in the previous process image.

If {_POSIX_CAP} is defined, the new process image shall not inherit any capability data objects nor any working storage associated with capabilities in the previous process image.

If {_POSIX_CAP} is defined, the *exec*() functions shall modify the state of each of the capabilities of the process as follows, where I_1, E_1, and P_1 are respectively the inheritable, effective, and permitted flags of the new process image; I_0 is the inheritable flags of the current process image; and I_f, E_f and P_f are respectively the inheritable, effective, and permitted flags associated with the file being executed:

$$I_1 = I_0$$
$$P_1 = (P_f \&\& X) \| (I_f \&\& I_0)$$
$$E_1 = E_f \&\& P_1$$

where X denotes possible additional implementation-defined restrictions.

If {_POSIX_INF} is defined and {_POSIX_INF_PRESENT} is in effect for the file being executed, the information label of the process shall automatically be set to the same value as returned by *inf_float(file information label, process information label).* If {_POSIX_INF} is defined but {_POSIX_INF_PRESENT} is not in effect for the file being executed, the information label of the process shall be set in an implementation defined manner.

⇒ **3.3.1.3 Signal Actions — Description (POSIX.1: line 556)** *Insert the following section before line 556:*

If {_POSIX_INF} is defined, the following functions shall also be reentrant with respect to signals:

inf_dominate() *inf_equal*() *inf_set_fd*() *inf_set_file*() *inf_set_proc*() *inf_size*()

If {_POSIX_MAC} is defined, the following functions shall also be reentrant with respect to signals:

mac_dominate() *mac_equal*() *mac_set_fd*() *mac_set_file*() *mac_set_proc*() *mac_size*()

⇒ **3.3.2.2 Send a Signal to a Process — Description (POSIX.1: line 594)** *Insert the following sentence after the word "privileges":*

If {_POSIX_CAP} is defined, then appropriate privilege shall include CAP_KILL.

⇒ 3.3.2.2 Send a Signal to a Process — Description (POSIX.1: line 616)

Insert the following after line 616:

If {_POSIX_MAC} is defined, then in addition to the restrictions defined above, the following restrictions apply depending on the MAC labels of the sending and receiving process. There are four cases to be considered for each potential receiving process specified by *pid*:

(1) If the MAC label of the sending process is equivalent to the MAC label of the receiving process, then no additional restrictions are imposed.

(2) If the MAC label of the sending process dominates the MAC label of the receiver (i.e., the signal is being written down), then the sending process must have appropriate privilege. If {_POSIX_CAP} is defined, then appropriate privilege shall include CAP_MAC_WRITE.

(3) If the MAC label of the receiving process dominates the MAC label of the sending process (i.e., the signal is being written up), then it is implementation defined whether the sending process requires appropriate privilege. If {_POSIX_CAP} is defined and appropriate privilege is required, then appropriate privilege shall include CAP_MAC_READ.

(4) If neither of the MAC labels of the sender and receiver dominates the other, then the sending process must have appropriate privilege. If {_POSIX_CAP} is defined, appropriate privilege shall include CAP_MAC_WRITE.

⇒ 3.3.2.4 Send a Signal to a Process — Errors (POSIX.1: line 625-628)

Replace lines 625-628 with the following:

[EPERM] The process does not have permission to send the signal to any receiving process.

If {_POSIX_MAC} is defined, the process has appropriate MAC access to a receiving process, but other access checks have denied the request.

[ESRCH] No process or process group can be found corresponding to that specified by *pid*.

If {_POSIX_MAC} is defined, a receiving process or processes may actually exist, but the sending process does not have appropriate MAC access to any of the receiving processes.

Section 4: Revisions to Process Environment

⇒ **4.2.2.2 Set User and Group IDs — Description (POSIX.1: line 48)** *Insert the following after line 48 of Section 4.2.2.2:*

If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_SETUID capability.

⇒ **4.2.2.2 Set User and Group IDs — Description (POSIX.1: line 52)** *Insert the following after line 52 of Section 4.2.2.2:*

If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_SETUID capability.

⇒ **4.2.2.2 Set User and Group IDs — Description (POSIX.1: line 54)** *Insert the following after line 54 of Section 4.2.2.2:*

If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_SETGID capability.

⇒ **4.2.2.2 Set User and Group IDs — Description (POSIX.1: line 58)** *Insert the following after line 58 of Section 4.2.2.2:*

If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_SETGID capability.

⇒ **4.2.2.2 Set User and Group IDs — Description (POSIX.1: line 61)** *Insert the following after line 61 of Section 4.2.2.2:*

If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_SETUID capability.

⇒ **4.2.2.2 Set User and Group IDs — Description (POSIX.1: line 64)** *Insert the following after line 64 of Section 4.2.2.2:*

If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_SETUID capability.

⇒ **4.2.2.2 Set User and Group IDs — Description (POSIX.1: line 66)** *Insert the following after line 66 of Section 4.2.2.2:*

If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_SETGID capability.

⇒ **4.2.2.2 Set User and Group IDs — Description (POSIX.1: line 69)** *Insert the following after line 69 of Section 4.2.2.2:*

If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_SETGID capability.

⇒ **4.8.1.2 Get Configurable System Variables — Description (POSIX.1: line 407)** *Insert the following entries in Table 4-2:*

Variable	*name* Value
{_POSIX_ACL_MAX}	{_SC_ACL_MAX}
{_POSIX_ACL}	{_SC_ACCESS_CONTROL_LIST}
{_POSIX_AUD}	{_SC_AUDIT}
{_POSIX_CAP}	{_SC_CAPABILITIES}
{_POSIX_INF}	{_SC_INFORMATION_LABEL}
{_POSIX_MAC}	{_SC_MANDATORY_ACCESS_CONTROL}

Section 5: Revisions to Files and Directories

⇒ **5.3.1.2 Open a File — Description (POSIX.1: lines 192-194)** *Replace the sentence beginning "The file permission bits ...", with the following:*

If {_POSIX_ACL} is defined and {_POSIX_ACL_EXTENDED} is in effect for the directory in which the file is being created (the "containing directory") and said directory has a default ACL, the following actions shall be performed:

(1) The default ACL of the containing directory is copied to the access ACL of the new file.

(2) Both the ACL_USER_OBJ ACL entry permission bits and the file owner class permission bits of the access ACL are set to the intersection of the default ACL's ACL_USER_OBJ permission bits and the file owner class permission bits in *mode*. The action taken for any implementation-defined permissions that may be in the ACL_USER_OBJ entry shall be implementation-defined.

(3) If the default ACL does not contain an ACL_MASK entry, both the ACL_GROUP_OBJ ACL entry permission bits and the file group class permission bits of the access ACL are set to the intersection of the default ACL's ACL_GROUP_OBJ permission bits and the file group class permission bits in *mode*. The action taken for any implementation-defined permissions that may be in the ACL_GROUP_OBJ entry shall be implementation-defined.

(4) If the default ACL contains an ACL_MASK entry, both the ACL_MASK ACL entry permission bits and the file group class permission bits of the access ACL are set to the intersection of the default ACL's ACL_MASK permission bits and the file group class permission bits in *mode*. The action taken for any implementation-defined permissions that may be in the ACL_MASK entry shall be implementation-defined.

(5) Both the ACL_OTHER ACL entry permission bits and the file other class permission bits of the access ACL are set to the intersection of the default ACL's ACL_OTHER permission bits and the file other class permission bits in *mode*. The action taken for any implementation-defined permissions that may be in the ACL_OTHER entry shall be implementation-defined.

Implementation-defined default ACL entries may affect the above algorithm but shall not alter the access permitted to any subject that does not match those implementation-defined ACL entries. Implementations may provide an additional default ACL mechanism that is applied if a default ACL as defined by this standard is not present. Such an implementation-defined default ACL interface may apply different access and/or default ACLs to created objects based upon implementation-defined criteria.

If {_POSIX_ACL} is not defined, or {_POSIX_ACL_EXTENDED} is not in effect for the directory in which the file is being created (the "containing directory"), or said directory does not have a default ACL, the file permission bits (see 5.6.1) shall be set to the value of *mode* except those set in the file mode creation mask of the process (see 5.3.3). In any of these cases (default ACL, implementation-defined default ACL, or file permission bits), access control decisions shall not be made on the newly created file until all access control information has been associated with the file.

⇒ **5.3.1.2 Open a File — Description (POSIX.1: line 197)** *Insert the following lines after line 197 in Section 5.3.1.2:*

If {_POSIX_MAC} is defined and {_POSIX_MAC_PRESENT} is in effect for the containing directory and the file is created, the MAC label of the newly created file shall be equivalent to the MAC label of the calling process. If {_POSIX_INF} is defined and the file is created, the information label of the file shall automatically be set to a value which dominates the value returned by *inf_default*().

⇒ **5.3.1.2 Open a File — Description (POSIX.1: line 234)** *Insert the following sentences after line 234 in Section 5.3.1.2:*

If {_POSIX_INF} is defined and {_POSIX_INF_PRESENT} is in effect for the file path, then the information label of the file shall automatically be set to a value which dominates the value returned by *inf_default*().

⇒ **5.3.1.2 Open a File — Description (POSIX.1: line 240)** *Insert the following paragraph after line 240 in Section 5.3.1.2:*

If {_POSIX_MAC} is defined and if the file exists and it is a FIFO special file, then the calling process shall have MAC write access to the file. If the file exists and is a FIFO special file, and the value of *oflag* includes O_RDONLY or O_RDWR then the calling process shall also have MAC read access to the file.

⇒ **5.3.4.2 Link a File — Description (POSIX.1: line 331)** *Insert the following sentence:*

If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_LINK_DIR capability.

⇒ **5.3.4.2 Link a File — Description (POSIX.1: line 336)** *Insert the following paragraph after line 336 in Section 5.3.4.2:*

If {_POSIX_MAC} is defined, the calling process shall have MAC write access to *existing*, MAC read access to the path to *existing* and *new*, and MAC read access to *new*.

If {_POSIX_MAC} is defined the calling process shall also have MAC write access to the directory in which the new entry is to be created.

If {_POSIX_INF} is defined and {_POSIX_INF_PRESENT} is in effect for the *existing* argument, the information label of *existing* remains unchanged.

⇒ **5.3.4.4 Link a File — Errors (POSIX.1: line 347)** *Insert the following after the line:*

or {_POSIX_MAC} is defined and MAC write access was denied to *existing* or to the directory in which *new* is to be created or MAC read access was denied to the path to *existing* or to *new*.

⇒ **5.4.1.2 Make a Directory — Description (POSIX.1: lines 378-380)** *Replace the second and third sentences of the paragraph with the following:*

If {_POSIX_ACL} is defined and {_POSIX_ACL_EXTENDED} is in effect for the directory in which the new directory is being created (the "containing directory") and said directory has a default ACL, the following actions shall be performed:

(1) The default ACL of the containing directory is copied to both the access ACL and the default ACL of the new directory.

(2) Both the ACL_USER_OBJ ACL entry permission bits and the file owner class permission bits of the access ACL are set to the intersection of the default ACL's ACL_USER_OBJ permission bits and the file owner class permission bits in *mode*. The action taken for any implementation-defined permissions that may be in the ACL_USER_OBJ entry shall be implementation-defined.

(3) If the default ACL does not contain an ACL_MASK entry, both the ACL_GROUP_OBJ ACL entry permission bits and the file group class permission bits of the access ACL are set to the intersection of the

default ACL's ACL_GROUP_OBJ permission bits and the file group class permission bits in *mode*. The action taken for any implementation-defined permissions that may be in the ACL_GROUP_OBJ entry shall be implementation-defined.

(4) If the default ACL contains an ACL_MASK entry, both the ACL_MASK ACL entry permission bits and the file group class permission bits of the access ACL are set to the intersection of the default ACL's ACL_MASK permission bits and the file group class permission bits in *mode*. The action taken for any implementation-defined permissions that may be in the ACL_MASK entry shall be implementation-defined.

(5) Both the ACL_OTHER ACL entry permission bits and the file other class permission bits of the access ACL are set to the intersection of the default ACL's ACL_OTHER permission bits and the file other class permission bits in *mode*. The action taken for any implementation-defined permissions that may be in the ACL_OTHER entry shall be implementation-defined.

Implementation-defined default ACL entries may affect the above algorithm but shall not alter the access permitted to any subject that does not match those implementation-defined ACL entries. Implementations may provide an additional default ACL mechanism that is applied if a default ACL as defined by this standard is not present. Such an implementation-defined default ACL interface may apply different access and/or default ACLs to created objects based upon implementation-defined criteria.

If {_POSIX_ACL} is not defined or {_POSIX_ACL_EXTENDED} is not in effect for the directory in which the file is being created (the "containing directory"), or said directory does not have a default ACL, the file permission bits of the new directory shall be set to the value of *mode* except those set in the file mode creation mask of the process (see 5.3.3). In any of these cases (default ACL, implementation-defined default ACL, or file permission bits), access control decisions shall not be made on the newly created directory until all access control information has been associated with the directory.

⇒ **5.4.1.2 Make a Directory — Description (POSIX.1: line 385)** *Insert the following paragraphs after line 385 in Section 5.4.1.2:*

If {_POSIX_MAC} is defined and {_POSIX_MAC_PRESENT} is in effect for the containing directory and the directory is created, the MAC label of the newly created directory shall be equivalent to the MAC label of the calling process.

If {_POSIX_MAC} is defined, the calling process shall require MAC write access to the containing directory.

⇒ 5.4.2.2 Make a FIFO Special File — Description (POSIX.1: lines 426-428)

Replace the second and third sentences in the paragraph with the following:

If {_POSIX_ACL} is defined and {_POSIX_ACL_EXTENDED} is in effect for the directory in which the FIFO is being created (the "containing directory") and said directory has a default ACL, the following actions shall be performed:

(1) The default ACL of the containing directory is copied to the access ACL of the new FIFO.

(2) Both the ACL_USER_OBJ ACL entry permission bits and the file owner class permission bits of the access ACL are set to the intersection of the default ACL's ACL_USER_OBJ permission bits and the file owner class permission bits in *mode*. The action taken for any implementation-defined permissions that may be in the ACL_USER_OBJ entry shall be implementation-defined.

(3) If the default ACL does not contain an ACL_MASK entry, both the ACL_GROUP_OBJ ACL entry permission bits and the file group class permission bits of the access ACL are set to the intersection of the default ACL's ACL_GROUP_OBJ permission bits and the file group class permission bits in *mode*. The action taken for any implementation-defined permissions that may be in the ACL_GROUP_OBJ entry shall be implementation-defined.

(4) If the default ACL contains an ACL_MASK entry, both the ACL_MASK ACL entry permission bits and the file group class permission bits of the access ACL are set to the intersection of the default ACL's ACL_MASK permission bits and the file group class permission bits in *mode*. The action taken for any implementation-defined permissions that may be in the ACL_MASK entry shall be implementation-defined.

(5) Both the ACL_OTHER ACL entry permission bits and the file other class permission bits of the access ACL are set to the intersection of the default ACL's ACL_OTHER permission bits and the file other class permission bits in *mode*. The action taken for any implementation-defined permissions that may be in the ACL_OTHER entry shall be implementation-defined.

Implementation-defined default ACL entries may affect the above algorithm but shall not alter the access permitted to any subject that does not match those implementation-defined ACL entries. Implementations may provide an additional default ACL mechanism that is applied if a default ACL as defined by this standard is not present. Such an implementation-defined default ACL interface may apply different access and/or default ACLs to created objects based upon implementation-defined criteria.

If {_POSIX_ACL} is not defined or {_POSIX_ACL_EXTENDED} is not in effect for the directory in which the file is being created (the "containing directory"), or said directory does not have a default ACL, the file permission bits of the new FIFO are initialized from *mode*. The file permission bits of the *mode* argument are modified by the file creation mask of the process (see 5.3.3).

⇒ 5.4.2.2 Make a FIFO Special File — Description (POSIX.1: lines 432)

Insert the following paragraphs after line 432 in Section 5.4.2.2:

If {_POSIX_MAC} is defined and {_POSIX_MAC_PRESENT} is in effect for the containing directory and the special file is created, the MAC label of the newly created special file shall be equivalent to the MAC label of the calling process and the calling process shall have MAC write access to the parent directory of the file to be created.

If {_POSIX_INF} is defined and {_POSIX_INF_PRESENT} is in effect for the file path, and the special file is created, then the information label of the special file shall automatically be set to a value which dominates the value returned by *inf_default*().

⇒ 5.5.1.2 Remove Directory Entries — Description (POSIX.1: line 474)

Insert the following paragraphs:

If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_ADMIN capability.

If {_POSIX_MAC} is defined the calling process shall have MAC write access to the directory containing the link to be removed.

⇒ 5.5.1.4 Remove Directory Entries — Errors (POSIX.1: line 487) *Insert the following phrase at the end of the line:*

or {_POSIX_MAC} is defined and MAC write access to the directory containing the link to be removed was denied.

⇒ 5.5.2.2 Remove a Directory — Description (POSIX.1: line 520) *Insert the following paragraph after line 520:*

If {_POSIX_MAC} is defined, the calling process shall have MAC write access to the parent directory of the directory being removed. If {_POSIX_MAC} is defined, the calling process shall have MAC read access to the parent directory of the directory being removed.

⇒ **5.5.2.4 Remove a Directory — Errors (POSIX.1: line 532)** *Insert the following phrase at the end of the line:*

or {_POSIX_MAC} is defined and MAC write access was denied to the parent directory of the directory being removed or MAC read access was denied to the directory containing *path*.

⇒ **5.5.3.2 Rename a File — Description (POSIX.1: line 583)** *Insert the following paragraph after line 566:*

If {_POSIX_MAC} is defined the calling process must have MAC write access to the directory containing *old* and to the directory that will contain *new*. If {_POSIX_MAC} is defined, and the link named by the *new* argument exists, the calling process shall have MAC write access to *new*.

⇒ **5.6.2.2 Get File Status — Description (POSIX.1: line 726)** *Insert the following sentence:*

If {_POSIX_ACL} is defined, and {_POSIX_ACL_EXTENDED} is in effect for the pathname, and the access ACL contains an ACL_MASK entry, then the file group class permission bits represent the ACL_MASK access ACL entry file permission bits. If {_POSIX_ACL} is defined, and {_POSIX_ACL_EXTENDED} is in effect for the pathname, and the access ACL does not contain an ACL_MASK entry, then the file group class permission bits represent the ACL_GROUP_OBJ access ACL entry file permission bits.

⇒ **5.6.2.2 Get File Status — Description (POSIX.1: line 727)** *Insert the following:*

If {_POSIX_MAC} is defined *stat*() shall require the calling process have MAC read access to the file. If {_POSIX_MAC} is defined *fstat*() shall require the calling process have the file open for read or have MAC read access to the file.

⇒ **5.6.2.4 Get File Status — Errors (POSIX.1: line 738)** *Insert the following phrase at the end of this line:*

or {_POSIX_MAC} is defined and MAC read access is denied to the file.

⇒ **5.6.4.2 Change File Modes — Description (POSIX.1: line 802)** *Insert the following sentence in line 802 of Section 5.6.4.2:*

If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_FOWNER capability. |

⇒ **5.6.4.2 Change File Modes — Description (POSIX.1: line 804)** *Insert the following sentence in line 804:*

If the process does not have appropriate privilege, then the S_ISUID bit in the *mode* is ignored. If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_FSETID capability. |

⇒ **5.6.4.2 Change File Modes — Description (POSIX.1: line 805)** *Insert the following paragraph after this line:*

If {_POSIX_ACL} is defined and {_POSIX_ACL_EXTENDED} is in effect for the pathname, then the following actions shall be performed. –

(1) The ACL_USER_OBJ access ACL entry permission bits shall be set equal to the file owner class permission bits. +

(2) If an ACL_MASK entry is not present in the access ACL, then the ACL_GROUP_OBJ access ACL entry permission bits shall be set equal to the file group class permission bits. Otherwise, the ACL_MASK access ACL entry permission bits shall be set equal to the file group class permission bits, and the ACL_GROUP_OBJ access ACL entry permission bits shall remain unchanged. +

(3) The ACL_OTHER access ACL entry permission bits shall be set equal to the file other class permission bits. +

⇒ **5.6.4.2 Change File Modes — Description (POSIX.1: line 809)** *Insert the following sentence after this line:*

If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_FSETID capability. |

⇒ **5.6.4.2 Change File Modes — Description (POSIX.1: line 811)** *Insert the following sentence after line 811 of Section 5.6.4.2:*

If {_POSIX_MAC} is defined, the calling process shall have MAC write access to the file.

⇒ **5.6.4.2 Change File Modes — Errors (POSIX.1: line 821)** *Insert the following phrase at the end of this line:*

or {_POSIX_MAC} is defined and MAC write access to the target file is denied.

⇒ **5.6.5.2 Change Owner and Group of a File — Description (POSIX.1: line 844)** *Insert the following sentence in this line:*

If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_FOWNER capability.

⇒ **5.6.5.2 Change Owner and Group of a File — Description (POSIX.1: line 847)** *Insert the following sentence after this line:*

If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_CHOWN capability.

⇒ **5.6.5.2 Change Owner and Group of a File — Description (POSIX.1: line 856)** *Insert the following sentence after the word "altered":*

If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_FSETID capability.

⇒ **5.6.5.2 Change Owner and Group of a File — Description (POSIX.1: line 858)** *Insert the following paragraph after line 858:*

If {_POSIX_MAC} is defined, the calling process shall have MAC write access to the file.

⇒ 5.6.5.4 Change Owner and Group of a File — Errors (POSIX.1: line 868)
Insert the following phrase at the end of this line:

or {_POSIX_MAC} is defined and MAC write access to the target file is denied.

⇒ 5.6.5.4 Change Owner and Group of a File — Errors (POSIX.1: line 879)
Insert the following sentences after this line:

If {_POSIX_CAP} is defined and {_POSIX_CHOWN_RESTRICTED} is defined, and the effective user ID matches the owner of the file, then appropriate privilege shall include the CAP_CHOWN capability. If {_POSIX_CAP} is defined, and the effective user ID does not match the owner of the file, then appropriate privilege shall include the CAP_FOWNER capability.

⇒ 5.6.6.2 Set File Access and Modification Times — Description (POSIX.1: line 899) *Insert the following sentence after this line:*

If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_FOWNER capability.

⇒ 5.6.6.2 Set File Access and Modification Times — Description (POSIX.1: line 899) *Insert the following paragraph after this:*

If {_POSIX_MAC} is defined, then the process shall have MAC write access to the file.

⇒ 5.6.6.2 Set File Access and Modification Times — Description (POSIX.1: line 903) *Insert the following sentence after this line:*

If {_POSIX_CAP} is defined, then appropriate privilege shall include the CAP_FOWNER capability.

⇒ 5.6.6.4 Set File Access and Modification Times — Errors (POSIX.1: line 927) *Insert the following phrase at the end of this line:*

or {_POSIX_MAC} is defined and MAC write access to the target file is denied.

⇒ **5.7.1.3 Get Configurable Pathname Variables — Returns (POSIX.1: line 965)** *Add the following variables to Table 5-2:*

Variable	*name* Value	**Notes**
{_POSIX_ACL_EXTENDED}	{_PC_ACL_EXTENDED}	(7)
{_POSIX_ACL_PATH_MAX}	{_PC_ACL_MAX}	(7)
{_POSIX_CAP_PRESENT}	{_PC_CAP_PRESENT}	(7)
{_POSIX_MAC_PRESENT}	{_PC_MAC_PRESENT}	(7)
{_POSIX_INF_PRESENT}	{_PC_INF_PRESENT}	(7)

Section 6: Revisions to Input and Output Primitives

⇒ **6.1.1.2 Create an Inter-Process Channel — Description (POSIX.1: line 21)** *Insert the following paragraphs after this line:*

If {_POSIX_MAC} is defined, then the MAC label of a pipe shall be equivalent to the MAC label of the process that created it. The MAC label is present for return by *mac_get_fd*(). This standard does not define that any access control decisions are made using the label.

If {_POSIX_INF} is defined, the information label of the pipe shall automatically be set to a value which dominates the value returned by *inf_default*().

⇒ **6.4.1.2 Read from a File — Description (POSIX.1: line 158)** *Insert the following paragraph after this line:*

If {_POSIX_INF} is defined and {_POSIX_INF_PRESENT} is in effect for the file being read, then the information label of the process shall automatically be set to an implementation-defined value that shall be the same as the value of *inf_float(file information label, process information label).*

⇒ **6.4.2.2 Write to a File — Description (POSIX.1: line 261)** *Insert the following paragraph after this line:*

If {_POSIX_INF} is defined and {_POSIX_INF_PRESENT} is in effect for the file being written, then the information label of the file shall automatically be set to an implementation-defined value which shall be the same as the value of *inf_float(process information label, file information label).*

Section 8: Revisions to C Programming Language Specific Services

⇒ 8.2.3 Interactions of Other File Type C Functions (POSIX.1: line 345)

Insert the following sentence after line 345:

In particular, if an optional portion of this standard is present, the traits specific to the option in the underlying function must be shared by the stream function.

Section 23: Access Control Lists

23.1 General Overview

The POSIX.1e ACL facility defines an interface for manipulating Access Control Lists. This interface is an extension of the POSIX.1 file permission bits. Support for the interfaces defined in this section is optional but shall be provided if the symbol {_POSIX_ACL} is defined.

The POSIX.1e ACL interface does not alter the syntax of existing POSIX.1 interfaces. However, the access control semantics associated with existing POSIX.1 interfaces are necessarily more complex as a result of ACLs. The POSIX.1e ACL facility includes:

(1) Definition and use of access and default ACLs

(2) Definition of initial access permissions on object creation

(3) Specification of the access check algorithm

(4) Functions to manipulate ACLs.

Every object can be thought of as having associated with it an ACL that governs the discretionary access to that object; this ACL is referred to as an access ACL. In addition, a directory may have an associated ACL that governs the initial access ACL for objects created within that directory; this ACL is referred to as a default ACL. Files, as defined by POSIX.1, are the only objects for which the POSIX.1e ACL facility defines ACLs. For the purposes of this document, the POSIX.1 file permission bits will be considered as a special case of an ACL. An ACL consists of a set of ACL entries. An ACL entry specifies the access permissions on the associated object for an individual user or a group of users. The POSIX.1e ACL facility does not dictate the actual implementation of ACLs or the existing POSIX.1 file permission bits. The POSIX.1e ACL facility does not dictate the specific internal representation of an ACL nor any ordering of entries within an ACL. In particular, the order of internal storage of entries within an ACL does not affect the order of evaluation.

In order to read an ACL from an object, a process must have read access to the object's attributes. In order to write (update) an ACL to an object, the process must have write access to the object's attributes.

23.1.1 ACL Entry Composition

An ACL entry contains, at a minimum, three distinct pieces of information:

(1) tag type: specifies the type of ACL entry

(2) qualifier: specifies an instance of an ACL entry tag type

(3) permissions set: specifies the discretionary access rights for processes identified by the tag type and qualifier

A conforming implementation may add implementation-defined pieces of information to an ACL entry.

A conforming ACL implementation shall define the following tag types:

— ACL_GROUP: an ACL entry of tag type ACL_GROUP denotes discretionary access rights for processes whose effective group ID or any supplemental group IDs match the ACL entry qualifier

— ACL_GROUP_OBJ: an ACL entry of tag type ACL_GROUP_OBJ denotes discretionary access rights for processes whose effective group ID or any supplemental group IDs match the group ID of the group of the file.

— ACL_MASK: an ACL entry of tag type ACL_MASK denotes the maximum discretionary access rights that can be granted to a process in the file group class.

— ACL_OTHER: an ACL entry of tag type ACL_OTHER denotes discretionary access rights for processes whose attributes do not match any other entry in the ACL

— ACL_USER: an ACL entry of tag type ACL_USER denotes discretionary access rights for processes whose effective user ID matches the ACL entry qualifier

— ACL_USER_OBJ: an ACL entry of tag type ACL_USER_OBJ denotes discretionary access rights for processes whose effective user ID matches the user ID of the owner of the file.

A conforming implementation may define additional tag types.

This standard extends the file group class, as defined in POSIX.1, to include processes which are not in the file owner class and which match ACL entries with the tag types ACL_GROUP, ACL_GROUP_OBJ, ACL_USER, or any implementation-defined tag types that are not in the file owner class.

An ACL shall contain exactly one entry for each of ACL_USER_OBJ, ACL_GROUP_OBJ, and ACL_OTHER tag types. ACL entries with ACL_GROUP and ACL_USER tag types shall appear zero or more times in an ACL. A conforming implementation shall support the maximum number of entries in an ACL, as defined by the value of {_POSIX_ACL_PATH_MAX}, on a non-empty set of objects.

The three ACL entries of tag type ACL_USER_OBJ, ACL_GROUP_OBJ, and ACL_OTHER are referred to as the *required ACL entries*. An ACL that contains only the required ACL entries is called a *minimum ACL*. An ACL which is not a minimum ACL is called an *extended ACL*.

An ACL that contains ACL_GROUP, ACL_USER, or implementation-defined ACL entries in the file group class shall contain exactly one ACL_MASK entry. If an ACL does not contain ACL_GROUP, ACL_USER, or implementation-defined ACL entries in the file group class, then the ACL_MASK entry shall be optional.

The qualifier field associated with the POSIX.1e ACL facility defined tag types shall not be extended to contain any implementation-defined information. The qualifier field associated with implementation-defined tag types may contain fully implementation-defined information. The qualifier field shall be unique among all entries of the same POSIX.1e ACL facility defined tag type in a given ACL. For entries of the ACL_USER and ACL_GROUP tag type, the qualifier field shall be present and contain either a user ID or a group ID respectively. The value of the qualifier field in entries of tag types ACL_GROUP_OBJ, ACL_MASK, ACL_OTHER, and ACL_USER_OBJ shall be unspecified.

The set of discretionary access permissions shall, at a minimum, include: read, write, and execute/search. Additional permissions may be added and shall be implementation-defined.

23.1.2 Relationship with File Permission Bits

ACL interfaces extend the file permission bit interfaces to provide a finer granularity of access control than is possible with permission bits alone. As a superset of the file permission bit interface, the ACL functionality specified preserves compatibility with applications using POSIX.1 interfaces to retrieve and manipulate access permission bits, e.g., *chmod*(), *creat*(), and *stat*().

The file permission bits shall correspond to three entries in an ACL. The permissions specified by the file owner class permission bits correspond to the permissions associated with the ACL_USER_OBJ entry. The permissions specified by the file group class permission bits correspond to the permissions associated with the ACL_GROUP_OBJ entry or the permissions associated with the ACL_MASK entry if the ACL contains an ACL_MASK entry. The permissions specified by the file other class permission bits correspond to the permissions associated with the ACL_OTHER entry.

The permissions associated with these ACL entries shall be identical to the permissions defined for the corresponding file permission bits. Modification of the permissions associated with these ACL entries shall modify the corresponding file permission bits and modification of the file permission bits shall modify the permissions of the corresponding ACL entries.

When the file permissions of an object are modified, e.g. using the *chmod*() function, then:

(1) the corresponding permissions associated with the ACL_USER_OBJ entry shall be set equal to each of the file owner class permission bits

(2) if the ACL does not contain an ACL_MASK entry, then the corresponding permissions associated with the ACL_GROUP_OBJ entry shall be set

equal to each of the file group class permission bits

(3) if the ACL contains an ACL_MASK entry, then the corresponding permissions associated with the ACL_MASK entry shall be set equal to each of the file group class permission bits and the permissions associated with the ACL_GROUP_OBJ entry shall not be modified.

(4) the corresponding permissions associated with the ACL_OTHER entry shall be set equal to each of the file other class permission bits

23.1.3 Default ACLs

A default ACL is an additional ACL which may be associated with a directory, but which has no operational effect on the discretionary access on that directory. It shall be possible to associate a default ACL with any directory for which {_POSIX_ACL_EXTENDED} is in effect. If there is a default ACL associated with a directory, then that default ACL shall be used, as specified in 23.1.4, to initialize the access ACL for any object created in that directory. If the newly created object is a directory and if the parent directory has a default ACL, then the new directory inherits the parent's default ACL as its default ACL. Entries within a default ACL are manipulated using the same interfaces as those used for an access ACL. A default ACL has the same minimum required entries as an access ACL as specified in 23.1.1.

Directories are not required to have a default ACL. While any particular directory for which {_POSIX_ACL_EXTENDED} is in effect may have a default ACL, a conforming implementation shall support the default ACL interface described here. If a default ACL does not exist on a directory, then any implementation-defined default ACL(s) may be applied to the access or default ACLs of objects created in that directory. If no default ACL is applied, the initial access control information shall be obtained as specified in 5.3 and 5.4.

23.1.4 Associating an ACL with an Object at Object Creation Time

When an object is created, its access ACL is always initialized. If a default ACL is associated with a directory, two components may be used to determine the initial access ACL for objects created within that directory:

(1) The *mode* parameter to functions which can create objects may be used by an application to specify the maximum discretionary access permissions to be associated with the resulting object. There are four POSIX.1 functions which can be used to create objects: *creat*(), *mkdir*(), *mkfifo*(), and *open*() (with the O_CREAT flag).

(2) The default ACL may be used by the owner of a directory to specify the maximum discretionary access permissions to be associated with objects created within that directory.

The initial access control information is obtained as is specified in 5.3 and 5.4. Implementations may provide an additional default ACL that is applied if a default ACL as defined by this standard is not present. Such an implementation-defined default ACL interface may apply different access and/or default ACLs to created objects based upon implementation-defined criteria.

The physical ordering of the ACL entries of a newly created object shall be unspecified.

23.1.5 ACL Access Check Algorithm

A process may request discretionary read, write, execute/search or any implementation-defined access mode of an object protected by an access ACL. The algorithm below matches specific attributes of the process to ACL entries. The process's request is granted only if a matching ACL entry grants all of the requested access modes.

The access check algorithm shall check the ACL entries in the following relative order:

(1) the ACL_USER_OBJ entry

(2) any ACL_USER entries

(3) the ACL_GROUP_OBJ entry as well as any ACL_GROUP entries

(4) the ACL_OTHER entry

Implementation-defined entries may be checked at any implementation-defined points in the access check algorithm, as long as the above relative ordering is maintained. Implementation-defined entries may grant or deny access but shall not alter the access permitted to any process that does not match those implementation entries.

If no ACL_USER_OBJ, ACL_USER, ACL_GROUP_OBJ, or ACL_GROUP entries apply and no implementation-defined entries apply, the permissions in the ACL_OTHER entry shall be used to determine access.

Note, the algorithm presented is a logical description of the access check. The physical code sequence may be different.

(1) **If** the effective user ID of the process matches the user ID of the object owner
then
    set matched entry to ACL_USER_OBJ entry

(2) **else if** the effective user ID of the process matches the user ID specified in any ACL_USER tag type ACL entry,
then
    set matched entry to the matching ACL_USER entry

(3) **else if** the effective group ID or any of the supplementary group IDs of the process match the group ID of the object or match the group ID specified in any ACL_GROUP or ACL_GROUP_OBJ tag type ACL entry
then

if the requested access modes are granted by at least one entry matched by the effective group ID or any of the supplementary group IDs of the process
then
  set matched entry to a granting entry
else
  access is denied
endif

(4) **else if** the requested access modes are granted by the ACL_OTHER entry of the ACL,
then
  set matched entry to the ACL_OTHER entry
endif

(5) **If** the requested access modes are granted by the matched entry
then
  if the matched entry is an ACL_USER_OBJ or ACL_OTHER entry
  then
    access is granted

  else if the requested access modes are also granted by the ACL_MASK entry or no ACL_MASK entry exists in the ACL
  then
    access is granted
  else
    access is denied
  endif
else
  access is denied
endif

23.1.6 ACL Functions

Functional interfaces are defined to manipulate ACLs and ACL entries. The functions provide a portable interface for editing and manipulating the entries within an ACL and the fields within an ACL entry.

Four groups of functions are defined to:

(1) manage the ACL working storage area

(2) manipulate ACL entries

(3) manipulate an ACL on an object

(4) translate an ACL into different formats.

23.1.6.1 ACL Storage Management

These functions manage the storage areas used to contain working copies of ACLs. An ACL in working storage shall not be used in any access control decisions.

acl_dup()	Duplicates an ACL in a working storage area
acl_free()	Release the working storage area allocated to an ACL data object
acl_init()	Allocates and initializes an ACL working storage area

23.1.6.2 ACL Entry Manipulation

These functions manipulate ACL entries in working storage. The functions are divided into several groups:

(1) Functions that manipulate complete entries in an ACL:

acl_copy_entry()	Copies an ACL entry to another ACL entry
acl_create_entry()	Creates a new entry in an ACL
acl_delete_entry()	Deletes an entry from an ACL
acl_get_entry()	Returns a descriptor to an ACL entry
acl_valid()	Validates an ACL by checking for duplicate, missing, and ill-formed entries

(2) Functions that manipulate permissions within an ACL entry:

acl_add_perm()	Adds a permission to a given permission set
acl_calc_mask()	Sets the permission granted by the ACL_MASK entry to the maximum permissions granted by the ACL_GROUP, ACL_GROUP_OBJ, ACL_USER and implementation-defined ACL entries
acl_clear_perms()	Clears all permissions from a given permission set
acl_delete_perm()	Deletes a permission from a given permission set
acl_get_permset()	Returns the permissions in a given ACL entry
acl_set_permset()	Sets the permissions in a given ACL entry

(3) Functions that manipulate the tag type and qualifier in an ACL entry:

acl_get_qualifier()	Returns the qualifier in a given ACL entry
acl_get_tag_type()	Returns the tag type in a given ACL entry
acl_set_qualifier()	Sets the qualifier in a given ACL entry
acl_set_tag_type()	Sets the tag type in a given ACL entry

23.1.6.3 ACL Manipulation on an Object

These functions read the contents of an access ACL or a default ACL into working storage and write an ACL in working storage to an object's access ACL or default ACL. The functions also delete a default ACL from an object:

acl_delete_def_file()	Deletes the default ACL associated with an object
acl_get_fd()	Reads the contents of an access ACL associated with a file descriptor into working storage
acl_get_file()	Reads the contents of an access ACL or default ACL associated with an object into working storage
acl_set_fd()	Writes the ACL in working storage to the object associated with a file descriptor as an access ACL
acl_set_file()	Writes the ACL in working storage to an object as an access ACL or default ACL

23.1.6.4 ACL Format Translation

The standard defines three different representations for ACLs:

external form	The exportable, contiguous, persistent representation of an ACL in user-managed space
internal form	The internal representation of an ACL in working storage
text form	The structured text representation of an ACL

These functions translate an ACL from one representation into another.

acl_copy_ext()	Translates an internal form of an ACL to an external form of an ACL
acl_copy_int()	Translates an external form of an ACL to an internal form of an ACL
acl_from_text()	Translates a text form of an ACL to an internal form of an ACL
acl_size()	Returns the size in bytes required to store the external form of an ACL that is the result of an *acl_copy_ext*()
acl_to_text()	Translates an internal form of an ACL to a text form of an ACL

23.1.7 POSIX.1 Functions Covered by ACLs

The following table lists the POSIX.1 interfaces that are changed to reflect Access Control Lists. There are no changes to the syntax of these interfaces.

Existing Function	POSIX.1 Section
access()	5.6.3
chmod()	5.6.4
creat()	5.3.2
fstat()	5.6.2
mkdir()	5.4.1
mkfifo()	5.4.2
open()	5.3.1
stat()	5.6.2

23.2 Header

The header `<sys/acl.h>` defines the symbols used in the ACL interfaces.

Some of the data types used by the ACL functions are not defined as part of this standard but shall be implementation-defined. If {_POSIX_ACL} is defined, these types shall be defined in the header `<sys/acl.h>`, which contains definitions for at least the types shown in Table 23-1.

Table 23-1 – ACL Data Types

Defined Type	Description
acl_entry_t	Used as a descriptor for a specific ACL entry in ACL working storage. This data type is non-exportable data.
acl_perm_t	Used for individual object access permissions. This data type is exportable data.
acl_permset_t	Used for the set of object access permissions. This data type is non-exportable data.
acl_t	Used as a pointer to an ACL in ACL working storage. This data type is non-exportable data.
acl_tag_t	Used to distinguish different types of ACL entries. This data type is exportable data.
acl_type_t	Used to distinguish different types of ACLs (e.g., access, default). This data type is exportable data.

The symbolic constants defined in Table 23-2, Table 23-3, Table 23-4, Table 23-5, Table 23-6, shall be defined in the header `<sys/acl.h>`. +

23.2.1 acl_entry_t

This typedef shall define an opaque, implementation-defined descriptor for an ACL entry. The internal structure of an *acl_entry_t* is unspecified.

23.2.2 acl_perm_t

This typedef shall define a data type capable of storing an individual object access permission.

Table 23-2 contains *acl_perm_t* values for *acl_add_perm*(), *acl_clear_perms*(), and *acl_delete_perm*().

Table 23-2 – acl_perm_t Values

Constant	Description
ACL_EXECUTE	ACL execute permission
ACL_READ	ACL read permission
ACL_WRITE	ACL write permission

These constants shall be implementation-defined unique values.

23.2.3 acl_permset_t

This typedef shall define the opaque, implementation-defined descriptor for a set of object access permissions. The internal structure of an *acl_permset_t* is unspecified.

23.2.4 acl_t

This typedef shall define a pointer to an opaque, implementation-defined ACL in ACL working storage, the internal structure of which is unspecified.

23.2.5 acl_tag_t

This typedef shall define a data type capable of storing an individual ACL entry tag type.

Table 23-3 contains *acl_tag_t* values for *acl_get_tag_type*() and *acl_set_tag_type*().

Table 23-3 – acl_tag_t Values

Constant	Description
ACL_GROUP	ACL entry for a specific group
ACL_GROUP_OBJ	ACL entry for the owning group
ACL_MASK	ACL entry that denotes the maximum permissions allowed on all other ACL entry types except for ACL_USER_OBJ and ACL_OTHER (including implementation-defined types in the file group class)
ACL_OTHER	ACL entry for users whose process attributes are not matched in any other ACL entry
ACL_UNDEFINED_TAG	Undefined ACL entry
ACL_USER	ACL entry for a specific user
ACL_USER_OBJ	ACL entry for the object owner

These constants shall be implementation-defined unique values.

23.2.6 acl_type_t

This typedef shall define a data type capable of storing an individual ACL type.

Table 23-4 contains *acl_type_t* values for *acl_get_file*() and *acl_set_file*().

Table 23-4 – acl_type_t Values

Constant	Description
ACL_TYPE_ACCESS	Indicates an access ACL
ACL_TYPE_DEFAULT	Indicates a default ACL

These constants shall be implementation-defined unique values.

23.2.7 ACL Qualifier

Table 23-5 contains the value of undefined user IDs or group IDs for the ACL qualifier.

Table 23-5 – ACL Qualifier Constants

Constant	Description
ACL_UNDEFINED_ID	Undefined ID

These constants shall be implementation-defined values.

23.2.8 ACL Entry

Table 23-6 contains the values used to denote ACL entries to be retrieved by the *acl_get_entry*() function.

Table 23-6 – ACL Entry Constants

Constant	Description
ACL_FIRST_ENTRY	Return the first ACL entry in the ACL.
ACL_NEXT_ENTRY	Return the next ACL entry in the ACL.

These constants shall be implementation-defined values.

23.3 Text Form Representation

This section defines the long and short text forms of ACLs. The long text form is defined first in order to give a complete specification with no exceptions. The short text form is defined second because it is specified relative to the long text form.

23.3.1 Long Text Form for ACLs

The long text form is used for either input or output of ACLs and is defined as follows:

<*acl_entry*>
[<*acl_entry*>] ...

Each <*acl_entry*> line shall contain one ACL entry with three required colon-separated fields: an ACL entry tag type, an ACL entry qualifier, and the discretionary access permissions. An implementation may define additional colon-separated fields after the required fields. Comments may be included on any <*acl_entry*> line. If a comment starts at the beginning of a line, then the entire line shall be interpreted as a comment.

The first field contains the ACL entry tag type. This standard defines the following ACL entry tag type keywords, one of which shall appear in the first field:

`user` A `user` ACL entry specifies the access granted to either the file owner or a specified user.

`group` An `group` ACL entry specifies the access granted to either the file owning group or a specified group.

`other` An `other` ACL entry specifies the access granted to any process that does not match any `user`, `group`, or implementation-defined ACL entries.

`mask` A `mask` ACL entry specifies the maximum access which can be granted by any ACL entry except the `user` entry for the file owner and the `other` entry.

An implementation may define additional ACL entry types.

The second field contains the ACL entry qualifier (referred to in the remainder of this section as qualifier). This standard defines the following qualifiers:

- *uid* — This qualifier specifies a user name or a user ID number.
- *gid* — This qualifier specifies a group name or a group ID number.
- *empty* — This qualifier specifies that no *uid* or *gid* information is to be applied to the ACL entry. An *empty* qualifier shall be represented by an empty string or by white space.

An implementation may define additional qualifiers.

The third field contains the discretionary access permissions. This standard defines the following symbolic discretionary access permissions:

- `r` — Read access
- `w` — Write access
- `x` — Execute/search access
- – — No access by this ACL entry.

The discretionary access permissions field shall contain exactly one each of the following characters in the following order: `r`, `w`, and `x`. Each of these may be replaced by the "-" character to indicate no access. An implementation may define additional characters following the required characters that represent implementation-defined permissions.

A `user` entry with an *empty* qualifier shall specify the access granted to the file owner. A `user` entry with a *uid* qualifier shall specify the access permissions granted to the user name matching the *uid* value. If the *uid* value does not match a user name, then the ACL entry shall specify the access permissions granted to the user ID matching the numeric *uid* value. +

A `group` entry with an *empty* qualifier shall specify the access granted to the file owning group. A `group` entry with a *gid* qualifier shall specify the access permissions granted to the group name matching the *gid* value. If the *gid* value does not match a group name, then the ACL entry shall specify the access permissions granted to the group ID matching the numeric *gid* value. +

The `mask` and `other` entries shall contain an *empty* qualifier. An implementation may define additional ACL entry types that use the *empty* qualifier.

A number-sign (#) starts a comment on an <*acl_entry*> line. A comment may start at the beginning of a line, after the required fields and after any implementation-defined, colon-separated fields. The end of the line denotes the end of the comment. –

If an ACL entry contains permissions that are not also contained in the `mask` entry, then the output text form for that <*acl_entry*> line shall be displayed as described above followed by a number-sign (#), the string "effective: ", and the effective access permissions for that ACL entry.

White space is permitted in <*acl_entry*> lines as follows: at the start of the line; immediately before and after a ":" separator; immediately before the first number-sign (#) character; at any point after the first number-sign (#) character.

Comments shall have no effect on the discretionary access check of the object with which they are associated. An implementation shall define whether or not comments are stored with an ACL.

If an implementation allows the colon character ":" to be present in an ACL entry qualifier, then that implementation shall provide a method for distinguishing between a colon character as a field separator in an ACL entry definition and a colon character as a component of the ACL entry qualifier value.

23.3.2 Short Text Form for ACLs

The short text form is used only for input of ACLs and is defined as follows:

<*acl_entry*>[,<*acl_entry*>]...

Each <*acl_entry*> shall contain one ACL entry, as defined in 23.3.1, with two exceptions.

The ACL entry tag type keyword shall appear in the first field in either its full unabbreviated form or its single letter abbreviated form. The abbreviation for `user` is "`u`", the abbreviation for `group` is "`g`", the abbreviation for `other` is "`o`", and the abbreviation for `mask` is "`m`". An implementation may define additional ACL entry tag type abbreviations.

There are no exceptions for the second field in the short text form for ACLs.

The discretionary access permissions shall appear in the third field. The symbolic string shall contain at most one each of the following characters in any order: `r`, `w`, and `x`; implementations may define additional characters that may appear in any order within the string.

23.4 Functions

Support for the ACL facility functions described in this section is optional. If the symbol {_POSIX_ACL} is defined, the implementation supports the ACL option and all of the ACL functions shall be implemented as described in this section. If {_POSIX_ACL} is not defined, the result of calling any of these functions is unspecified.

The error [ENOTSUP] shall be returned in those cases where the system supports the ACL facility but the particular ACL operation cannot be applied because of restrictions imposed by the implementation.

23.4.1 Add a Permission to an ACL Permission Set

Function: *acl_add_perm*()

23.4.1.1 Synopsis

```
#include <sys/acl.h>
int acl_add_perm (acl_permset_t permset_d, acl_perm_t perm);
```

23.4.1.2 Description

The *acl_add_perm*() function shall add the permission contained in argument *perm* to the permission set referred to by argument *permset_d*. An attempt to add a permission that is already granted by the permission set shall not be considered an error.

Any existing descriptors that refer to *permset_d* shall continue to refer to that permission set.

23.4.1.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error.

23.4.1.4 Errors

If any of the following conditions occur, the *acl_add_perm*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL] Argument *permset_d* is not a valid descriptor for a permission set within an ACL entry.

Argument *perm* does not contain a valid *acl_perm_t* value.

23.4.1.5 Cross-References

acl_clear_perms(), 23.4.3; *acl_delete_perm*(), 23.4.10; *acl_get_permset*(), 23.4.17; *acl_set_permset*(), 23.4.23.

23.4.2 Calculate the File Group Class Mask

Function: *acl_calc_mask*()

23.4.2.1 Synopsis

```
#include <sys/acl.h>

int acl_calc_mask (acl_t *acl_p);
```

23.4.2.2 Description

The *acl_calc_mask*() function shall calculate and set the permissions associated with the ACL_MASK ACL entry of the ACL referred to by *acl_p*. The value of the new permissions shall be the union of the permissions granted by the ACL_GROUP, ACL_GROUP_OBJ, ACL_USER, and any implementation-defined tag types which match processes in the file group class contained in the ACL referred to by *acl_p*. If the ACL referred to by *acl_p* already contains an ACL_MASK entry, its permissions shall be overwritten; if it does not contain an ACL_MASK entry, one shall be added. If the ACL referred to by *acl_p* does not contain enough space for the new ACL entry, then additional working storage may be allocated. If the working storage cannot be increased in the current location, then it may be relocated and the previous working storage shall be released and a pointer to the new working storage shall be returned via *acl_p*.

The order of existing entries in the ACL is undefined after this function.

Any existing ACL entry descriptors that refer to entries in the ACL shall continue to refer to those entries. Any existing ACL pointers that refer to the ACL referred to by *acl_p* shall continue to refer to the ACL.

23.4.2.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error.

23.4.2.4 Errors

If any of the following conditions occur, the *acl_calc_mask*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL] Argument *acl_p* does not point to a pointer to a valid ACL.

[ENOMEM] The *acl_calc_mask*() function is unable to allocate the memory required for an ACL_MASK ACL entry. –

23.4.2.5 Cross-References

acl_get_entry(), 23.4.14; *acl_valid*(), 23.4.28.

23.4.3 Clear All Permissions from an ACL Permission Set

Function: *acl_clear_perms*()

23.4.3.1 Synopsis

```
#include <sys/acl.h>
int acl_clear_perms (acl_permset_t permset_d);
```

23.4.3.2 Description

The *acl_clear_perms*() function shall clear all permissions from the permission set referred to by argument *permset_d*.

Any existing descriptors that refer to *permset_d* shall continue to refer to that permission set.

23.4.3.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

23.4.3.4 Errors

If any of the following conditions occur, the *acl_clear_perms*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] Argument *permset_d* is not a valid descriptor for a permission set within an ACL entry.

23.4.3.5 Cross-References

acl_add_perm(), 23.4.1; *acl_delete_perm*(), 23.4.10; *acl_get_permset*(), 23.4.17; *acl_set_permset*(), 23.4.23.

23.4.4 Copy an ACL Entry

Function: *acl_copy_entry*()

23.4.4.1 Synopsis

```
#include <sys/acl.h>
int acl_copy_entry (acl_entry_t dest_d, acl_entry_t src_d);
```

23.4.4.2 Description

The *acl_copy_entry*() function shall copy the contents of the ACL entry indicated by the *src_d* descriptor to the existing ACL entry indicated by the *dest_d* descriptor. The *src_d* and *dest_d* descriptors may refer to entries in different ACLs.

The *src_d*, *dest_d* and any other ACL entry descriptors that refer to entries in either ACL shall continue to refer to those entries. The order of all existing entries in both ACLs shall remain unchanged.

23.4.4.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

23.4.4.4 Errors

If any of the following conditions occur, the *acl_copy_entry*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] Argument *src_d* or *dest_d* is not a valid descriptor for an ACL entry.

Arguments *src_d* and *dest_d* reference the same ACL entry.

23.4.4.5 Cross-References

acl_get_entry(), 23.4.14.

23.4.5 Copy an ACL From System to User Space

Function: *acl_copy_ext*()

23.4.5.1 Synopsis

```
#include <sys/acl.h>

ssize_t acl_copy_ext (void *buf_p, acl_t acl, ssize_t size);
```

23.4.5.2 Description

The *acl_copy_ext*() function shall copy an ACL, pointed to by *acl*, from system-managed space to the user managed space pointed to by *buf_p*. The *size* parameter represents the size in bytes of the buffer pointed to by *buf_p*. The format of the ACL placed in the user-managed space pointed to by *buf_p* shall be a contiguous, persistent data item, the format of which is unspecified. It is the responsibility of the invoker to allocate an area large enough to hold the copied ACL. The size of the exportable, contiguous, persistent form of the ACL may be obtained by invoking the *acl_size*() function.

Any ACL entry descriptors that refer to an entry in the ACL referenced by *acl* shall continue to refer to those entries. Any existing ACL pointers that refer to the ACL referenced by *acl* shall continue to refer to the ACL.

23.4.5.3 Returns

Upon successful completion, the *acl_copy_ext*() function shall return the number of bytes placed in the user-managed space pointed to by *buf_p*. Otherwise, a value of (*ssize_t*) −1 shall be returned and *errno* shall be set to indicate the error.

23.4.5.4 Errors

If any of the following conditions occur, the *acl_copy_ext*() function shall return a value of (*ssize_t*) −1 and set *errno* to the corresponding value:

[EINVAL] The *size* parameter is zero or negative.

Argument *acl* does not point to a valid ACL.

The ACL referenced by *acl* contains one or more improperly formed ACL entries, or for some other reason cannot be translated into the external form ACL.

[ERANGE] The *size* parameter is greater than zero but smaller than the length of the contiguous, persistent form of the ACL.

23.4.5.5 Cross-References

acl_copy_int(), 23.4.6; *acl_size*(), 23.4.26.

23.4.6 Copy an ACL From User to System Space

Function: *acl_copy_int*()

23.4.6.1 Synopsis

```
#include <sys/acl.h>

acl_t acl_copy_int (const void *buf_p);
```

23.4.6.2 Description

The *acl_copy_int*() function shall copy an exportable, contiguous, persistent form of an ACL, pointed to by *buf_p*, from user-managed space to system-managed space.

This function may cause memory to be allocated. The caller should free any releaseable memory, when the new ACL is no longer required, by calling *acl_free*() with the (*void* *)*acl_t* as an argument.

Upon successful completion, this function shall return a pointer that references the ACL in ACL working storage.

23.4.6.3 Returns

Upon successful completion, the *acl_copy_int*() function shall return a pointer referencing the ACL in ACL working storage. Otherwise, a value of (*acl_t*)**NULL** shall be returned, and *errno* shall be set to indicate the error.

23.4.6.4 Errors

If any of the following conditions occur, the *acl_copy_int*() function shall return a value of (*acl_t*)**NULL** and set *errno* to the corresponding value:

[EINVAL] The buffer pointed to by argument *buf_p* does not contain a valid external form ACL.

[ENOMEM] The ACL working storage requires more memory than is allowed by the hardware or system-imposed memory management constraints.

23.4.6.5 Cross-References

acl_copy_ext(), 23.4.5; *acl_get_entry*(), 23.4.14; *acl_free*(), 23.4.12.

23.4.7 Create a New ACL Entry

Function: *acl_create_entry*()

23.4.7.1 Synopsis

```
#include <sys/acl.h>

int acl_create_entry (acl_t *acl_p, acl_entry_t *entry_p);
```

23.4.7.2 Description

The *acl_create_entry*() function creates a new ACL entry in the ACL pointed to by the contents of the pointer argument *acl_p*.

This function may cause memory to be allocated. The caller should free any releaseable memory, when the ACL is no longer required, by calling *acl_free*() with (*void* *)*acl_t* as an argument.

If the ACL working storage cannot be increased in the current location, then the working storage for the ACL pointed to by *acl_p* may be relocated and the previous working storage shall be released. A pointer to the new working storage shall be returned via *acl_p*. Upon successful completion, the *acl_create_entry*() function shall return a descriptor for the new ACL entry via *entry_p*.

The components of the new ACL entry are initialized in the following ways: the ACL tag type component shall contain ACL_UNDEFINED_TAG, the qualifier component shall contain ACL_UNDEFINED_ID, and the set of permissions shall have no permissions enabled. Other features of a newly created ACL entry shall be implementation-defined. Any existing ACL entry descriptors that refer to entries in the ACL shall continue to refer to those entries.

23.4.7.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error.

23.4.7.4 Errors

If any of the following conditions occur, the *acl_create_entry*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL] Argument *acl_p* does not point to a pointer to a valid ACL.

[ENOMEM] The ACL working storage requires more memory than is allowed by the hardware or system-imposed memory management constraints.

23.4.7.5 Cross-References

acl_delete_entry(), 23.4.9; *acl_get_entry*(), 23.4.14.

23.4.8 Delete a Default ACL by Filename

Function: *acl_delete_def_file*()

23.4.8.1 Synopsis

```
#include <sys/acl.h>

int acl_delete_def_file (const char *path_p);
```

23.4.8.2 Description

The *acl_delete_def_file*() function deletes a default ACL from the directory whose pathname is pointed to by the argument *path_p*. The effective user ID of the process must match the owner of the directory or the process must have appropriate privilege to delete the default ACL from *path_p*. If {_POSIX_CAP} is defined, then appropriate privilege shall include CAP_FOWNER. In addition, if {_POSIX_MAC} is defined, then the process must have MAC write access to the directory.

If the argument *path_p* is not a directory, then the function shall fail. It shall not be considered an error if *path_p* is a directory and either

{_POSIX_ACL_EXTENDED} is not in effect for *path_p*, or *path_p* does not have a default ACL.

Upon successful completion, *acl_delete_def_file*() shall delete the default ACL associated with the argument *path_p*. If *acl_delete_def_file*() is unsuccessful, the default ACL associated with the argument *path_p* shall not be changed.

23.4.8.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error.

23.4.8.4 Errors

If any of the following conditions occur, the *acl_delete_def_file*() function shall return –1 and set *errno* to the corresponding value:

[EACCES] Search permission is denied for a component of the path prefix or the object exists and the process does not have appropriate access rights.

If {_POSIX_MAC} is defined, MAC write access to *path_p* is denied.

[ENAMETOOLONG]
The length of the *path_p* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {POSIX_NO_TRUNC} is in effect.

[ENOENT] The named object does not exist or the *path_p* argument points to an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

Argument *path_p* does not refer to a directory.

[EPERM] The process does not have appropriate privilege to perform the operation to delete the default ACL.

[EROFS] This function requires modification of a file system which is currently read-only.

23.4.8.5 Cross-References

acl_get_file(), 23.4.16; *acl_set_file*(), 23.4.22.

23.4.9 Delete an ACL Entry

Function: *acl_delete_entry*()

23.4.9.1 Synopsis

```
#include <sys/acl.h>
int acl_delete_entry (acl_t acl, acl_entry_t entry_d);
```

23.4.9.2 Description

The *acl_delete_entry*() function shall remove the ACL entry indicated by the *entry_d* descriptor from the ACL pointed to by *acl*.

Any existing ACL entry descriptors that refer to entries in *acl* other than that referred to by *entry_d* shall continue to refer to the same entries. The argument *entry_d* and any other ACL entry descriptors that refer to the same ACL entry are undefined after this function completes. Any existing ACL pointers that refer to the ACL referred to by *acl* shall continue to refer to the ACL.

23.4.9.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

23.4.9.4 Errors

If any of the following conditions occur, the *acl_delete_entry*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] Argument *acl* does not point to a valid ACL. Argument *entry_d* is not a valid descriptor for an ACL entry in *acl*.

[ENOSYS] This function is not supported by the implementation.

23.4.9.5 Cross-References

acl_copy_entry(), 23.4.4; *acl_create_entry*(), 23.4.7; *acl_get_entry*(), 23.4.14.

23.4.10 Delete Permissions from an ACL Permission Set

Function: *acl_delete_perm*()

23.4.10.1 Synopsis

```
#include <sys/acl.h>
int acl_delete_perm (acl_permset_t permset_d, acl_perm_t perm);
```

23.4.10.2 Description

The *acl_delete_perm*() function shall delete the permission contained in argument *perm* from the permission set referred to by argument *permset_d*. An attempt to delete a permission that is not granted by the ACL entry shall not be considered an error.

Any existing descriptors that refer to *permset_d* shall continue to refer to that permission set.

23.4.10.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

23.4.10.4 Errors

If any of the following conditions occur, the *acl_delete_perm*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] Argument *permset_d* is not a valid descriptor for a permission set within an ACL entry.

Argument *perm* does not contain a valid *acl_perm_t* value.

[ENOSYS] This function is not supported by the implementation.

23.4.10.5 Cross-References

acl_add_perm(), 23.4.1; *acl_clear_perms*(), 23.4.3; *acl_get_permset*(), 23.4.17; *acl_set_permset*(), 23.4.23.

23.4.11 Duplicate an ACL

Function: *acl_dup*()

23.4.11.1 Synopsis

```
#include <sys/acl.h>

acl_t acl_dup (acl_t acl);
```

23.4.11.2 Description

The *acl_dup*() function returns a pointer to a copy of the ACL pointed to by argument *acl*.

This function may cause memory to be allocated. When the new ACL is no longer required, the caller should free any releaseable memory by calling *acl_free*() with the (*void* ∗)*acl_t* as an argument.

Any existing ACL pointers that refer to the ACL referred to by *acl* shall continue to refer to the ACL.

23.4.11.3 Returns

Upon successful completion, the function shall return a pointer to the duplicate ACL. Otherwise, a value of (*acl_t*)**NULL** shall be returned and *errno* shall be set to indicate the error.

23.4.11.4 Errors

If any of the following conditions occur, the *acl_dup*() function shall return a value of (*acl_t*)**NULL** and set *errno* to the corresponding value:

[EINVAL] Argument *acl* does not point to a valid ACL.

[ENOMEM] The ACL working storage requires more memory than is allowed by the hardware or system-imposed memory management constraints. –

23.4.11.5 Cross-References

acl_free(), 23.4.12; *acl_get_entry*(), 23.4.14.

23.4.12 Release Memory Allocated to an ACL Data Object

Function: *acl_free*()

23.4.12.1 Synopsis

```
#include <sys/acl.h>

int acl_free (void *obj_p);
```

23.4.12.2 Description

The *acl_free*() function shall free any releasable memory currently allocated to the ACL data object identified by *obj_p*. The argument *obj_p* may identify an ACL, an ACL entry qualifier, or a pointer to a string allocated by one of the ACL functions.

If the item identified by *obj_p* is an *acl_t*, the *acl_t* and any existing descriptors that refer to parts of the ACL shall become undefined. If the item identified by *obj_p* is a string (*char**), then use of the *char** shall become undefined. If the item identified by *obj_p* is an ACL entry qualifier (*void**), then use of the *void** shall become undefined.

23.4.12.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

23.4.12.4 Errors

If any of the following conditions occur, the *acl_free*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The value of the *obj_p* argument is invalid.

23.4.12.5 Cross-References

acl_copy_int(), 23.4.6; *acl_create_entry*(), 23.4.7; *acl_dup*(), 23.4.11; *acl_from_text*(), 23.4.13; *acl_get_fd*(), 23.4.15; *acl_get_file*(), 23.4.16; *acl_get_permset*(), 23.4.17; *acl_init*(), 23.4.20.

23.4.13 Create an ACL from Text

Function: *acl_from_text*()

23.4.13.1 Synopsis

```
#include <sys/acl.h>

acl_t acl_from_text (const char *buf_p);
```

23.4.13.2 Description

The *acl_from_text*() function converts the text form of the ACL referred to by *buf_p* into the internal form of an ACL and returns a pointer to the working storage that contains the ACL. The *acl_from_text*() function shall accept as input the long text form and short text form of an ACL as described in sections 23.3.1. and 23.3.2.

This function may cause memory to be allocated. The caller should free any releaseable memory, when the new ACL is no longer required, by calling *acl_free*() with the (*void* ∗)*acl_t* as an argument.

Permissions within each ACL entry within the short text form of the ACL shall be specified only as absolute values.

23.4.13.3 Returns

Upon successful completion, the function shall return a pointer to the internal representation of the ACL in working storage. Otherwise, a value of (*acl_t*)**NULL** shall be returned and *errno* shall be set to indicate the error.

23.4.13.4 Errors

If any of the following conditions occur, the *acl_from_text*() function shall return a value of (*acl_t*)**NULL** and set *errno* to the corresponding value:

[EINVAL] Argument *buf_p* cannot be translated into an ACL.

[ENOMEM] The ACL working storage requires more memory than is allowed by the hardware or system-imposed memory management constraints.

23.4.13.5 Cross-References

acl_free(), 23.4.12; *acl_get_entry*(), 23.4.14; *acl_to_text*(), 23.4.27.

23.4.14 Get an ACL Entry

Function: *acl_get_entry*()

23.4.14.1 Synopsis

```
#include <sys/acl.h>

int acl_get_entry (acl_t acl,
                   int entry_id,
                   acl_entry_t *entry_p);
```

23.4.14.2 Description

The *acl_get_entry*() function shall obtain a descriptor for an ACL entry as specified by *entry_id* within the ACL indicated by argument *acl*. If the value of *entry_id* is ACL_FIRST_ENTRY, then the function shall return in *entry_p* a descriptor for the first ACL entry within *acl*. If a call is made to *acl_get_entry*() with *entry_id* set to ACL_NEXT_ENTRY when there has not been either an initial successful call to *acl_get_entry*(), or a previous successful call to *acl_get_entry*() following a call to *acl_calc_mask*(), *acl_copy_int*(), *acl_create_entry*(), *acl_delete_entry*(), *acl_dup*(), *acl_from_text*(), *acl_get_fd*(), *acl_get_file*(), *acl_set_fd*(), *acl_set_file*(), or *acl_valid*(), then the effect is unspecified.

Upon successful execution, the *acl_get_entry*() function shall return a descriptor for the ACL entry via *entry_p*.

Calls to *acl_get_entry*() shall not modify any ACL entries. Subsequent operations using the returned ACL entry descriptor shall operate on the ACL entry within the ACL in ACL working storage. The order of all existing entries in the ACL shall remain unchanged. Any existing ACL entry descriptors that refer to entries within the ACL shall continue to refer to those entries. Any existing ACL pointers that refer to the ACL referred to by *acl* shall continue to refer to the ACL.

23.4.14.3 Returns

If the function successfully obtains an ACL entry, the function shall return a value of 1. If the ACL has no ACL entries, the function shall return a value of zero. If the value of *entry_id* is ACL_NEXT_ENTRY and the last ACL entry in the ACL has already been returned by a previous call to *acl_get_entry*(), the function shall return a value of zero until a successful call with *entry_id* of ACL_FIRST_ENTRY is made. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

23.4.14.4 Errors

If any of the following conditions occur, the *acl_get_entry*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] Argument *acl* does not point to a valid ACL. Argument *entry_id* is neither ACL_NEXT_ENTRY nor ACL_FIRST_ENTRY.

23.4.14.5 Cross-References

acl_calc_mask(), 23.4.2; *acl_copy_int*(), 23.4.6; *acl_create_entry*(), 23.4.7; *acl_delete_entry*(), 23.4.9; *acl_dup*(), 23.4.11; *acl_from_text*(), 23.4.13; *acl_get_fd*(), 23.4.15; *acl_get_file*(), 23.4.16; *acl_init*(), 23.4.20; *acl_set_fd*(), 23.4.21; *acl_set_file*(), 23.4.22; *acl_valid*(), 23.4.28.

23.4.15 Get an ACL by File Descriptor

Function: *acl_get_fd*()

23.4.15.1 Synopsis

```
#include <sys/acl.h>

acl_t acl_get_fd (int fd);
```

23.4.15.2 Description

The *acl_get_fd*() function retrieves the access ACL for the object associated with the file descriptor, *fd*. If {_POSIX_MAC} is defined, then the process must have MAC read access to the object associated with *fd*. The ACL shall be placed into working storage and *acl_get_fd*() shall return a pointer to that storage.

This function may cause memory to be allocated. The caller should free any releaseable memory, when the new ACL is no longer required, by calling *acl_free*() with the (*void* ∗)*acl_t* as an argument.

The ACL in the working storage is an independent copy of the ACL associated with the object referred to by *fd*. The ACL in the working storage shall not participate in any access control decisions.

23.4.15.3 Returns

Upon successful completion, the function shall return a pointer to the ACL that was retrieved. Otherwise, a value of (*acl_t*)**NULL** shall be returned and *errno* shall be set to indicate the error.

23.4.15.4 Errors

If any of the following conditions occur, the *acl_get_fd*() function shall return a value of (*acl_t*)**NULL** and set *errno* to the corresponding value:

[EACCES] If {_POSIX_MAC} is defined, MAC read access to the object is denied.

[EBADF] The *fd* argument is not a valid file descriptor.

[ENOMEM] The ACL working storage requires more memory than is allowed by the hardware or system-imposed memory management constraints.

23.4.15.5 Cross-References

acl_free(), 23.4.12; *acl_get_entry*(), 23.4.14; *acl_get_file*(), 23.4.16; *acl_set_fd*(), 23.4.21.

23.4.16 Get an ACL by Filename

Function: *acl_get_file*()

23.4.16.1 Synopsis

```
#include <sys/acl.h>

acl_t acl_get_file (const char *path_p, acl_type_t type);
```

23.4.16.2 Description

The *acl_get_file*() function retrieves the access ACL associated with an object or the default ACL associated with a directory. The pathname for the object or directory is pointed to by the argument *path_p*. If {_POSIX_MAC} is defined, then the process must have MAC read access to *path_p*. The ACL shall be placed into working storage and *acl_get_file*() shall return a pointer to that storage.

This function may cause memory to be allocated. The caller should free any releaseable memory, when the new ACL is no longer required, by calling *acl_free*() with the (*void* *)*acl_t* as an argument.

The value of the argument *type* is used to indicate whether the access ACL or the default ACL associated with *path_p* is returned. If *type* is ACL_TYPE_ACCESS, then the access ACL shall be returned. If *type* is ACL_TYPE_DEFAULT, then the default ACL shall be returned. If *type* is ACL_TYPE_DEFAULT and no default

ACL is associated with *path_p*, then an ACL containing zero ACL entries shall be returned. If the argument *type* specifies a type of ACL that cannot be associated with *path_p*, then the function shall fail.

The ACL in the working storage is an independent copy of the ACL associated with the object referred to by *path_p*. The ACL in the working storage shall not participate in any access control decisions.

23.4.16.3 Returns

Upon successful completion, the function shall return a pointer to the ACL that was retrieved. Otherwise, a value of (*acl_t*)**NULL** shall be returned and *errno* shall be set to indicate the error.

23.4.16.4 Errors

If any of the following conditions occur, the *acl_get_file*() function shall return a value of (*acl_t*)**NULL** and set *errno* to the corresponding value:

[EACCES] Search permission is denied for a component of the path prefix or the object exists and the process does not have appropriate access rights.

If {_POSIX_MAC} is defined, MAC read access to the object is denied.

Argument *type* specifies a type of ACL that cannot be associated with *path_p*.

[EINVAL] Argument *type* is not ACL_TYPE_ACCESS, ACL_TYPE_DEFAULT, or a valid implementation-defined value.

[ENAMETOOLONG]
The length of the *path_p* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {POSIX_NO_TRUNC} is in effect.

[ENOENT] The named object does not exist or the *path_p* argument points to an empty string.

[ENOMEM] The ACL working storage requires more memory than is allowed by the hardware or system-imposed memory management constraints.

[ENOTDIR] A component of the path prefix is not a directory.

23.4.16.5 Cross-References

acl_delete_def_file(), 23.4.8; *acl_free*(), 23.4.12; *acl_get_entry*(), 23.4.14; *acl_get_fd*(), 23.4.15; *acl_set_file*(), 23.4.22.

23.4.17 Retrieve the Permission Set from an ACL Entry

Function: *acl_get_permset*()

23.4.17.1 Synopsis

```
#include <sys/acl.h>

int acl_get_permset (acl_entry_t entry_d, acl_permset_t *permset_p);
```

23.4.17.2 Description

The *acl_get_permset*() function returns via *permset_p* a descriptor to the permission set in the ACL entry indicated by *entry_d*. Subsequent operations using the returned permission set descriptor operate on the permission set within the ACL entry.

Any ACL entry descriptors that refer to the entry referred to by *entry_d* shall continue to refer to those entries.

23.4.17.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

23.4.17.4 Errors

If any of the following conditions occur, the *acl_get_permset*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry.

23.4.17.5 Cross-References

acl_add_perm(), 23.4.1; *acl_clear_perms*(), 23.4.3; *acl_delete_perm*(), 23.4.10; *acl_set_permset*(), 23.4.23.

23.4.18 Get ACL Entry Qualifier

Function: *acl_get_qualifier*()

23.4.18.1 Synopsis

```
#include <sys/acl.h>
void *acl_get_qualifier (acl_entry_t entry_d);
```

23.4.18.2 Description

The *acl_get_qualifier*() function retrieves the qualifier of the tag for the ACL entry indicated by the argument *entry_d* into working storage and returns a pointer to that storage.

If the value of the tag type in the ACL entry referred to by *entry_d* is ACL_USER, then the value returned by *acl_get_qualifier*() shall be a pointer to type *uid_t*. If the value of the tag type in the ACL entry referred to by *entry_d* is ACL_GROUP, then the value returned by *acl_get_qualifier*() shall be a pointer to type *gid_t*. If the value of the tag type in the ACL entry referred to by *entry_d* is implementation-defined, then the value returned by *acl_get_qualifier*() shall be a pointer to an implementation-defined type. If the value of the tag type in the ACL entry referred to by *entry_d* is ACL_UNDEFINED_TAG, ACL_USER_OBJ, ACL_GROUP_OBJ, ACL_OTHER, ACL_MASK, or an implementation-defined value for which a qualifier is not supported, then *acl_get_qualifier*() shall return a value of (*void* *)**NULL** and the function shall fail. Subsequent operations using the returned pointer shall operate on an independent copy of the qualifier in working storage.

This function may cause memory to be allocated. The caller should free any releaseable memory, when the new qualifier is no longer required, by calling *acl_free*() with the *void** as an argument.

The argument *entry_d* and any other ACL entry descriptors that refer to entries within the ACL containing the entry referred to by *entry_d* shall continue to refer to those entries. The order of all existing entries in the ACL containing the entry referred to by *entry_d* shall remain unchanged.

23.4.18.3 Returns

Upon successful completion, the function shall return a pointer to the tag qualifier that was retrieved into ACL working storage. Otherwise, a value of (*void* *)**NULL** shall be returned and *errno* shall be set to indicate the error.

23.4.18.4 Errors

If any of the following conditions occur, the *acl_get_qualifier*() function shall return a value of (*void* *)**NULL** and set *errno* to the corresponding value:

[EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry.

The value of the tag type in the ACL entry referenced by argument *entry_d* is not ACL_USER, ACL_GROUP, nor a valid implementation-defined value.

[ENOMEM] The value to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

23.4.18.5 Cross-References

acl_create_entry(), 23.4.7; *acl_free*(), 23.4.12; *acl_get_entry*(), 23.4.14; *acl_get_tag_type*(), 23.4.19; *acl_set_qualifier*(), 23.4.24; *acl_set_tag_type*(), 23.4.25.

23.4.19 Get ACL Entry Tag Type

Function: *acl_get_tag_type*()

23.4.19.1 Synopsis

```
#include <sys/acl.h>

int acl_get_tag_type (acl_entry_t entry_d, acl_tag_t *tag_type_p);
```

23.4.19.2 Description

The *acl_get_tag_type*() function returns the tag type for the ACL entry indicated by the argument *entry_d*. Upon successful completion, the location referred to by the argument *tag_type_p* shall be set to the tag type of the ACL entry referred to by *entry_d*.

The argument *entry_d* and any other ACL entry descriptors that refer to entries in the same ACL shall continue to refer to those entries. The order of all existing entries in the ACL shall remain unchanged.

23.4.19.3 Returns

Upon successful completion, the function shall set the location referred to by *tag_type_p* to the tag type that was retrieved and shall return a value of zero. Otherwise, a value of –1 shall be returned, the location referred to by *tag_type_p*, shall not be changed, and *errno* shall be set to indicate the error.

23.4.19.4 Errors

If any of the following conditions occur, the *acl_get_tag_type*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry.

23.4.19.5 Cross-References

acl_create_entry(), 23.4.7; *acl_get_entry*(), 23.4.14; *acl_get_qualifier*(), 23.4.18; *acl_set_qualifier*(), 23.4.24; *acl_set_tag_type*(), 23.4.25.

23.4.20 Initialize ACL Working Storage

Function: *acl_init*()

23.4.20.1 Synopsis

```
#include <sys/acl.h>

acl_t acl_init (int count);
```

23.4.20.2 Description

The *acl_init*() function allocates and initializes working storage for an ACL of at least *count* ACL entries. A pointer to the working storage is returned. The working storage allocated to contain the ACL is freed by a call to *acl_free*(). When the area is first allocated, it shall contain an ACL that contains no ACL entries. The initial state of any implementation-defined attributes of the ACL shall be implementation-defined.

This function may cause memory to be allocated. The caller should free any releaseable memory, when the new ACL is no longer required, by calling *acl_free*() with the (*void* ∗)*acl_t* as an argument.

23.4.20.3 Returns

Upon successful completion, this function shall return a pointer to the working storage. Otherwise, a value of (*acl_t*)**NULL** shall be returned and *errno* shall be set to indicate the error.

23.4.20.4 Errors

If any of the following conditions occur, the *acl_init*() function shall return a value of (*acl_t*)**NULL** and set *errno* to the corresponding value:

[EINVAL] The value of *count* is less than zero.

[ENOMEM] The *acl_t* to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

23.4.20.5 Cross-References

acl_free(), 23.4.12.

23.4.21 Set an ACL by File Descriptor

Function: *acl_set_fd*()

23.4.21.1 Synopsis

```
#include <sys/acl.h>
int acl_set_fd (int fd, acl_t acl);
```

23.4.21.2 Description

The *acl_set_fd*() function associates an access ACL with the object referred to by *fd*. The effective user ID of the process must match the owner of the object or the process must have appropriate privilege to set the access ACL on the object. If {_POSIX_CAP} is defined, then appropriate privilege shall include CAP_FOWNER. In addition, if {_POSIX_MAC} is defined, then the process must have MAC write access to the object.

The *acl_set_fd*() function will succeed only if the ACL referred to by *acl* is valid as defined by the *acl_valid*() function.

Upon successful completion, *acl_set_fd*() shall set the access ACL of the object referred to by argument *fd* to the ACL contained in the argument *acl*. The object's previous access ACL shall no longer be in effect. The invocation of this function may result in changes to the object's file permission bits. If *acl_set_fd*() is unsuccessful, the access ACL and the file permission bits of the object referred to by argument *fd* shall not be changed.

The ordering of entries within the ACL referred to by *acl* may be changed in some implementation-defined manner.

Existing ACL entry descriptors that refer to entries within the ACL referred to by *acl* shall continue to refer to those entries. Existing ACL pointers that refer to the ACL referred to by *acl* shall continue to refer to the ACL.

23.4.21.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

23.4.21.4 Errors

If any of the following conditions occur, the *acl_set_fd*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] If {_POSIX_MAC} is defined, MAC write access to the object is denied.

[EBADF] The *fd* argument is not a valid file descriptor.

[EINVAL] Argument *acl* does not point to a valid ACL. The function *acl_valid*() may be used to determine what errors are in the ACL.

fpathconf() indicates that {_POSIX_ACL_EXTENDED} is in effect for the object referenced by the argument *fd*, but the ACL has more entries than the value returned by *fpathconf*() for {_POSIX_ACL_PATH_MAX} for the object.

[ENOSPC] The directory or file system that would contain the new ACL cannot be extended or the file system is out of file allocation resources.

[EPERM] The process does not have appropriate privilege to perform the operation to set the ACL.

[EROFS] This function requires modification of a file system which is currently read-only.

23.4.21.5 Cross-References

acl_delete_def_file(), 23.4.8; *acl_get_entry*(), 23.4.14; *acl_get_fd*(), 23.4.15; *acl_get_file*(), 23.4.16; *acl_set_file*(), 23.4.22; *acl_valid*(), 23.4.28.

23.4.22 Set an ACL by Filename

Function: *acl_set_file*()

23.4.22.1 Synopsis

```
#include <sys/acl.h>

int acl_set_file (const char *path_p, acl_type_t type, acl_t acl);
```

23.4.22.2 Description

The *acl_set_file*() function associates an access ACL with an object or associates a default ACL with a directory. The pathname for the object or directory is pointed to by the argument *path_p*. The effective user ID of the process must match the owner of the object or the process must have appropriate privilege to set the access ACL or the default ACL on *path_p*. If {_POSIX_CAP} is defined, then appropriate privilege shall include CAP_FOWNER. In addition, if {_POSIX_MAC} is defined, then the process must have MAC write access to the object.

The value of the argument *type* is used to indicate whether the access ACL or the default ACL associated with *path_p* is being set. If *type* is ACL_TYPE_ACCESS, then the access ACL shall be set. If *type* is ACL_TYPE_DEFAULT, then the default ACL shall be set. If the argument *type* specifies a type of ACL that cannot be associated with *path_p*, then the function shall fail.

The *acl_set_file*() function will succeed only if the access or default ACL is valid as defined by the *acl_valid*() function.

If {_POSIX_ACL_EXTENDED} is not in effect for *path_p*, then the function shall fail if:

(1) the value of *type* is ACL_TYPE_DEFAULT, or

(2) the value of *type* is ACL_TYPE_ACCESS and *acl* is not a minimum ACL.

If the value of *type* is ACL_TYPE_ACCESS or ACL_TYPE_DEFAULT, then the function shall fail if the number of entries in *acl* is greater than the value *pathconf*() returns for {_POSIX_ACL_PATH_MAX} for *path_p*.

Upon successful completion, *acl_set_file*() shall set the access ACL or the default ACL, as indicated by *type_d*, of the object *path_p* to the ACL contained in the argument *acl*. The object's previous access ACL or default ACL, as indicated by *type_d*, shall no longer be in effect. The invocation of this function may result in changes to the object's file permission bits. If *acl_set_file*() is unsuccessful, the access ACL, the default ACL, and the file permission bits of the object referred to by argument *path_p* shall not be changed.

The ordering of entries within the ACL referred to by *acl* may be changed in some implementation-defined manner.

Existing ACL entry descriptors that refer to entries within the ACL referred to by *acl* shall continue to refer to those entries. Existing ACL pointers that refer to the ACL referred to by *acl* shall continue to refer to the ACL.

23.4.22.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error.

23.4.22.4 Errors

If any of the following conditions occur, the *acl_set_file*() function shall return –1 and set *errno* to the corresponding value:

[EACCES] Search permission is denied for a component of the path prefix or the object exists and the process does not have appropriate access rights.

If {_POSIX_MAC} is defined, MAC write access to *path_p* is denied.

Argument *type* specifies a type of ACL that cannot be associated with *path_p*.

[EINVAL] Argument *acl* does not point to a valid ACL. The function *acl_valid*() may be used to determine what errors are in the ACL.

Argument *type* is not ACL_TYPE_ACCESS, ACL_TYPE_DEFAULT, or a valid implementation-defined value.

pathconf() indicates that {_POSIX_ACL_EXTENDED} is in effect for the object referenced by the argument *path_p*, but the ACL has more entries than the value returned by *pathconf*() for {_POSIX_ACL_PATH_MAX} for the object.

[ENAMETOOLONG] The length of the *path_p* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {POSIX_NO_TRUNC} is in effect.

[ENOENT] The named object does not exist or the *path_p* argument points to an empty string.

[ENOSPC] The directory or file system that would contain the new ACL cannot be extended or the file system is out of file allocation resources.

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The process does not have appropriate privilege to perform the operation to set the ACL.

[EROFS] This function requires modification of a file system which is currently read-only.

23.4.22.5 Cross-References

acl_delete_def_file(), 23.4.8; *acl_get_entry*(), 23.4.14; *acl_get_fd*(), 23.4.15; *acl_get_file*(), 23.4.16; *acl_set_fd*(), 23.4.21; *acl_valid*(), 23.4.28.

23.4.23 Set the Permissions in an ACL Entry

Function: *acl_set_permset*()

23.4.23.1 Synopsis

```
#include <sys/acl.h>

int acl_set_permset (acl_entry_t entry_d, acl_permset_t permset_d);
```

23.4.23.2 Description

The *acl_set_permset*() function shall set the permissions of the ACL entry indicated by argument *entry_d* to the permissions contained in the argument *permset_d*.

Any ACL entry descriptors that refer to the entry containing the permission set referred to by *permset_d* shall continue to refer to those entries. Any ACL entry descriptors that refer to the entry referred to by *entry_d* shall continue to refer to that entry.

23.4.23.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

23.4.23.4 Errors

If any of the following conditions occur, the *acl_set_permset*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry.

Argument *permset_d* is not a valid descriptor for a permission set within an ACL entry.

Argument *permset_d* contains values which are not valid *acl_permset_t* values.

23.4.23.5 Cross-References

acl_add_perm(), 23.4.1; *acl_clear_perms*(), 23.4.3; *acl_delete_perm*(), 23.4.10; *acl_get_permset*(), 23.4.17.

23.4.24 Set ACL Entry Tag Qualifier

Function: *acl_set_qualifier*()

23.4.24.1 Synopsis

```
#include <sys/acl.h>

int acl_set_qualifier (acl_entry_t entry_d,
                       const void *tag_qualifier_p);
```

23.4.24.2 Description

The *acl_set_qualifier*() function shall set the qualifier of the tag for the ACL entry indicated by the argument *entry_d* to the value referred to by the argument *tag_qualifier_p*.

If the value of the tag type in the ACL entry referred to by *entry_d* is ACL_USER, then the value referred to by *tag_qualifier_p* shall be of type *uid_t*. If the value of the tag type in the ACL entry referred to by *entry_d* is ACL_GROUP, then the value referred to by *tag_qualifier_p* shall be of type *gid_t*. If the value of the tag type in the ACL entry referred to by *entry_d* is ACL_UNDEFINED_TAG, ACL_USER_OBJ, ACL_GROUP_OBJ, ACL_OTHER or ACL_MASK, then *acl_set_qualifier*() shall return an error. If the value of the tag type in the ACL entry referred to by *entry_d* is an implementation-defined value, then the value referred to by *tag_qualifier_p* shall be implementation-defined.

Any ACL entry descriptors that refer to the entry referred to by *entry_d* shall continue to refer to that entry. This function may cause memory to be allocated. The caller should free any releaseable memory, when the ACL is no longer required, by calling *acl_free*() with a pointer to the ACL as an argument.

23.4.24.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error.

23.4.24.4 Errors

If any of the following conditions occur, the *acl_set_qualifier*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry.

The tag type of the ACL entry referred to by the argument *entry_d* is not ACL_USER, ACL_GROUP, nor a valid implementation-defined value.

The value pointed to by the argument *tag_qualifier_p* is not valid.

[ENOMEM] The *acl_set_qualifier*() function is unable to allocate the memory required for an ACL tag qualifier.

23.4.24.5 Cross-References

acl_get_qualifier(), 23.4.18.

23.4.25 Set ACL Entry Tag Type

Function: *acl_set_tag_type*()

23.4.25.1 Synopsis

```
#include <sys/acl.h>

int acl_set_tag_type (acl_entry_t entry_d, acl_tag_t tag_type);
```

23.4.25.2 Description

The *acl_set_tag_type*() function shall set the tag type for the ACL entry referred to by the argument *entry_d* to the value of the argument *tag_type*.

Any ACL entry descriptors that refer to the entry referred to by *entry_d* shall continue to refer to that entry.

23.4.25.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error.

23.4.25.4 Errors

If any of the following conditions occur, the *acl_set_tag_type*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry.

Argument *tag_type* is not a valid tag type.

23.4.25.5 Cross-References

acl_get_tag_type(), 23.4.19.

23.4.26 Get the Size of an ACL

Function: *acl_size*()

23.4.26.1 Synopsis

```
#include <sys/acl.h>

ssize_t acl_size (acl_t acl);
```

23.4.26.2 Description

The *acl_size*() function shall return the size, in bytes, of the buffer required to hold the exportable, contiguous, persistent form of the ACL pointed to by argument *acl*, when converted by *acl_copy_ext*().

Any existing ACL entry descriptors that refer to entries in *acl* shall continue to refer to the same entries. Any existing ACL pointers that refer to the ACL referred to by *acl* shall continue to refer to the ACL. The order of ACL entries within *acl* shall remain unchanged.

23.4.26.3 Returns

Upon successful completion, the *acl_size*() function shall return the size in bytes of the contiguous, persistent form of the ACL. Otherwise, a value of (ssize_t) −1 shall be returned and *errno* shall be set to indicate the error.

23.4.26.4 Errors

If any of the following conditions occur, the *acl_size*() function shall return (ssize_t) −1 and set *errno* to the corresponding value:

[EINVAL] Argument *acl* does not point to a valid ACL.

23.4.26.5 Cross-References

acl_copy_ext(), 23.4.5.

23.4.27 Convert an ACL to Text

Function: *acl_to_text*()

23.4.27.1 Synopsis

```
#include <sys/acl.h>
char *acl_to_text (acl_t acl, ssize_t *len_p);
```

23.4.27.2 Description

The *acl_to_text*() function translates the ACL pointed to by argument *acl* into a **NULL** terminated character string. If the pointer *len_p* is not **NULL**, then the function shall return the length of the string (not including the **NULL** terminator) in the location pointed to by *len_p*. The format of the text string returned by *acl_to_text*() shall be the long text form defined in 23.3.1.

This function allocates any memory necessary to contain the string and returns a pointer to the string. The caller should free any releaseable memory, when the new string is no longer required, by calling *acl_free*() with the (*void* *)*char* as an argument.

Any existing ACL entry descriptors that refer to entries in *acl* shall continue to refer to the same entries. Any existing ACL pointers that refer to the ACL referred to by *acl* shall continue to refer to the ACL. The order of ACL entries within *acl* shall remain unchanged.

23.4.27.3 Returns

Upon successful completion, the function shall return a pointer to the long text form of an ACL. Otherwise, a value of (*char* *)**NULL** shall be returned and *errno* shall be set to indicate the error.

23.4.27.4 Errors

If any of the following conditions occur, the *acl_to_text*() function shall return a value of (*char* *)**NULL** and set *errno* to the corresponding value:

[EINVAL] Argument *acl* does not point to a valid ACL.

The ACL denoted by *acl* contains one or more improperly formed ACL entries, or for some other reason cannot be translated into a text form of an ACL.

[ENOMEM] The character string to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

23.4.27.5 Cross-References

acl_free(), 23.4.12; *acl_from_text*(). 23.4.13.

23.4.28 Validate an ACL

Function: *acl_valid*()

23.4.28.1 Synopsis

```
#include <sys/acl.h>

int acl_valid (acl_t acl);
```

23.4.28.2 Description

The *acl_valid*() function checks the ACL referred to by the argument *acl* for validity.

The three required entries (ACL_USER_OBJ, ACL_GROUP_OBJ, and ACL_OTHER) shall exist exactly once in the ACL. If the ACL contains any ACL_USER, ACL_GROUP, or any implementation-defined entries in the file group class, then one ACL_MASK entry shall also be required. The ACL shall contain at most one ACL_MASK entry.

The qualifier field shall be unique among all entries of the same POSIX.1e ACL facility defined tag type. The tag type field shall contain valid values including any implementation-defined values. Validation of the values of the qualifier field is implementation-defined.

The ordering of entries within the ACL referred to by *acl* may be changed in some implementation-defined manner.

Existing ACL entry descriptors that refer to entries within the ACL referred to by *acl* shall continue to refer to those entries. Existing ACL pointers that refer to the ACL referred to by *acl* shall continue to refer to the ACL.

If multiple errors occur in the ACL, the order of detection of the errors and, as a result, the ACL entry descriptor returned by *acl_valid*() shall be implementation-defined.

23.4.28.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

23.4.28.4 Errors

If any of the following conditions occur, the *acl_valid*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] Argument *acl* does not point to a valid ACL.

One or more of the required ACL entries is not present in *acl*.

The ACL contains entries that are not unique. –

23.4.28.5 Cross-References

acl_get_entry(), 23.4.14; *acl_get_fd*(), 23.4.15; *acl_get_file*(), 23.4.16; *acl_init*(), 23.4.20; *acl_set_fd*(), 23.4.21; *acl_set_file*(), 23.4.22.

Section 24: Audit

24.1 General Overview

There are four major functional components of the POSIX.1 audit interface specification:

(1) Interfaces for a conforming application to construct and write records to an audit log and control the auditing of the current process

(2) Interfaces for reading an audit log and manipulating audit records

(3) The definition of a standard set of events, based on the POSIX.1 function interfaces, that shall be reportable in conforming implementations

(4) The definition of the contents of audit records.

This standard defines which interfaces require an appropriate privilege, and the relevant capabilities if the POSIX capability option is in use.

Support for the interfaces defined in this section is optional but shall be provided if the symbol {_POSIX_AUD} is defined.

24.1.1 Audit Logs

The standard views the destination of audit records that are recorded, and the source of records read by an audit post-processing application, as an "audit log". Audit logs map to the POSIX abstraction of a "file": that is, POSIX file interfaces such as *open*() can generally be used to gain access to audit logs, subject to the access controls of the system.

As viewed at the POSIX interface, a log contains a sequence of audit records; interfaces are provided to write records to a log, and to read records from it.

A conforming implementation shall support a "system audit log": that is, a log that is the destination of system-generated audit records (e.g. reporting on use of security-relevant POSIX.1 interfaces), and of application-generated records that an application sends to that log. The system audit log may correspond to different files at different times. An application that sends records to the system audit log does not have to be able to *open*() the corresponding file; instead an appropriate privilege is required. This protects the integrity of the system audit log. A post-processing application that reads records from the system audit log can gain access to the log through *open*() of the file that currently corresponds to it.

The internal format of audit logs, and of the records within them, is unspecified (because of this, the POSIX *read*() and *write*() interfaces should not generally be used to access audit logs).

24.1.2 Audit Records

Audit records describe events; that is, there is a correspondence between some actual event that occurred and the audit record reporting it. An audit record provides a description of one event. With an audit record, a report is given of what happened, who will be held accountable for it, what it affected, and when.

Audit records are generated in two ways:

- By a system conforming to the POSIX.1 audit option, to report on use of its security relevant interfaces. This is known as *system auditing*, and the records are known as *system-generated records*.
- By an application with the appropriate privilege, to report on its own activities. These are known as *application-generated records*.

This standard does not specify the method by which audit records are written to the audit log nor does it specify the internal format in which audit records are stored. The standard specifies only the interfaces by which *application-generated records* are delivered to the system and by which system- and application-generated records are reported to a conforming application.

Note that the standard does not specify the manner by which *system-generated records* are delivered to the system audit log; this is left up to the implementation.

An audit record that is generated by an application, or an auditable event that occurs in a system conforming to the POSIX.1 audit option, may or may not actually be reported to a conforming application. This standard specifies that these events shall be reportable on a conforming implementation, but not that they always be reported. The record will be reported only if {_POSIX_AUD} was defined at the time the event occurred and was defined at the time the event completed. The results are indeterminate if {_POSIX_AUD} was not defined through the lifetime of the event. There may also be other implementation-specific controls on the events that are actually reported (in particular, a conforming implementation may have some configurable selectivity of the events that are reported).

24.1.2.1 Audit Record Contents

Although there is no requirement on how the system stores an audit record, logically it appears to the post-processing application, and to a self-auditing application constructing a record, to have several parts:

- one or more headers, see below
- one or more sets of subject attributes, describing the process(es) that caused the event to be reported

- zero or more sets of event-specific data
- zero or more sets of object attributes, describing objects affected by the event.

Records are required to have at least one header and set of subject attributes. Conforming implementations and self-auditing applications may add further parts, of any type; the contents of each of the required parts is also extensible.

A post-processing application can obtain a descriptor to each of the parts, and using these descriptors can then obtain the contents of each part. An audit record header contains, amongst other things, the event type, time and result. There is also a record format indicator, currently limited to defining that the data in the record is in the format used by the current system. The header also contains a version number, identifying the version of this standard to which the record content conforms. Post-processing applications should examine this value to ensure that the version is one for which they can process the information in the record.

The event type in the header defines the minimum set of information found in the record. This standard specifies the required content for POSIX.1 events that are required to be auditable: that is, the content of the event-specific and object parts of the record; the event type for these system-generated events is an integer. Implementations may define additional content for such events, and additional events and their content. Self-auditing applications may add further events, with application-specific types and contents; the event type for these application-generated events is a text string.

To ensure that users can be made individually accountable for their security-relevant actions, an "audit identifier", or *audit ID*, that an implementation can use to uniquely identify each accountable user, is included in the header of each record. If the record is related to an event that is not associated with any individual user (e.g., events recorded before a user has completed authentication, or events from daemons), the implementation may report a null audit ID for that record.

24.1.3 Audit Interfaces

Self-auditing applications need a standard means of constructing records and adding them into an audit log. Additionally, applications having the appropriate privilege may need to suspend system auditing of their actions. However, the request to suspend system auditing is advisory and may be rejected by the implementation.

Portable audit post-processing utilities need a standard means to access records in an audit log and a standard means to analyze the content of the records.

Several groups of functions are defined for use by portable applications. These functions are used to:

(1) Construct audit records

(2) Write audit records

(3) Control system auditing of the current process

(4) Read audit records

(5) Analyze an audit record

(6) Save audit records in user-managed store and return them to system managed store.

The following sections provide an overview of those functions.

24.1.3.1 Accessing an Audit Log

Audit logs are accessed via the POSIX.1 *open*() and *close*() functions. The system audit log is also written directly by the *aud_write*() function (see below).

24.1.3.2 Constructing Audit Records

Functions are provided to get access to an unused audit record in working store, and to duplicate an existing record:

aud_init_record() Get access to an unused audit record in working store.

aud_dup_record() Create a duplicate of an existing audit record in working store.

Various other functions manipulate audit records. New sections can be added to an audit record:

aud_put_hdr() Add an empty header to an audit record

aud_put_subj() Add an empty set of subject attributes to an audit record

aud_put_event() Add an empty set of event-specific data to an audit record

aud_put_obj() Add an empty set of object attributes to an audit record

And data can be added to each type of section:

aud_put_hdr_info() Add a data item to a header in an audit record

aud_put_subj_info() Add a data item to a set of subject attributes in an audit record

aud_put_event_info() Add a data item to a set of event-specific data in an audit record

aud_put_obj_info() Add a data item to a set of object attributes in an audit record.

Data items can also be deleted from each type of section:

aud_delete_hdr_info() Delete a data item from a header in an audit record

aud_delete_subj_info() Delete a data item from a set of subject attributes in an audit record *aud_delete_event_info*() Delete a data item from a set of event-specific data in an audit record

aud_delete_obj_info() Delete a data item from a set of object attributes in an audit record.

And whole sections can be deleted too:

aud_delete_hdr() Delete a header from an audit record

aud_delete_subj() Delete a set of subject attributes from an audit record

aud_delete_event() Delete a set of event-specific data from an audit record

aud_delete_obj() Delete a set of object attributes from an audit record.

A function is provided to obtain the audit ID of the user accountable for the actions of a specified process:

aud_get_id() Get the audit ID of a process with a specified process ID. This allows, for example, a server process to include the audit ID of a client in a record it generates.

A function is provided to check the validity of an audit record:

aud_valid() Validates an audit record by checking for, at least, a valid header.

24.1.3.3 Writing Audit Records

A single function is provided to write a record to an audit log:

aud_write() When a program wants to write a record to an audit log, it calls *aud_write*(). The system then adds the record to the log. This could be used by a self-auditing application that has constructed the record, or by an audit post-processing application that has read the record from an audit log and now wants to preserve it in another log for later processing. Appropriate privilege is required to use this interface to write to the system audit log.

24.1.3.4 Controlling System Auditing

A single function is provided to allow a self-auditing application to control system auditing of its operations:

aud_switch() Suspend or resume system auditing of the current process, or query the current state of system auditing for the current process. The system may or may not actually suspend (either partially or completely) its auditing of the process, depending on the implementation-specific audit policy currently in use. Appropriate privilege is required to use this interface.

24.1.3.5 Reading Audit Records

A single function is provided to read an audit record from an audit log into system managed store.

aud_read()	Read the next record from the audit log and return a descriptor to it in working store. The descriptor can then be used as an argument to any of the audit functions that manipulate audit records.

24.1.3.6 Analyzing an Audit Record

Functions are provided to get descriptors for the various sections of an audit record, and to get data items from within each type of section:

aud_get_hdr()	Get the descriptor for a header from an audit record.
aud_get_hdr_info()	Get an item from within a header of an audit record.
aud_get_subj()	Get the descriptor for a subject attribute set from an audit record.
aud_get_subj_info()	Get an item from within a subject attribute set from an audit record.
aud_get_event()	Get the descriptor for a set of event-specific data from an audit record.
aud_get_event_info()	Get an item from within a set of event-specific data from an audit record.
aud_get_obj()	Get a descriptor for an object attribute set from an audit record.
aud_get_obj_info()	Get an item from within an object attribute set in an audit record.

To allow a post-processing application to interact with an audit administrator, either to display records or to obtain record selection criteria from the administrator, interfaces are provided to convert a record to text, to convert between the internal and human-readable forms of event types and audit IDs, and to find out all the system event types reportable in the audit log:

aud_rec_to_text()	Convert an entire audit record into human-readable text.
aud_evid_to_text()	Map a numeric identifier for a system audit event to a text string.
aud_evid_from_text()	Map a text string, representing an system audit event type, to a numeric audit event.
aud_id_to_text()	Map an audit ID to text identifying an individual user.
aud_id_from_text()	Map text identifying an individual user to an audit ID.
aud_get_all_evid()	Get a list of all system generated audit event types currently reportable on the system. This interface retrieves both

POSIX.1 and implementation-specific event types.

24.1.3.7 Storing Audit Records

A pair of functions are provided for placing audit records in user-managed space and conversely, returning audit records to system-managed space; for the former, a function is provided that determines how much space is needed. This facility provides applications with the ability to save selected records outside an audit log for later processing.

aud_copy_ext() The *aud_copy_ext*() function is provided to convert the record to a “byte-copyable” format in user-managed space.

aud_copy_int() The *aud_copy_int*() function is provided to convert the record from a “byte-copyable” format in user-managed space into system-dependent, internal format in system-managed space.

aud_size() Return the size of user-managed space needed to hold a record.

Note that it is also possible to transfer an audit record from one log to another, without using user-managed space, by use of *aud_read*() and *aud_write*().

Finally, an interface is provided to allow an application to free any memory allocated by the various audit functions:

aud_free() Many of the above interfaces may allocate memory space. The *aud_free*() interface frees all the releasable space.

24.1.4 Summary of POSIX.1 System Interface Impact

When {_POSIX_AUD} is defined, there is no impact on the interface syntax of any POSIX.1 function, nor on the function semantics defined by POSIX.1. However, use of some POSIX.1 functions may cause audit records to be reported, see section 24.2.1.1, below.

24.2 Audit Record Content

Section 24.1.2.1, defines the overall structure of an audit record, viewed through these interfaces, as consisting of headers, subject attribute sets, sets of event-specific data items, and object attribute sets. This section specifies the minimum set of event types which shall be reportable in a conforming implementation, and for each of these event types defines the minimum required contents of the set of event-specific items for the event and the minimum required object attribute sets.

24.2.1 Auditable Interfaces and Event Types

This section defines the minimum set of audit event types that shall be reportable by a conforming system.

Two kinds of auditing are defined. First there is auditing, by the system, of operations performed by programs at the system interface level. Second there is auditing by applications of their own operations.

24.2.1.1 Auditing at the System Interface

The following interfaces, which are derived from POSIX.1 and the POSIX.1e options, are defined as the minimum set of system interface functions that shall be reportable on a conforming implementation. For each interface, a corresponding POSIX.1e audit event is defined. For each defined event, a numeric constant uniquely identifying the audit event is defined in the <sys/audit.h> header. For all the interfaces except *fork*(), a single audit record shall be reportable for each occasion that the interface is used.

If {_POSIX_AUD} is defined, the following interfaces shall be auditable:

Table 24-1 – Interfaces and Corresponding Audit Events

Interface	Event Type
aud_switch()	AUD_AET_AUD_SWITCH
aud_write()	AUD_AET_AUD_WRITE
chdir()	AUD_AET_CHDIR
chmod()	AUD_AET_CHMOD
chown()	AUD_AET_CHOWN
creat()	AUD_AET_CREAT
dup()	AUD_AET_DUP
dup2()	AUD_AET_DUP
exec()	AUD_AET_EXEC
execl()	AUD_AET_EXEC
execlp()	AUD_AET_EXEC
execv()	AUD_AET_EXEC
execvp()	AUD_AET_EXEC
execle()	AUD_AET_EXEC
execve()	AUD_AET_EXEC
_exit()	AUD_AET_EXIT
fork()	AUD_AET_FORK
kill()	AUD_AET_KILL
link()	AUD_AET_LINK
mkdir()	AUD_AET_MKDIR
mkfifo()	AUD_AET_MKFIFO
open()	AUD_AET_OPEN
opendir()	AUD_AET_OPEN
pipe()	AUD_AET_PIPE
rename()	AUD_AET_RENAME
rmdir()	AUD_AET_RMDIR
setgid()	AUD_AET_SETGID
setuid()	AUD_AET_SETUID
unlink()	AUD_AET_UNLINK
utime()	AUD_AET_UTIME

The *aud_write*() function is auditable only when an attempt to write to the system audit log fails.

The *fcntl*() function when used with command *F_DUPFD* also generates audit events of type *AUD_AET_DUP*.

If {_POSIX_ACL} is defined, the following interfaces shall be auditable:

Interface	Event Type
acl_delete_def_file()	AUD_AET_ACL_DELETE_DEF_FILE
acl_set_fd()	AUD_AET_ACL_SET_FD
acl_set_file()	AUD_AET_ACL_SET_FILE

If {_POSIX_CAP} is defined, the following interfaces shall be auditable:

Interface	Event Type
cap_set_fd()	AUD_AET_CAP_SET_FD
cap_set_file()	AUD_AET_CAP_SET_FILE
cap_set_proc()	AUD_AET_CAP_SET_PROC

If {_POSIX_INF} is defined, the following interfaces shall be auditable:

Interface	Event Type
inf_set_fd()	AUD_AET_INF_SET_FD
inf_set_file()	AUD_AET_INF_SET_FILE
inf_set_proc()	AUD_AET_INF_SET_PROC

If {_POSIX_MAC} is defined, the following interfaces shall be auditable:

Interface	Event Type
mac_set_fd()	AUD_AET_MAC_SET_FD
mac_set_file()	AUD_AET_MAC_SET_FILE
mac_set_proc()	AUD_AET_MAC_SET_PROC

Event types recording use of other system interfaces shall be implementation-defined; a complete set of such events shall be obtainable through the *aud_get_all_evid*() interface.

24.2.1.2 Auditing by Applications

No specific types are defined for auditing by applications. The event types used by applications are character strings (to reduce the chances of different applications using the same types and ensure they do not clash with the integer event types used for system-generated events) and applications are free to add their own audit event types. Applications which generate their own audit records will use the *aud_write*() function passing the event type in the record header.

24.2.2 Audit Event Types and Record Content

This clause defines the minimum required content of audit records for each of the standard event types. The required contents of the header is the same for all records, and is defined in *aud_get_hdr_info*(); the required content of the set of subject attributes is similar for all records, and is defined in *aud_get_subj_info*(); the required contents of a set of object attributes is defined in *aud_get_obj_info*(). This section defines the required minimum content for the set of items specific to each event, and the required minimum object attribute sets for each event. A conforming implementation may include additional items in the required header, set of subject attributes, set of event-specific items, and object attribute sets, or may add additional sets, but the required content must be reported before these implementation-specific additions.

A header, subject attribute set, set of event-specific items, and object attribute set from an audit record are not C-language structures; each is a separate logical section within the record, with components accessed using the *aud_get_*_info*() interfaces described below. An argument *item_id* of these interfaces identifies the component to access; a value for this argument for each component is defined in the tables below.

Unless otherwise specified, event-specific data contains the argument values requested for the operation. If the argument is not available (for example, if the caller supplied a **NULL** or invalid pointer for a pathname), the *aud_get_event_info*() function shall return an *aud_info_t* structure with a zero *len* member. *Pathname* values reported as arguments may be the exact values passed as arguments, or may be expanded by the implementation to full pathnames.

24.2.2.1 AUD_AET_ACL_DELETE_DEF_FILE

This event will be encountered only if {_POSIX_ACL} was defined when the audit log was generated.

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_ACL_DELETE_DEF_FILE event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Pathname	AUD_PATHNAME

The *Pathname* contains the value passed as an argument to the *acl_delete_def_file*() function.

If the call succeeded a set of object attributes shall also be available from the record, describing the object affected; if an ACL is reported in the set of object attributes it shall contain the ACL before the event. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.2 AUD_AET_ACL_SET_FD

This event will be encountered only if {_POSIX_ACL} was defined when the audit log was generated.

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_ACL_SET_FD event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_INT	File desc	AUD_FILE_ID
AUD_TYPE_ACL_TYPE	ACL type	AUD_ACL_TYPE
AUD_TYPE_ACL	ACL	AUD_ACL

The *File desc*, *ACL type*, and *ACL* contain the values passed as arguments to the *acl_set_fd*() function.

If the call succeeded a set of object attributes shall also be available from the record, describing the object affected; if an ACL is reported in the set of object attributes it shall contain the ACL before the event. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.3 AUD_AET_ACL_SET_FILE

This event will be encountered only if {_POSIX_ACL} was defined when the audit log was generated.

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_ACL_SET_FILE event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Pathname	AUD_PATHNAME
AUD_TYPE_ACL_TYPE	ACL type	AUD_ACL_TYPE
AUD_TYPE_ACL	ACL	AUD_ACL

The *Pathname*, *ACL type*, and *ACL* contain the values passed as arguments to the *acl_set_file*() function.

If the call succeeded a set of object attributes shall also be available from the record, describing the object affected; if an ACL is reported in the set of object attributes it shall contain the ACL before the event. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.4 AUD_AET_AUD_SWITCH

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_AUD_SWITCH event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_AUD_STATE	Audit state	AUD_AUDIT_STATE

The *Audit state* contains the value passed as an argument to the *aud_switch*() function: AUD_STATE_ON, AUD_STATE_OFF or AUD_STATE_QUERY.

24.2.2.5 AUD_AET_AUD_WRITE

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_AUD_WRITE event are not required to report any event-specific data. This event is required to be reportable only if an attempt to use *aud_write*(), to write a record to the system audit log, fails (e.g. due to lack of appropriate privilege). The header of the record shall give details of the attempt to use *aud_write*(), and the set of subject attributes shall relate to the caller of *aud_write*(); that is, the record is not required to contain data from the record that the application tried to write to the system audit log.

24.2.2.6 AUD_AET_CAP_SET_FD

This event will be encountered only if {_POSIX_CAP} was defined when the audit log was generated.

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_CAP_SET_FD event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_INT	File desc	AUD_FILE_ID
AUD_TYPE_CAP	Capability state	AUD_CAP

The *File desc* and *Capability state* contain the values passed as arguments to the *cap_set_fd*() function.

If the call succeeded a set of object attributes shall also be available from the record, describing the object affected; if a file capability state is reported in the set of object attributes it shall contain the file capability state before the event. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.7 AUD_AET_CAP_SET_FILE

This event will be encountered only if {_POSIX_CAP} was defined when the audit log was generated.

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_CAP_SET_FILE event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Pathname	AUD_PATHNAME
AUD_TYPE_CAP	Capability state	AUD_CAP

The *Pathname* and *Capability state* contain the values passed as arguments to the *cap_set_file*() function.

If the call succeeded a set of object attributes shall also be available from the record, describing the object affected. If a file capability state is reported in the set of object attributes it shall contain the file capability state before the event. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.8 AUD_AET_CAP_SET_PROC

This event will be encountered only if {_POSIX_CAP} was defined when the audit log was generated.

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_CAP_SET_PROC event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_CAP	Capability state	AUD_CAP

The *Capability state* records the value passed as an argument to the *cap_set_proc*() function. If a capability state is reported in the set of subject attributes in the record, this shall record the process capability state of the process before the event.

24.2.2.9 AUD_AET_CHDIR

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_CHDIR event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Pathname	AUD_PATHNAME

The *Pathname* contains the value passed as an argument to the *chdir*() function.

24.2.2.10 AUD_AET_CHMOD

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_CHMOD event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Pathname	AUD_PATHNAME
AUD_TYPE_MODE	Mode	AUD_MODE

The *Pathname* and *Mode* contain the values passed as arguments to the *chmod*() function.

If the call succeeded a set of object attributes shall also be available from the record, describing the object affected; if a mode is reported in the set of object attributes it shall contain the mode before the event. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.11 AUD_AET_CHOWN

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_CHOWN event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Pathname	AUD_PATHNAME
AUD_TYPE_UID	Owner	AUD_UID
AUD_TYPE_GID	Group	AUD_GID

The *Pathname*, *Owner*, and *Group* contain the values passed as arguments to the *chown*() function.

If the call succeeded a set of object attributes shall also be available from the record, describing the object affected; if an owner and group are reported in the set of object attributes they shall contain the object owner and group before the event. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.12 AUD_AET_CREAT

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_CREAT event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Pathname	AUD_PATHNAME
AUD_TYPE_MODE	Mode	AUD_MODE
AUD_TYPE_INT	Return value (file descriptor)	AUD_RETURN_ID

The *Pathname* and *Mode* contain the values passed as arguments to the *creat*() function.

If the call succeeded a set of object attributes shall also be available from the record, describing the object created. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.13 AUD_AET_DUP

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_DUP event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_INT	File descriptor	AUD_FILE_ID
AUD_TYPE_INT	Return value (file descriptor)	AUD_RETURN_ID

This event is recorded for any of the functions *dup*(), *dup2*(), or *fcntl*() with command *F_DUPFD*.

The *File descriptor* contains the value passed as the first argument to the function.

24.2.2.14 AUD_AET_EXEC

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_EXEC event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Pathname	AUD_PATHNAME
AUD_TYPE_STRING_ARRAY	Command-args (Records *arg0...argn*)	AUD_CMD_ARGS
AUD_TYPE_STRING_ARRAY	Env_args (Records *envp*)	AUD_ENVP
AUD_TYPE_UID	Effective UID	AUD_UID_ID
AUD_TYPE_GID	Effective GID	AUD_GID_ID
AUD_TYPE_CAP	Process capability state	AUD_CAP

This event is recorded for any of the functions *exec*(), *execl*(), *execlp*(), *execv*(), *execvp*(), *execle*(), or *execve*().

The *Pathname* contains the value passed as an argument to the function.

An implementation may choose not to report the value of *Command_args*. If this is the case, or the arrays pointed to by the argument contained any invalid pointers, the *aud_get_event_info*() function shall return an *aud_info_t* with a zero *aud_info_length* member.

For calls other than *execle*() and *execve*(), the *aud_get_event_info*() function may return an *aud_info_t* with a zero *aud_info_length* member for *Env_args*. For *execle*() and *execve*() an implementation may choose not to report the value of *Env_args*. If this is the case, or the arrays pointed to by the arguments contained any invalid pointers, the *aud_get_event_info*() function shall return an *aud_info_t* with a zero *aud_info_length* member.

The *Effective UID* and *GID* are those in effect after the call to *exec*(). The values previous to the call to *exec*() are reportable in the record's subject attributes. The *aud_info_length* member of the *aud_info_t* reporting these values may be zero length if the effective UID and GID of the process are the same before and after the *exec*().

If {_POSIX_CAP} was in effect when the record was generated, then the *process capability state* in the event-specific data shall record the state at the end of the call, and if a process capability state is reported in the subject attributes in the audit record, it shall be that at the start of the call. If {_POSIX_CAP} was not in effect when the record was generated, the *aud_get_event_info*() function shall return an *aud_info_t* with a zero *aud_info_length* member for the process capability state.

If the call succeeded a set of object attributes shall also be available from the record, describing the object executed. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.15 AUD_AET_EXIT

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_EXIT event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_INT	Exit code	AUD_EXIT_CODE

The *Exit code* contains the value passed as an argument to the *_exit*() function.

24.2.2.16 AUD_AET_FORK

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_FORK event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_PID	Return value	AUD_RETURN_ID

The audit record shall be reportable on behalf of the parent, when the *Return value* shall be the child's process ID, thus the parent's process ID is recorded in the record header, and the child's is the return value. A conforming implementation may also report a record for the child process; in this case the *Return value* shall be zero. No events that are reported for the child shall be reported before the parent's AUD_AET_FORK record.

24.2.2.17 AUD_AET_INF_SET_FD

This event will be encountered only if {_POSIX_INF} was defined when the audit log was generated.

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_INF_SET_FD event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_INT	File desc	AUD_FILE_ID
AUD_TYPE_INF	Label	AUD_INF_LBL

The *File desc* and *Label* contain the values passed as arguments to the *inf_set_fd*() function.

If the call succeeded a set of object attributes shall also be available from the record, describing the object affected; if an information label is reported in the set of object attributes it shall contain the information label before the event. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.18 AUD_AET_INF_SET_FILE

This event will be encountered only if {_POSIX_INF} was defined when the audit log was generated.

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_INF_SET_FILE event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Pathname	AUD_PATHNAME
AUD_TYPE_INF	Label	AUD_INF_LBL

The *Pathname* and *Label* contain the values passed as arguments to the *inf_set_file*() function.

If the call succeeded a set of object attributes shall also be available from the record, describing the object affected; if an information label is reported in the set of object attributes it shall contain the information label before the event. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.19 AUD_AET_INF_SET_PROC

This event will be encountered only if {_POSIX_INF} was defined when the audit log was generated.

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_INF_SET_PROC event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_INF	Label	AUD_INF_LBL

The *Label* contains the value passed as an argument to the *inf_set_proc*() function. If an information label is reported in the record header it shall contain the process's information label before the event.

24.2.2.20 AUD_AET_KILL

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_KILL event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_PID	Pid	AUD_PID
AUD_TYPE_INT	Signal Number	AUD_SIG

The *Pid* and *Signal Number* shall record the values passed as arguments to the *kill*() function.

If the call succeeded, or if the call failed because of access control restrictions, sets of object attributes shall also be available from the record, one describing each object to which the signal was directed. In addition, following the content normally required from each set of object attributes, there shall also be available from each an item:

Type	Description	item_id
AUD_TYPE_AUD_STATUS	The audit status of the event	AUD_STATUS

recording whether the signal was successfully sent to that object. If the call failed for reasons other than access control, it is not defined whether any sets of object attributes are available.

24.2.2.21 AUD_AET_LINK

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_LINK event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Path1	AUD_PATHNAME
AUD_TYPE_STRING	Path2	AUD_LINKNAME

The *Path1* and *Path2* contain the values passed as arguments to the *link*() function. *Path1* contains the pathname of the existing file, *Path2* contains the pathname of the new directory entry to be created.

If the call succeeded a set of object attributes shall also be available from the record, describing the file to which the link is made. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.22 AUD_AET_MAC_SET_FD

This event will be encountered only if {_POSIX_MAC} was defined when the audit log was generated.

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_MAC_SET_FD event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_INT	File desc	AUD_FILE_ID
AUD_TYPE_MAC	Label	AUD_MAC_LBL

The *File desc* and *Label* contain the values passed as arguments to the *mac_set_fd*() call.

If the call succeeded a set of object attributes shall also be available from the record, describing the object affected; if a MAC label is reported in the set of object attributes it shall contain the MAC label before the event. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.23 AUD_AET_MAC_SET_FILE

This event will be encountered only if {_POSIX_MAC} was defined when the audit log was generated.

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_MAC_SET_FILE event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Pathname	AUD_PATHNAME
AUD_TYPE_MAC	Label	AUD_MAC_LBL

The *Pathname* and *Label* contain the values passed as arguments to the *mac_set_file*() call.

If the call succeeded a set of object attributes shall also be available from the record, describing the object affected; if a MAC label is reported in the set of object attributes it shall contain the MAC label before the event. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.24 AUD_AET_MAC_SET_PROC

This event will be encountered only if {_POSIX_MAC} was defined when the audit log was generated.

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_MAC_SET_PROC event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_MAC	Label	AUD_MAC_LBL

The *Label* contains the value passed as an argument to the *mac_set_proc*() function. If a MAC label is reported in the record header it shall contain the process

MAC label before the event.

24.2.2.25 AUD_AET_MKDIR

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_MKDIR event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Pathname	AUD_PATHNAME
AUD_TYPE_MODE	Mode	AUD_MODE

The *Pathname* and *Mode* contain the values passed as arguments to the *mkdir*() function.

If the call succeeded a set of object attributes shall also be available from the record, describing the object created. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.26 AUD_AET_MKFIFO

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_MKFIFO event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Pathname	AUD_PATHNAME
AUD_TYPE_MODE	Mode	AUD_MODE

The *Pathname* and *Mode* contain the values passed as arguments to the *mkfifo*() function.

If the call succeeded a set of object attributes shall also be available from the record, describing the object created. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.27 AUD_AET_OPEN

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_OPEN event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Pathname	AUD_PATHNAME
AUD_TYPE_INT	Oflag	AUD_OFLAG
AUD_TYPE_MODE	Mode	AUD_MODE
AUD_TYPE_INT	Return value (file descriptor)	AUD_RETURN_ID

The *Pathname*, *Oflag* and *Mode* contain the values passed as arguments to the *open*() function. If the O_CREAT flag is not set in *Oflag*, the *aud_get_event_info*() function shall return an *aud_info_t* with a zero *aud_info_length* field if an attempt is made to read *Mode*.

If the call succeeded a set of object attributes shall also be available from the record, describing the object opened. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.28 AUD_AET_PIPE

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_PIPE event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_INT	Read file descriptor	AUD_RD_FILE_ID
AUD_TYPE_INT	Write file descriptor	AUD_WR_FILE_ID

If the call succeeded, the *File descriptors* shall contain the values returned to the caller. Otherwise, the *aud_get_event_info*() function shall return *aud_info_t* structures with zero *aud_info_length* members for these items.

24.2.2.29 AUD_AET_RENAME

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_RENAME event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Old pathname	AUD_OLD_PATHNAME
AUD_TYPE_STRING	New pathname	AUD_NEW_PATHNAME

The *pathnames* contain the values passed as arguments to the *rename*() call.

If the call succeeded a set of object attributes shall also be available from the record, describing the object renamed; the name reported in the set of object attributes shall contain the name before the event. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.30 AUD_AET_RMDIR

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_RMDIR event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Pathname	AUD_PATHNAME

The *pathname* contains the value passed as an argument to the *rmdir*() call.

If the call succeeded a set of object attributes shall also be available from the record, describing the object removed. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.31 AUD_AET_SETGID

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_SETGID event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_GID	gid	AUD_GID

The *gid* contains the value passed as an argument. The value before the call is reportable in the subject attributes.

24.2.2.32 AUD_AET_SETUID

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_SETUID event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_UID	uid	AUD_UID

The *uid* contains the value passed as an argument. The value before the call is reportable in the subject attributes.

24.2.2.33 AUD_AET_UNLINK

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_UNLINK event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Pathname	AUD_PATHNAME

The *Pathname* contains the value passed as an argument to the *unlink*() function.

If the call succeeded a set of object attributes shall also be available from the record, describing the object unlinked. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.2.2.34 AUD_AET_UTIME

Calls on *aud_get_event_info*() for the audit record of an AUD_AET_UTIME event shall return *aud_info_t* structures for the following event-specific items, with *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_STRING	Pathname	AUD_PATHNAME
AUD_TYPE_TIME	Access time	AUD_ATIME
AUD_TYPE_TIME	Modification time	AUD_MTIME

The *Pathname* contains the value passed as an argument to the *utime*() function. The *Access time* and *Modification time* contain the values from the *timebuf* structure passed as an argument.

If the call succeeded a set of object attributes shall also be available from the record, describing the object affected. If the call failed due to access controls, and a set of object attributes is still available from the record, it shall describe the object at which the failure occurred. Otherwise it is unspecified whether a set of object attributes is available, or what object is defined by such a set.

24.3 Header

Some of the data types used by the audit functions are not defined as part of this standard, but shall be implementation-defined. If {_POSIX_AUD} is defined, these types shall be defined in the header `<sys/audit.h>`, which contains definitions for at least the types shown in the following table.

Table 24-2 – Audit Data Types

Defined Type	Description
aud_evinfo_t	Used to access the set of event-specific data within an audit record. This data type is non-exportable data.
aud_hdr_t	Used to access the header of an audit record. This data type is non-exportable data.
aud_id_t	Item in an audit record header used to provide individual accountability for the audit event. This data type is exportable data.
aud_info_t	Defines the type, size and location of various items from an audit record. This data type is non-exportable data.
aud_obj_t	Used to access an object attribute set within an audit record. This data type is non-exportable data.
aud_obj_type_t	Item in an object attribute set that defines the type of the object. This data type is exportable data.
aud_rec_t	A pointer to an opaque audit record. This data type is non-exportable data.
aud_state_t	Controls whether system-generated records are auditable for a process. This data type is exportable data.
aud_status_t	Item in an audit record header giving the success/failure status of the audit event. This data type is exportable data.
aud_subj_t	Used to access the subject attribute set within an audit record. This data type is non-exportable data.
aud_time_t	The time of an audit event. This data type is exportable data.

Further details of these types are given below.

In addition, the header `<sys/audit.h>` shall define the following constants:

- All the AUD_AET_∗ constants defined in section 24.2.2, for the POSIX-defined event types
- All the constants defined in sections 24.2.2, 24.4.17, 24.4.19, 24.4.22, and 24.4.24, (including AUD_FIRST_ITEM and AUD_NEXT_ITEM) for the *item_id* arguments that can be supplied to the *aud_get_∗_info*() functions
- The following miscellaneous constants:

Table 24-3 – Other Constants

Constant	Description
AUD_SYSTEM_LOG	Value of the *filedes* argument for *aud_write*().
AUD_NATIVE	Value of the format item in a record header.
AUD_LAST_ITEM	Value of the *position* argument for the *aud_put_∗_info*() functions.
AUD_STD_NNNN_N	Value of the version item in a record header.

The NNNN_N in AUD_STD_NNNN_N is merely a placeholder for the year (e.g., 1997) this standard is approved and standard (e.g., _1 implying POSIX.1) it is placed into.

Further constants are identified in the rest of this section.

24.3.1 aud_evinfo_t

This typedef shall define an opaque, implementation-defined descriptor for the set of event-specific data in an audit record. The internal structure of an *aud_evinfo_t* is unspecified.

24.3.2 aud_hdr_t

This typedef shall define an opaque, implementation-defined descriptor for an audit record header. The internal structure of an *aud_hdr_t* is unspecified.

24.3.3 aud_id_t

The *aud_id_t* obtainable from an audit record header is an implementation-defined typedef for holding a value which uniquely identifies a user.

24.3.4 aud_info_t

The *aud_info_t* structure defines the type, length and location of some data from an audit record. The *aud_info_t* structure shall contain at least the following members:

Table 24-4 – *aud_info_t* **members**

Defined Type	Name	Description
int	aud_info_type	The type of the data
size_t	aud_info_length	The length of the data
*void **	aud_info_p	Pointer to the data

The *aud_info_type* member may be used to interpret the data referenced by the *aud_info_p* member. Values for *aud_info_type* shall be defined in the header `<sys/audit.h>`. At least the following values of *aud_info_type* shall be defined, and shall have the specified interpretation:

Table 24-5 – Values for *aud_info_type* Member

Value of *aud_info_type*	Interpretation of *aud_info_p*
AUD_TYPE_ACL	*acl_t**
AUD_TYPE_ACL_TYPE	*acl_type_t**
AUD_TYPE_AUD_ID	*aud_id_t**
AUD_TYPE_AUD_OBJ_TYPE	*aud_obj_type_t**
AUD_TYPE_AUD_STATE	*aud_state_t**
AUD_TYPE_AUD_STATUS	*aud_status_t**
AUD_TYPE_AUD_TIME	*aud_time_t**
AUD_TYPE_CAP	*cap_t**
AUD_TYPE_CHAR	*char**
AUD_TYPE_GID	*gid_t**
AUD_TYPE_INF	*inf_t**
AUD_TYPE_INT	*int**
AUD_TYPE_LONG	*long**
AUD_TYPE_MAC	*mac_t**
AUD_TYPE_MODE	*mode_t**
AUD_TYPE_OPAQUE	*void**
AUD_TYPE_PID	*pid_t**
AUD_TYPE_SHORT	*short**
AUD_TYPE_STRING	*char**, pointing to a null terminated character string
AUD_TYPE_STRING_ARRAY	*char***
AUD_TYPE_TIME	*time_t**
AUD_TYPE_UID	*uid_t**

With the exception of AUD_TYPE_STRING and AUD_TYPE_OPAQUE, *aud_info_p* should be interpreted as a pointer to zero or more items of the type specified. In the case of AUD_TYPE_STRING, *aud_info_p* should interpreted as a (*char* *) value. For AUD_TYPE_OPAQUE *aud_info_p* is interpreted as a pointer to zero or more bytes of opaque data.

A conforming implementation may define further values for *aud_info_type*, that can be treated in the same way as AUD_TYPE_OPAQUE.

In all cases, the *aud_info_length* member gives the length, in bytes, of the data to which *aud_info_p* points.

24.3.5 aud_obj_t

This typedef shall define an opaque, implementation-defined descriptor for a set of object attributes in an audit record. The internal structure of an *aud_obj_t* is unspecified.

24.3.6 aud_obj_type_t

The *aud_obj_type_t* obtainable from an object attribute set indicates the object type. This data type shall support a unique value for each of the object types for which object attribute sets can be generated in the implementation. The implementation shall define in `<sys/audit.h>` at least the following unique values:

Table 24-6 – *aud_obj_type_t* Values

Defined Type	Description
AUD_OBJ_BLOCK_DEV	Block device
AUD_OBJ_CHAR_DEV	Character device
AUD_OBJ_DIR	Directory
AUD_OBJ_FIFO	FIFO object
AUD_OBJ_FILE	Regular file
AUD_OBJ_PROC	Process object

24.3.7 aud_rec_t

This typedef shall define a pointer to an opaque data item capable of holding a specific audit record, the format and storage of which are unspecified. Thus, an application cannot depend on performing normal byte-copy operations on the data item to which an *aud_rec_t* points.

24.3.8 aud_state_t

An *aud_state_t* describes whether system events are being audited for a process. An implementation shall define in `<sys/audit.h>` at least the following unique values for this type:

Table 24-7 – *aud_state_t* Values

Defined Type	Description
AUD_STATE_OFF	System events not audited
AUD_STATE_ON	System events audited
AUD_STATE_QUERY	Enquiry value for *aud_switch*()

24.3.9 aud_status_t

The *aud_status_t* item obtainable from an audit record header indicates the status of the event. This data type shall define in `<sys/audit.h>` at least the following unique values for this type:

Table 24-8 – ***aud_status_t*** **Values**

Defined Type	Description
AUD_FAIL_PRIV	The event failed because the process did not have appropriate privilege (see below).
AUD_FAIL_DAC	The event failed because of DAC access checks.
AUD_FAIL_MAC	The event failed because of MAC access checks.
AUD_FAIL_OTHER	The event failed for some reason not included in other AUD_FAIL_* values.
AUD_PRIV_USED	The event completed successfully; appropriate privilege was used (see below).
AUD_SUCCESS	The event completed successfully.

The value AUD_PRIV_USED indicates that the operation succeeded, but would not have done so if the process had not had appropriate privilege.

If the process fails a DAC or MAC access check, and does not have appropriate privilege to override this check, and does not fail any other checks for appropriate privilege, then the AUD_FAIL_DAC or AUD_FAIL_MAC status, respectively, shall be reported in preference to the AUD_FAIL_PRIV one.

A conforming implementation may add additional status values.

24.3.10 aud_subj_t

This typedef shall define an opaque, implementation-defined descriptor for the set of subject attributes in an audit record. The internal structure of an *aud_subj_t* is unspecified.

24.3.11 aud_time_t

An *aud_time_t* structure specifies a single time value and shall include at least the following members:

Table 24-9 – ***aud_time_t*** **Members**

Defined Type	Name	Description
time_t	sec	Seconds
long	nsec	Nanoseconds

The *nsec* member specifies the subsecond portion of time; it is valid only if greater than or equal to zero, and less than the number of nanoseconds in a second (1000 million). A conforming implementation shall provide the subsecond portion of time to a resolution of at least 20 milliseconds (1/50 of a second).

24.4 Functions

The functions in this section comprise the set of services that permit a process to construct, write, read and analyze audit records. Support for the audit facility functions described in this section is optional. If the symbol {_POSIX_AUD} is defined the implementation supports the audit option and all of the audit functions shall be implemented as described in this section. If {_POSIX_AUD} is not defined, the result of calling any of these functions is unspecified.

The error [ENOTSUP] shall be returned in those cases where the system supports the audit facility but the particular audit operation cannot be applied because of restrictions imposed by the implementation.

24.4.1 Copy an Audit Record From System to User Space

Function: *aud_copy_ext*()

24.4.1.1 Synopsis

```
#include <sys/audit.h>

ssize_t aud_copy_ext (void *aud_rec_ext_p, aud_rec_t aud_rec_int,
                      ssize_t size);
```

24.4.1.2 Description

The *aud_copy_ext*() function shall copy an audit record, pointed to by *aud_rec_int*, from system-managed space to user-managed space (pointed to by *aud_rec_ext_p*). The *size* argument represents the size in bytes of the buffer pointed to by the *aud_rec_ext_p* argument.

The *aud_copy_ext*() function will do any conversions necessary to convert the record from internal format. The audit record returned by *aud_copy_ext*() will be a contiguous, persistent data item. It is the responsibility of the user to allocate a record buffer large enough to hold the copied record. The size of the buffer needed can be obtained by a call to the *aud_size*() function.

The *aud_copy_ext*() call shall not affect the record pointed to by *aud_rec_int*.

It is the responsibility of the user to release any space required to store the converted record.

24.4.1.3 Returns

Upon successful completion, the *aud_copy_ext*() function returns the size of the converted record placed in *aud_rec_ext_p*. Otherwise, a value of ((ssize_t)–1) shall be returned and *errno* shall be set to indicate the error.

24.4.1.4 Errors

If any of the following conditions occur, the *aud_copy_ext*() function shall return ((ssize_t)–1) and set *errno* to the corresponding value:

[EINVAL] The value for the *aud_rec_int* argument is invalid.

The *size* argument is zero or negative. –

[ERANGE] The *size* argument is greater than zero but smaller than the length of the audit record.

24.4.1.5 Cross-References

aud_copy_int(), 24.4.2; *aud_size*(), 24.4.38; *aud_valid*(), 24.4.40.

24.4.2 Copy an Audit Record From User to System Space

Function: *aud_copy_int*()

24.4.2.1 Synopsis

```
#include <sys/audit.h>
aud_rec_t aud_copy_int (const void *aud_rec_ext_p);
```

24.4.2.2 Description

The *aud_copy_int*() function shall copy an audit record, pointed to by *aud_rec_ext_p*, from user-managed space to system-managed space. Upon successful completion, the function shall return an *aud_rec_t* pointing to the internal version of the audit record.

Once copied to system-managed space, the record can be manipulated by the *aud_get_**() functions, and other functions that manipulate audit records.

The record pointed to by *aud_rec_ext_p* must have been obtained from a previous, successful call to *aud_copy_ext*() for this function to work successfully.

This function may cause memory to be allocated. The caller should free any releasable memory, when the new record is no longer required, by calling *aud_free*() with the (*void**)*aud_rec_t* as an argument.

The *aud_copy_int*() call shall not affect the record pointed to by *aud_rec_ext_p*.

24.4.2.3 Returns

Upon successful completion, the *aud_copy_int*() function returns an audit record pointer set to point to the internal version of the audit record. Otherwise, a value of (*aud_rec_t*)**NULL** shall be returned, the caller shall not have to free any releasable memory, and *errno* shall be set to indicate the error.

24.4.2.4 Errors

If any of the following conditions occur, the *aud_copy_int*() function shall return (*aud_rec_t*)**NULL** and set *errno* to the corresponding value:

[EINVAL] The value of the *aud_rec_ext_p* argument is invalid.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. –

24.4.2.5 Cross-References

aud_copy_ext(), 24.4.1; *aud_free*(), 24.4.14; *aud_get_event*(), 24.4.16; *aud_get_hdr*(), 24.4.18; *aud_get_obj*(), 24.4.21; *aud_get_subj*(). 24.4.23.

24.4.3 Delete Set of Event-specific Data from a Record

Function: *aud_delete_event*()

24.4.3.1 Synopsis

```
#include <sys/audit.h>
int aud_delete_event (aud_evinfo_t aud_event_d);
```

24.4.3.2 Description

The *aud_delete_event*() function deletes a set of event-specific data from an audit record, including any data items within the set. The set to be deleted is defined by the *aud_event_d* descriptor. Upon successful execution, the set of event-specific data shall no longer be accessible, and the *aud_event_d* descriptor shall become undefined.

Calls to this function shall not affect the status of descriptors for any other set of data in this or any other audit record.

24.4.3.3 Returns

Upon successful completion, the *aud_delete_event*() function returns 0. Otherwise, it returns a value of –1 and *errno* is set to indicate the error. The audit record shall not be changed if the return value is –1.

24.4.3.4 Errors

If any of the following conditions occur, the *aud_delete_event*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] Argument *aud_event_d* is not a valid descriptor for a set of event-specific data within an audit record. –

24.4.3.5 Cross-References

aud_delete_event_info(), 24.4.4; *aud_init_record*(), 24.4.27; *aud_put_event*(), 24.4.28; *aud_valid*(), 24.4.40; *aud_write*(), 24.4.41.

24.4.4 Delete Item from Set of Event-specific Data

Function: *aud_delete_event_info*()

24.4.4.1 Synopsis

```
#include <sys/audit.h>

int aud_delete_event_info (aud_evinfo_t aud_event_d,
                           int item_id);
```

24.4.4.2 Description

The *aud_delete_event_info*() function deletes a data item from a set of event-specific data in an audit record. Upon successful execution of *aud_delete_event_info*(), the item defined by *item_id* shall no longer be accessible in the set of event-specific data defined by *aud_event_d*.

The value of *item_id* specifies an item within the set of event-specific data. For system-generated records, the items available are dependent upon the *event type* of the audit record being examined; for each POSIX-defined event type the minimum set of items that shall be available, together with values of *item_id* to access them, are specified in section 24.2.2. For application-generated records, the values of *item_id* match the calls on *aud_put_event_info*() that put the items into the set of event-specific data.

Calls to this function shall not affect the status of descriptors for any other data item in this or any other audit record.

24.4.4.3 Returns

Upon successful completion, the *aud_delete_event_info*() function returns 0. Otherwise, it returns a value of −1 and *errno* is set to indicate the error. The audit record shall not be changed if the return value is −1.

24.4.4.4 Errors

If any of the following conditions occur, the *aud_delete_event_info*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] Argument *aud_event_d* is not a valid descriptor for a set of event-specific data within an audit record.

Argument *item_id* does not reference a valid data item within *aud_event_d*.

24.4.4.5 Cross-References

aud_delete_event(), 24.4.3; *aud_init_record*(), 24.4.27; *aud_put_event*(), 24.4.28; *aud_put_event_info*(), 24.4.29; *aud_valid*(), 24.4.40; *aud_write*(), 24.4.41.

24.4.5 Delete Header from an Audit Record

Function: *aud_delete_hdr*()

24.4.5.1 Synopsis

```
#include <sys/audit.h>
int aud_delete_hdr (aud_hdr_t aud_hdr_d);
```

24.4.5.2 Description

The *aud_delete_hdr*() function deletes a header from an audit record, including any data items within the header. The header to be deleted is defined by the *aud_hdr_d* descriptor. Upon successful execution, the header shall no longer be accessible in the record, and the *aud_hdr_d* descriptor shall become undefined.

Calls to this function shall not affect the status of descriptors for any other set of data in this or any other audit record.

24.4.5.3 Returns

Upon successful completion, the *aud_delete_hdr*() function returns 0. Otherwise, it returns a value of −1 and *errno* is set to indicate the error. The audit record shall not be changed if the return value is −1.

24.4.5.4 Errors

If any of the following conditions occur, the *aud_delete_hdr*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] Argument *aud_hdr_d* is not a valid descriptor for a header within an audit record.

24.4.5.5 Cross-References

aud_delete_hdr_info(), 24.4.6; *aud_init_record*(), 24.4.27; *aud_put_hdr*(), 24.4.30; *aud_valid*(), 24.4.40; *aud_write*(), 24.4.41.

24.4.6 Delete Item from Audit Record Header

Function: *aud_delete_hdr_info*()

24.4.6.1 Synopsis

```
#include <sys/audit.h>

int aud_delete_hdr_info (aud_hdr_t aud_hdr_d,
                         int item_id);
```

24.4.6.2 Description

The *aud_delete_hdr_info*() function deletes a data item from a header in an audit record. Upon successful execution of *aud_delete_hdr_info*(), the item defined by *item_id* shall no longer be accessible in the header defined by *aud_hdr_d*.

The value of *item_id* specifies an item within the audit record header. For records read from an audit log, the minimum set of items that shall be available from the first header, together with values of *item_id* to access them, are specified in section 24.4.19. For application-generated records the values of *item_id* match the calls on *aud_put_hdr_info*() that put the items into the header.

Calls to this function shall not affect the status of descriptors for any other data item in this or any other audit record.

24.4.6.3 Returns

Upon successful completion, the *aud_delete_hdr_info*() function returns 0. Otherwise, it returns a value of –1 and *errno* is set to indicate the error. The audit record shall not be changed if the return value is –1.

24.4.6.4 Errors

If any of the following conditions occur, the *aud_delete_hdr_info*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL] Argument *aud_hdr_d* is not a valid descriptor for a header within an audit record.

Argument *item_id* does not reference a valid data item within *aud_hdr_d*.

24.4.6.5 Cross-References

aud_delete_hdr(), 24.4.5; *aud_init_record*(), 24.4.27; *aud_put_hdr*(), 24.4.30; *aud_put_hdr_info*(), 24.4.31; *aud_valid*(), 24.4.40; *aud_write*(), 24.4.41.

24.4.7 Delete Set of Object Attributes from a Record

Function: *aud_delete_obj*()

24.4.7.1 Synopsis

```
#include <sys/audit.h>
int aud_delete_obj (aud_obj_t aud_obj_d);
```

24.4.7.2 Description

The *aud_delete_obj*() function deletes a set of object attributes from an audit record, including any data items within the set. The set to be deleted is defined by the *aud_obj_d* descriptor. Upon successful execution, the set of object attributes shall no longer be accessible in the record, and the *aud_obj_d* descriptor shall become undefined.

Calls to this function shall not affect the status of descriptors for any other set of data in this or any other audit record.

24.4.7.3 Returns

Upon successful completion, the *aud_delete_obj*() function returns 0. Otherwise, it returns a value of −1 and *errno* is set to indicate the error. The audit record shall not be changed if the return value is −1.

24.4.7.4 Errors

If any of the following conditions occur, the *aud_delete_obj*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] Argument *aud_obj_d* is not a valid descriptor for a set of object attributes within an audit record.

24.4.7.5 Cross-References

aud_delete_obj_info(), 24.4.8; *aud_init_record*(), 24.4.27; *aud_put_obj*(), 24.4.32; *aud_valid*(), 24.4.40; *aud_write*(), 24.4.41.

24.4.8 Delete Item from Set of Object Attributes

Function: *aud_delete_obj_info*()

24.4.8.1 Synopsis

```
#include <sys/audit.h>

int aud_delete_obj_info (aud_obj_t aud_obj_d,
                         int item_id);
```

24.4.8.2 Description

The *aud_delete_obj_info*() function deletes a data item from a set of object attributes in an audit record. Upon successful execution of *aud_delete_obj_info*(), the item defined by *item_id* shall no longer be accessible in the set of object attributes defined by *aud_obj_d*.

The value of *item_id* specifies an item within the set of object attributes. For system-generated records, the minimum set of items that shall be available, together with values of *item_id* to access them, are specified in section 24.4.22. For application-generated records, the values of *item_id* match the calls on *aud_put_obj_info*() that put the items into the set of object attributes.

Calls to this function shall not affect the status of descriptors for any other data item in this or any other audit record.

24.4.8.3 Returns

Upon successful completion, the *aud_delete_obj_info*() function returns 0. Otherwise, it returns a value of –1 and *errno* is set to indicate the error. The audit record shall not be changed if the return value is –1.

24.4.8.4 Errors

If any of the following conditions occur, the *aud_delete_obj_info*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL] Argument *aud_obj_d* is not a valid descriptor for a set of object attributes within an audit record.

Argument *item_id* does not reference a valid data item within *aud_obj_d*. –

24.4.8.5 Cross-References

aud_delete_obj(), 24.4.7; *aud_init_record*(), 24.4.27; *aud_put_obj*(), 24.4.32; *aud_put_obj_info*(), 24.4.33; *aud_valid*(), 24.4.40; *aud_write*(), 24.4.41.

24.4.9 Delete Set of Subject Attributes from a Record

Function: *aud_delete_subj*()

24.4.9.1 Synopsis

```
#include <sys/audit.h>

int aud_delete_subj (aud_subj_t aud_subj_d);
```

24.4.9.2 Description

The *aud_delete_subj*() function deletes a set of subject attributes from an audit record, including any data items within the set. The set to be deleted is defined by the *aud_subj_d* descriptor. Upon successful execution, the set of subject attributes shall no longer be accessible in the record, and the *aud_subj_d* descriptor shall become undefined.

Calls to this function shall not affect the status of descriptors for any other set of data in this or any other audit record.

24.4.9.3 Returns

Upon successful completion, the *aud_delete_subj*() function returns 0. Otherwise, it returns a value of –1 and *errno* is set to indicate the error. The audit record shall not be changed if the return value is –1.

24.4.9.4 Errors

If any of the following conditions occur, the *aud_delete_subj*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] Argument *aud_subj_d* is not a valid descriptor for a set of subject attributes within an audit record.

24.4.9.5 Cross-References

aud_delete_subj_info(), 24.4.10; *aud_init_record*(), 24.4.27; *aud_put_subj*(), 24.4.34; *aud_valid*(), 24.4.40; *aud_write*(), 24.4.41.

24.4.10 Delete Item from Set of Subject Attributes

Function: *aud_delete_subj_info*()

24.4.10.1 Synopsis

```
#include <sys/audit.h>

int aud_delete_subj_info (aud_subj_t aud_subj_d,
                          int item_id);
```

24.4.10.2 Description

The *aud_delete_subj_info*() function deletes a data item from a set of subject attributes in an audit record. Upon successful execution of *aud_delete_subj_info*(), the item defined by *item_id* shall no longer be accessible in the set of subject attributes defined by *aud_subj_d*.

The value of *item_id* specifies an item within the set of subject attributes. For system-generated records, the minimum set of items that shall be available, together with values of *item_id* to access them, are specified in section 24.4.24. For application-generated records, the values of *item_id* match the calls on *aud_put_subj_info*() that put the items into the set of subject attributes.

Calls to this function shall not affect the status of descriptors for any other data item in this or any other audit record.

24.4.10.3 Returns

Upon successful completion, the *aud_delete_subj_info*() function returns 0. Otherwise, it returns a value of –1 and *errno* is set to indicate the error. The audit record shall not be changed if the return value is –1.

24.4.10.4 Errors

If any of the following conditions occur, the *aud_delete_subj_info*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL] Argument *aud_subj_d* is not a valid descriptor for a set of subject attributes within an audit record.

Argument *item_id* does not reference a valid data item within *aud_subj_d*. –

24.4.10.5 Cross-References

aud_delete_subj(), 24.4.9; *aud_init_record*(), 24.4.27; *aud_put_subj*(), 24.4.34; *aud_put_subj_info*(), 24.4.35; *aud_valid*(), 24.4.40; *aud_write*(), 24.4.41.

24.4.11 Duplicate an Audit Record

Function: *aud_dup_record*()

24.4.11.1 Synopsis

```
#include <sys/audit.h>
aud_rec_t aud_dup_record (aud_rec_t ar);
```

24.4.11.2 Description

The *aud_dup_record*() function creates a duplicate of the audit record pointed to by argument *ar*. The duplicate shall be independent of the original record; subsequent operations on either shall not affect the other. Upon successful execution, the *aud_dup_record*() function returns a pointer to the duplicate record.

Any existing descriptors that refer to *ar* shall continue to refer to that record. Calls to *aud_dup_record*() shall not affect the status of any existing records.

This function may cause memory to be allocated. The caller should free any releasable memory, when the new record is no longer required, by calling *aud_free*() with the (*void**)*aud_rec_t* as an argument.

24.4.11.3 Returns

Upon successful completion, the *aud_dup_record*() function returns an *aud_rec_t* pointing to the new record. Otherwise, a value of (*aud_rec_t*)**NULL** shall be returned, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error.

24.4.11.4 Errors

If any of the following conditions occur, the *aud_dup_record*() function shall return (*aud_rec_t*)**NULL** and set *errno* to the corresponding value:

[EINVAL] Argument *ar* does not point to a valid audit record.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. –

24.4.11.5 Cross-References

aud_free(), 24.4.14; *aud_init_record*(), 24.4.27; *aud_valid*(), 24.4.40; *aud_write*(), 24.4.41.

24.4.12 Map Text to Event Type

Function: *aud_evid_from_text*()

24.4.12.1 Synopsis

```
#include <sys/audit.h>

int aud_evid_from_text  (const char *text);
```

24.4.12.2 Description

The *aud_evid_from_text*() function returns the audit event type of the system audit event identified by the string pointed to by *text*. The means by which this information is obtained is unspecified.

24.4.12.3 Returns

Upon successful completion, the *aud_evid_from_text*() function returns the event type associated with *text*. On error, or if the requested entry is not found a value of -1 is returned and *errno* is set to indicate the error.

24.4.12.4 Errors

If any of the following conditions occur, the *aud_evid_from_text*() function shall return a value of -1 and set *errno* to the corresponding value:

[EINVAL] The *text* argument does not identify a valid system audit event type.

24.4.12.5 Cross-References

aud_evid_to_text(), 24.4.13; *aud_get_hdr_info*(), 24.4.19; *aud_put_hdr_info*(), 24.4.31.

24.4.13 Map Event Type to Text

Function: *aud_evid_to_text*()

24.4.13.1 Synopsis

```
#include <sys/audit.h>

char *aud_evid_to_text (int event_type, ssize_t *aud_info_length);
```

24.4.13.2 Description

The *aud_evid_to_text*() function shall transform the system audit *event_type* into a human-readable, null terminated character string identifying an event type. The means by which this information is obtained is unspecified. The function shall return the address of the string, and set the location pointed to by *aud_info_length* to the length of the string (not including the null terminator).

This function may cause memory to be allocated. The caller should free any releasable memory when the string is no longer required, by calling the *aud_free*() function with the string address (cast to a (*void**)) as an argument.

24.4.13.3 Returns

Upon successful completion, the *aud_evid_to_text*() function returns a pointer to a string containing the event name associated with *event_type*. On error, or if the requested entry is not found, (*char* ∗)**NULL** is returned, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error.

24.4.13.4 Errors

If any of the following conditions occur, the *aud_evid_to_text*() function shall return (*char* ∗)**NULL** and set *errno* to the corresponding value:

[EINVAL] The *event_type* argument does not contain a valid system audit event type.

[ENOMEM] The string to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

24.4.13.5 Cross-References

aud_evid_from_text(); 24.4.12. *aud_get_hdr_info*(), 24.4.19; *aud_put_hdr_info*(), 24.4.31.

24.4.14 Release Memory Allocated to an Audit Data Object

Function: *aud_free*()

24.4.14.1 Synopsis

```
#include <sys/audit.h>

int aud_free (void *obj_p);
```

24.4.14.2 Description

The function *aud_free*() shall free any releasable memory currently allocated to the item identified by *obj_p*. This may identify an audit record (i.e., be a (*void*∗)*aud_rec_t*) or a pointer to a string or event list allocated by one of the audit functions.

If the item identified by *obj_p* is an *aud_rec_t*, the *aud_rec_t* and any existing descriptors and *aud_info_t* items that refer to parts of the audit record shall become undefined. If it is a string (*char*∗), then use of the *char*∗ shall become undefined.

24.4.14.3 Returns

Upon successful completion, the *aud_free*() function returns 0. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error, and the memory shall not be freed.

24.4.14.4 Errors

If any of the following conditions occur, the *aud_free*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The *obj_p* argument does not identify an audit record, string or event list allocated by one of the audit functions.

24.4.14.5 Cross-References

aud_copy_int(), 24.4.2; *aud_dup_record*(), 24.4.11; *aud_get_all_evid*(), 24.4.15; *aud_get_event*(), 24.4.16; *aud_get_event_info*(), 24.4.17; *aud_get_hdr*(), 24.4.18; *aud_get_hdr_info*(), 24.4.19; *aud_get_obj*(), 24.4.21; *aud_get_obj_info*(), 24.4.22; *aud_get_subj*(), 24.4.23; *aud_get_subj_info*(), 24.4.24; *aud_id_to_text*(), 24.4.26; *aud_init_record*(), 24.4.27; *aud_read*(), 24.4.36; *aud_rec_to_text*(), 24.4.37; *aud_valid*(), 24.4.40.

24.4.15 Get All Audit Event Types

Function: *aud_get_all_evid*()

24.4.15.1 Synopsis

```
#include <sys/audit.h>

int *aud_get_all_evid (void)
```

24.4.15.2 Description

The *aud_get_all_evid*() function returns the list of event types for system-generated events currently reportable on a conforming implementation. Each event type is a non-negative integer; the list is terminated by a negative value. The means by which this information is obtained is unspecified. These event types can be converted into textual format by the *aud_evid_to_text*() function.

This function may cause memory to be allocated. The caller should free any releasable memory when the event list is no longer required, by calling the *aud_free*() function with the event list address (cast to a *void**) as an argument.

24.4.15.3 Returns

Upon successful completion, the *aud_get_all_evid*() function returns a pointer to a list of the system-generated event types currently reportable on a conforming implementation. Otherwise, (*int* ∗)**NULL** is returned, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error.

24.4.15.4 Errors

If any of the following conditions occur, the *aud_get_all_evid*() function shall return (*int* ∗)**NULL** and set *errno* to the corresponding value:

[ENOMEM] The event types to be returned require more memory than is allowed by the hardware or system-imposed memory management constraints.

24.4.15.5 Cross-References

aud_free(), 24.4.14; *aud_evid_from_text*(), 24.4.12; *aud_evid_to_text*(), 24.4.13.

24.4.16 Get Audit Record Event-specific Data Descriptor

Function: *aud_get_event*()

24.4.16.1 Synopsis

```
#include <sys/audit.h>

int aud_get_event (aud_rec_t ar,
                   int index,
                   aud_evinfo_t *aud_event_p);
```

24.4.16.2 Description

The *aud_get_event*() function returns a descriptor to a set of event-specific data from an audit record. The function accepts an audit record pointer *ar* returned from a previously successful call to *aud_read*(), *aud_init_record*() or *aud_dup_record*(). If *aud_event_p* is not **NULL**, then upon successful execution the *aud_get_event*() function shall return a descriptor via *aud_event_p* for the set of event-specific data identified by *index*. The descriptor returned by this call can then be used in subsequent calls on *aud_get_event_info*() to extract the data items from the set of event-specific data from the audit record. If *aud_event_p* is **NULL**, then the value of the *index* argument is ignored and the function just returns a value as described below.

Calls to *aud_get_event*() shall not affect the status of any other existing descriptors. Calls on the various *aud_get_∗*() functions can be interleaved without affecting each other.

This function may cause memory to be allocated. The caller should free any releasable memory, when the record is no longer required, by calling *aud_free*() with the (*void**)*aud_rec_t* as an argument.

A descriptor for the first set of event-specific data in the record is obtained by supplying an *index* of 1. While the standard does not require more than one set of event-specific data to be present in a record, an implementation or application may add additional sets that can be read by supplying values of *index* that are greater than 1.

24.4.16.3 Returns

Upon successful completion, the *aud_get_event*() function returns a non-negative value. This value indicates the number of sets of event-specific data in the record.

In the event of failure the *aud_get_event*() function returns a value of −1, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error. The *aud_evinfo_t* referenced by *aud_event_p* shall not be affected if the return value is -1.

24.4.16.4 Errors

If any of the following conditions occur, the *aud_get_event*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] Argument *ar* does not point to a valid audit record.

Argument *index* does not identify a valid set of event-specific data in the record.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints.

24.4.16.5 Cross-References

aud_free(), 24.4.14; *aud_get_event_info*(), 24.4.17; *aud_put_event*(), 24.4.28; *aud_read*(), 24.4.36; *aud_valid*(), 24.4.40.

24.4.17 Examine Audit Record Event-specific Data

Function: *aud_get_event_info*()

24.4.17.1 Synopsis

```
#include <sys/audit.h>

int aud_get_event_info (aud_evinfo_t aud_event_d,
                        int item_id,
                        aud_info_t *aud_event_info_p);
```

24.4.17.2 Description

The *aud_get_event_info*() function returns a data item from within a set of event-specific data. The set of event-specific data within an audit record to be examined is identified by *aud_event_d* which was obtained from a previous successful call to *aud_get_event*() or *aud_put_event*(). If *aud_event_info_p* is not **NULL**, then upon successful execution the *aud_get_event_info*() function shall return via *aud_event_info_p* an *aud_info_t* for the data identified by *item_id*. If *aud_event_info_p* is **NULL**, then the value of the *item_id* argument is ignored, and the function just returns a value as described in the Returns section below.

The value of *item_id* may specify a named item within the set of event-specific data, or may specify the 'first' item or the 'next' item. The named items available are dependent upon the *event type* of the audit record being examined; for each POSIX-defined event type the minimum set of items that shall be available, together with values of *item_id* to access them, are specified in section 24.2.2.

If *item_id* is AUD_FIRST_ITEM, then this specifies the first item of event-specific data in the set. A call of *aud_get_event_info*() with *item_id* set to AUD_NEXT_ITEM shall return the item that follows the previous one read; for POSIX-defined events, the required items are returned in the order they are defined for each event type in section 24.2.2; implementations may report additional items after the required items. If AUD_NEXT_ITEM is used when there has not been a previous successful call of this function for this set of event information, the effect is unspecified.

Any existing descriptors shall not be affected by use of this function. Calls on the various *aud_get_**() functions can be interleaved without affecting each other.

This function may cause memory to be allocated. The caller should free any releasable memory, when the record containing *aud_event_d* is no longer required, by calling *aud_free*() with the *aud_rec_t* for the record (cast to a (*void**)) as an argument.

24.4.17.3 Returns

Upon successful completion, the *aud_get_event_info*() function returns a non-negative value. This value indicates the number of items of event-specific data in the set.

In the event of failure the *aud_get_event_info*() function returns a value of –1, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error. The *aud_info_t* referenced by *aud_event_info_p* shall not be affected if the return value is -1.

24.4.17.4 Errors

If any of the following conditions occur, the *aud_get_event_info*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL]	Argument *aud_event_d* is not a valid descriptor for a set of event-specific data within an audit record.
	Argument *item_id* does not identify a valid item from the set of event-specific data.
[ENOMEM]	The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. −

24.4.17.5 Cross-References

aud_free(), 24.4.14; *aud_get_event*(), 24.4.16; *aud_put_event_info*(), 24.4.29; *aud_read*(), 24.4.36; *aud_valid*(), 24.4.40.

24.4.18 Get an Audit Record Header Descriptor

Function: *aud_get_hdr*()

24.4.18.1 Synopsis

```
#include <sys/audit.h>

int aud_get_hdr (aud_rec_t ar,
                 int index,
                 aud_hdr_t *aud_hdr_p);
```

24.4.18.2 Description

The *aud_get_hdr*() function returns a descriptor to the header of an audit record. The function accepts an audit record pointer *ar* returned from a previously successful call to *aud_read*(), *aud_init_record*() or *aud_dup_record*(). If *aud_hdr_p* is not **NULL**, then upon successful execution the *aud_get_hdr*() function shall return a descriptor via *aud_hdr_p* for the header identified by *index*. The descriptor returned by this call can then be used in subsequent calls to *aud_get_hdr_info*() to extract the data from the audit record header. If *aud_hdr_p* is **NULL**, then the value of the *index* argument is ignored, and the function just returns a value as described below.

Calls to *aud_get_hdr*() shall not affect the status of any other existing descriptors. Calls on the various *aud_get_**() functions can be interleaved without affecting each other.

This function may cause memory to be allocated. The caller should free any releasable memory, when the record is no longer required, by calling *aud_free*() with the (*void**)*aud_rec_t* as an argument.

A descriptor for the first header in the record is obtained by supplying an *index* of 1. While the standard does not require more than one header to be present in a record, an implementation or application may add additional headers that can be read by supplying values of *index* that are greater than 1.

24.4.18.3 Returns

Upon successful completion, the *aud_get_hdr*() function returns a non-negative value. This value indicates the number of headers in the record.

In the event of failure the *aud_get_hdr*() function returns a value of −1, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error. The *aud_hdr_t* referenced by *aud_hdr_p* shall not be affected if the return value is -1.

24.4.18.4 Errors

If any of the following conditions occur, the *aud_get_hdr*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] Argument *ar* does not point to a valid audit record.

Argument *index* does not identify a valid header in the record.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. –

24.4.18.5 Cross-References

aud_free(), 24.4.14; *aud_get_hdr_info*(), 24.4.19; *aud_put_hdr*(), 24.4.30; *aud_read*(), 24.4.36; *aud_valid*(), 24.4.40.

24.4.19 Examine an Audit Record Header

Function: *aud_get_hdr_info*()

24.4.19.1 Synopsis

```
#include <sys/audit.h>

int aud_get_hdr_info (aud_hdr_t aud_hdr_d,
                      int item_id,
                      aud_info_t *aud_hdr_info_p);
```

24.4.19.2 Description

The *aud_get_hdr_info*() function returns a data item from within a header of an audit record. The audit record header to be examined is identified by *aud_hdr_d* which was obtained from a previous successful call to *aud_get_hdr*() or *aud_put_hdr*(). If *aud_hdr_info_p* is not **NULL**, then upon successful execution the *aud_get_hdr_info*() function shall return via *aud_hdr_info_p* an *aud_info_t* for the item of event-specific data identified by *item_id*. If *aud_hdr_info_p* is **NULL**, then the value of the *item_id* argument is ignored, and the function just returns a value as described in the Returns section below.

The value of *item_id* may specify a named item within the set of header data, or may specify the 'first' item or the 'next' item.

The minimum set of named items to be available from the first header of an audit record is specified in the table below, together with values of *item_id* to access the items. If a record contains more than one header, the contents of the second and subsequent headers is not specified by this standard.

Table 24-10 – aud_hdr_info_p Values

Type	Description	item_id	Notes
AUD_TYPE_SHORT	The format of the audit record	AUD_FORMAT	(1)
AUD_TYPE_SHORT	The version number of the record	AUD_VERSION	(2)
AUD_TYPE_AUD_ID	The audit ID of the process	AUD_AUD_ID	(3)
AUD_TYPE_INT or AUD_TYPE_STRING	The event type of the record	AUD_EVENT_TYPE	(4)
AUD_TYPE_AUD_TIME	The time the event occurred	AUD_TIME	
AUD_TYPE_AUD_STATUS	The audit status of the event	AUD_STATUS	
AUD_TYPE_INT	Value returned for event (*errno*)	AUD_ERRNO	(5)

Notes on the table:

(1) Only one value is currently defined for the *format* item: AUD_NATIVE. All data in any given record shall be in the same format.

(2) The *version* item provides a means of identifying the version of the POSIX.1e audit option to which the audit record conforms. Conforming applications can make use of the *version* to provide for backward compatibility or to ignore records which they are not prepared to handle.

Currently only one value for *version* is defined: AUD_STD_NNNN_N. This identifies records which conform to the initial version of this standard. Further revisions of this standard may define additional values for the header version. The NNNN_N is merely a placeholder for the year (e.g., 1997) this standard is approved and standard (e.g., _1 implying POSIX.1) it is placed into.

(3) If the record is not associated with any accountable user (e.g., it was recorded before a user had completed authentication), then the *aud_get_hdr_info*() function shall return an *aud_info_t* with a zero *aud_info_length* member.

(4) The event type is an integer if this is a system-generated event, or a string if it is an application-generated event.

(5) For system-generated events, the return value reported contains the *errno* on return from the function audited; if the operation succeeded (as shown by the *status*), this value is undefined. For application-generated records there may be no *errno* reported.

If *item_id* is AUD_FIRST_ITEM, then this specifies the first of the items of information from the header. A call of *aud_get_hdr_info*() with *item_id* set to AUD_NEXT_ITEM shall return the item that follows the previous one read; for the POSIX-defined header, the required items are returned in the order they are defined in the table above; implementations may report additional items after the required items. If AUD_NEXT_ITEM is used when there has not been a previous successful call of this function for this header, the effect is unspecified.

This function may cause memory to be allocated. The caller should free any releasable memory, when the record containing *aud_hdr_d* is no longer required, by calling *aud_free*() with the *aud_rec_t* for the record (cast to a (*void*∗)) as an argument.

Any existing descriptors shall not be affected by use of this function. Calls on the various *aud_get_*∗() functions can be interleaved without affecting each other.

24.4.19.3 Returns

Upon successful completion, the *aud_get_hdr_info*() function returns a non-negative value. This value indicates the number of items of header information in the set.

In the event of failure the *aud_get_hdr_info*() function returns a value of –1, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error. The *aud_info_t* referenced by *aud_hdr_info_p* shall not be affected if the return value is -1.

24.4.19.4 Errors

If any of the following conditions occur, the *aud_get_hdr_info*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] Argument *aud_hdr_d* is not a valid descriptor for an audit record header within an audit record.

Argument *item_id* does not identify a valid item from the header.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. –

24.4.19.5 Cross-References

aud_free(), 24.4.14; *aud_get_hdr*(), 24.4.18; *aud_put_hdr_info*(), 24.4.31; *aud_read*(), 24.4.36; *aud_valid*(), 24.4.40.

24.4.20 Get a Process Audit ID

Function: *aud_get_id*()

24.4.20.1 Synopsis

```
#include <sys/audit.h>
#include <sys/types.h>

aud_id_t aud_get_id (pid_t pid);
```

24.4.20.2 Description

The *aud_get_id*() function returns the audit ID of the user who is accountable for auditable actions of the existing process identified by *pid*.

It is unspecified whether appropriate privilege is required to use this function.

24.4.20.3 Returns

Upon successful completion, the *aud_get_id*() function returns the audit ID of the nominated process. Otherwise, a value of ((*aud_id_t*)–1) is returned and *errno* is set to indicate the error.

24.4.20.4 Errors

If any of the following conditions occur, the *aud_get_id*() function shall return a value of ((*aud_id_t*)–1) and set *errno* to the corresponding value:

[EINVAL] The value of the *pid_t* argument is invalid.

24.4.20.5 Cross-References

aud_id_to_text(), 24.4.26; *aud_put_hdr_info*(), 24.4.31; *aud_write*(), 24.4.41.

24.4.21 Get an Audit Record Object Descriptor

Function: *aud_get_obj*()

24.4.21.1 Synopsis

```
#include <sys/audit.h>

int aud_get_obj (aud_rec_t ar,
                 int index,
                 aud_obj_t *aud_obj_p);
```

24.4.21.2 Description

The *aud_get_obj*() function returns a descriptor to a set of object attributes from an audit record. The function accepts an audit record pointer *ar* returned from a previously successful call to *aud_read*(), *aud_init_record*() or *aud_dup_record*(). If *aud_obj_p* is not **NULL**, then upon successful execution the *aud_get_obj*() function shall return a descriptor via *aud_obj_p* for the set of object data identified by *index*. The descriptor returned by this call can then be used in subsequent calls to *aud_get_obj_info*() to extract the object data for that object. If *aud_obj_p* is **NULL**, then the function just returns a value as described below.

Calls to *aud_get_obj*() shall not affect the status of any other existing descriptors. Calls on the various *aud_get_**() functions can be interleaved without affecting each other.

This function may cause memory to be allocated. The caller should free any releasable memory, when the record is no longer required, by calling *aud_free*() with the (*void**)*aud_rec_t* as an argument.

A descriptor for the first set of object attributes in the record is obtained by supplying an *index* of 1. Any additional sets can be read by supplying values of *index* that are greater than 1.

24.4.21.3 Returns

Upon successful completion, the *aud_get_obj*() function returns a non-negative value. This value indicates the number of sets of object attributes in the record.

In the event of failure the *aud_get_obj*() function returns a value of −1, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error. The *aud_obj_t* referenced by *aud_obj_p* shall not be affected if the return value is -1.

24.4.21.4 Errors

If any of the following conditions occur, the *aud_get_obj*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] Argument *ar* does not point to a valid audit record.

Argument *index* does not identify a valid set of object attributes in the record.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. –

24.4.21.5 Cross-References

aud_free(), 24.4.14; *aud_get_obj_info*(), 24.4.22; *aud_put_obj*(), 24.4.32; *aud_read*(), 24.4.36; *aud_valid*(), 24.4.40.

24.4.22 Examine Audit Record Object Data

Function: *aud_get_obj_info*()

24.4.22.1 Synopsis

```
#include <sys/audit.h>

int aud_get_obj_info (aud_obj_t aud_obj_d,
                      int item_id,
                      aud_info_t *aud_obj_info_p);
```

24.4.22.2 Description

The *aud_get_obj_info*() function returns a data item from within a set of object data. For system-generated events recording use of an interface that changes object attributes, the attributes reported are those at the start of the event. The set of object data to be examined is identified by *aud_obj_d* which was obtained from a previous successful call to *aud_get_obj*() or *aud_put_obj*(). If *aud_obj_info_p* is not **NULL**, then upon successful execution the *aud_get_obj_info*() function shall return via *aud_obj_info_p* an *aud_info_t* for the object attribute identified by *item_id*. If *aud_obj_info_p* is **NULL**, then the value of the *item_id* argument is ignored, and the function just returns a value as described in the Returns section below.

The value of *item_id* may specify a named item within the set of object data or may specify the 'first' item or the 'next' item.

The minimum set of named items that shall be available for system generated events that are required to report object attributes is specified in the table below, together with values of *item_id* to access them:

Table 24-11 – aud_obj_info_p Values

Type	Description	item_id	Notes
AUD_TYPE_AUD_OBJ_TYPE	The type of the object	AUD_TYPE	
AUD_TYPE_UID	The user ID of the object owner	AUD_UID	(1)
AUD_TYPE_GID	The group ID of the object owner	AUD_GID	(2)
AUD_TYPE_MODE	The mode bits of the object	AUD_MODE	(3)
AUD_TYPE_STRING	The name of the object	AUD_NAME	(4)
AUD_TYPE_ACL	The ACL of the object	AUD_ACL	(5)
AUD_TYPE_MAC	The MAC label of the object	AUD_MAC_LBL	(6)
AUD_TYPE_INF	The information label of the object	AUD_INF_LBL	(7)
AUD_TYPE_CAP	The capability set of the object	AUD_CAP	

Notes on the table:

(1) For a process object, the object owner is the effective UID of the process.

(2) For a process object, the object group is the effective GID of the process.

(3) For a process object, the *aud_get_obj_info*() function may return an *aud_info_t* with a zero *aud_info_length* member for the mode bits.

(4) This item contains the name of the object, which shall provide sufficient information to identify the object.

(5) This item contains an *acl_t* recording the ACL of the object at the start of the event. If {_POSIX_ACL} was not defined at that time, or the object

does not have a POSIX.1e conformant ACL, the *aud_get_obj_info*() function shall return an *aud_info_t* with a zero *aud_info_length* member.

(6) This item contains a *mac_t* recording the MAC label of the object at the start of the event. If {_POSIX_MAC} was not defined at that time, the *aud_get_obj_info*() function shall return an *aud_info_t* with a zero *aud_info_length* member.

(7) This item contains an *inf_t* recording the information label of the object at the start of the event. If {_POSIX_INF} was not defined at that time, the *aud_get_obj_info*() function shall return an *aud_info_t* with a zero *aud_info_length* member.

(8) This item contains a *cap_t* recording the capability set of the object at the start of the event. If {_POSIX_CAP} was not in effect at that time or if the object does not have a POSIX.1e conformant capability set, the *aud_get_obj_info*() function shall return an *aud_info_t* with a zero *aud_info_length* member.

If *item_id* is AUD_FIRST_ITEM, this specifies the first of the items of information from the set. A call of *aud_get_obj_info*() with *item_id* set to AUD_NEXT_ITEM shall return the item that follows the previous one read; for system-generated events that are required to report object attributes, the required items are returned in the order they are defined in the table above; implementations may report additional items after the required items. If AUD_NEXT_ITEM is used when there has not been a previous successful call of this function for this set of object attributes, the effect is unspecified.

Only the object type and object owner items are required. The other specified items are optional. If an item is not available, the function *aud_get_obj_info*() shall return a *aud_info_t* with a zero *aud_info_length* member.

Any existing descriptors shall not be affected by use of this function. Calls on the various *aud_get_*∗() functions can be interleaved without affecting each other.

This function may cause memory to be allocated. The caller should free any releasable memory, when the record containing *aud_obj_d* is no longer required, by calling *aud_free*() with the *aud_rec_t* for the record (cast to a (*void*∗)) as an argument.

24.4.22.3 Returns

Upon successful completion, the *aud_get_obj_info*() function returns a non-negative value. This value indicates the number of items of object attributes in the set.

In the event of failure the *aud_get_obj_info*() function returns a value of −1, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error. The *aud_info_t* referenced by *aud_obj_info_p* shall not be affected if the return value is -1.

24.4.22.4 Errors

If any of the following conditions occur, the *aud_get_obj_info*() function shall return -1 and set *errno* to the corresponding value: If any of the following conditions occur, this function will fail and set *errno* to one of the following values:

[EINVAL] Argument *aud_obj_d* is not a valid descriptor for a set of object attributes within an audit record.

Argument *item_id* does not identify a valid item from the set of object attributes.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. –

24.4.22.5 Cross-References

aud_free(), 24.4.14; *aud_get_obj*(), 24.4.21; *aud_put_obj_info*(), 24.4.33; *aud_read*(), 24.4.36; *aud_valid*(), 24.4.40.

24.4.23 Get an Audit Record Subject Descriptor

Function: *aud_get_subj*()

24.4.23.1 Synopsis

```
#include <sys/audit.h>

int aud_get_subj (aud_rec_t ar,
                  int index,
                  aud_subj_t *aud_subj_p);
```

24.4.23.2 Description

The *aud_get_subj*() function returns a descriptor to a set of subject attributes from an audit record. The function accepts an audit record pointer *ar* returned from a previously successful call to *aud_read*(), *aud_init_record*() or *aud_dup_record*(). If *aud_subj_p* is not **NULL**, then upon successful execution the *aud_get_subj*() function shall return a descriptor via *aud_subj_p* for the set of subject attributes identified by *index*. The descriptor returned by this call can then be used in subsequent calls to *aud_get_subj_info*() to extract the attributes for that process. If *aud_subj_p* is **NULL**, then the function just returns a value as described below.

Calls to *aud_get_subj*() shall not affect the status of any other existing descriptors. Calls on the various *aud_get_**() functions can be interleaved without affecting each other.

This function may cause memory to be allocated. The caller should free any releasable memory, when the record is no longer required, by calling *aud_free*() with the (*void**)*aud_rec_t* as an argument.

A descriptor for the first set of subject attributes in the record is obtained by supplying an *index* of 1. While the standard does not require more than one set of subject attributes to be present in a record, an implementation or application may add additional sets that can be read by supplying values of *index* that are greater than 1.

24.4.23.3 Returns

Upon successful completion, the *aud_get_subj*() function returns a non-negative value. This value indicates the number of sets of subject attributes in the record.

In the event of failure the *aud_get_subj*() function returns a value of −1, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error. The *aud_subj_t* referenced by *aud_subj_p* shall not be affected if the return value is -1.

24.4.23.4 Errors

If any of the following conditions occur, the *aud_get_subj*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] Argument *ar* does not point to a valid audit record.

Argument *index* does not identify a valid set of subject attributes in the record.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. −

24.4.23.5 Cross-References

aud_free(), 24.4.14; *aud_get_subj_info*(), 24.4.24; *aud_put_subj*() 24.4.34; *aud_read*(), 24.4.36; *aud_valid*(), 24.4.40.

24.4.24 Examine Audit Record Subject Data

Function: *aud_get_subj_info*()

24.4.24.1 Synopsis

```
#include <sys/audit.h>

int aud_get_subj_info (aud_subj_t aud_subj_d,
                       int item_id,
                       aud_info_t *aud_subj_info_p);
```

24.4.24.2 Description

The *aud_get_subj_info*() function returns a data item from within a set of subject attributes in an audit record. For system-generated events recording use of an interface that changes subject attributes, the attributes reported are those at the start of the event. The set of attributes to be examined is identified by *aud_subj_d* which was obtained from a previous successful call to *aud_get_subj*() or *aud_put_subj*(). If *aud_subj_info_p* is not **NULL**, then upon successful execution the *aud_get_subj_info*() function shall return via *aud_subj_info_p* an *aud_info_t* for the attribute identified by *item_id*. If *aud_subj_info_p* is **NULL**, then the value of the *item_id* argument is ignored, and the function just returns a value as described in the Returns section below.

The value of *item_id* may specify a named item within the set of subject attributes, or may specify the 'first' item or the 'next' item. The minimum set of named items that shall be available from system-generated records is specified in the table below, together with values of *item_id* to access them:

Table 24-12 – aud_subj_info_p Values

Type	Description	item_id	Notes
AUD_TYPE_PID	The process ID	AUD_PID	
AUD_TYPE_UID	The effective user ID	AUD_EUID	
AUD_TYPE_GID	The effective group ID	AUD_EGID	
AUD_TYPE_GID	The supplementary group IDs	AUD_SGIDS	(1)
AUD_TYPE_UID	The real user ID	AUD_RUID	
AUD_TYPE_GID	The real group ID	AUD_RGID	
AUD_TYPE_MAC	The process MAC label	AUD_MAC_LBL	(2)
AUD_TYPE_INF	The process information label	AUD_INF_LBL	(3)
AUD_TYPE_CAP	The process capability state	AUD_CAP	(4)

Notes on the table:

(1) The number of supplementary groups can be calculated from the *aud_info_length* member of the *aud_info_t*.

(2) If {_POSIX_MAC} was not defined at that time, the *aud_get_subj_info*() function shall return an *aud_info_t* with a zero *aud_info_length* member.

(3) If {_POSIX_INF} was not defined at that time, the *aud_get_subj_info*() function shall return an *aud_info_t* with a zero *aud_info_length* member.

(4) If {_POSIX_CAP} was not defined at that time, the *aud_get_subj_info*() function shall return an *aud_info_t* with a zero *aud_info_length* member.

If *item_id* is AUD_FIRST_ITEM, then this specifies the first of the items from the set of subject attributes. A call of *aud_get_subj_info*() with *item_id* set to AUD_NEXT_ITEM shall return the item that follows the previous one read; for system-generated records, the required items are returned in the order they are defined in the table above; implementations may report additional items after the required items. If AUD_NEXT_ITEM is used when there has not been a previous successful call of this function for this set of subject attributes, the effect is unspecified.

For system-generated records, the first three items are required; the MAC label, information label and capability state are required if the relevant POSIX options are in effect; the other specified items are optional. If an item is not available, the function *aud_get_subj_info*() shall return an *aud_info_t* with a zero *aud_info_length* member.

Any existing descriptors shall not be affected by use of this function. Calls on the various *aud_get_**() functions can be interleaved without affecting each other.

This function may cause memory to be allocated. The caller should free any releasable memory, when the record containing *aud_subj_d* is no longer required, by calling *aud_free*() with the *aud_rec_t* for the record (cast to a (*char**)) as an argument.

24.4.24.3 Returns

Upon successful completion, the *aud_get_subj_info*() function returns a non-negative value. This value indicates the number of items of subject attributes in the set.

In the event of failure the *aud_get_subj_info*() function returns a value of −1, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error. The *aud_info_t* referenced by *aud_subj_info_p* shall not be affected if the return value is -1.

24.4.24.4 Errors

If any of the following conditions occur, the *aud_get_subj_info*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] Argument *aud_subj_d* is not a valid descriptor for a set of subject attributes within an audit record.

Argument *item_id* does not identify a valid item from the set of subject attributes.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. −

24.4.24.5 Cross-References

aud_free(), 24.4.14; *aud_get_subj*(), 24.4.23; *aud_put_subj_info*(), 24.4.35; *aud_read*(), 24.4.36; *aud_valid*(), 24.4.40.

24.4.25 Map Text to Audit ID

Function: *aud_id_from_text*()

24.4.25.1 Synopsis

```
#include <sys/audit.h>
aud_id_t aud_id_from_text  (const char *text_p);
```

24.4.25.2 Description

The *aud_id_from_text*() function returns the audit ID identified by the string pointed to by *text_p*. The means by which this information is obtained is unspecified.

24.4.25.3 Returns

Upon successful completion, the *aud_id_from_text*() function returns the audit ID associated with *text_p*. On error, or if the requested entry is not found, a value of ((*aud_id_t*)–1) is returned and *errno* is set to indicate the error.

24.4.25.4 Errors

If any of the following conditions occur, the *aud_id_from_text*() function shall return a value of ((*aud_id_t*)–1) and set *errno* to the corresponding value:

[EINVAL] The *text_p* argument does not identify a valid user. –

24.4.25.5 Cross-References

aud_get_hdr_info(), 24.4.19; *aud_id_to_text*(), 24.4.26; *aud_put_hdr_info*(), 24.4.31.

24.4.26 Map Audit ID to Text

Function: *aud_id_to_text*()

24.4.26.1 Synopsis

```
#include <sys/audit.h>
char *aud_id_to_text (aud_id_t audit_ID, ssize_t *len_p);
```

24.4.26.2 Description

The *aud_id_to_text*() function transforms the *audit_ID* into a human-readable, null terminated character string. The means by which this information is obtained is unspecified. Upon successful completion, the function shall return the address of the string, and set the location pointed to by *len_p* to the length of the string (not including the null terminator).

This function may cause memory to be allocated. The caller should free any releasable memory when the text form of *audit_ID* is no longer required, by calling *aud_free*() with the string address (cast to a (*void*∗)) as an argument.

24.4.26.3 Returns

Upon successful completion, the *aud_id_to_text*() function returns a pointer to a string identifying the user associated with *audit_ID*. On error, or if the requested entry is not found, the caller shall not have to free any releasable memory, (*char*∗)**NULL** is returned, the location pointed to by *len_p* is not changed, and *errno* is set to indicate the error.

24.4.26.4 Errors

If any of the following conditions occur, the *aud_id_to_text*() function shall return (*char* ∗)**NULL** and set *errno* to the corresponding value:

[EINVAL] The *audit_ID* argument does not contain a valid audit identifier.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints.

24.4.26.5 Cross-References

aud_free(), 24.4.14; *aud_get_hdr_info*(), 24.4.19; *aud_id_from_text*(), 24.4.25; *aud_put_hdr_info*(), 24.4.31.

24.4.27 Create a New Audit Record

Function: *aud_init_record*()

24.4.27.1 Synopsis

```
#include <sys/audit.h>

aud_rec_t aud_init_record (void);
```

24.4.27.2 Description

The *aud_init_record*() function returns a pointer to an audit record that is otherwise not in use. The record shall contain no headers or sets of subject, event-specific, or object information.

Upon successful execution of the *aud_init_record*() function, the pointer returned can be used in subsequent calls to the *aud_put_*∗() functions to add information to the record, and in other functions that manipulate audit records, and the record can be written to an audit log by a call of *aud_write*().

Calls to *aud_init_record*() shall not affect the status of any existing records.

This function may cause memory to be allocated. The caller should free any releasable memory, when the record is no longer required, by calling *aud_free*() with the (*void∗*)*aud_rec_t* as an argument.

24.4.27.3 Returns

Upon successful completion, the *aud_init_record*() function returns an *aud_rec_t* pointing to the new record. Otherwise, a value of (*aud_rec_t*)**NULL** shall be returned, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error.

24.4.27.4 Errors

If any of the following conditions occur, the *aud_init_record*() function shall return (*aud_rec_t*)**NULL** and set *errno* to the corresponding value:

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. –

24.4.27.5 Cross-References

aud_dup_record(), 24.4.11; *aud_free*(), 24.4.14; *aud_put_event*(), 24.4.28; *aud_put_hdr*(), 24.4.30; *aud_put_obj*(), 24.4.32; *aud_put_subj*(), 24.4.34; *aud_write*(), 24.4.41.

24.4.28 Add Set of Event-specific Data to Audit Record

Function: *aud_put_event*()

24.4.28.1 Synopsis

```
#include <sys/audit.h>

int aud_put_event (aud_rec_t ar,
                   const aud_evinfo_t *next_p,
                   aud_evinfo_t *new_p);
```

24.4.28.2 Description

The *aud_put_event*() function creates a new set of event-specific data, containing no data items, in an audit record, and returns a descriptor to the set. The function accepts an audit record pointer *ar*, and puts the new set of event-specific data logically before the existing set *next_p* in the record. If *next_p* is **NULL**, then the new set shall be logically the last in the record.

Upon successful execution the *aud_put_event*() function shall return via *new_p* a descriptor for the new set of event-specific data. The descriptor returned by this call can then be used in subsequent calls to *aud_put_event_info*() to add data to this set of event-specific data in the audit record.

Calls to *aud_put_event*() shall not affect the status of any existing descriptors for this or any other audit record. Calls on the various *aud_put_∗*() functions can be interleaved without affecting each other.

This function may cause memory to be allocated. The caller should free any releasable memory, when the record is no longer required, by calling *aud_free*() with the (*void∗*)*aud_rec_t* as an argument.

24.4.28.3 Returns

Upon successful completion, the *aud_put_event*() function returns 0. Otherwise, a value of −1 shall be returned, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error. The audit record referenced by *ar* shall not be affected if the return value is -1.

24.4.28.4 Errors

If any of the following conditions occur, the *aud_put_event*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] Argument *ar* does not point to a valid audit record.

Argument *next_p* is neither **NULL** nor does it indicate an existing set of event-specific data in record *ar*.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. –

24.4.28.5 Cross-References

aud_free(), 24.4.14; *aud_delete_event*(), 24.4.3; *aud_get_event*(), 24.4.16; *aud_init_record*(), 24.4.27; *aud_put_event_info*(), 24.4.29; *aud_valid*(), 24.4.40; *aud_write*(), 24.4.41.

24.4.29 Add Item to Set of Event-specific Data

Function: *aud_put_event_info*()

24.4.29.1 Synopsis

```
#include <sys/audit.h>

int aud_put_event_info (aud_evinfo_t aud_event_d,
                        int position,
                        int item_id,
                        const aud_info_t *aud_event_info_p);
```

24.4.29.2 Description

The *aud_put_event_info*() function adds a data item to a set of event-specific data within an audit record. The function accepts a descriptor for a set of event-specific data *aud_event_d* in an audit record, and puts into the set of event-specific data the item with type, size and address defined in the structure referenced by *aud_event_info_p*. The item shall subsequently be identifiable by *item_id* in calls to functions as the record is manipulated, including after being written to and read back from an audit log; no item identifiable by *item_id* shall already exist in the set of event-specific information.

The *position* argument shall specify either

— the *item_id* of an item that already exists in the set of event-specific data; in this case the new data item shall be placed logically before the existing item

— AUD_LAST_ITEM; in this case the new item shall be logically the last in the set.

After the call of *aud_put_event_info*(), the caller can continue to manipulate the data item indicated by the *aud_info_t*, and the *aud_info_t* itself, and changes to them shall not affect the record unless they are used in a further call to *aud_put_*_info*().

Calls to *aud_put_event_info*() shall not affect the status of any other existing descriptors for this or any other audit record. Calls on the various *aud_put_*_info*() functions can be interleaved without affecting each other.

This function may cause memory to be allocated. The caller should free any releasable memory, when the record is no longer required, by calling *aud_free*() with the (*void**)*aud_rec_t* as an argument.

24.4.29.3 Returns

Upon successful completion, the *aud_put_event_info*() function returns 0. Otherwise, it returns a value of –1, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error. The set of event-specific data referenced by *aud_event_d* shall not be affected if the return value is -1.

24.4.29.4 Errors

If any of the following conditions occur, the *aud_put_event_info*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] Argument *aud_event_d* is not a valid descriptor for a set of event-specific data within an audit record.

Argument *position* is not AUD_LAST_ITEM and does not identify a valid item from the set of event-specific data.

The value of the *aud_info_type* field of the structure referenced by *aud_event_info_p* is invalid.

An item with identifier *item_id* already exists in the set of event-specific data.

The argument *item_id* is equal to AUD_FIRST_ITEM, AUD_NEXT_ITEM, or AUD_LAST_ITEM.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. –

24.4.29.5 Cross-References

aud_delete_event_info(), 24.4.4; *aud_free*(), 24.4.14; *aud_get_event_info*(), 24.4.17; *aud_put_event*(), 24.4.28; *aud_valid*(), 24.4.40; *aud_write*(), 24.4.41.

24.4.30 Add Header to Audit Record

Function: *aud_put_hdr*()

24.4.30.1 Synopsis

```
#include <sys/audit.h>

int aud_put_hdr (aud_rec_t ar,
                  const aud_hdr_t *next_p,
                 aud_hdr_t *new_p);
```

24.4.30.2 Description

The *aud_put_hdr*() function creates a new header, containing no data items, in an audit record, and returns a descriptor to the header. The function accepts an audit record pointer *ar*, and puts the new header logically before the existing header *next_p* in the record. If *next_p* is **NULL**, then the new header shall be logically the last in the record.

Upon successful execution the *aud_put_hdr*() function shall return via *new_p* a descriptor for the new header. The descriptor returned by this call can then be used in subsequent calls to *aud_put_hdr_info*() to add data to this header in the audit record.

Calls to *aud_put_hdr*() shall not affect the status of any existing descriptors for this or any other audit record. Calls on the various *aud_put_**() functions can be interleaved without affecting each other.

This function may cause memory to be allocated. The caller should free any releasable memory, when the record is no longer required, by calling *aud_free*() with the (*void**)*aud_rec_t* as an argument.

24.4.30.3 Returns

Upon successful completion, the *aud_put_hdr*() function returns 0. Otherwise, a value of −1 shall be returned, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error. The audit record referenced by *ar* shall not be affected if the return value is -1.

24.4.30.4 Errors

If any of the following conditions occur, the *aud_put_hdr*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] Argument *ar* does not point to a valid audit record.

Argument *next_p* is neither **NULL** nor does it indicate an existing header in record *ar*.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. −

24.4.30.5 Cross-References

aud_delete_hdr(), 24.4.5; *aud_free*(), 24.4.14; *aud_get_hdr*(), 24.4.18; *aud_init_record*(), 24.4.27; *aud_put_hdr_info*(), 24.4.31; *aud_valid*(), 24.4.40; *aud_write*(), 24.4.41.

24.4.31 Add Item to Audit Record Header

Function: *aud_put_hdr_info*()

24.4.31.1 Synopsis

```
#include <sys/audit.h>

int aud_put_hdr_info (aud_hdr_t aud_hdr_d,
                      int position,
                      int item_id,
                      const aud_info_t *aud_hdr_info_p);
```

24.4.31.2 Description

The *aud_put_hdr_info*() function adds a data item to a header within an audit record. The function accepts a descriptor for a header *aud_hdr_d* in an audit record, and puts into the header the item with type, size and address defined in the structure referenced by *aud_hdr_info_p*. The item shall subsequently be identifiable by *item_id* in calls to functions as the record is manipulated, including after being written to and read back from an audit log; no item identifiable by *item_id* shall already exist in the header.

The *position* argument shall specify either

— the *item_id* of an item that already exists in the header; in this case the new data item shall be placed logically before the existing item

— AUD_LAST_ITEM; in this case the new item shall be logically the last in the header.

After the call of *aud_put_hdr_info*(), the caller can continue to manipulate the data item indicated by the *aud_info_t*, and the *aud_info_t*, and changes to them shall not affect the record unless they are used in a further call to *aud_put_*_info*().

Calls to *aud_put_hdr_info*() shall not affect the status of any other existing descriptors for this or any other audit record. Calls on the various *aud_put_*_info*() functions can be interleaved without affecting each other.

This function may cause memory to be allocated. The caller should free any releasable memory, when the record is no longer required, by calling *aud_free*() with the (*void**)*aud_rec_t* as an argument.

24.4.31.3 Returns

Upon successful completion, the *aud_put_hdr_info*() function returns 0. Otherwise, it returns a value of –1, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error. The header referenced by *aud_hdr_d* shall not be affected if the return value is -1.

24.4.31.4 Errors

If any of the following conditions occur, the *aud_put_hdr_info*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] Argument *aud_hdr_d* is not a valid descriptor for a header within an audit record.

Argument *position* is not AUD_LAST_ITEM and does not identify a valid item from the header.

The value of the *aud_info_type* field of the structure referenced by *aud_hdr_info_p* is invalid.

An item with identifier *item_id* already exists in the header.

The argument *item_id* is equal to AUD_FIRST_ITEM, AUD_NEXT_ITEM, or AUD_LAST_ITEM.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. –

24.4.31.5 Cross-References

aud_delete_hdr_info(), 24.4.6; *aud_free*(), 24.4.14; *aud_get_hdr_info*(), 24.4.20; *aud_put_hdr*(), 24.4.30; *aud_valid*(), 24.4.40; *aud_write*(), 24.4.41.

24.4.32 Add Set of Object Attributes to Audit Record

Function: *aud_put_obj*()

24.4.32.1 Synopsis

```
#include <sys/audit.h>

int aud_put_obj (aud_rec_t ar,
                 const aud_obj_t *next_p,
                 aud_obj_t *new_p);
```

24.4.32.2 Description

The *aud_put_obj*() function creates a new set of object attributes, containing no data items, in an audit record, and returns a descriptor to the set. The function accepts an audit record pointer *ar*, and puts the new set of object attributes logically before the existing set *next_p* in the record. If *next_p* is **NULL**, then the new set shall be logically the last in the record.

Upon successful execution the *aud_put_obj*() function shall return via *new_p* a descriptor for the new set of object attributes. The descriptor returned by this call can then be used in subsequent calls to *aud_put_obj_info*() to add data to this set of object attributes in the audit record.

Calls to *aud_put_obj*() shall not affect the status of any existing descriptors for this or any other audit record. Calls on the various *aud_put_**() functions can be interleaved without affecting each other.

This function may cause memory to be allocated. The caller should free any releasable memory, when the record is no longer required, by calling *aud_free*() with the (*void**)*aud_rec_t* as an argument.

24.4.32.3 Returns

Upon successful completion, the *aud_put_obj*() function returns 0. Otherwise, a value of −1 shall be returned, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error. The audit record referenced by *ar* shall not be affected if the return value is -1.

24.4.32.4 Errors

If any of the following conditions occur, the *aud_put_obj*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] Argument *ar* does not point to a valid audit record.

Argument *next_p* is neither **NULL** nor does it indicate an existing set of object attributes in record *ar*.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. –

24.4.32.5 Cross-References

aud_delete_obj(), 24.4.8; *aud_free*(), 24.4.14; *aud_get_obj*(), 24.4.21; *aud_init_record*(), 24.4.27; *aud_put_obj_info*(), 24.4.33; *aud_valid*(), 24.4.40; *aud_write*(), 24.4.41.

24.4.33 Add Item to Set of Object Attributes

Function: *aud_put_obj_info*()

24.4.33.1 Synopsis

```
#include <sys/audit.h>

int aud_put_obj_info (aud_obj_t aud_obj_d,
                      int position,
                      int item_id,
                      const aud_info_t *aud_obj_info_p);
```

24.4.33.2 Description

The *aud_put_obj_info*() function adds a data item to a set of object attributes within an audit record. The function accepts a descriptor for a set of object attributes *aud_obj_d* in an audit record, and puts into the set of object attributes the item with type, size and address defined in the structure referenced by *aud_obj_info_p*. The item shall subsequently be identifiable by *item_id* in calls to functions as the record is manipulated, including after being written to and read back from an audit log; no item identifiable by *item_id* shall already exist in the set of object attributes.

The *position* argument shall specify either

- the *item_id* of an item that already exists in the set of object attributes; in this case the new data item shall be placed logically before the existing item
- AUD_LAST_ITEM; in this case the new item shall be logically the last in the set.

After the call of *aud_put_obj_info*(), the caller can continue to manipulate the data item indicated by the *aud_info_t*, and the *aud_info_t*, and changes to them shall not affect the record unless they are used in a further call to *aud_put_*_info*().

Calls to *aud_put_obj_info*() shall not affect the status of any other existing descriptors for this or any other audit record. Calls on the various *aud_put_*_info*() functions can be interleaved without affecting each other.

This function may cause memory to be allocated. The caller should free any releasable memory, when the record is no longer required, by calling *aud_free*() with the (*void**)*aud_rec_t* as an argument.

24.4.33.3 Returns

Upon successful completion, the *aud_put_obj_info*() function returns 0. Otherwise, it returns a value of –1, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error. The set of object attributes referenced by *aud_obj_d* shall not be affected if the return value is -1.

24.4.33.4 Errors

If any of the following conditions occur, the *aud_put_obj_info*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] Argument *aud_obj_d* is not a valid descriptor for a set of object attributes within an audit record.

Argument *position* is not AUD_LAST_ITEM and does not identify a valid item from the set of object attributes.

The value of the *aud_info_type* field of the structure referenced by *aud_obj_info_p* is invalid.

An item with identifier *item_id* already exists in the set of object attributes.

The argument *item_id* is equal to AUD_FIRST_ITEM, AUD_NEXT_ITEM, or AUD_LAST_ITEM.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. –

24.4.33.5 Cross-References

aud_delete_obj_info(), 24.4.8; *aud_free*(), 24.4.14; *aud_get_obj_info*(), 24.4.22; *aud_put_obj*(), 24.4.32; *aud_valid*(), 24.4.40; *aud_write*(), 24.4.41.

24.4.34 Add Set of Subject Attributes to Audit Record

Function: *aud_put_subj*()

24.4.34.1 Synopsis

```
#include <sys/audit.h>

int aud_put_subj (aud_rec_t ar,
                  const aud_subj_t *next_p,
                  aud_subj_t *new_p);
```

24.4.34.2 Description

The *aud_put_subj*() function creates a new set of subject attributes, containing no data items, in an audit record, and returns a descriptor to the set. The function accepts an audit record pointer *ar*, and puts the new set of subject attributes logically before the existing set *next_p* in the record. If *next_p* is **NULL**, then the new set shall be logically the last in the record.

Upon successful execution the *aud_put_subj*() function shall return via *new_p* a descriptor for the new set of subject attributes. The descriptor returned by this call can then be used in subsequent calls to *aud_put_subj_info*() to add data to this set of subject attributes in the audit record.

Calls to *aud_put_subj*() shall not affect the status of any existing descriptors for this or any other audit record. Calls on the various *aud_put_**() functions can be interleaved without affecting each other.

This function may cause memory to be allocated. The caller should free any releasable memory, when the record is no longer required, by calling *aud_free*() with the (*void**)*aud_rec_t* as an argument.

24.4.34.3 Returns

Upon successful completion, the *aud_put_subj*() function returns 0. Otherwise, a value of –1 shall be returned, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error. The audit record referenced by *ar* shall not be affected if the return value is -1.

24.4.34.4 Errors

If any of the following conditions occur, the *aud_put_subj*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] Argument *ar* does not point to a valid audit record.

Argument *next_p* is neither **NULL** nor does it indicate an existing set of subject attributes in record *ar*.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. –

24.4.34.5 Cross-References

aud_delete_subj(), 24.4.9; *aud_free*(), 24.4.14; *aud_get_subj*(), 24.4.23; *aud_init_record*(), 24.4.27; *aud_put_subj_info*(), 24.4.35; *aud_valid*(), 24.4.40; *aud_write*(), 24.4.41.

24.4.35 Add Item to Set of Subject Attributes

Function: *aud_put_subj_info*()

24.4.35.1 Synopsis

```
#include <sys/audit.h>

int aud_put_subj_info (aud_subj_t aud_subj_d,
                       int position,
                       int item_id,
                       const aud_info_t *aud_subj_info_p);
```

24.4.35.2 Description

The *aud_put_subj_info*() function adds a data item to a set of subject attributes within an audit record. The function accepts a descriptor for a set of subject attributes *aud_subj_d* in an audit record, and puts into the set of subject attributes the item with type, size and address defined in the structure referenced by *aud_subj_info_p*. The item shall subsequently be identifiable by *item_id* in calls to functions as the record is manipulated, including after being written to and read back from an audit log; no item identifiable by *item_id* shall already exist in the set of subject attributes.

The *position* argument shall specify either

— the *item_id* of an item that already exists in the set of subject attributes; in this case the new data item shall be placed logically before the existing item

— AUD_LAST_ITEM; in this case the new item shall be logically the last in the set.

After the call of *aud_put_subj_info*(), the caller can continue to manipulate the data item indicated by the *aud_info_t*, and the *aud_info_t*, and changes to them shall not affect the record unless they are used in a further call to *aud_put_*_info*().

Calls to *aud_put_subj_info*() shall not affect the status of any other existing descriptors for this or any other audit record. Calls on the various *aud_put_*_info*() functions can be interleaved without affecting each other.

This function may cause memory to be allocated. The caller should free any releasable memory, when the record is no longer required, by calling *aud_free*() with the (*void**)*aud_rec_t* as an argument.

24.4.35.3 Returns

Upon successful completion, the *aud_put_subj_info*() function returns 0. Otherwise, it returns a value of –1, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error. The set of subject attributes referenced by *aud_subj_d* shall not be affected if the return value is -1.

24.4.35.4 Errors

If any of the following conditions occur, the *aud_put_subj_info*() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] Argument *aud_subj_d* is not a valid descriptor for a set of subject attributes within an audit record.

Argument *position* is not AUD_LAST_ITEM and does not identify a valid item from the set of subject attributes.

The value of the *aud_info_type* field of the structure referenced by *aud_subj_info_p* is invalid.

An item with identifier *item_id* already exists in the set of subject attributes.

The argument *item_id* is equal to AUD_FIRST_ITEM, AUD_NEXT_ITEM, or AUD_LAST_ITEM.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints. –

24.4.35.5 Cross-References

aud_delete_subj_info(), 24.4.10; *aud_free*(), 24.4.14; *aud_get_subj_info*(), 24.4.24; *aud_put_subj*(), 24.4.34; *aud_valid*(), 24.4.40; *aud_write*(), 24.4.41.

24.4.36 Read an Audit Record

Function: *aud_read*()

24.4.36.1 Synopsis

```
#include <sys/audit.h>

aud_rec_t aud_read (int filedes);
```

24.4.36.2 Description

This function attempts to read an audit record from the current file offset of the file identified by *filedes*. If the function successfully reads an audit record, the file offset shall be incremented such that a further call of the function will operate on the next audit record in the log. If the file contains records that were written to the system audit log, it is left to the implementation to provide any sequencing information required to ensure that successive calls of *aud_read*() each obtain the "next" available record that was written to the log. If the file contains records that were written to a file, the ordering of the records depends on the position of the file offset at the time *aud_write*() was called. If no more records are in the file, a value of zero is returned. In other cases, if a call is unsuccessful, the effect of further calls is unspecified.

Upon successful completion, the function returns an audit record pointer, *aud_rec_t*, identifying the audit record. The format of the audit record is unspecified, but the *aud_rec_t* can be supplied as an input argument to functions such as the *aud_get_∗*() functions .

Any existing audit record pointers that refer to records from the audit log shall continue to refer to those records.

This function may cause memory to be allocated. The caller should free any releasable memory allocated by this function (and by other functions that are used to process the record), when the caller is finished with the record, by a call to *aud_free*() with the (*void∗*)*aud_rec_t* as an argument.

If {_POSIX_INF} is defined, and {_POSIX_INF_PRESENT} is in effect for the file designated by *filedes*, then the information label of the process shall automatically be set to an implementation-defined value which shall be the same as the value returned by *inf_float(file information label, process information label).*

24.4.36.3 Returns

Upon successful completion, the *aud_read*() function returns an *aud_rec_t* pointing to the record. If there are no more records in the audit log, the caller shall not have to free any releasable memory, and the function returns a value of (*aud_rec_t*) 0. Otherwise, a value of (*aud_rec_t*) −1 is returned, the caller shall not have to free any releasable memory, and *errno* is set to indicate the error.

24.4.36.4 Errors

If any of the following conditions occur, the *aud_read*() function shall return a value of −1 and set *errno* to the corresponding value:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor *filedes* and the process would be delayed in the read operation.

[EBADF] The *filedes* argument is not a valid file descriptor open for reading.

[EINTR] The operation was interrupted by a signal, and no data was transferred.

[EINVAL] The value of the *filedes* argument does not identify an audit log positioned at a valid audit record.

The header of the next record in the audit log identified by *filedes* indicates the record has an AUD_FORMAT or AUD_VERSION that is not supported by the implementation.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints.

24.4.36.5 Cross-References

aud_free(), 24.4.14; *aud_get_event*(), 24.4.16; *aud_get_hdr*(), 24.4.18; *aud_get_obj*(), 24.4.21; *aud_get_subj*(), 24.4.23; *aud_rec_to_text*(), 24.4.37.

24.4.37 Convert an Audit Record to Text

Function: *aud_rec_to_text*()

24.4.37.1 Synopsis

```
#include <sys/audit.h>
char *aud_rec_to_text (aud_rec_t ar, ssize_t *len_p);
```

24.4.37.2 Description

The *aud_rec_to_text*() function transforms the audit record identified by *ar* into a human-readable, null terminated character string. The function shall return the address of the string and, if *len_p* is not **NULL**, set the location pointed to by *len_p* to the length of the string (not including the null terminator).

The text string produced by *aud_rec_to_text*() shall contain a text form of the various sections of the audit record; the record header(s) shall be given first, followed by any set(s) of subject attributes, followed by any set(s) of event specific information, followed by any set(s) of object attributes. Items within each section shall be given in the order they would be returned by the *aud_get_**() functions. Other than this, the form of the text string is unspecified by this standard.

This function may cause memory to be allocated. The caller should free any releasable memory when the text form of the record is no longer required, by calling *aud_free*() with the string address (cast to a (*void**)) as an argument.

24.4.37.3 Returns

Upon successful completion, the *aud_rec_to_text*() function returns a pointer to the text record. Otherwise, a value of **NULL** shall be returned, the caller shall not have to free any releasable memory, and *errno* shall be set to indicate the error.

24.4.37.4 Errors

If any of the following conditions occur, the *aud_rec_to_text*() function shall return a value of **NULL** and set *errno* to the corresponding value:

[EINVAL] The value of the *ar* argument does not identify a valid audit record.

[ENOMEM] The text to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints. –

24.4.37.5 Cross-References

aud_free(), 24.4.14; *aud_read*(), 24.4.36; *aud_valid*(), 24.4.40.

24.4.38 Get the Size of an Audit Record

Function: *aud_size*()

24.4.38.1 Synopsis

```
#include <sys/audit.h>

ssize_t aud_size (aud_rec_t ar);
```

24.4.38.2 Description

The *aud_size*() function returns the total length (in bytes) that the audit record identified by *ar* would use when converted by *aud_copy_ext*(). The audit record *ar* will have been obtained by a previous, successful call to the *aud_read*(), *aud_init_record*() or *aud_dup_record*() function. The *aud_size*() function is used to ascertain the buffer size required to copy an audit record (via *aud_copy_ext*()) into user-allocated space.

24.4.38.3 Returns

Upon successful completion, the *aud_size*() function returns the length of the audit record.

In the event of failure the *aud_size*() function returns a value of −1 and *errno* is set to indicate the error.

24.4.38.4 Errors

If any of the following conditions occur, the *aud_size*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The value of the *ar* argument does not identify a valid audit record.

24.4.38.5 Cross-References

aud_copy_ext(), 24.4.1; *aud_dup_record*(), 24.4.11; *aud_init_record*(), 24.4.27; *aud_read*(), 24.4.36; *aud_valid*(), 24.4.40.

24.4.39 Control the Generation of Audit Records

Function: *aud_switch*()

24.4.39.1 Synopsis

```
#include <sys/audit.h>
aud_state_t aud_switch (aud_state_t aud_state);
```

24.4.39.2 Description

The *aud_switch*() function requests that recording of system-generated audit records for the current process be suspended (using AUD_STATE_OFF) or resumed (using AUD_STATE_ON), or enquires about the current state (using AUD_STATE_QUERY). A request to set the state is advisory and may be ignored either wholly or partially if the auditing policy of the system prohibits the suspension of process auditing. A request to suspend auditing does not affect auditing performed by the *aud_write*() function.

The current state of this switch is inherited by a child if the process calls the *fork*() function.

Appropriate privilege is required to use this function. If {_POSIX_CAP} is defined, then appropriate privilege is provided by the CAP_AUDIT_CONTROL capability.

24.4.39.3 Returns

Upon successful completion, the *aud_switch*() function returns the value of the audit state for the calling process at the start of the call. Otherwise, a value of ((*aud_state_t*)–1) is returned and no change shall be made to the calling process's audit state.

24.4.39.4 Errors

If any of the following conditions occur, the *aud_switch*() function shall return a value of ((*aud_state_t*)–1) and set *errno* to the corresponding value:

[EINVAL] The value of the *aud_state* argument is invalid.

[EPERM] The process does not have appropriate privileges to call this function.

24.4.39.5 Cross-References

aud_write(), 24.4.41.

24.4.40 Validate an Audit Record

Function: *aud_valid*()

24.4.40.1 Synopsis

```
#include <sys/audit.h>
int aud_valid (aud_rec_t ar);
```

24.4.40.2 Description

The *aud_valid*() function checks the audit record referred to by the argument *ar* for validity.

The audit record *ar* shall have been created by a previous call to *aud_init_record*(), *aud_copy_int*() or *aud_dup_record*(), or shall have been read from an audit log by *aud_read*(). The record shall contain at least one header, and the first or only header shall contain at least the following items:

- The event type for the event (identified by a *item_id* of AUD_EVENT_TYPE_ID). The corresponding *aud_info_t* shall have its *aud_info_type* member equal to AUD_TYPE_STRING or AUD_TYPE_INT.
- The audit status for the event (identified by a *item_id* of AUD_STATUS_ID). The corresponding *aud_info_t* shall have its *aud_info_type* member equal to AUD_TYPE_AUD_STATUS.

Calls to *aud_valid*() shall not affect the status of any existing descriptors for this or any other audit record.

24.4.40.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

24.4.40.4 Errors

If any of the following conditions occur, the *aud_valid*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] Argument *ar* does not point to an *aud_rec_t* structure as recognized by the implementation.

One or more of the required entries is not present.

24.4.41 Write an Audit Record

Function: *aud_write*()

24.4.41.1 Synopsis

```
#include <sys/audit.h>
int aud_write (int filedes, aud_rec_t ar);
```

24.4.41.2 Description

The *aud_write*() function writes an application-specific audit record to an audit log. Upon successful completion the audit record identified by *aud_rec_t* shall be written into the audit log file identified by *filedes;* if *filedes* is equal to AUD_SYSTEM_LOG then the record shall be written to the system audit log. If *filedes* is not equal to AUD_SYSTEM_LOG then the record shall be written at the position in the file defined for the POSIX *write*() interface.

The record *ar* shall be a valid audit record, as defined by the *aud_valid*() function. The *aud_write*() call shall not alter the record *ar*; after the call of *aud_write*(), the caller can continue to manipulate the record, and changes to it shall not affect the record reportable from this call of *aud_write*().

If the first or only header in the record does not contain an item with *item_id* set to AUD_TIME_ID, then the time reported by a later call on *aud_get_hdr_info*() for the AUD_TIME_ID field of this header shall be the time at which the *aud_write*() function was executed. If the header does not contain items with *item_id*s set to AUD_FORMAT_ID and AUD_VERSION_ID, then the values of these fields reported for this record shall be the same as those that would be reported for records generated by the system at the time the *aud_write*() function was called. If the header does not contain an item with *item_id* set to AUD_AUD_ID, then the audit ID reported by a later call on *aud_get_hdr_info*() for the AUD_AUD_ID field of this header shall be the audit ID of the user accountable for the current process.

The application may include in the record one or more sets of subject attributes. If the application is auditing an action performed on behalf of a client process, the first set of subject attributes should describe the client, and the header should include the client's audit ID in an item with *item_id* set to AUD_AUD_ID and *aud_info_type* field AUD_TYPE_AUD_ID. If the application is writing a record that was read from another log, the record will already contain one or more sets of subject attributes. If the record does not contain any sets of subject attributes, then later calls to *aud_get_subj*() and *aud_get_subj_info*() for this record shall report one set of subject attributes, containing details of the process that invoked *aud_write*().

If the record has been constructed by the application, later reading of the record using *aud_read*(), *aud_get_**() and the *aud_get_*_info*() functions shall report the items from the record *ar* in the logical order specified by the *aud_put_**() and *aud_put_*_info*() calls used to construct the record. The content of the record, reported by calls to the *aud_read*(), *aud_get_∗*() and the *aud_get_∗_info*() functions, shall be the content at the time *aud_write*() was invoked.

If {_POSIX_INF} is defined, and {_POSIX_INF_PRESENT} is in effect for the log designated by *filedes*, then the information label of the log shall automatically be set to an implementation-defined value which should be the same as the value returned by *inf_float(process information label, log information label).*

Appropriate privilege is required to use *aud_write*() to write to the system audit log. If {_POSIX_CAP} is defined then appropriate privilege is provided by the CAP_AUDIT_WRITE capability.

24.4.41.3 Returns

Upon successful completion, the *aud_write*() function returns a value of 0 and the specified record is written to the specified audit log. Otherwise, a value of −1 is returned and *errno* is set to indicate the error, and the specified record is not written to the specified audit log.

24.4.41.4 Errors

If any of the following conditions occur, the *aud_write*() function shall return −1 and set *errno* to the corresponding value:

[EBADF] The value of the *filedes* argument is not a valid file descriptor open for writing and is not AUD_SYSTEM_LOG.

[EINTR] The operation was interrupted by a signal, and no data was transferred.

[EINVAL] The value of the *ar* argument does not identify a valid audit record.

The audit record identified by *ar* does not contain the required header data.

[EPERM] The process does not have appropriate privilege to perform the requested operation.

24.4.41.5 Cross-References

aud_dup_record(), 24.4.11; *aud_get_id*(), 24.4.20; *aud_init_record*(), 24.4.27; *aud_read*(), 24.4.36; *aud_switch*(), 24.4.39; *aud_valid*(), 24.4.40.

Section 25: Capabilities

25.1 General Overview

This section defines a set of portable interfaces that permit one or more capabilities to be associated with a process or file, for the capabilities associated with a process to be enabled or disabled, and for a set of these capabilities to be passed on to the next program associated with a process. This specification also identifies a minimum set of capabilities required for the support of portable security-relevant programs, and specifies the circumstances in POSIX.1 under which these capabilities shall be used. Support for the interfaces defined in this section is optional, but shall be provided if the symbol {_POSIX_CAP} is defined.

POSIX.1 specifies that certain actions require a process to possess appropriate privilege in order to complete those actions. This section specifies the names of the capabilities which constitute appropriate privilege to perform those actions on a system that supports the POSIX Capability Option.

This section describes a set of interfaces by which *capabilities* may be associated with a process and the method by which a process's *capabilities* are derived. Specific capabilities of a process that exec's a particular file may be revoked, inherited from the previous process image, or granted to the process, depending on the value(s) of the file *capability state* of the file and the process *capability state* of the previous process image.

The set of interfaces defined by this standard provide the means to support the *principle of least privilege*. Note, however, it does not require that a conforming implementation actually enforce a least privilege (least capability) security policy. The capability related interfaces and semantics specified in this standard permit individual capabilities to be defined down to a per-function level and permit them to be granted or denied to the granularity of an individual process image. They also permit a process image to control the effectiveness of the capabilities assigned to it during its execution. These capabilities are necessary, but not sufficient, for the implementation of a least privilege security policy. Implementations may extend the capability interfaces such that use of and/or access to capabilities by programs are further constrained.

This section also defines a minimal number of capabilities that shall be supported by conforming implementations. Implementations may define additional capabilities that affect the behavior of POSIX defined and/or other system functions.

25.1.1 Major Features

25.1.1.1 Task Bounding of Capability

Another major characteristic of the capability interfaces is that capabilities may be bounded in the extent of code they are effective over. That is, they can be enabled for only as long as they are actually needed to perform a task (or tasks), and then disabled. The extent of code that could exercise a particular capability can be bounded both at the program level and within a particular program.

At the program level, a process may be assigned or denied specific capabilities by setting the capability flags and attributes associated with the program file. When the file is executed, these flags and attributes are examined by *exec*(). The *exec*() function then modifies the capability state of the process in a specific manner according to those flags and attributes. In this way, a process may gain additional capabilities by executing certain programs, or it may lose capabilities it currently possesses.

Within itself, a process image may enable, disable, or permanently revoke its capabilities. For example, a process modifies the effective flag of a given capability to either enable or disable that capability. This flag shall be set in order for the capability to be available for use. A process image permanently, i.e., for the duration of that process image, revokes a given capability by resetting both the effective and permitted flags for that capability. More information on these two flags is provided in section 25.1.1.4 below.

25.1.1.2 Capability Inheritance

Following the *exec*() of a program, the capabilities that have their permitted flags set in the new process image depend on the capability states of both the previous process image and the exec'd program file. Each capability marked as permitted may have been forced to be set by the program file or inherited from the previous image (if the capability attributes of the program file allow the inheritance).

Inheritance permits a process image to request that all, some or none of its capabilities be passed to the next process image, subject to restrictions set by the system security policy. For example, a backup program may *exec*() the pax utility, granting it the capabilities required to read all files in a file system (providing it is allowed to inherit those capabilities). However, the same backup utility may *exec*() other utilities to which it does not pass any capabilities.

25.1.1.3 Capability Flags

The capability flags defined by this standard are permitted, effective, and inheritable. These flags apply to each capability separately, and together their values determine a capability state. Capability states shall be assignable to at least two entities: processes and files. Implementations may define additional flags for capabilities and may provide for the assignment of capability states to additional entities.

25.1.1.4 Capability Flags and Processes

The capability state is the attribute of a process which contains the value of all of the process's capability flags. A conforming implementation shall support the assignment of a capability state to processes. When the process permitted flag for a capability is set, a process shall be able to set all its flags defined by this standard for that capability. A process shall be able to clear any flag for any of its capabilities regardless of the state of the permitted flags. A process can exercise a particular capability only when that capability's process effective flag is set. The process effective flag shall be the only flag considered by system functions when determining if the process possesses appropriate privilege. The process inheritable flag is used by the *exec*() function when determining the capability flags of the new process image. A capability may be passed from one process image to the next through an *exec*() only if the inheritable flag of that capability is set. This inheritance may or may not actually occur, depending on the capability state of the file as described in the next section. The new process image may also acquire capabilities based upon the capability state of the file used to create the new process image, as defined in section 3.1.2.2.

25.1.1.5 Capability Flags and Files

The capability state is the attribute of a file which contains the value of all of the file's capability flags. A conforming implementation shall support the assignment of capability states to files. The purpose of assigning capability states to files is to provide the *exec*() function with information regarding the capabilities that any process image created with the program in the file is capable of dealing with and have been granted by some authority to use.

If *pathconf*() indicates that {_POSIX_CAP_PRESENT} is not in effect for a file, then the capability state of that file shall be implementation-defined.

25.1.1.6 File System Support of Capability

The capability state of a process after an *exec*() of a file for which the value of the pathname variable {_POSIX_CAP_PRESENT} is zero shall be implementation-defined.

25.1.1.7 Application

The POSIX.1 functions listed in Table 25-1 are affected by the capability functionality defined in this standard.

Table 25-1 – POSIX.1 Functions Covered by Capability Policies

Existing Function	POSIX.1 Section
chmod	5.6.4
chown	5.6.5
creat	5.3.2
exec	3.1.2
fpathconf	5.7.1
fstat	5.6.2
kill	3.3.2
link	5.3.4
mkdir	5.4.1
mkfifo	5.4.2
open	5.3.1
pathconf	5.7.1
read	6.4.1
rename	5.5.3
rmdir	5.5.2
setgid	4.2.2
setuid	4.2.2
stat	5.6.2
unlink	5.5.1
utime	5.6.6
write	6.4.2

New Function	POSIX.1e Synopsis
acl_delete_def_fd	Delete a Default ACL by File Descriptor
acl_delete_def_file	Delete a Default ACL by Filename
acl_get_fd	Get an ACL by File Descriptor
acl_get_file	Get an ACL by Filename
acl_set_fd	Set an ACL by File Descriptor
acl_set_file	Set an ACL Filename
aud_switch	Control the Generation of Audit Records
aud_write	write an application-generated record to an audit log
inf_get_fd	Get the Information Label of a File Identified by File Descriptor
inf_get_file	Get the Information Label of a File Identified by File Pathname
inf_set_fd	Set the Information Label of a File Identified by File Descriptor
inf_set_file	Set the Information Label of a File Identified by File Pathname
inf_set_proc	Set the Process Information Label
mac_get_fd	Get the Label of a File Designated by File Descriptor
mac_get_file	Get the Label of a File Designated by File Pathname
mac_set_fd	Set the Label of a File Designated by File Descriptor
mac_set_file	Set the Label of a File
mac_set_proc	Set the Process Label
cap_get_fd	Get the Capability State of an Open File
cap_get_file	Read the Capability State of a File
cap_set_fd	Set the Capability State of an Open File
cap_set_file	Write the Capability State of a File

25.1.2 Capability Functions

Functional interfaces are defined to manipulate capability states, to assign them to files and processes and to obtain them for files and processes. These functions comprise a set of interfaces that permit portable programs to manipulate their own capability state and a minimal set of interfaces to manipulate the capability state of files.

Four groups of functions are defined to:

(1) manage the working storage area used by capability states

(2) manipulate the capability flags within a capability state

(3) manipulate (read and write) a capability state on a file or process

(4) translate a capability state into different formats.

25.1.2.1 Capability Data Object Storage Management

The capabilities associated with a file or process are never edited directly. Rather, a working storage area is allocated to contain a representation of the capability state. Capabilities are edited and manipulated only within this working storage area. Once the editing of the capability state is completed, the updated capability state is used to replace the capability state associated with the file or process.

Working storage is allocated as needed by the capability manipulation functions. The *cap_init*() and *cap_dup*() functions also allow the application to allocate working storage for the creation of a new capability state. The working storage area may be released by the application once the capability state is no longer needed by use of the *cap_free*() function.

cap_dup() Duplicates a capability state in a working storage area

cap_free() Releases working storage area previously allocated by capability manipulation functions

cap_init() Allocates and initializes working storage area for a capability state.

25.1.2.2 Capability Data Object Manipulation

These functions manipulate capability state only in working storage not associated with file or process.

cap_get_flag() Obtain the value of a specific flag for a specific capability.

cap_set_flag() Sets the value of a specific flag for a specific capability.

cap_clear() Initializes or resets a capability state such that all flags for all capabilities are cleared.

25.1.2.3 Capability Manipulation on an Object

These functions read the capability state of a file or process into working storage and write the capability state in working storage to a file or process.

cap_get_fd() Reads the capability state associated with a file descriptor into working storage.

cap_get_file() Reads the capability state associated with a file into working storage.

cap_get_proc() Reads the capability state associated with the calling process into working storage.

cap_set_fd() Writes the capability state in working storage to the object associated with a file descriptor.

cap_set_file() Writes the capability state in working storage to a file.

cap_set_proc() Sets the process capability state of the calling process to a capability state in working storage.

25.1.2.4 Capability State Format Translation

This standard defines three different representations for a capability state: –

external form The exportable, contiguous, persistent representation of a capability state in user-managed space.

internal form The internal representation of a capability state in working storage managed by the capability functions.

text form The structured text representation of a capability state.

These functions translate a capability state from one representation into another. +

cap_copy_ext() Translates an internal form of a capability state to the external form of a capability state.

cap_copy_int() Translates the external form of a capability state to the internal form of a capability state.

cap_from_text() Translates a text form of a capability state to the internal form of a capability state.

cap_size() Returns the size in bytes required to store the external form of a capability state that is the result of an *cap_copy_ext*().

cap_to_text() Translates an internal form of a capability state to the text form of a capability state.

25.2 Header

Some of the data types used by the capability functions are not defined as part of this standard, but shall be implementation-defined. These types shall be defined in the header `<sys/capability.h>`, which contains definitions for at least the types shown in the following table.

Table 25-2 – Capability Data Types

Defined Type	Description
cap_flag_t	Used to identify capability flags. This data type is exportable data.
cap_t	Used as a pointer to an opaque data object that is used as capability state working storage. This data type is non-exportable data.
cap_flag_value_t	Used to specify the value of capability flags. This data type is exportable data.
cap_value_t	Used to identify capabilities. This data type is exportable data.

The symbolic constants specified in the remainder of this section shall be defined in the header `<sys/capability.h>`.

Table 25-3 contains *cap_flag_t* values for the *cap_get_flag*() and *cap_set_flag*() functions.

Table 25-3 – cap_flag_t Values

Constant	Description
CAP_EFFECTIVE	Specifies the effective flag.
CAP_INHERITABLE	Specifies the inheritable flag.
CAP_PERMITTED	Specifies the permitted flag.

Table 25-4 contains *cap_flag_value_t* values for the *cap_get_flag*() and *cap_set_flag*() functions.

Table 25-4 – cap_flag_value_t Values

Constant	Description
CAP_CLEAR	The flag is cleared/disabled.
CAP_SET	The flag is set/enabled.

Table 25-5 through Table 25-8 contains *cap_value_t* values for *cap_get_flag*() and *cap_set_flag*(). Note that the description of each capability specifies exactly what restriction the capability is intended to affect. Possession of a capability that overrides one restriction should not imply that any other restrictions are overridden. For example, possession of the CAP_DAC_OVERRIDE capability should not imply that a process can read files with MAC labels that dominate that of the process, nor should it override any restrictions that the file owner ID match the user ID of the process.

If the {_POSIX_CAP} system configuration option is defined, the implementation shall define at least the following set of *cap_value_t* values:

Table 25-5 – cap_value_t Values

Constant	Description
CAP_CHOWN	In a system in which the {_POSIX_CHOWN_RESTRICTED} option is defined, this capability shall override the restriction that a process cannot change the user ID of a file it owns and the restriction that the group ID supplied to the *chown*() function shall be equal to either the group ID or one of the supplementary group IDs of the calling process.
CAP_DAC_EXECUTE	This capability shall override file mode execute access restrictions when accessing an object, and, if the {_POSIX_ACL} option is defined, this capability shall override the ACL execute access restrictions when accessing an object.
CAP_DAC_WRITE	This capability shall override file mode write access restrictions when accessing an object, and, if the {_POSIX_ACL} option is defined, this capability shall override the ACL write access restrictions when accessing an object.
CAP_DAC_READ_SEARCH	This capability shall override file mode read and search access restrictions when accessing an object, and, if the {_POSIX_ACL} option is defined, this capability shall override the ACL read and search access restrictions when accessing an object.

CAP_FOWNER	This capability overrides the requirement that the user ID associated with a process be equal to the file owner ID, except in the cases where the CAP_FSETID capability is applicable. In general, this capability, when effective, will permit a process to perform all the functions that any file owner would have for their files.
CAP_FSETID	This capability shall override the following restrictions: that the effective user ID of the calling process shall match the file owner when setting the set-user-ID (S_ISUID) and set-group-ID (S_ISGID) bits on that file; that the effective group ID or one of the supplementary group IDs of the calling process shall match the group ID of the file when setting the set-group-ID bit of that file; and that the set-user-ID and set-group-ID bits of the file mode shall be cleared upon successful return from *chown*().
CAP_KILL	This capability shall override the restriction that the real or effective user ID of a process sending a signal must match the real or effective user ID of the receiving process.
CAP_LINK_DIR	This capability overrides the restriction that a process cannot create or delete a hard link to a directory.

CAP_SETFCAP	This capability shall override the restriction that a process cannot set the file capability state of a file.
CAP_SETGID	This capability shall override the restriction in the *setgid*() function that a process cannot change its real group ID or change its effective group ID to a value other than its real group ID. If {_POSIX_SAVED_IDS} is defined, then this capability also overrides any restrictions on setting the saved set-group-ID to a value other than the current real or saved set-group ID.
CAP_SETUID	This capability shall override the restriction in the *setuid*() function that a process cannot change its real user ID or change its effective user ID to a value other than the current real user ID. If {_POSIX_SAVED_IDS} is defined, then this capability also overrides any restrictions on setting the saved set-user-ID.

If the {_POSIX_MAC} system configuration option is defined, the implementation – shall define at least the following set of *cap_value_t* values:

Table 25-6 – cap_value_t Values for Mandatory Access Controls

Constant	Description
CAP_MAC_DOWNGRADE	This capability shall override the restriction that no process may downgrade the MAC label of a file.
CAP_MAC_READ	This capability shall override mandatory read access restrictions when accessing objects.
CAP_MAC_RELABEL_SUBJ	This capability shall override the restriction that a process may not modify its own MAC label.
CAP_MAC_UPGRADE	This capability shall override the restriction that no process may upgrade the MAC label of a file.
CAP_MAC_WRITE	This capability shall override mandatory write access restrictions when accessing objects.

If the {_POSIX_INF} system configuration option is defined, the implementation shall define at least the following set of *cap_value_t* values:

Table 25-7 – cap_value_t Values for Information Labels

Constant	Description
CAP_INF_NOFLOAT_OBJ	This capability shall override the requirement that an object's information label shall automatically float when a write operation is performed by a process.
CAP_INF_NOFLOAT_SUBJ	This capability shall override the requirement that a process' information label shall automatically float when a read or execute operation is performed on an object.
CAP_INF_RELABEL_OBJ	This capability shall override the restriction against changing the information label of an object.
CAP_INF_RELABEL_SUBJ	This capability shall override the restriction that a process may not modify its own information label in violation of the information labeling policy.

If the {_POSIX_AUD} system configuration option is defined, the implementation shall define at least the following set of *cap_value_t* values:

Table 25-8 – cap_value_t Values for Audit

Constant	Description
CAP_AUDIT_CONTROL	This capability shall override the restriction that a process cannot modify audit control parameters.
CAP_AUDIT_WRITE	This capability shall override the restriction that a process cannot write data into the system audit trail.

The symbolic constants defined in this section shall be implementation-defined unique values.

25.3 Text Form Representation

The text form of a capability state shall consist of one or more `clauses` contained within a single, **NULL**-terminated character string. `Clauses` are separated by whitespace characters. Each valid *clause* identifies a capability or a set of capabilities, an *op* (operation), and one or more *flags* that the operation applies to:

clause [SEP *clause*]...

where *clause* has the following format:

[*caplist*] *actionlist*

and SEP is ":" or any whitespace character.

caplist has the following format:

capability_name[,*capability_name*]... .

actionlist has the following format:

op [*flags*] [*op* [*flags*]]...

op is one of "`=`", "–", or "`+`".

flags is a token consisting of one or more alphabetic characters.

The string shall be interpreted in order, e.g., the *op* specified in a later *clause* shall supplant or modify *op* that apply to the same capabilities in an earlier *clause*.

The *capability_name* symbols shall specify which capability or capabilities the *clause* is to operate on. The symbols to be used are those defined in the `capability.h` header file for each capability, e.g., "CAP_FOWNER", "CAP_SETUID", etc. More than one *capability_name* may be specified in a `clause` by separating them with a comma. A *capability_name* consisting of the string "`all`" shall be equivalent to a list containing every capability defined by the implementation. *Capability_names* are case insensitive on input, and the case used for output shall be implementation defined.

The *flags* symbols `e`, `i` and `p` shall represent the *effective*, *inheritable* and *permitted* capability flags, respectively. All lowercase characters for use as *flags* symbols are reserved for use by future versions of this standard. Implementations may define uppercase characters for *flags* to represent implementation-defined flags.

If multiple *actionlists* are grouped with a single *caplist* in the grammar, each *actionlist* shall be applied in the order specified with that *caplist*. The *op* symbols shall represent the operation performed, as follows:

+ If *flags* is not specified and *caplist* contains one or more capability names, the + operation shall not change the capability state; else,

  if *caplist* is not specified, this shall be considered an error; otherwise

  if *caplist* is specified as "`all`", the capability flags represented by *flags* for all capabilities defined for the target by the implementation shall be set; otherwise,

  the flags specified in *flags* for all the capabilities specified in *caplist* shall be set.

− If *flags* is not specified and *caplist* contains one or more capability names, the − operation shall not change the target capability state; else,

  if *caplist* is not specified or is specified as "`all`", the capability flags represented by *flags* for all capabilities defined by the implementation shall be cleared; otherwise,

  the capability flags specified in *flags* for all the capabilities specified in *caplist* shall be cleared.

= Clear all the capability flags for the capabilities specified in *caplist*, or, if no *caplist* is specified, clear all capability flags for all capabilities defined by the implementation, then:

  if *flags* is not specified, the = operation shall make no further modification to the target capability state; else,

  if *caplist* is not specified or is specified as "`all`", the capability flags represented by *flags* shall be set for all capabilities defined for the target by the implementation; otherwise,

  the capability flags represented by *flags* shall be set for all the capabilities specified in *caplist* in the target capability state.

25.3.1 Grammar

The grammar and lexical conventions in this subclause describe the syntax for the textual representation of capability state. The general conventions for this style of grammar are described in POSIX.2, "Grammar Conventions", 2.1.2. A valid *capability state* can be represented as the nonterminal symbol *capability state* in the grammar. The formal syntax description in this grammar shall take precedence over the textual descriptions in this clause.

The lexical processing shall be based on single characters except for capability name recognition. Implementations need not allow whitespace characters within the single argument being processed.

```
%start          capability_state
%%
capability_state  :  clause
                  |  capability_state clause
                  ;

clause            :  actionlist
                  |  caplist actionlist
                  ;

caplist           :  capability_name
                  |  caplist ',' capability_name
                  ;

actionlist        :  action
                  |  actionlist action
                  ;

action            :  op
                  |  op flaglist
                  ;

op                :  ''+''
                  |  ''−''
                  |  ''=''
                  ;

flaglist          :  flag
                  |  flaglist flag
                  ;

flag              :  ''e''
                  |  ''i''
                  |  ''p''
                  ;
```

25.4 Functions

The functions in this section comprise the set of services that permit a process image to acquire, manipulate, and pass capabilities on to new process images they create. Support for the capability facility functions identified in this section is optional. If the symbol {_POSIX_CAP} is defined, the implementation supports the capability option and all of the capability functions shall be implemented as described in this section. If {_POSIX_CAP} is not defined, the result of calling any of these functions is unspecified.

The error [ENOTSUP] shall be returned in those cases where the system supports the capability facility but the particular capability operation cannot be applied because of restrictions imposed by the implementation.

25.4.1 Initialize a Capability State in Working Storage

Function: *cap_clear*()

25.4.1.1 Synopsis

```
#include <sys/capability.h>

int cap_clear (cap_t cap_p);
```

25.4.1.2 Description

The function *cap_clear*() shall initialize the capability state in working storage identified by *cap_p* so that all capability flags for all capabilities defined in the implementation shall be cleared.

25.4.1.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

25.4.1.4 Errors

If any of the following conditions occur, the *cap_clear*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The value of the *cap_p* argument does not refer to a capability state in working storage.

25.4.1.5 Cross-References

cap_init(), 25.4.11; *cap_set_flag*(), 25.4.14.

25.4.2 Copy a Capability State From System to User Space

Function: *cap_copy_ext*()

25.4.2.1 Synopsis

```
#include <sys/capability.h>

ssize_t cap_copy_ext (void *ext_p, cap_t cap_p, ssize_t size)
```

25.4.2.2 Description

The *cap_copy_ext*() function shall copy a capability state in working storage, identified by *cap_p*, from system managed space to user-managed space (pointed to by *ext_p*) and returns the length of the resulting data record. The *size* parameter represents the maximum size, in bytes, of the resulting data record. The *cap_copy_ext*() function will do any conversions necessary to convert the capability state from the unspecified internal format to an exportable, contiguous, persistent data record. It is the responsibility of the user to allocate a buffer large enough to hold the copied data. The buffer length required to hold the copied data may be obtained by a call to the *cap_size*() function.

25.4.2.3 Returns

Upon successful completion, the function shall return the number of bytes placed in the user managed space pointed to by *ext_p*. Otherwise, a value of (*ssize_t*)–1 shall be returned and *errno* shall be set to indicate the error.

25.4.2.4 Errors

If any of the following conditions occur, the *cap_copy_ext*() function shall return (*ssize_t*)–1 and set *errno* to the corresponding value:

[EINVAL] The value of the *cap_p* argument does not refer to a capability state in working storage or the value of the *size* argument is zero or negative.

[ERANGE] The *size* parameter is greater than zero, but smaller than the length of the contiguous, persistent form of the capability state.

25.4.2.5 Cross-References

cap_copy_int() 25.4.3.

25.4.3 Copy a Capability State From User to System Space

Function: *cap_copy_int*()

25.4.3.1 Synopsis

```
#include <sys/capability.h>

cap_t cap_copy_int (const void *ext_p)
```

25.4.3.2 Description

The *cap_copy_int*() function shall copy a capability state from a capability data record in user-managed space to a new capability state in working storage, allocating any memory necessary, and returning a pointer to the newly created capability state. The function shall initialize the capability state and then copy the capability state from the record pointed to by *ext_p* into the capability state, converting, if necessary, the data from a contiguous, persistent format to an unspecified internal format. Once copied into internal format, the object can be manipulated by the capability state manipulation functions.

Note that the record pointed to by *ext_p* must have been obtained from a previous, successful call to *cap_copy_ext*() for this function to work successfully.

This function may cause memory to be allocated. The caller should free any releasable memory, when the capability state in working storage is no longer required, by calling *cap_free*() with the cap_t as an argument.

25.4.3.3 Returns

Upon successful completion, the *cap_copy_int*() function returns a pointer to the newly created capability state in working storage. Otherwise, a value of (*cap_t*)**NULL** shall be returned and *errno* shall be set to indicate the error.

25.4.3.4 Errors

If any of the following conditions occur, the *cap_copy_int*() function shall return (*cap_t*)**NULL** and set *errno* to the corresponding value:

[EINVAL] The value of the *ext_p* argument does not refer to a capability data record as defined in section 25.3.

[ENOMEM] The capability state to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

25.4.3.5 Cross-References

cap_copy_ext(), 25.4.2; *cap_free*(), 25.4.5; *cap_init*(), 25.4.11.

25.4.4 Duplicate a Capability State in Working Storage

Function: *cap_dup*()

25.4.4.1 Synopsis

```
#include <sys/capability.h>

cap_t cap_dup (cap_t cap_p);
```

25.4.4.2 Description

The *cap_dup*() function returns a duplicate capability state in working storage given the source object *cap_p*, allocating any memory necessary, and returning a pointer to the newly created capability state. Once duplicated, no operations on either capability state shall affect the other in any way.

This function may cause memory to be allocated. The caller should free any releasable memory, when the capability state in working storage is no longer required, by calling *cap_free*() with the cap_t as an argument.

25.4.4.3 Returns

Upon successful completion, the *cap_dup*() function returns a pointer to the newly created capability state in working storage. Otherwise, a value of (*cap_t*)**NULL** shall be returned and *errno* shall be set to indicate the error.

25.4.4.4 Errors

If any of the following conditions occur, the *cap_dup*() function shall return (*cap_t*)**NULL** and set *errno* to the corresponding value:

[EINVAL] The value of the *cap_p* argument does not refer to a capability state in working storage.

[ENOMEM] The capability state to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

25.4.4.5 Cross-References

cap_free(), 25.4.5.

25.4.5 Release Memory Allocated to a Capability State in Working Storage

Function: *cap_free*()

25.4.5.1 Synopsis

```
#include <sys/capability.h>
int cap_free (void *obj_d);
```

25.4.5.2 Description

The function *cap_free*() shall free any releasable memory currently allocated to the capability state in working storage identified by *obj_d*. The *obj_d* argument may identify either a cap_t entity, or a char * entity allocated by the *cap_to_text*() function.

25.4.5.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

25.4.5.4 Errors

If any of the following conditions occur, the *cap_free*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The value of the *obj_d* argument does not refer to memory recognized as releasable by the implementation.

25.4.5.5 References

cap_copy_int(), 25.4.3; *cap_dup*(), 25.4.4; *cap_from_text*(), 25.4.6; *cap_get_fd*(), 25.4.7; *cap_get_file*(), 25.4.8; *cap_get_proc*(), 25.4.10; *cap_init*(), 25.4.11; *cap_to_text*(), 25.4.17.

25.4.6 Convert Text to a Capability State in Working Storage

Function: *cap_from_text*()

25.4.6.1 Synopsis

```
#include <sys/capability.h>

cap_t cap_from_text (const char *buf_p);
```

25.4.6.2 Description

This function shall allocate and initialize a capability state in working storage. It shall then set the contents of this newly-created capability state to the state represented by the human–readable, null terminated character string pointed to by *buf_p*. It shall then return a pointer to the newly created capability state.

This function may cause memory to be allocated. The caller should free any releasable memory, when the capability state in working storage is no longer required, by calling *cap_free*() with the cap_t as an argument.

The function shall recognize and correctly parse any string that meets the specification in 25.3. The function shall return an error if the implementation cannot parse the contents of the string pointed to by *buf_p* or does not recognize any *capability_name* or flag character as valid. The function shall also return an error if any flag is both set and cleared within a single clause.

25.4.6.3 Returns

Upon successful completion, a non-**NULL** value is returned. Otherwise, a value of (*cap_t*)**NULL** shall be returned and *errno* shall be set to indicate the error.

25.4.6.4 Errors

If any of the following conditions occur, the *cap_from_text*() function shall return (*cap_t*)**NULL** and set *errno* to the corresponding value:

[EINVAL] The *buf_p* argument does not refer to a character string as defined in section 25.3, the string pointed to by *buf_p* is not parseable by the function, the text string contains a *capability_name* or a flag character that the implementation does not recognize as valid.

[ENOMEM] The capability state to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

25.4.6.5 Cross-References

cap_to_text(), 25.4.17; *cap_free*(), 25.4.5; *cap_init*(), 25.4.11; *cap_set_flag*(), 25.4.14.

25.4.7 Get the Capability State of an Open File

Function: *cap_get_fd*()

25.4.7.1 Synopsis

```
#include <sys/capability.h>

cap_t cap_get_fd (int fd);
```

25.4.7.2 Description

The function *cap_get_fd*() shall allocate a capability state in working storage and set it to represent the capability state of the file open on the descriptor *fd*, then return a pointer to the newly created capability state.

A process can get the capability state of any regular file for which the process has a valid file descriptor. If the file open on the descriptor *fd* is not a regular file, then *cap_get_fd*() shall return an error. If {_POSIX_CAP_PRESENT} is not in effect for the file, then the results of *cap_get_fd*() shall be implementation-defined.

If {_POSIX_MAC} is defined, the process must also have mandatory access control read access to the file.

This function may cause memory to be allocated. The caller should free any releasable memory, when the capability state in working storage is no longer required, by calling *cap_free*() with the cap_t as an argument.

25.4.7.3 Returns

Upon successful completion, this function returns a non-**NULL** value. Otherwise, a value of (*cap_t*)**NULL** shall be returned and *errno* shall be set to indicate the error.

25.4.7.4 Errors

If any of the following conditions occur, the *cap_get_fd*() function shall return (*cap_t*)**NULL** and set *errno* to the corresponding value:

[EACCES] If the {_POSIX_MAC} system configuration option is enabled, MAC read access to the file is denied.

[EBADF] The *fd* argument is not a valid open file descriptor.

[EINVAL] The file open on *fd* is not a regular file.

[ENOMEM] The capability state to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

25.4.7.5 Cross-References

cap_init(), 25.4.11; *cap_free*(), 25.4.5; *cap_get_file*(), 25.4.8; *cap_set_fd*(), 25.4.12.

25.4.8 Get the Capability State of a File

Function: *cap_get_file*()

25.4.8.1 Synopsis

```
#include <sys/capability.h>

cap_t cap_get_file (const char *path_p);
```

25.4.8.2 Description

The function *cap_get_file*() shall allocate a capability state in working storage and set it to be equal to the capability state of the pathname pointed to by *path_p*, then return a pointer for the newly created capability state in working storage.

A process can get the capability state of any regular file for which the process has search access to the path specified. If the file pointed to by *path_p* is not a regular file, then *cap_get_file*() shall return an error. If {_POSIX_CAP_PRESENT} is not in effect for the file, then the results of *cap_get_file*() shall be implementation-defined.

If {_POSIX_MAC} is defined, the process must also have MAC read access to the file.

This function may cause memory to be allocated. The caller should free any releasable memory, when the capability state in working storage is no longer required, by calling *cap_free*() with the cap_t as an argument.

25.4.8.3 Returns

Upon successful completion, this function returns a non-**NULL** value. Otherwise, a value of (*cap_t*)**NULL** shall be returned and *errno* shall be set to indicate the error.

25.4.8.4 Errors

If any of the following conditions occur, the *cap_get_file*() function shall return (*cap_t*)**NULL** and set *errno* to the corresponding value:

- [EACCES] Search permission is denied for a component of the path prefix, or, if {_POSIX_MAC} is defined, MAC read access to the file *path_p* is denied.
- [EINVAL] The file pointed to by *path_p* is not a regular file.
- [ENAMETOOLONG] The length of the *path_p* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {POSIX_NO_TRUNC} is in effect.
- [ENOENT] The named file does not exist or the *path_p* argument points to an empty string.
- [ENOMEM] The capability state to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.
- [ENOTDIR] A component of the path prefix is not a directory.

25.4.8.5 Cross-References

cap_free(), 25.4.5; *cap_init*(), 25.4.11; *cap_set_file*(), 25.4.13; *cap_get_fd*(), 25.4.7.

25.4.9 Get the Value of a Capability Flag

Function: *cap_get_flag*()

25.4.9.1 Synopsis

```
#include <sys/capability.h>

int cap_get_flag (cap_t cap_p,
                  cap_value_t cap,
                  cap_flag_t flag,
                  cap_flag_value_t *value_p);
```

25.4.9.2 Description

The function *cap_get_flag*() shall obtain the current value of the capability flag *flag* of the capability *cap* from the capability state in working storage identified by *cap_p* and place it into the location pointed to by *value_p*.

25.4.9.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

25.4.9.4 Errors

If any of the following conditions occur, the *cap_get_flag*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] At least one of the values of the *cap_p*, *cap*, *flag* and *value_p* arguments does not refer to the corresponding entity.

25.4.9.5 Cross-References

cap_set_flag(), 25.4.14.

25.4.10 Obtain the Current Process Capability State

Function: *cap_get_proc*()

25.4.10.1 Synopsis

```
#include <sys/capability.h>

cap_t cap_get_proc (void);
```

25.4.10.2 Description

The function *cap_get_proc*() shall allocate a capability state in working storage, set its state to that of the calling process, and return a pointer to the newly created capability state.

This function may cause memory to be allocated. The caller should free any releasable memory, when the capability state in working storage is no longer required, by calling *cap_free*() with the cap_t as an argument.

25.4.10.3 Returns

Upon successful completion, this function shall return a cap_t value. Otherwise, a value of (*cap_t*)**NULL** shall be returned and *errno* shall be set to indicate the error.

25.4.10.4 Errors

If any of the following conditions occur, the *cap_get_proc*() function shall return (*cap_t*)**NULL** and set *errno* to the corresponding value:

[ENOMEM] The capability state to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

25.4.10.5 Cross-References

cap_free(), 25.4.5; *cap_init*(), 25.4.11; *cap_get_flag*(), 25.4.9; *cap_set_proc*(), 25.4.15.

25.4.11 Allocate and Initialize a Capability State in Working Storage

Function: *cap_init*()

25.4.11.1 Synopsis

```
#include <sys/capability.h>
cap_t cap_init (void);
```

25.4.11.2 Description

The function *cap_init*() shall create a capability state in working storage and return a pointer to the capability state. The initial value of all flags for all capabilities defined by the implementation shall be cleared.

This function may cause memory to be allocated. The caller should free any releasable memory, when the capability state in working storage is no longer required, by calling *cap_free*() with the cap_t as an argument.

25.4.11.3 Returns

Upon successful completion, this function returns a non-**NULL** cap_t value. Otherwise, a value of (*cap_t*)**NULL** shall be returned and *errno* shall be set to indicate the error.

25.4.11.4 Errors

If any of the following conditions occur, the *cap_init*() function shall return (*cap_t*)**NULL** and set *errno* to the corresponding value:

[ENOMEM] The capability state to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

25.4.11.5 Cross-References

cap_free(), 25.4.5.

25.4.12 Set the Capability State of an Open File

Function: *cap_set_fd*()

25.4.12.1 Synopsis

```
#include <sys/capability.h>
int cap_set_fd (int fd, cap_t cap_p);
```

25.4.12.2 Description

The function *cap_set_fd*() shall set the values for all capability flags for all capabilities defined in the implementation for the file opened on descriptor *fd* with the capability state identified by *cap_p*. The new capability state of the file identified by *fd* shall be completely determined by the contents of *cap_p*.

For this function to succeed, the process calling it must have the CAP_SETFCAP capability enabled and either the effective user ID of the process must match the file owner or the calling process must have the effective CAP_FOWNER capability flag set. In addition, if {_POSIX_MAC} is defined, then the process must have MAC write access to the file. Implementations may place additional restrictions on setting the capability state of a file.

If the file open on the descriptor *fd* is not a regular file, then *cap_set_fd*() shall return an error.

25.4.12.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error. The capability state of the file shall not be affected if the return value is −1.

25.4.12.4 Errors

If any of the following conditions occur, the *cap_set_fd*() function shall return −1 and set *errno* to the corresponding value:

[EACCES]	The requested access to the file specified is denied, or the {_POSIX_MAC} system configuration option is enabled and MAC write access to the file opened on descriptor *fd* is denied.
[EBADF]	The *fd* argument is not a valid open file descriptor.
[EINVAL]	The value of the *cap_p* argument does not refer to a capability state in working storage.

The file open on *fd* is not a regular file.

[EPERM] The process does not have appropriate privilege or does not meet other restrictions imposed by the implementation to perform the operation.

[EROFS] This function requires modification of a file resident on a file system which is currently read-only.

25.4.12.5 Cross-References

cap_get_fd(), 25.4.7; *cap_set_file*(), 25.4.13.

25.4.13 Set the Capability State of a File

Function: *cap_set_file*()

25.4.13.1 Synopsis

```
#include <sys/capability.h>
int cap_set_file (const char *path_p, cap_t cap_p);
```

25.4.13.2 Description

The function *cap_set_file*() shall set the values for all capability flags for all capabilities defined in the implementation for the pathname pointed to by *path_p* with the capability state identified by *cap_p*. The new capability state of the file identified by *path_p* shall be completely determined by the contents of *cap_p*.

For this function to succeed, the process must have the CAP_SETFCAP capability enabled and either the effective user ID of the process must match the file owner or the calling process must have the effective flag of the CAP_FOWNER capability set. In addition, if {_POSIX_MAC} is defined, then the process must have MAC write access to the file. Implementations may place additional restrictions on setting the capability state of a file.

If the file pointed to by *path_p* is not a regular file, then *cap_set_file*() shall return an error. The effects of writing capability state to any file type other than a regular file are undefined.

25.4.13.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error. The capability state of the file shall not be affected if the return value is −1.

25.4.13.4 Errors

If any of the following conditions occur, the *cap_set_file*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] Search/read permission is denied for a component of the path prefix, or the {_POSIX_MAC} system configuration option is enabled and MAC write access to the file referred to by *path_p* is denied.

[EINVAL] The value of the *cap_p* argument does not refer to a capability state in working storage or the capability state specified is not permitted for a file on the implementation.

The file pointed to by *path_p* is not a regular file.

[ENAMETOOLONG] The length of the *path_p* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {POSIX_NO_TRUNC} is in effect.

[ENOENT] The named file/directory does not exist or the *path_p* argument points to an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The process does not have appropriate privilege or does not meet other restrictions imposed by the implementation to perform the operation.

[EROFS] This function requires modification of a file resident on a file system which is currently read-only.

25.4.13.5 Cross-References

cap_get_file(), 25.4.8; *cap_set_fd*(), 25.4.12.

25.4.14 Set the Value of a Capability Flag

Function: *cap_set_flag*()

25.4.14.1 Synopsis

```
#include <sys/capability.h>

int cap_set_flag (cap_t cap_p,
                  cap_flag_t flag,
                  int ncap,
                  cap_value_t caps[],
                  cap_flag_value_t value);
```

25.4.14.2 Description

This function shall set the flag *flag* of each capability in the array *caps* in the capability state in working storage identified by *cap_p* to *value*. The argument *ncap* is used to specify the number of capabilities in the array *caps*. Implementations may place restrictions on the setting of the flags in a capability state.

25.4.14.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error. The capability state identified by *cap_p* shall not be affected if the return value is –1.

25.4.14.4 Errors

If any of the following conditions occur, the *cap_set_flag*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL] At least one of the values of *cap_p*, *ncap*, *flag* and *value*, or at least one of the first *ncap* elements in *caps*, does not refer to the corresponding entity. –

The resulting capability state identified by *cap_p* violates one or more implementation restrictions. % –

25.4.14.5 Cross-References

cap_get_flag(), 25.4.16.

25.4.15 Set the Process Capability State

Function: *cap_set_proc*()

25.4.15.1 Synopsis

```
#include <sys/capability.h>
int cap_set_proc (cap_t cap_p);
```

25.4.15.2 Description

The function *cap_set_proc*() shall set the values for all capability flags for all capabilities defined in the implementation with the capability state identified by *cap_p*. The new capability state of the process shall be completely determined by the contents of *cap_p* upon successful return from this function. If any flag in *cap_p* is set for any capability not currently permitted for the calling process, the function shall fail, and the capability state of the process shall remain unchanged.

25.4.15.3 Returns

Upon successful completion, the function shall return a value of zero. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error. Neither the state represented in the object identified by *cap_p* nor the capability state of the calling process shall be affected if the return value is −1.

25.4.15.4 Errors

If any of the following conditions occur, *cap_set_proc*() shall return −1 and set *errno* to the corresponding value:

[EINVAL] The value of the *cap_p* argument does not refer to a capability state in working storage.

[EPERM] The caller attempted to set a capability flag of a capability that was not permitted to the invoking process.

[ENOMEM] The function requires more memory than is allowed by the hardware or system-imposed memory management constraints.

25.4.15.5 Cross-References

cap_get_proc(), 25.4.10.

25.4.16 Get the Size of a Capability Data Record

Function: *cap_size*()

25.4.16.1 Synopsis

```
#include <sys/capability.h>
ssize_t cap_size (cap_t cap_p)
```

25.4.16.2 Description

The *cap_size*() function returns the total length (in bytes) that the capability state in working storage identified by *cap_p* would require when converted by *cap_copy_ext*(). This function is used primarily to determine the amount of buffer space that must be provided to the *cap_copy_ext*() function in order to hold the capability data record created from *cap_p*.

25.4.16.3 Returns

Upon successful completion, the *cap_size*() function returns the length required to hold a capability data record. Otherwise, a value of (*ssize_t*)–1 shall be returned and *errno* shall be set to indicate the error.

25.4.16.4 Errors

If any of the following conditions occur, *cap_size*() shall return −1 and set *errno* to one of the following values:

[EINVAL] The value of the *cap_p* argument does not refer to a capability state in working storage.

25.4.16.5 Cross-References

cap_copy_ext(), 25.4.2.

25.4.17 Convert a Capability State in Working Storage to Text

Function: *cap_to_text*()

25.4.17.1 Synopsis

```
#include <sys/capability.h>

char *cap_to_text (cap_t cap_p, size_t *len_p);
```

25.4.17.2 Description

This function shall convert the capability state in working storage identified by *cap_p* into a null terminated human-readable string. This function allocates any memory necessary to contain the string, and returns a pointer to the string. If the pointer *len_p* is not (*size_t*)**NULL**, the function shall also return the full length of the string (not including the null terminator) in the location pointed to by *len_p*. The capability state in working storage identified by *cap_p* shall be completely represented in the returned character string.

The format of the string pointed to by the returned pointer shall comply with the specification in 25.3.

This function may cause memory to be allocated. The caller should free any releasable memory, when the capability state in working storage is no longer required, by calling *cap_free*() with the cap_t as an argument.

25.4.17.3 Returns

Upon successful completion, a non-**NULL** value is returned. Otherwise, a value of (*char* *)**NULL** shall be returned and *errno* shall be set to indicate the error.

25.4.17.4 Errors

If any of the following conditions occur, *cap_to_text*() shall return (*char* *)**NULL** and set *errno* to the corresponding value:

[EINVAL] Either the *cap_p* argument does not refer to a capability state in working storage or the *len_p* argument is invalid, or both.

[ENOMEM] The string to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

25.4.17.5 Cross-References

cap_free(), 25.4.5; *cap_get_flag*(), 25.4.16; *cap_from_text*(), 25.4.6.

Section 26: Mandatory Access Control

26.1 General Overview

This section describes the Mandatory Access Control Option. The section defines and discusses MAC concepts, outlines the MAC policy adopted in this standard and the impact of MAC on existing POSIX.1 functions. Support for the interfaces defined in this section is optional but shall be provided if the symbol {_POSIX_MAC} is defined.

26.1.1 MAC Concepts

MAC Labels

MAC labels form the basis for mandatory access control decisions. In order to promote flexibility in which conforming implementations may define a MAC policy, specific components of MAC labels and their textual representations are implementation-defined.

Label Relationships

Two relationships are defined between MAC labels: *equivalence*, and *dominance*. The details of *dominance* are left to the definition of the conforming implementation, however the dominance relation shall constitute a partial order on MAC labels. *Equivalence* is defined relative to *dominance*. If two MAC labels are equivalent, then each dominates the other.

MAC Objects

MAC objects are the interface-visible data containers, i.e., entities that receive or contain data, to which MAC is applied. In POSIX.1, these include the following:

Files

This includes regular files, directories, FIFO-special files, and (unnamed) pipes.

Processes

In cases where a process is the target of some request by another process, that target process shall be considered an object.

MAC Subjects

A subject is an active entity that can cause information to flow between controlled objects. Since processes are the only such interface-visible element of POSIX.1

they are the only subjects treated in this document.

26.1.2 MAC Policy

The MAC policy presented below is logically structured into the following named policies:

- **P:** The fundamental statement of mandatory access control policy
- **FP.*:** The refinements of **P** that apply to file objects (**FP.1, FP.2,** etc.)
- **PP.*:** The refinements of **P** that apply to process objects

The following labeling requirement shall be imposed:

> Each subject and each object shall have a MAC label associated with it at all times.

A physically unique MAC label is not required to be associated with each subject and object. The requirement is only that a MAC label shall always be associated with each subject and object. For example, all files in a file system could share a single MAC label.

Policies for initial assignment and constraints on the changing of MAC labels are given in the refining policies below.

The fundamental MAC restriction **P** is simply stated:

P: Subjects cannot cause information labeled at some MAC label $\mathbf{L_1}$ to become accessible to subjects at $\mathbf{L_2}$ unless $\mathbf{L_2}$ dominates $\mathbf{L_1}$.

This covers all data entities visible at the POSIX.1e interface, and includes restrictions on re-labeling data, i.e., changing the label of an object, as well as movement of that data between objects. **P** covers all forms of data transmission visible through the POSIX interface.

There are several important exceptions or limitations to the application of **P** and its refinements to POSIX.1 functions:

Covert Channel Exceptions

Policy statement **P** strictly requires that there be no covert channels. Consistent with this policy statement the new POSIX.1e functions and the changes to existing POSIX.1 functions have been specified such that covert channels are not inherent in their definition. This standard does not require conforming implementations to be free of covert channels.

Processes Possessing Appropriate Privileges

Implicit in the statement of **P** is the assumption that none of the policies need necessarily apply to processes possessing appropriate privilege unless explicitly stated. If {_POSIX_CAP} is defined, the list of capabilities that satisfy the appropriate privilege requirements are defined in this standard in section 25.2.

Devices

The MAC policy on devices may have additional restrictions or refinements not addressed here. The MAC policy on devices is unspecified.

Additional Implementation Restrictions

It is understood that a conforming implementation may enforce additional security restrictions consistent with these policies.

26.1.2.1 FP: File Function Policies

Mandatory access control for files results from the application of basic policies **(FP.∗)** to a simple assumption of the file data object. The straightforward application of these rules to the object model determines the specific MAC restrictions for a large number of file-related interfaces. The object that encompasses a POSIX.1 file shall be defined to consist of a data portion and an attribute portion. For the purposes of mandatory access control, the following assumption is made:

> Both the data and attribute portion of a file are considered a single MAC-labeled data container. Note that the MAC label shall be considered to be in the attribute portion.

Note that, within this standard, and as a basis for defining interface behavior, link names are considered as the contents of directories, and are not a property of the file that they indicate. They are protected by and considered labeled at the MAC label of their containing directory.

The following policy rules apply:

FP.1: The MAC label of a file shall be dominated by the MAC label of a subject for the subject to read the data or attributes of a file.

FP.2: The MAC label of a file shall dominate the MAC label of a subject for the subject to write the data or attributes of a file.

The general POSIX.1e mandatory access control policy shall be that subjects may write objects if the MAC label of the subject is dominated by the object's MAC label. In accordance with the policy in 2.3.2 that further restrictions may be placed on a policy, an implementation could choose to be more restrictive by allowing a subject to write to a file only when the MAC labels are equivalent.

FP.3: If reading from a FIFO-special file changes either the attributes or the data of the FIFO object, both **FP.1** and **FP.2** shall be satisfied.

FP.4: A newly created object shall be assigned a MAC label which dominates the MAC label of the creating subject.

The general POSIX.1e mandatory access control policy shall be that newly created objects shall be assigned a MAC label which dominates the MAC label of the creating subject. Although this policy statement allows creation of upgraded objects, this standard only provides interfaces which will create objects with equivalent MAC labels to the MAC label of the creating subject.

The MAC label of a file object cannot be modified in violation of **P,** e.g., processes which do not possess appropriate privilege cannot downgrade the label of a file object.

(Unnamed) pipes are considered objects, although they are not addressable by pathname.

26.1.2.1.1 Summary of POSIX.1 System Interface Impact

This policy shall be applied to the POSIX.1 functions listed in Table 26-1.

Table 26-1 – POSIX.1 Functions Covered by MAC File Policies

Existing Function	POSIX.1 Section
access	5.6.3
chdir	5.2.1
chmod	5.6.4
chown	5.6.5
creat	5.3.2
execl	3.1.2
execv	3.1.2
execle	3.1.2
execve	3.1.2
execlp	3.1.2
execvp	3.1.2
fcntl	6.5.2
getcwd	5.2.2
link	5.3.4
mkdir	5.4.1
mkfifo	5.4.2
open	5.3.1
opendir	5.1.2
pipe	6.1.1
rename	5.5.3
rmdir	5.5.2
stat	5.6.2
unlink	5.5.1
utime	5.6.6

New Function	POSIX.1e Synopsis	
acl_delete_def_file	Delete a Default ACL of a File	
acl_get_fd	Get an ACL of an Open File	
acl_get_file	Get an ACL of a File	
acl_set_fd	Set an ACL of an Open File	
acl_set_file	Set an ACL of a File	
inf_get_fd	Get the Information Label of an Open File	
inf_get_file	Get the Information Label of a File	
inf_set_fd	Set the Information Label of an Open File	
inf_set_file	Set the Information Label of a File	
mac_get_fd	Get the MAC Label of an Open File	
mac_get_file	Get the MAC Label of a File	
mac_set_fd	Set the MAC Label of an Open File	
mac_set_file	Set the MAC Label of a File	+
cap_get_fd	Get the Capability State of an Open File	
cap_get_file	Get the Capability State of a File	
cap_set_fd	Set the Capability State of an Open File	
cap_set_file	Set the Capability State of a File	

26.1.2.2 PP: Process Function Policies

Mandatory access control for processes stems from the application of the basic MAC restriction to the affected POSIX.1 functions. When treated as an object, the process shall consist of its internal data (including the environment data), its executable image, and its status information.

The following policy rules apply:

PP.1: No process at MAC label $\mathbf{L_1}$ may write to a process at label $\mathbf{L_2}$ unless $\mathbf{L_2}$ dominates $\mathbf{L_1}$.

PP.2: A newly created process shall be assigned a MAC label which dominates the MAC label of the creating process.

The general POSIX.1 mandatory access control policy shall be that newly created processes shall be assigned a MAC label which dominates the MAC label of the creating process. Although this policy statement allows creation of upgraded processes, this standard only provides interfaces which create processes with equivalent MAC labels as the creating process.

The MAC label of a process cannot be altered in violation of **P,** e.g., a process which do not possess appropriate privilege cannot downgrade its own MAC label.

26.1.2.2.1 POSIX.1 Functions Covered by MAC Process Policies

This policy shall be applied to the POSIX.1 functions listed in Table 26-2.

Table 26-2 – POSIX.1 Functions Covered by MAC Process Policies

Existing Function	POSIX.1 Section
fork	3.1.1
kill	3.3.2

New Function	POSIX.1e Synopsis
mac_set_proc	Set the Process Label

26.2 Header

Some of the data types used by the MAC label functions are not defined as part of this standard, but shall be implementation-defined. If {_POSIX_MAC} is defined, these types shall be defined in the header `<sys/mac.h>`, which contains definitions for at least the following type: *mac_t*.

26.2.1 mac_t

This type defines a pointer to an "exportable" object capable of holding a MAC label. The object is opaque, persistent, and self-contained. It shall be possible to create an independent copy of the entire MAC label in a user-defined location using normal byte-copy of *mac_size*() bytes starting at the location pointed to by the *mac_t*. It shall be possible to byte-copy the copy back into system-managed space, and recommence processing of it there, even if the copy has been stored in a file or elsewhere, or moved to a different process. The internal structure of the MAC label is otherwise unspecified.

26.3 Functions

The functions in this section comprise the set of services that permit processes to retrieve, compare, set, and convert MAC labels. Support for the functions and policy described in this section is optional. If the symbol {_POSIX_MAC} is defined, the implementation supports the Mandatory Access Control (MAC) labels option and all of the MAC functions shall be implemented as described in this section. If {_POSIX_MAC} is not defined, the result of calling any of these functions is unspecified.

The error [ENOTSUP] shall be returned in those cases where the system supports MAC labeling but the particular MAC label operation cannot be applied because of restrictions imposed by the implementation.

26.3.1 Test MAC Labels for Dominance

Function: *mac_dominate*()

26.3.1.1 Synopsis

```
#include <sys/mac.h>

int mac_dominate (mac_t labela, mac_t labelb);
```

26.3.1.2 Description

The function *mac_dominate*() determines whether *labela* dominates *labelb*. The precise method for determining domination is implementation-defined.

This function is provided to allow conforming applications to test for dominance since a comparison of the labels themselves may yield an indeterminate result.

26.3.1.3 Returns

If an error occurs, the *mac_dominate*() function shall return a value of −1 and *errno* shall be set to indicate the error. Otherwise, a value of 1 shall be returned if label *labela* dominates *labelb*, and a value of 0 shall be returned if *labela* does not dominate *labelb*.

26.3.1.4 Errors

If any of the following conditions occur, the *mac_dominate*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] At least one of the labels is not a valid MAC label as defined by *mac_valid*().

26.3.1.5 Cross-References

mac_equal(), 26.3.2; *mac_valid*(), 26.3.15.

26.3.2 Test MAC Labels for Equivalence

Function: *mac_equal*()

26.3.2.1 Synopsis

```
#include <sys/mac.h>

int mac_equal (mac_t labela, mac_t labelb);
```

26.3.2.2 Description

The function *mac_equal*() determines whether *labela* is equivalent to *labelb*. The precise method for determining equivalence is implementation-defined.

This function is provided to allow conforming applications to test for equivalence since a comparison of the labels themselves may yield an indeterminate result.

26.3.2.3 Returns

If an error occurs, a value of −1 shall be returned and *errno* shall be set to indicate the error. Otherwise, the *mac_equal*() function returns 1 if *labela* is equivalent to *labelb*, and a value of 0 shall be returned if *labela* is not equivalent to *labelb*.

26.3.2.4 Errors

If any of the following conditions occur, the *mac_equal*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] At least one of the labels is not a valid MAC label as defined by *mac_valid*().

26.3.2.5 Cross-References

mac_dominate(), 26.3.1; *mac_valid*(), 26.3.15.

26.3.3 Free MAC Label Storage Space

Function: *mac_free*()

26.3.3.1 Synopsis

```
#include <sys/mac.h>

int mac_free (void *buf_p);
```

26.3.3.2 Description

The function *mac_free*() shall free any releasable memory currently allocated to the buffer identified by *buf_p*. The *buf_p* argument may be either a (*void**)*mac_t*, or a (*void**)char* allocated by the *mac_to_text*() function.

26.3.3.3 Returns

Upon successful completion, the function *mac_free*() returns a value of 0. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

26.3.3.4 Errors

This standard does not specify any error conditions that are required to be detected for the *mac_free*() function. Some errors may be detected under conditions that are unspecified by this part of the standard.

26.3.3.5 Cross-References

mac_from_text(), 26.3.4; *mac_get_fd*(), 26.3.5; *mac_get_file*(), 26.3.6; *mac_get_proc*(), 26.3.7; *mac_glb*(), 26.3.8; *mac_lub*(), 26.3.9.

26.3.4 Convert Text MAC Label to Internal Representation

Function: *mac_from_text*()

26.3.4.1 Synopsis

```
#include <sys/mac.h>

mac_t mac_from_text (const char *text_p);
```

26.3.4.2 Description

The function *mac_from_text*() converts the text representation of a MAC label *text_p* into its internal representation.

This function may cause memory to be allocated. The caller should free any releasable memory, when the MAC label is no longer required, by calling *mac_free*() with the *mac_t* as an argument. In event an error occurs, no memory shall be allocated and **NULL** shall be returned.

26.3.4.3 Returns

Upon successful completion, the function *mac_from_text*() shall return a pointer to the MAC label. Otherwise, no space shall be allocated, a (*mac_t*) **NULL** pointer shall be returned, and *errno* shall be set to indicate the error.

26.3.4.4 Errors

If any of the following conditions occur, the *mac_from_text*() function shall return a **NULL** pointer and set *errno* to the corresponding value:

[EINVAL] The string *text_p* is not a valid textual representation of a MAC label as defined by *mac_valid*().

[ENOMEM] The MAC label requires more memory than is allowed by the hardware or system-imposed memory management constraints.

26.3.4.5 Cross-References

mac_free(), 26.3.3; *mac_valid*(), 26.3.15.

26.3.5 Get the Label of a File Designated by a File Descriptor

Function: *mac_get_fd*()

26.3.5.1 Synopsis

```
#include <sys/mac.h>

mac_t mac_get_fd (int fildes)
```

26.3.5.2 Description

The *mac_get_fd*() function returns the MAC label associated with an open file. The function accepts a valid file descriptor to the file, allocates memory in which to store the MAC label to be returned and copies the file MAC label into the allocated memory.

A process can get the MAC label for any file for which the process has a valid file descriptor and MAC read access.

This function may cause memory to be allocated. The caller should free any releasable memory, when the MAC label is no longer required, by calling *mac_free*() with the *mac_t* as an argument. In event an error occurs, no memory shall be allocated and **NULL** shall be returned.

26.3.5.3 Returns

Upon successful completion, the function shall return a pointer to the MAC label. Otherwise, no space shall be allocated, a (*mac_t*)**NULL** pointer shall be returned and *errno* shall be set to indicate the error.

26.3.5.4 Errors

If any of the following conditions occur, the *mac_get_fd*() function shall return a (*mac_t*)**NULL** and set *errno* to the corresponding value:

[EACCES] MAC read access is denied to the file referred to by *fildes*.

[EBADF] The *fildes* argument is not a valid file descriptor.

[ENOMEM] The MAC label requires more memory than is allowed by the hardware or system-imposed memory management constraints.

26.3.5.5 Cross-References

mac_free(), 26.3.3; *mac_get_file*(), 26.3.6; *mac_set_fd*(), 26.3.10; *mac_set_file*(), 26.3.11.

26.3.6 Get the Label of a File Designated by a Pathname

Function: *mac_get_file*()

26.3.6.1 Synopsis

```
#include <sys/mac.h>

mac_t mac_get_file (const char *path_p);
```

26.3.6.2 Description

The *mac_get_file*() function returns the MAC label associated with the pathname pointed to by *path_p*. The function allocates memory in which to store the MAC label to be returned and copies the file MAC label into the allocated memory.

A process can get the MAC label for any file for which the process has search access to the path specified and MAC read access to the file.

This function may cause memory to be allocated. The caller should free any releasable memory, when the MAC label is no longer required, by calling *mac_free*() with the *mac_t* as an argument. In event an error occurs, no memory shall be allocated and **NULL** shall be returned.

26.3.6.3 Returns

Upon successful completion, the function shall return a pointer to the MAC label. Otherwise, no space shall be allocated, a (*mac_t*)**NULL** pointer shall be returned and *errno* shall be set to indicate the error.

26.3.6.4 Errors

If any of the following conditions occur, the *mac_get_file*() function shall return a (*mac_t*)**NULL** and set *errno* to the corresponding value:

- [EACCES] Search permission is denied for a component of the path prefix or MAC read access to the file is denied.
- [ENAMETOOLONG] The length of the *path_p* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX} while {POSIX_NO_TRUNC} is in effect.
- [ENOENT] The named file/directory does not exist, or the *path_p* argument points to an empty string.
- [ENOMEM] The MAC label requires more memory than is allowed by the hardware or system-imposed memory management constraints. –
- [ENOTDIR] A component of the path prefix is not a directory.

26.3.6.5 Cross-References

mac_free(), 26.3.3; *mac_get_fd*(), 26.3.5; *mac_set_fd*(), 26.3.10; *mac_set_file*(), 26.3.11.

26.3.7 Get the Process Label

Function: *mac_get_proc*()

26.3.7.1 Synopsis

```
#include <sys/mac.h>

mac_t mac_get_proc (void);
```

26.3.7.2 Description

The *mac_get_proc*() function returns the MAC label associated with the requesting process. The function allocates memory in which to store the MAC label to be returned and copies the process MAC label into the allocated memory.

Any process may so query its MAC label.

This function may cause memory to be allocated. The caller should free any releasable memory, when the MAC label is no longer required, by calling *mac_free*() with the *mac_t* as an argument. In event an error occurs, no memory shall be allocated and **NULL** shall be returned.

26.3.7.3 Returns

Upon successful completion, *mac_get_proc*() returns a pointer to the MAC label of the process. Otherwise, no space shall be allocated, a (*mac_t*)**NULL** pointer shall be returned and *errno* shall be set to indicate the error.

26.3.7.4 Errors

If any of the following conditions occur, the *mac_get_proc*() function shall return a (*mac_t*)**NULL** and set *errno* to the corresponding value:

[ENOMEM] The MAC label requires more memory than is allowed by the hardware or system-imposed memory management constraints.

26.3.7.5 Cross-References

mac_free(), 26.3.3; *mac_set_proc*(), 26.3.12.

26.3.8 Compute the Greatest Lower Bound

Function: *mac_glb*()

26.3.8.1 Synopsis

```
#include <sys/mac.h>

mac_t mac_glb (mac_t labela, mac_t labelb);
```

26.3.8.2 Description

The function *mac_glb*() returns a pointer to the (valid) MAC label, if it exists, that is dominated by both the MAC label *labela* and the MAC label *labelb* and dominates all other valid MAC labels that are dominated by both the MAC label *labela* and the MAC label *labelb*.

This function may cause memory to be allocated. The caller should free any releasable memory, when the MAC label is no longer required, by calling *mac_free*() with the *mac_t* as an argument. In event an error occurs, no memory shall be allocated and **NULL** shall be returned.

26.3.8.3 Returns

Upon successful completion, this returns a pointer to the allocated bounding MAC label. Otherwise, no space shall be allocated, a (*mac_t*)**NULL** pointer shall be returned, and *errno* shall be set to indicate the error.

26.3.8.4 Errors

If any of the following conditions occur, the *mac_glb*() function shall return a (*mac_t*)**NULL** and set *errno* to the corresponding value:

[EINVAL] At least one of the input labels is not a valid MAC label as defined by *mac_valid*().

[ENOENT] The bounding MAC label does not exist or is not valid.

[ENOMEM] The MAC label requires more memory than is allowed by the hardware or system-imposed memory management constraints.

26.3.8.5 Cross-References

mac_free(), 26.3.3; *mac_lub*(), 26.3.9; *mac_valid*(), 26.3.15.

26.3.9 Compute the Least Upper Bound

Function: *mac_lub*()

26.3.9.1 Synopsis

```
#include <sys/mac.h>

mac_t mac_lub (mac_t labela, mac_t labelb);
```

26.3.9.2 Description

The function *mac_lub*() returns a pointer to the (valid) MAC label (if it exists) that dominates both the MAC label *labela* and the MAC label *labelb* and is dominated by all other valid MAC labels that dominate both the MAC label *labela* and the MAC label *labelb*.

This function may cause memory to be allocated. The caller should free any releasable memory, when the MAC label is no longer required, by calling *mac_free*() with the *mac_t* as an argument. In event an error occurs, no memory shall be allocated and **NULL** shall be returned.

26.3.9.3 Returns

Upon successful completion, this function shall return a pointer to the bounding MAC label. Otherwise, a (*mac_t*)**NULL** pointer shall be returned and *errno* shall be set to indicate the error.

26.3.9.4 Errors

If any of the following conditions occur, the *mac_lub*() function shall return a (*mac_t*)**NULL** and set *errno* to the corresponding value:

[EINVAL] At least one of the input labels is not a valid MAC label as defined by *mac_valid*().

[ENOENT] The bounding MAC label does not exist or is not valid.

[ENOMEM] The MAC label requires more memory than is allowed by the hardware or system-imposed memory management constraints.

26.3.9.5 Cross-References

mac_free(), 26.3.3; *mac_glb*(), 26.3.8; *mac_valid*(), 26.3.15.

26.3.10 Set the Label of a File Identified by File Descriptor

Function: *mac_set_fd*()

26.3.10.1 Synopsis

```
#include <sys/mac.h>

int mac_set_fd (int fildes, mac_t label);
```

26.3.10.2 Description

This function sets the MAC label of a file to *label*. The function requires that *fildes* be a valid file descriptor to indicate the file.

A process can set the MAC label for a file only if the process has a valid file descriptor for the file and has MAC write access to the file. Additionally, only processes with an effective user ID equal to the owner of the file or with appropriate privileges may change the label of the file. If {_POSIX_CAP} is defined, then appropriate privilege shall include CAP_FOWNER.

The *mac_set_fd*() function shall fail if the new MAC label is not equivalent to the file's previous label and the process does not possess appropriate privilege. If {_POSIX_CAP} is defined, and the new MAC label dominates, but is not equivalent to the file's previous MAC label, then appropriate privilege shall include CAP_MAC_UPGRADE. If {_POSIX_CAP} is defined, and the new MAC label does not dominate the file's previous MAC label then appropriate privilege shall include CAP_MAC_DOWNGRADE.

It is implementation-defined whether an implementation will return [EBUSY] or will perform revocation of access if other processes have current access to the file at the time of MAC label modification.

26.3.10.3 Returns

Upon successful completion, the function shall return a value of 0. Otherwise, a value of –1 shall be returned and *errno* shall be set to indicate the error.

26.3.10.4 Errors

If any of the following conditions occur, the *mac_set_fd*() function shall return –1 and set *errno* to the corresponding value:

[EACCES]	MAC write access is denied to the file specified.
[EBADF]	The *fildes* argument is not a valid file descriptor.
[EBUSY]	The file named by the *fildes* argument is currently in a state in which the implementation does not allow the label to be changed.
[EINVAL]	The MAC label *label* is not a valid MAC label as defined by *mac_valid*().
[ENOTSUP]	{_POSIX_MAC} is defined, but this function is not supported on the file referred to by *fildes*, i.e., {_POSIX_MAC_PRESENT} is not in effect for the file referred to by *fildes*.
[EPERM]	An attempt was made to change the MAC label of a file and the process does not possess appropriate privilege.
[EROFS]	This function requires modification of a file system which is currently read-only.

26.3.10.5 Cross-References

mac_get_fd(), 26.3.5; *mac_set_file*(), 26.3.11; *mac_valid*(), 26.3.15.

26.3.11 Set the Label of a File Designated by Pathname

Function: *mac_set_file*()

26.3.11.1 Synopsis

```
#include <sys/mac.h>

int mac_set_file (const char *path_p, mac_t label);
```

26.3.11.2 Description

This function sets the MAC label of the pathname pointed to by *path_p* to *label*.

A process can set the MAC label for a file only if the process has search access to the path and has MAC write access to the file. Additionally, only processes with an effective user ID equal to the owner of the file or with appropriate privileges may change the label of the file. If {_POSIX_CAP} is defined, then appropriate privilege shall include CAP_FOWNER.

The *mac_set_file*() function shall fail if the new MAC label is not equivalent to the file's previous MAC label and the process does not possess appropriate privilege. If {_POSIX_CAP} is defined, and the new MAC label dominates, but is not equivalent to the file's previous MAC label, then appropriate privilege shall include CAP_MAC_UPGRADE. If {_POSIX_CAP} is defined, and the new MAC label does not dominate the file's previous MAC label then appropriate privilege shall include CAP_MAC_DOWNGRADE.

It is implementation-defined whether an implementation will return [EBUSY] or will perform revocation of access if other processes have current access to the file at the time of MAC label modification.

26.3.11.3 Returns

Upon successful completion, the function shall return a value of 0. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

26.3.11.4 Errors

If any of the following conditions occur, the *mac_set_file*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] Search permission is denied for a component of the path prefix or MAC write access to the target file is denied.

[EBUSY] The file or directory indicated by *path_p* is currently in a state in which the implementation does not allow the label to be changed.

[EINVAL] The MAC label *label* is not a valid MAC label as defined by *mac_valid*().

[ENAMETOOLONG] The length of the *path_p* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {POSIX_NO_TRUNC} is in effect.

[ENOENT] The named file/directory does not exist, or the *path_p* argument points to an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

[ENOTSUP] {_POSIX_MAC} is defined, but this function is not supported on the file specified, i.e., {_POSIX_MAC_PRESENT} is not in effect for the file specified.

[EPERM] An attempt was made to change the MAC label of a file and the process does not possess appropriate privilege.

[EROFS] This function requires modification of a file system which is currently read-only.

26.3.11.5 Cross-References

mac_get_file(), 26.3.6; *mac_set_fd*(), 26.3.10; *mac_valid*(), 26.3.15.

26.3.12 Set the Process Label

Function: *mac_set_proc*()

26.3.12.1 Synopsis

```
#include <sys/mac.h>

int mac_set_proc (mac_t label);
```

26.3.12.2 Description

The *mac_set_proc*() function is used to set (write) the MAC label of the requesting process. The new label is specified by *label*. A process may only alter its MAC label if it possesses appropriate privilege. If {_POSIX_CAP} is defined, then appropriate privilege shall include CAP_MAC_RELABEL_SUBJ.

26.3.12.3 Returns

Upon successful completion, *mac_set_proc*() shall return a value of 0. Otherwise, a value of −1 shall be returned and *errno* shall be set to indicate the error.

26.3.12.4 Errors

If any of the following conditions occur, the *mac_set_proc*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The MAC label *label* is not a valid MAC label as defined by *mac_valid*().

[EPERM] The process does not have appropriate privilege to perform the operation requested.

26.3.12.5 Cross-References

mac_valid(), 26.3.15.

26.3.13 Get the Size of a MAC Label

Function: *mac_size*()

26.3.13.1 Synopsis

```
#include <sys/mac.h>

ssize_t mac_size (mac_t label);
```

26.3.13.2 Description

The *mac_size*() function returns the size in bytes of the MAC label specified by *label* if the label is valid. Note: this is the size of the internal MAC label, not the size of the text representation as produced by the *mac_to_text*() function.

26.3.13.3 Returns

Upon successful completion, this function shall return the size of the MAC label. Otherwise, a value of −1 shall be returned and *errno*

26.3.13.4 Errors

If any of the following conditions occur, the *mac_size*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The MAC label *label* is invalid as defined by *mac_valid*().

26.3.13.5 Cross-References

mac_valid(), 26.3.15.

26.3.14 Convert Internal MAC Label to Textual Representation

Function: *mac_to_text*()

26.3.14.1 Synopsis

```
#include <sys/mac.h>

char *mac_to_text (mac_t label, size_t *len_p);
```

26.3.14.2 Description

The function *mac_to_text*() converts the internal representation of the MAC label pointed to by *label* into a human-readable, **NULL** terminated, character string. The output of *mac_to_text*() shall be suitable for re-input as the *text_p* parameter to *mac_from_text*() in 26.3.4, or as the *label* operand to the `setfmac` utility as defined in section 11 of POSIX.2c on the same system or other systems with identical MAC label definitions. The function returns a pointer to the text representation of the MAC label. If the pointer *len_p* is not **NULL**, the function shall return the length of the string (not including the **NULL** terminator) in the location pointed to by *len_p*.

This function may cause memory to be allocated. The caller should free any releasable memory, when the text label is no longer required, by calling *mac_free*() with the string address as an argument. In event an error occurs, no memory shall be allocated and **NULL** shall be returned.

26.3.14.3 Returns

Upon successful completion, the function *mac_to_text*() returns a pointer to the text representation of the MAC label, and if the pointer *len_p* is not **NULL**, returns the length of the string (not including the **NULL** terminator) in the location pointer to by *len_p*. Otherwise, no memory shall be allocated, the memory referred to by *len_p* shall be unchanged, a (*char* *) **NULL** pointer shall be returned and *errno* shall be set to indicate the error.

26.3.14.4 Errors

If any of the following conditions occur, the *mac_to_text*() function shall return a **NULL** pointer and set *errno* to the corresponding value:

[EINVAL] The MAC label *label* is not a valid MAC label as defined by *mac_valid*().

[ENOMEM] The text to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

26.3.14.5 Cross-References

mac_from_text(), 26.3.4; *mac_valid*(), 26.3.15; `setfmac`, POSIX.2c - 11.3.

26.3.15 Label Validity

Function: *mac_valid*()

26.3.15.1 Synopsis

```
#include <sys/mac.h>

int mac_valid (mac_t label);
```

26.3.15.2 Description

The *mac_valid*() function determines if *label* is a valid MAC label. The meaning of validity is implementation-defined.

26.3.15.3 Returns

Upon successful completion, the function shall return 1 if *label* is valid, and 0 if it is invalid. Otherwise a value of −1 shall be returned and *errno* is set to indicate the error.

26.3.15.4 Errors

This standard does not specify any error conditions that are required to be detected for the *mac_valid*() function. Some errors may be detected under conditions that are unspecified by this part of the standard.

26.3.15.5 Cross-References

None.

Section 27: Information Labeling

27.1 General Overview

This section describes the Information Label Option. The section defines and discusses the information label concepts, outlines the information label policy adopted in this standard, and outlines the impact of information labels on existing POSIX.1 functions. Support for the interfaces defined in this section is optional but shall be provided if the symbol {POSIX_INF} is defined.

27.1.1 Information Label Concepts

Information Labels

The Information Label is the item visible at the POSIX.1 interface that is used for associating labeling information with data. This labeling information is not related to Mandatory Access Control, nor does the information labeling policies in any way override the MAC or DAC options, if they are in effect.

In order to promote the flexibility with which conforming implementations may define an information labeling policy, specific components of information labels and their textual representation are not defined by this standard.

Information Label Relationships

Two relationships are defined between information labels: *equivalence* and *dominance*. A conforming implementation must provide the interfaces for determining whether two information labels have these relationships. Note that it would be acceptable for a conforming implementation to implement information labels in such a manner that no information label is equivalent to, nor dominates, any information label other than itself. Thus, while interfaces for determining dominance and equivalence must be provided, the detailed definitions of these relationships are left undefined.

Information Label Floating

The *inf_float*() operation is used in the statement of the information label policy. The operation *inf_float(inf_p1, inf_p2)* returns an information label whose value is dependent on the values of *inf_p1* and *inf_p2* and the implementation-defined floating policy. The precise definition of *inf_float*() is left to the conforming implementation, however, its intended use is described in 27.1.2. (As a result of this permitted flexibility, a conforming implementation could, for example, choose to always return just *inf_p2*.)

Information Label Subjects

In the broad sense, a subject is an active entity that can cause information of any kind to flow between controlled objects. Since processes are the only such interface-visible element in this standard, they are the only subjects treated in the information label section.

Information Label Objects

Objects are passive entities that contain or receive data. Access to objects potentially implies access to the data they contain. However, objects not only contain data, but also possess attributes. The data portion of an object is that portion that contains the bytes intended to be stored by the object (e.g., the bytes written to a regular file comprise that file's data portion). The attribute portion of an object is that portion that contains descriptive, or control, information pertaining to the object (e.g., a regular file's access and modification times, permission bits, length, and so forth). The granting of access to an object's data and to that object's attributes may be based upon different criteria. Information labeling, as described in greater detail below, relies on this distinction.

The objects to which information labeling applies include the data portion of the following objects: regular files, FIFO-special files, and (unnamed) pipes. Note that conforming implementations may choose to apply the information labeling policy more broadly by including, for example, object attributes.

27.1.2 Information Label Policy

The information label policy presented below is logically structured into the following named policies:

I: The fundamental statement of information labeling

FI.∗**:** The refinements of **I** that apply to file objects (**FI.1, FI.2**, etc.)

PI.∗**:** The refinements of **I** that apply to process objects

The following information labeling requirement is imposed:

> Each subject and each object that contains data, as opposed to attributes (e.g., mandatory access control label and access time), shall have as an additional attribute an information label at all times.

Policies for initial assignment and constraints on the changing of information labels are given in the refining policies below.

The fundamental information label policy **I** is:

I: When subjects cause data (as opposed to attributes) to flow from a source with information label *inf_p1* to a destination with information label *inf_p2*, the destination's information label shall be automatically set to the value returned by *inf_float (inf_p1, inf_p2)*.

There are several important exceptions or limitations to the application of **I** and its refinements to POSIX.1 functions:

Processes Possessing Appropriate Privilege

Implicit in the statement of **I** is the assumption that none of the policies need necessarily apply to processes possessing appropriate privilege unless explicitly stated. If {POSIX_CAP} is defined, the list of capabilities that satisfy the appropriate privilege requirements are defined by this standard in section 25.2. Note that conforming implementations can further restrict the policies that can be bypassed using capabilities. For example, if {POSIX_CAP} is defined, the effect of the CAP_INF_RELABEL_OBJ capability may be limited to a range of information labels, where such a range is implementation defined.

Additional Implementation-Defined Floating

It is understood that a conforming implementation may cause the floating described above through the automatic application of the *inf_float*() operation to occur at other times in addition to those covered by the general policy. Additionally it may cause other changes (including "downward" adjustments) of information labels under implementation-defined circumstances.

27.1.2.1 FI: File Function Policies

Information labeling for files results from the application of basic policies (**FI.***) to the file data object. The straightforward application of these rules to the object model determines the specific information label restrictions for a large number of file-related interfaces.

The object that encompasses a POSIX.1 file is defined to consist of a data portion, and an attribute portion that contains the POSIX-defined attributes including the information label. For the purposes of information labeling, the information label of a file applies only to the data portion of the file.

The following policy rules apply:

FI.1: When an process with information label *inf_p1* writes data to a file with information label *inf_p2*, the information label of the file shall automatically be set to the value returned by *inf_float(inf_p1, inf_p2)*.

FI.2: The information label of a newly created file object shall automatically be set to a value that dominates the value returned by *inf_default*().

A conforming implementation may modify these policy rules for certain objects. For example, some objects may be designated "non-floating." The information label of these objects will not change on process writes. Other objects may support additional or finer-grained labeling which will modify the application of **FI.1** (as well as **PI.1** below.) Precisely which objects are subject to modified rules is implementation-defined.

27.1.2.1.1 POSIX.1 Functions Covered by IL File Policies

This policy is applied to the following POSIX.1 functions:

Table 27-1 – POSIX.1 Functions Covered by Information Label File Policies

Existing Function	POSIX.1 Section
creat	5.3.2
mkfifo	5.4.2
open	5.3.1
pipe	6.1.1
write	6.4.2

New Function	POSIX.1e Synopsis
aud_write	Write an Audit Record
inf_get_fd	Get the Information Label of a File Identified by File Descriptor
inf_get_file	Get the Information Label of a File Identified by Pathname
inf_set_fd	Set the Information Label of a File Identified by File Descriptor
inf_set_file	Set the Information Label of a File Identified by Pathname

27.1.2.2 PI: Process Function Policies

Information labeling for processes stems from the application of the basic information label policy to the few affected POSIX.1 functions.

When treated as an object, the process shall consist of its internal data (including the environment data), its executable image, and its status information.

The following policy rules apply:

- **PI.1:** When a process with information label *inf_p1* reads data from a file with information label *inf_p2*, the information label of the process shall be automatically set to the value returned by *inf_float(inf_p2, inf_p1)*.
- **PI.2:** When a process with information label *inf_p1* executes a file with information label *inf_p2*, the information label of the process shall be automatically set to the value returned by *inf_float(inf_p2, inf_p1)*.
- **PI.3:** A newly created process shall be assigned the information label of the creating subject (process).

As mentioned previously, a conforming implementation may modify these rules for certain objects. For example, some objects may support additional or finer-grained labeling which will modify the application of **PI.1**. Precisely which objects are subject to modified rules is implementation defined.

27.1.2.2.1 POSIX.1 Functions Covered by IL Process Policies

This policy is applied to the following POSIX.1 functions:

Table 27-2 – POSIX.1 Functions Covered by Information Label Process Policies

Existing Function	POSIX.1 Section
execl	3.1.2
execv	3.1.2
execle	3.1.2
execve	3.1.2
execlp	3.1.2
execvp	3.1.2
fork	3.1.1
read	6.4.1

New Function	POSIX.1e Synopsis
aud_read	Read an Audit Record
inf_get_proc	Get the Process Information Label
inf_set_proc	Set the Process Information Label

27.2 Header

Some of the data types used by the information label functions are not defined as part of this standard, but shall be implementation-defined. If {POSIX_INF} is defined, these types shall be defined in the header `<sys/inf.h>`, which contains definitions for at least the following type.

27.2.1 inf_t

This type defines a pointer to an "exportable" object containing an information label. The object is opaque, persistent, and self-contained. Thus, the object can be copied by duplicating the bytes without knowledge of any underlying structure.

27.3 Functions

The functions in this section comprise the set of services that permit a process to get, set, and manipulate information labels. Support for the information label facility functions described in this section is optional. If the symbol {_POSIX_INF} is defined, the implementation supports the information label option and all of the information label functions shall be implemented as described in this section. If {_POSIX_INF} is not defined, the result of calling any of these functions is unspecified.

The error [ENOTSUP] shall be returned in those cases where the system supports the information label facility but the particular information label operation cannot be applied because of restrictions imposed by the implementation.

27.3.1 Initial Information Label

Function: *inf_default*()

27.3.1.1 Synopsis

```
#include <sys/inf.h>

inf_t inf_default (void)
```

27.3.1.2 Description

The *inf_default*() function returns a pointer to an information label with an initial information label value that a conforming application may associate with newly-created or fully truncated objects.

The system may allocate space for the information label to be returned. The caller should free any releasable memory when the new label is no longer required by calling *inf_free*() with the inf_t as an argument. In the event an error occurs, no memory shall be allocated and (*inf_t*)**NULL** shall be returned.

The precise method by which this label is determined is implementation-defined and therefore may vary arbitrarily (e.g., based on process ID). As a result, the initial information label may not be the same on all newly created objects. However, this label is guaranteed to be a valid label which, if applied to a newly-created object, will be consistent with the implementation's information label policy.

27.3.1.3 Returns

The function *inf_default*() returns a pointer to the initial information label unless one of the errors below occurs, in which case no space is allocated, a value of (*inf_t*)**NULL** is returned, and *errno* is set to indicate the error.

27.3.1.4 Errors

If any of the following conditions occur, the *inf_default*() function shall return a value of (*inf_t*)**NULL** and set *errno* to the corresponding value:

[ENOMEM] The label to be returned required more memory than was allowed by the hardware or by system-imposed memory management constraints.

27.3.1.5 Cross-References

inf_free(), 27.3.5; *inf_set_fd*(), 27.3.10; *inf_set_file*(), 27.3.11.

27.3.2 Test Information Labels For Dominance

Function: *inf_dominate*()

27.3.2.1 Synopsis

```
#include <sys/inf.h>
int inf_dominate (inf_t labela, inf_t labelb);
```

27.3.2.2 Description

The *inf_dominate*() function determines whether *labela* dominates *labelb*. The precise method for determining dominance is implementation-defined. Dominance includes equivalence. Hence, if one label is equivalent to another, then each shall dominate the other. Note that it is possible for neither of two labels to dominate the other.

27.3.2.3 Returns

The function *inf_dominate*() returns 1 if *labela* dominates *labelb*. A value of 0 is returned if *labela* does not dominate *labelb*. Otherwise, a result of −1 is returned, and *errno* is set to indicate the error.

27.3.2.4 Errors

If any of the following conditions occur, the *inf_dominate*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] One or both of the labels is not a valid information label as defined by *inf_valid*().

27.3.2.5 Cross-References

inf_equal(), 27.3.3; *inf_valid*(), 27.3.15.

27.3.3 Test Information Labels For Equivalence

Function: *inf_equal*()

27.3.3.1 Synopsis

```
#include <sys/inf.h>
int inf_equal (inf_t labela, inf_t labelb);
```

27.3.3.2 Description

The *inf_equal*() function determines whether *labela* is equivalent to *labelb*. The precise method for determining equivalence is implementation-defined.

This function is provided to allow conforming applications to test for equivalence since a comparison of the labels themselves may yield an indeterminate result.

27.3.3.3 Returns

The function *inf_equal*() returns 1 if *labela* is equivalent to *labelb*. A value of 0 is returned if *labela* not equivalent to *labelb*. Otherwise, a value of −1 is returned, and *errno* is set to indicate the error.

27.3.3.4 Errors

If any of the following conditions occur, the *inf_equal*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] One or both of the labels is not a valid information label as defined by *inf_valid*().

27.3.3.5 Cross-References

inf_dominate(), 27.3.2; *inf_valid*(), 27.3.15.

27.3.4 Floating Information Labels

Function: *inf_float*()

27.3.4.1 Synopsis

```
#include <sys/inf.h>
inf_t inf_float (inf_t labela, inf_t labelb);
```

27.3.4.2 Description

The *inf_float*() function returns a pointer to an information label that represents a combination of *labela* and *labelb* in a manner dependent on the implementation-defined floating policy.

The system may allocate space for the information label to be returned. The caller should free any releasable memory when the new label is no longer required by calling *inf_free*() with the returned inf_t as an argument. In the event an error occurs, no memory shall be allocated and (*inf_t*)**NULL** shall be returned.

Note, that the notion of floating presupposes the introduction of data with one label into a separately labeled subject or object. The *labela* argument represents the information label of the data being introduced, the argument *labelb* represents the subject's or object's current information label.

27.3.4.3 Returns

Upon successful completion, this function returns a pointer to the new information label. Otherwise, no space is allocated, a value of (*inf_t*)**NULL** is returned, and *errno* is set to indicate the error.

27.3.4.4 Errors

If any of the following conditions occur, the *inf_float*() function shall return a value of (*inf_t*)**NULL** and set *errno* to the corresponding value:

[EINVAL] One or both of the labels is not a valid information label as defined by *inv_valid*()

[ENOMEM] The label to be returned required more memory than was allowed by the hardware or by system-imposed memory management constraints.

27.3.4.5 Cross-References

inf_free(), 27.3.5; *inf_valid*(), 27.3.15.

27.3.5 Free Allocated Information Label Memory

Function: *inf_free*()

27.3.5.1 Synopsis

```
#include <sys/inf.h>

int inf_free (void *buf_p);
```

27.3.5.2 Description

The *inf_free*() function frees any releasable memory currently allocated to the buffer identified by *buf_p*. The *buf_p* argument may be either a (*void*∗)inf_t, or a (*void*∗)char∗ allocated by the *inf_to_text()* function.

27.3.5.3 Returns

Upon successful completion, the function *inf_free*() returns a value of 0. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

27.3.5.4 Errors

This standard does not specify any error conditions that are required to be detected for the *inf_free*() function. Some errors may be detected under conditions that are unspecified by this part of the standard.

27.3.5.5 Cross-References

inf_default(), 27.3.1; *inf_float*(), 27.3.4; *inf_get_fd*(), 27.3.7; *inf_get_file*(), 27.3.8; *inf_get_proc*(), 27.3.9; *inf_from_text*(), 27.3.6; *inf_to_text*(), 27.3.14.

27.3.6 Convert Text Label to Internal Representation

Function: *inf_from_text*()

27.3.6.1 Synopsis

```
#include <sys/inf.h>

inf_t inf_from_text (const char *text_p);
```

27.3.6.2 Description

The *inf_from_text*() function converts the text representation of an information label, *text_p* into its internal representation, and returns a pointer to a copy of the internal representation.

The system may allocate space for the information label to be returned. The caller should free any releasable memory when the new label is no longer required by calling *inf_free*() with the inf_t as an argument. In the event an error occurs, no memory shall be allocated and (*inf_t*)**NULL** shall be returned.

27.3.6.3 Returns

Upon successful completion, this function returns a pointer to the information label. Otherwise, no space is allocated, a value of (*inf_t*)**NULL** is returned, and *errno* is set to indicate the error.

27.3.6.4 Errors

If any of the following conditions occur, the *inf_from_text*() function shall return a value of (*inf_t*)**NULL** and set *errno* to the corresponding value:

[EINVAL] *text_p* is not a valid textual representation of an information label as defined by *inf_valid*().

[ENOMEM] The label to be returned required more memory than was allowed by the hardware or by system-imposed memory management constraints.

27.3.6.5 Cross-References

inf_free(), 27.3.5; *inf_to_text*(), 27.3.14; *inf_valid*(), 27.3.15.

27.3.7 Get the Information Label of a File Identified by File Descriptor

Function: *inf_get_fd*()

27.3.7.1 Synopsis

```
#include <sys/inf.h>

inf_t inf_get_fd (int fildes);
```

27.3.7.2 Description

The *inf_get_fd*() function returns the information label associated with a file. The function accepts a valid file descriptor and returns a pointer to the information label of the file referenced by the descriptor.

The system may allocate space for the information label to be returned. The caller should free any releasable memory when the new label is no longer required by calling *inf_free*() with the inf_t as an argument. In the event an error occurs, no memory shall be allocated and (*inf_t*)**NULL** shall be returned.

A process can get the information label of any file for which the process has a valid file descriptor. If {_POSIX_MAC} is defined, the process must also have MAC read access to the file.

27.3.7.3 Returns

Upon successful completion, this function returns the information label. Otherwise, no space is allocated, a value of (*inf_t*)**NULL** is returned, and *errno* is set to indicate the error.

27.3.7.4 Errors

If any of the following conditions occur, the *inf_get_fd*() function shall return a value of (*inf_t*)**NULL** and set *errno* to the corresponding value:

[EACCES] The required access to the file referred to by *fildes* was denied.

[EBADF] The *fildes* argument is not a valid file descriptor.

[ENOMEM] The label to be returned required more memory than was allowed by the hardware or by system-imposed memory management constraints.

27.3.7.5 Cross-References

inf_free(), 27.3.5; *inf_get_file*(), 27.3.8; *inf_set_fd*(), 27.3.10.

27.3.8 Get the Information Label of a File Identified by Pathname

Function: *inf_get_file*()

27.3.8.1 Synopsis

```
#include <sys/inf.h>

inf_t inf_get_file (const char *path_p);
```

27.3.8.2 Description

The *inf_get_file*() function returns the information label associated with a file. The function accepts a pathname to indicate the file. The function returns a pointer to the information label of the pathname pointed to by *path_p*.

The system may allocate space for the information label to be returned. The caller should free any releasable memory when the new label is no longer required by calling *inf_free*() with the inf_t as an argument. In the event an error occurs, no memory shall be allocated and (*inf_t*)**NULL** shall be returned.

A process can get the information label of any file for which the process has search access to the path specified. If {_POSIX_MAC} is defined, the process must also have MAC read access to the file.

27.3.8.3 Returns

Upon successful completion, this function returns the information label. Otherwise, no space is allocated, a value of (*inf_t*)**NULL** is returned, and *errno* is set to indicate the error.

27.3.8.4 Errors

If any of the following conditions occur, the *inf_get_file*() function shall return a value of (*inf_t*)**NULL** and set *errno* to the corresponding value:

[EACCES] Search permission is denied for a component of the path prefix or the required access to *path_p* is denied.

[ENAMETOOLONG]
The length of the pathname exceeds {PATH_MAX}, or a

pathname component is longer than {NAME_MAX} while {POSIX_NO_TRUNC} is in effect.

[ENOENT] The named file does not exist or the *path_p* argument points to an empty string.

[ENOMEM] The label to be returned required more memory than was allowed by the hardware or by system-imposed memory management constraints.

[ENOTDIR] A component of the path prefix is not a directory.

27.3.8.5 Cross-References

inf_free(), 27.3.5; *inf_get_fd*(), 27.3.7; *inf_set_file*(), 27.3.11.

27.3.9 Get the Process Information Label

Function: *inf_get_proc*()

27.3.9.1 Synopsis

```
#include <sys/inf.h>

inf_t inf_get_proc (void);
```

27.3.9.2 Description

The *inf_get_proc*() function returns a pointer to the information label associated with the requesting process.

The system may allocate space for the information label to be returned. The caller should free any releasable memory when the new label is no longer required by calling *inf_free*() with the inf_t as an argument. In the event an error occurs, no memory shall be allocated and (*inf_t*)**NULL** shall be returned.

27.3.9.3 Returns

Upon successful completion, this function returns the information label. Otherwise, no space is allocated, a value of (*inf_t*)**NULL** is returned, and *errno* is set to indicate the error.

27.3.9.4 Errors

If any of the following conditions occur, the *inf_get_proc*() function shall return a value of (*inf_t*)**NULL** and set *errno* to the corresponding value:

[ENOMEM] The label to be returned required more memory than was allowed by the hardware or by system-imposed memory management constraints.

27.3.9.5 Cross-References

inf_free(), 27.3.5; *inf_set_proc*(), 27.3.12.

27.3.10 Set the Information Label of a File Identified by File Descriptor

Function: *inf_set_fd*()

27.3.10.1 Synopsis

```
#include <sys/inf.h>
int inf_set_fd (int fildes, inf_t label);
```

27.3.10.2 Description

The *inf_set_fd*() function sets (writes) the information label of a file. The new information label is *label*. The function accepts a valid file descriptor to indicate the file.

A process can set the information label for a file using this function only if the process has a valid file descriptor for the file. If {_POSIX_MAC} is defined, the process must have mandatory write access to the file. Use of this function may also require appropriate privilege. If {_POSIX_CAP} is defined, and the effective user ID of the process is not equal to the file owner, appropriate privilege includes the CAP_FOWNER capability. In addition, if *label* is not equivalent to the information label associated with the file referred to by *fildes*, appropriate privilege includes the CAP_INF_RELABEL_OBJ capability.

27.3.10.3 Returns

Upon successful completion, this function returns a value of 0. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

27.3.10.4 Errors

If any of the following conditions occur, the *inf_set_fd*() function shall return –1 and set *errno* to the corresponding value:

[EACCES] The required access to the file referred to by *fildes* is denied.

[EBADF] The *fildes* argument is not a valid file descriptor.

[EINVAL] The label in *label* is not a valid information label as defined by *inf_valid*().

[ENOTSUP] *pathconf*() indicates that {_POSIX_INF_PRESENT} is not in effect for the file referenced.

[EPERM] The process does not have appropriate privilege to perform this operation.

[EROFS] This function requires modification of a file system which is currently read-only.

27.3.10.5 Cross-References

inf_get_fd(), 27.3.7; *inf_set_file*(), 27.3.11; *inf_valid*(), 27.3.15.

27.3.11 Set the Information Label of a File Identified by Pathname

Function: *inf_set_file*()

27.3.11.1 Synopsis

```
#include <sys/inf.h>
int inf_set_file (const char *path_p, inf_t label);
```

27.3.11.2 Description

The *inf_set_file*() function sets (writes) the information label of a file. The new information label is *label*. The function accepts a pathname to indicate the file.

A process can set the information label for a file only if the process has search access to the path specified. If {_POSIX_MAC} is defined, the process must have mandatory write access to the file. Use of this function may also require appropriate privilege. If {_POSIX_CAP} is defined, and the effective user ID of the process is not equal to the file owner, then appropriate privilege includes the CAP_FOWNER capability. In addition, if *label* is not equivalent to the information label associated with the file referred to by *path_p*, appropriate privilege includes the CAP_INF_RELABEL_OBJ capability.

27.3.11.3 Returns

Upon successful completion, this function returns a value of 0. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

27.3.11.4 Errors

If any of the following conditions occur, the *inf_set_file*() function shall return −1 and set *errno* to the corresponding value:

[EACCES] Search permission is denied for a component of the path prefix or the required access to *path_p* is denied.

[EINVAL] The label in *label* is not a valid information label as defined by *inf_valid*().

[ENAMETOOLONG]
The length of the pathname exceeded {PATH_MAX}, or a

pathname component is longer than {NAME_MAX} while {POSIX_NO_TRUNC} is in effect.

[ENOENT] The named file does not exist or the *path_p* argument points to an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

[ENOTSUP] *pathconf*() indicates that {_POSIX_INF_PRESENT} is not in effect for *path_p*.

[EPERM] The process does not have appropriate privilege to perform this operation.

[EROFS] This function requires modification of a file system which is currently read only.

27.3.11.5 Cross-References

inf_get_file(), 27.3.8; *inf_set_fd*(), 27.3.10; *inf_valid*(), 27.3.15.

27.3.12 Set the Process Information Label

Function: *inf_set_proc*()

27.3.12.1 Synopsis

```
#include <sys/inf.h>
int inf_set_proc (inf_t label);
```

27.3.12.2 Description

The *inf_set_proc*() function sets (writes) the information label of the requesting process. The new information label is *label*. If *label* is not equivalent to the information label associated with the process, then appropriate privilege is required for this operation. If {_POSIX_CAP} is defined, appropriate privilege includes the CAP_INF_RELABEL_SUBJ capability.

27.3.12.3 Returns

Upon successful completion, *inf_set_proc*() returns a value of 0. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

27.3.12.4 Errors

If any of the following conditions occur, the *inf_set_proc*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The label in *label* is not a valid information label as defined by *inf_valid*().

[EPERM] The process does not have appropriate privilege to perform this operation.

27.3.12.5 Cross-References

inf_get_proc(), 27.3.9; *inf_valid*(), 27.3.15.

27.3.13 Get the Size of an Information Label

Function: *inf_size*()

27.3.13.1 Synopsis

```
#include <sys/inf.h>

ssize_t inf_size (inf_t label);
```

27.3.13.2 Description

The *inf_size*() function returns the size in bytes of the internal representation of the information label in *label*, if it is valid.

27.3.13.3 Returns

Upon successful completion, the function returns the size of the information label. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

27.3.13.4 Errors

If any of the following conditions occur, the *inf_size*() function shall return –1 and set *errno* to the corresponding value:

[EINVAL] The *label* argument is not a valid information label as defined by *inf_valid*().

27.3.13.5 Cross-References

inf_free(), 27.3.5; *inf_valid*(), 27.3.15.

27.3.14 Convert Internal Label Representation to Text

Function: *inf_to_text*()

27.3.14.1 Synopsis

```
#include <sys/inf.h>
char *inf_to_text (inf_t label, size_t *len_p);
```

27.3.14.2 Description

The *inf_to_text*() function converts the information label contained in *label* into a human readable, **NULL**-terminated, character string which shall be suitable for the *text_p* parameter to *inf_from_text()* in section 27.3.9 and for re-input as the *inflabel* operand to the `setfinf` utility as defined in section 12 of POSIX.2c. This function returns a pointer to the string. If the pointer *len_p* is not **NULL**, the function shall also return the length of the string (not including the **NULL** terminator) in the location pointed to by *len_p*. The information label in *label* shall be completely represented in the returned character string.

The system may allocate space for the string to be returned. The caller should free any releasable memory when the string is no longer required by calling *inf_free*() with the char * as an argument. In the event an error occurs, no memory shall be allocated and (*inf_t*)**NULL** shall be returned.

27.3.14.3 Returns

Upon successful completion, *inf_to_text*() returns a pointer to the text representation. Otherwise, in all cases, the memory referred to by *len_p* shall remain unchanged, a value of (*char* *)**NULL** is returned, and *errno* is set to indicate the error.

27.3.14.4 Errors

If any of the following conditions occur, the *inf_to_text*() function shall return a value of (*char* *)**NULL** and set *errno* to the corresponding value:

[EINVAL] The label in *label* is not a valid information label as defined by *inf_valid*().

[ENOMEM] The text to be returned required more memory than was allowed by the hardware or by system-imposed memory management constraints.

27.3.14.5 Cross-References

inf_free(), 27.3.5; *inf_from_text*(), 27.3.6; *inf_valid*(), 27.3.15; `setfinf`, 12.3.

27.3.15 Information Label Validity

Function: *inf_valid*()

27.3.15.1 Synopsis

```
#include <sys/inf.h>
int inf_valid (inf_t label);
```

27.3.15.2 Description

The *inf_valid*() function determines whether the label in *label* is a valid information label. The precise meaning of validity is implementation-defined. Examples of some reasons why a label may be considered invalid include: the label is malformed, the label contains components that are not currently defined on the system, or the label is simply forbidden to be dealt with by the system.

27.3.15.3 Returns

Upon successful completion, the function returns 1 if *label* is valid, and 0 if it is invalid. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

27.3.15.4 Errors

This standard does not specify any error conditions that are required to be detected for the *inf_valid*() function. Some errors may be detected under conditions that are unspecified by this part of the standard.

27.3.15.5 Cross-References

None.

Annex B
(informative)

Revisions to Rationale and Notes

B.1 Revisions to Scope and Normative References

⇒ **B.1.1 Scope** *This rationale is to be revised and integrated appropriately into the scope rationale when POSIX.1e is approved:*

The goal of this standard is to specify an interface to protection, audit, and control functions for a POSIX.1 system in order to promote application portability. Implementation of any or all of these interfaces does not ensure the security of the conforming system or of conforming applications. In particular, there is no assurance that a vendor will implement the interfaces in a secure fashion or that the implementation of the interfaces will not cause additional security flaws. Even if such assurances were required or provided, there are many more aspects of a "secure system" than the interfaces defined in this standard.

This interface is extendible to allow for innovations that provide greater (or different) security functions in various markets. It is expected that conforming implementations may augment the mechanisms defined in this standard and may also provide security functions in areas not included in this standard.

It was not a goal of this document to address assurance requirements which constrain the implementation and not the interface. POSIX.1 standards define operating system interfaces only and attempt to allow for the greatest possible latitude in implementation so as to promote greater acceptance of the standards.

The United States Department of Defense Trusted Computer System Evaluation Criteria (TCSEC) document was a main source of requirements for this standard. The TCSEC is a comprehensive set of guidelines which has received extensive review. The TCSEC requirements are themselves general, and have been used to guide the development of a variety of computer systems, ranging from general purpose time-sharing systems to specialized networking components. The TCSEC has received broad distribution and acceptance and has been the basis for much of the work which followed it. Functions are drawn from all TCSEC classes where it is agreed that inclusion of the function in the standard will enhance application portability.

Even though the TCSEC was a source of requirements for the interfaces defined in this standard, this standard is not to be construed as defining a set of interfaces intended to satisfy the requirements of any particular level.

⇒ **B.1.3.6 Supported Security Mechanisms (POSIX.1: line 474)** *Add the following new section:*

B.1.3.6 Supported Security Mechanisms

The security mechanisms supported by this standard were chosen for their generality. The specific interfaces defined were selected because they were perceived to be generally useful to applications (trusted and untrusted). Two mechanisms, access control lists and privilege, are defined specifically to address areas in the POSIX.1 standard that were deferred to this standard.

⇒ **B.1.3.7 Unsupported Security Mechanisms** *Add the following new sections B.1.3.7 - B.1.3.7.11:*

B.1.3.7 Unsupported Security Mechanisms

The purpose of this standard is to provide for application portability between conforming systems. As a result, this standard does not address several functional security-related issues. Specifically, the POSIX.1e standard does not address:

(1) Identification and Authentication

(2) Networking Services and Protocols

(3) Administrative Services and Management of Security Information

(4) Covert Channels –

(5) Assurance Issues

(6) Evaluation Ratings Based on Current Trust Criteria

(7) The General Terminal Interface as described in the POSIX.1 standard

The rationale for excluding these and other potentially relevant topics is provided below.

B.1.3.7.1 Identification and Authentication

I&A mechanisms are being deferred to a future version of this standard. It was felt that the I&A mechanism should take into consideration third-party authentication schemes. It was also felt that deferring this area to a future standard would allow existing practice to become more stabilized prior to standardization.

B.1.3.7.2 Networking Services

Networking services are being deferred to a future version of this standard. This was done to allow the various POSIX Distributed Services working groups to further progress their work prior to standardization. It was also felt that deferring this area to a future standard would allow existing practice to become more stabilized prior to standardization.

B.1.3.7.3 Administrative Services and Management of Security Information

Administrative services and the management of security information are being deferred to a future version of this standard. This was done to allow the POSIX System Administration working group to further progress their work prior to standardization. System administration will ultimately be standardized through a document that is distinct from the POSIX.1 or POSIX.2 standards. The current POSIX.1e work is limited to modifications to the POSIX.1 and POSIX.2 standards.

B.1.3.7.4 Covert Channels

Covert channel analysis is undertaken from the perspective of the interface, and not the underlying implementation. This means that covert channels associated with resource exhaustion, e.g., process IDs, i-nodes, and file descriptors, are not considered. Covert channels visible at the interface are treated. These include the use of exclusive locks and the updating of file access times.

B.1.3.7.5 Assurance Issues

Assurance issues that do not require function or utility interfaces are not explicitly treated as part of the standard. But assurance requirements that constrain the system interfaces are implicitly part of the standard. The principal issues here are:

B.1.3.7.5.1 Modularity, Security Kernels, Software Engineering

These are mostly kernel internals design and implementation issues, which are beyond the scope of POSIX standards.

B.1.3.7.5.2 Minimality

The TCB minimality assurance requirement is not addressed by this standard. This is an implementation question only.

The minimality requirement, introduced at the B3 level of the TCSEC, does not constrain the definition of any POSIX.1e interface, because minimality pertains only to the definition of the partition between the trusted code of the system,i.e., the TCB, and the untrusted code of the system. This standard does not specify that the interfaces it defines must be TCB interfaces.

B.1.3.7.5.3 System Integrity

System Integrity interfaces are being deferred to a future version of this standard. It was felt that deferring this area to a future standard would allow existing practice to become more stabilized prior to standardization.

B.1.3.7.5.4 Formal Security Policy Model

No security policy models are defined as part of this standard because the standard is not intended to define a complete system. In some areas the implementation may want to extend the standard, and in other areas the implementation will have to extend the standard. Given this incompleteness, a model would be difficult (and perhaps impossible) to define. Also, a full, formal model would constrain implementations beyond the point necessary for application portability.

B.1.3.7.5.5 Separation of Administrative Roles

Without a complete definition of administrative function, this is clearly beyond the scope of this standard. Also, this is an area where implementations may wish to target particular and isolated installations.

B.1.3.7.5.6 Resource Controls

Resource controls (quotas) are used to support a system availability policy. They are not included in this standard because of a lack of existing practice in UNIX systems and, more importantly, the resources controlled tend to reflect implementation limits (static tables, ...) rather than physical ones.

B.1.3.7.5.7 Trusted Path

A Trusted Path mechanism is not defined because the notion of terminal defined in POSIX.1 is limited to dumb ttys, and is incomplete as well. Existing practice is lacking here as well. The standardization of the key sequence used for invoking the trusted path is possible, but it would also be necessary to define the behavior of the system upon trusted path invocation. It was felt that this would be impossible without a well-defined Trusted Path.

B.1.3.7.5.8 Protected Subsystems

The UNIX-protected subsystem mechanism (programs with the set-user-ID or set-group-ID mode bits set) is subject to abuse by knowledgeable users and misuse by naive users. Its shortcomings are not addressed due to some notable disagreements concerning the desirability of the mechanism. It also doesn't add much to portability.

B.1.3.7.6 Evaluation Ratings Based on Current Trust Criteria

Evaluations of products under current trust criteria involve analysis of all aspects of the product, especially of implementation details. This standard only deals with interfaces. Therefore, it is inherently incomplete and unsuited for evaluation under these criteria. In addition, a conforming system could implement the functionality under the interfaces in an insecure manner. Therefore, conformance to this standard does not guarantee that a system should be trusted.

B.1.3.7.7 General Terminal Interface

This standard does not extend General Terminal Interfaces described in sections 7.1 and 7.2. This section explains some of the problems with the GTI from a security perspective.

The existing interfaces do not require that the file descriptor used for changing terminal attributes be opened for writing. Given the MAC policy of read-down, a process could open a terminal which it dominates, and by manipulating terminal attributes perform data downgrade. This violates the basic MAC policies. Requiring that the device is opened for write (or that the process have MAC write access) solves this problem.

Manipulation of device attributes can interfere with invocation of trusted path. For example, a process could change the baud rate of its controlling terminal. The trusted path would be unable to determine if the baud rate was changed at the user's request, i.e., because the baud rate was adjusted on the physical terminal, or by a malicious or malfunctioning application. Thus, the user might be unable to communicate via the trusted path. Changing the baud rate should be restricted using privilege or trusted path.

Applications may cause output to be suspended (using the *tcflow*() function with the action set to TCOOFF). If the trusted path is invoked in such a case, the standard would need to define what happens, i.e., the trusted path can re-enable output, but the status of queued output would need to be determined. An appropriate solution to this problem is not clear.

While these problems generally involve trusted path (which is not a part of the standard), it is important not to enact a standard which would preclude building a system that includes a trusted path mechanism.

⇒ **B.1.3.8 Portable Trusted Applications** *Add the following new sections B.1.3.8:*

B.1.3.8 Portable Trusted Applications

Portable trusted applications are those applications that are: portable because the system call interface they use is that defined by POSIX.1e; and trusted, because they perform some security-related functionality and/or need some privilege from the system in order to function correctly, and which therefore must be trusted to perform the security-related functionality correctly and/or to not abuse the privilege granted to the application.

Such portable trusted applications may rely on the TCB of the host system to perform certain security-critical functions that are necessary to ensure the correct and secure operation of the portable trusted application. For example: a portable trusted application may need to protect some persistent data from tampering by unauthorized processes, and may therefore use DAC features to control access to the persistent data as stored in a file.

If the secure operation of the portable trusted application depends on the correct operation of such POSIX.1e functions, then those POSIX.1e functions must be implemented by the TCB of the host system on which the application is running; otherwise, the portable trusted application would be relying on untrusted code to perform functionality upon which the security of the portable trusted application depends.

Furthermore, the secure state of the entire system may be at stake if the portable trusted application runs with system privileges, because the portable trusted application may operate incorrectly and abuse its privilege as a result of malfunction of untrusted code performing functionality which is security-related as used by the portable trusted application. However, the interfaces defined in this standard are not required to be TCB interfaces.

As a result, a portable trusted application may be portable to various POSIX.1e-conformant systems, but only some of those conformant systems may actually implement as TCB interfaces those POSIX.1e interface functions upon which depends the secure operation of the portable trusted application. Therefore, portable trusted applications under some circumstances may not be trust-worthy even when run on conformant systems. Proper use of portable trusted applications depends on the specification of the system interfaces which are security-critical to the portable trusted application, and the determination of whether all those interfaces are implemented by the TCB of a system which can run the portable trusted application.

B.2 Revisions to Definitions and General Requirements

⇒ **B.2.2.2 General Terms** *Insert the following after line 986:*

user: the term user is used in this document to denote a person who interacts with a computer system. It is not meant to include programs that "look like" users.

⇒ **B.2.10 Security Interface (POSIX.1: line 1741)** *Add the following sections B.2.10 and B.2.10.1:*

B.2.10 Security Interface

B.2.10.1 Opaque Data Objects

Each functional area (MAC, ACL, IL, capabilities, and audit) defines one or more opaque data objects. Certain restrictions are applied to some of those opaque data objects, namely persistence and self-containment. This section describes the rationale for these requirements and their implications.

Opaque data objects by definition can contain any type of data, in any form, so long as the functions which manipulate those objects understand that form. For example, Access Control Lists are frequently implemented as linked lists. However, some applications need to pass opaque objects to other processes (e.g., by writing them in FIFOs), or to store them in files. For example, a trusted database system might store a MAC label for each record in the database. Truly opaque data cannot be stored, because an application does not know how much to store, and there is no guarantee that the data will be meaningful when retrieved from the database.

In each section, an interface is provided to free memory associated with data structures. (Thus, for example, there are *mac_free*(), *inf_free*(), etc., routines). The description of these routines state that they free any "releasable" memory. Once these routines have been called, the data structure freed can no longer be used by applications: in general, these routines will deallocate all memory associated with the data structure. That is, the **_free*() routines generally work analogously to the *malloc*() and *free*() routines of standard C. However, no requirement imposed by this standard that requires all allocated memory to be freed. Conforming implementations, then, can use their own memory management schemes. Nevertheless, portable applications must assume that the memory freed has been completely deallocated and that any pointers to the freed data structure are no longer valid.

B.3 Revisions to Process Primitives

⇒ **B.3.1.2 Process Creation (POSIX.1: 1770)** *Rationale for changes to this section in POSIX.1 is provided below:*

When a new process is created via a *fork*() call, the new process is an exact copy of its parent, including the current MAC label, information label, etc. Because this standard does not define the contents of many data structures, it is important to note that both the parent and child may continue using data structures independently.

For example, consider an implementation where a MAC label structure (that is an object referenced by *mac_t*) is simply a number. That number could be an index into a kernel table. Functions which use the MAC label could make kernel calls, and all manipulation of the MAC label would take place in the kernel. When the *fork*() function is executed, the system must duplicate the kernel table so both the parent and child processes are able to modify the MAC label without interfering with each other.

⇒ **B.3.1.2 Execute a File (POSIX.1: line 1821)** *Rationale for changes to this section in POSIX.1 is provided below:*

At first glance it might appear that a child's information label should be set either to the information label of the file being executed, or to the lowest label in the system. However, the process performing the *exec*∗() operation can pass information to the new process image by way of file descriptors and environment variables. Hence, the old process information label should be incorporated in the new process information label. Note that the standard recommends an information label, but does not require it; other information label policies are possible and allowed by this standard. Additionally, the standard does not require use of the *inf_float*() function to calculate the new information label; this is a suggestion of one way to perform the calculation.

⇒ **B.3.3.2 Send a Signal to a Process (POSIX.1: line 2428)** *Rationale for changes to this section in POSIX.1 is provided below:*

Using a signal between two processes is effectively sending data. While the amount of data (the signal number) is small, this standard is careful to avoid requiring information flow which contradicts the MAC security policy. Hence, the four cases described in the standard:

MAC label of sender equivalent to MAC label of receiver: no MAC restrictions

MAC label of sender dominates MAC label of receiver (i.e., write-down): appropriate privilege is required, and if {_POSIX_CAP} is defined, appropriate privilege includes the capability CAP_MAC_WRITE.

MAC label of receiver dominates MAC label of sender (i.e., write-up): appropriate privilege may or may not be required. A write-up is not an inherent violation of the security policy, except that the sender is able to determine the existence of a higher level process. Systems which address covert channels may wish to close this channel by requiring appropriate privilege. If {_POSIX_CAP} is defined, appropriate privilege includes the capability CAP_MAC_READ (because the existence of the higher level process is read).

MAC label of sender and receiver are incomparable: in this case, appropriate privilege is certainly required at least as strong as the case where the label of the sender dominates that of the receiver. If {_POSIX_CAP} is defined, appropriate privilege includes the capability CAP_MAC_WRITE. In addition, implementations may require appropriate privilege to perform the read-up, viewing the operation as a write-down followed by a read-up. In this case, if {_POSIX_CAP} is defined, appropriate privilege includes the capability CAP_MAC_READ. However, the additional capability is not defined by the standard, since implementations are free to add additional restrictions as desired.

The *kill*() function allows notification of a process group. The error code is defined in POSIX.1 as success if any signal was sent, and a failure only if no processes could be signaled. This standard extends that notion: if a process group contains processes with different MAC labels, then a signal is successfully sent to the process group if even a single process in the group can be signaled. This is consistent with the notion in IEEE Std 1003.1-1990 where a signal could be successful even if processes in the process group have different user IDs, and hence only some of them can be signaled.

If not even one process can be signaled, then there are two possible errors returned: [EPERM] and [ESRCH]. [EPERM] is used when the sending process dominates at least one of the potential receiving processes, but did not have the required appropriate privilege to send the signal. In this case, the sending process could determine the existence of the potential receiver, so no information channel exists by returning [EPERM]. By contrast, [ESRCH] is returned to indicate that either the process group did not exist, or none of the processes in the process group were visible to the sending process.

While this standard imposes no information label requirements on signals, implementations may consider the signal as having an information label, and hence float the information label of the receiving process to include the information label of the sending process.

This standard does not extend the notion of access control based on user IDs to include the notion of an access control list on a process.

Another architecture not discussed by this standard is to allow overrides of the signaling policy based on the privileges of the receiver. In such an architecture, a daemon process could be set up to accept signals from any process, regardless of the MAC label of the sender. However, the POSIX.1 standard does not recognize this notion for user ID based privileges, so this standard does not extend it for MAC.

⇒ **B.4 Revisions to Process Environment (POSIX.1: line 2645)** *Rationale for changes to this section in POSIX.1 is provided below:*

As previously described, each of the options described in this standard may be selected independently. The *sysconf*() variables listed in this section are to allow programs to determine at runtime whether the option is available.

⇒ **B.5 Files and Directories (POSIX.1: line 2896)** *Rationale for changes to this section in POSIX.1 is provided below:*

The extensions specified in this standard for file access avoid changing the interfaces specified in POSIX.1 any more than necessary. Specifically, no changes are made to parameter types, and where data structures are involved, no changes are made to add or remove elements from the structure. In some cases the data returned by the interface may be changed. This is most noticeable when examining the file permission bits of a file which has an access control list.

⇒ **B.5.3.1 Open a File (POSIX.1: line 3077)** *Rationale for changes to this section in POSIX.1 is provided below:*

While it might appear that a newly created file would always have the information label *inf_default*() this is not true. For example, implementations might set the information label of a new file to the information label of the containing directory or the information label of the creating process.

When opening a FIFO, the MAC restriction should be that process and FIFO MAC labels should be equivalent to avoid massive covert channels associated with MAC inequalities. Since the MAC policy defined by this standard allows MAC write-up, it is possible to be POSIX compliant and still include this covert channel. However, since the normal MAC policy is write-equals, this is not a major concern.

⇒ **B.5.6.2 Get File Status (POSIX.1: line 3208)** *Rationale for changes to this section in POSIX.1 is provided below:*

The *stat*() call in POSIX.1 provides the caller with all file attributes. This standard does not extend *stat*() to return the extended attributes such as MAC label or access control list. There were several reasons:

This standard had as a goal to leave the syntax of existing interfaces unchanged.

The data structures defined in this standard are potentially variable length, unlike in POSIX.1 where they are all fixed length. Thus, the *stat* structure would have to be adapted to handle pointers to the variable length items. This would make the interface more complicated.

Each portion of this standard is independent, so not all data types are necessarily defined. Thus, the *stat* structure would have to be set up differently depending which options are provided.

Existing programs designed to use a version of *stat*() as defined in POSIX.1 might get back additional information. If the program had not been recompiled to allow for a larger structure, this might overwrite other data, and cause the program to fail.

Thus, the standard leaves *stat*() unchanged, and adds new functions for getting the individual extended file attributes.

Note that if {_POSIX_ACL} is defined and {_POSIX_ACL_EXTENDED} is in effect for the pathname, the semantics of *stat*() and *fstat*() are changed. Specifically, *stat*() and *fstat*() no longer return all the discretionary access information, so applications that depend on it doing so (e.g., when copying discretionary file attributes to another file) may have to be changed.

⇒ **B.5.6.3 File Access (POSIX.1: line 3216)** *Rationale for changes to this section in POSIX.1 is provided below:*

POSIX.1 does not list the specific permissions required for each function (e.g., *open*(), *mkdir*(). Rather, it relies on the descriptions of pathname resolution and file access in POSIX.1, 2.3, together with additional information (e.g., error codes) in the individual function descriptions. For example, the description of *open*() does not specify that the caller must have search access to each pathname component, and must also have write access to the directory if a new file is being created. The pathname resolution portion is implicit from POSIX.1, 2.3, and write access to the parent directory is provided by the description of the EACCES error number.

In a similar fashion, this standard does not describe the MAC requirements for file access, instead referring to POSIX.1, 2.3. Additional information is provided where appropriate, such as linking files (which requires MAC write permission to the existing file) and opening a FIFO (which requires MAC write permission to the FIFO file).

Unlinking a file might appear to need MAC write access to the containing directory only. However, the unlink operation updates the link count on the file, which is effectively a write operation to the file. Hence, MAC write access is required. Similarly, removing a directory updates the directory link count, and consequently MAC write access is required to the directory being removed.

Clearing setuid/setgid and Privileges

One security-relevant issue not addressed by this standard is resetting of the setuid/setgid bits. For example, most historical implementations clear the setuid and setgid bits when a file is written into. The security risk is that if a setuid utility is improperly installed (e.g., with write permission) and the setuid bit is not cleared, a malicious user could replace the utility with a different version. However, neither IEEE Std 1003.1-1990 nor this standard require (nor prohibit) clearing the setuid and setgid bits.

There were several reasons for not specifying the behavior. The most important was determining which interfaces should trigger clearing setuid/setgid bits. Should they be cleared when the file is opened, when it is written to, when it is closed, or some combination? Each leaves certain timing windows, and has potential performance implications.

The capability flags provided by this standard provide an extension to the notion of setuid/setgid, with somewhat finer granularity. If setuid/setgid bits are to be cleared, should capability flags also be cleared? Just as this standard makes no statements about setuid/setgid, it does not require (nor prohibit) clearing of capability flags.

If capability flags are cleared when a file is written, the implementor should also consider whether they should be cleared when file attributes are changed. For example, consider a program file which has the MAC read-up exemption capability, and the file has a MAC label of secret. When executed, that program may read top secret data, but at worst it can relabel it as secret (because only a user with at least a secret security level will be able to access the file, and hence execute the program). If the file's MAC label is changed to unclassified, then an uncleared user may be able to execute it, thus allowing top secret data to be written into an unclassified file. Thus, the change in the MAC label of a file impacted the system security, by allowing additional risks. System implementors may wish to consider these types of threats, even though they are not required by this standard.

Finally, system implementors should consider whether capability and setuid/setgid bits should be cleared when the file owner is changed.

Object Reuse and File Erasure

Another topic of concern in trusted systems is object reuse, particularly as it applies to files. POSIX.1 requires that newly allocated files be cleared, so the previous contents of the file are inaccessible. While some historical systems overwrite the contents of a file when the file is deleted, this standard imposes no such requirement. Because the contents are cleared when the file is first read, this is not an issue except when the device which stores the file (i.e., the disk) can be accessed outside the file system (e.g., through a raw device). Such concepts are beyond the scope of this standard.

Initial Information Labels

When a file (including a directory or FIFO) is created, the initial information label on the file must be set. This standard does not specify an information label policy. Hence, the standard does not specify what the initial label will be. In most cases the initial label will be the same as the result of a call to *inf_default*().

⇒ **B.6.1 Pipes (POSIX.1: line 3380)** *Rationale for changes to this section in POSIX.1 is provided below:*

Pipes provide communication between related processes (typically a parent and child). Excluding the effects of privileged processes, the related processes by definition have the same MAC label. Hence, specifying the MAC label of the pipe is somewhat irrelevant. However, processes can request the MAC label of the file associated with a file descriptor. This standard defines the MAC label of the pipe as the MAC label of the creating process so such a request can be answered.

⇒ B.6.5.2 File Locking (POSIX.1: line 3613)

The file locking mechanism defined in IEEE Std 1003.1-1990 allows advisory locks to be placed and detected on a file. The mechanism does not specify the file mode used by processes placing or testing the locks. When a MAC policy is added, the locking mechanism can be used as an information flow channel. At earlier stages of development of this standard strict requirements for MAC access were specified and varying capabilities specified to obtain MAC access. Due to significant ballot objections to the granularity of the capabilities required, it was decided to let this standard be mute on the enforcement of MAC for file locking operations. Implementations concerned with closing the information flow channel have been left free to handle the channel in whatever way they choose. See B.25.4.3 for more discussion of this issue.

⇒ B.8 Language-Specific Services for the C Programming Language

Rationale for changes to this section in POSIX.1 is provided below:

Historical implementations implement the interfaces defined in this section using the base POSIX.1 interfaces. This concept is reflected by the description of the interfaces as having *underlying functions*. However, there is no requirement that implementations use the underlying functions, as noted in POSIX.1 Section 8, lines 341-345. As a result, this standard defines the extensions to the C standard I/O primitives.

Some consideration was given to defining security effects of making a *longjmp*() call. For example, to provide time bounding of capabilities the current capability set could be restored to its state as of the *setjmp*() call. This standard makes no such requirements, as applications are not required to time bound capabilities. Rather, applications developers are encouraged to clear appropriate capabilities in the code invoked from the *longjmp*() call.

B.23 Access Control Lists

The overall requirements for an Access Control List (ACL) mechanism in a secure system include the following:

(1) Allow authorized users to specify and control sharing of objects

(2) Supply discretionary access controls for objects.

(3) Specify discretionary access by a list of users and groups with their respective access rights to the protected objects

(4) Allow discretionary access to an object to be denied for a user or, in certain cases, a group of users.

(5) Allow changes to the ACL only by the owner of the object or by a process with the required access or appropriate privilege.

(6) Not allow more permissive discretionary access than either the initial or final access rights while the ACL is being written by *acl_set_file*() or *acl_set_fd*().

The primary goal in defining access control lists in a POSIX.1e system is to provide a finer granularity of control in specifying user and/or group access to objects. Additional goals for the ACL mechanism are:

(1) The mechanism should be compatible with the existing POSIX.1 and POSIX.2 standards and, to the extent possible, existing interfaces should continue to work as expected.

(2) Reasonable vendor extensions to the ACL mechanism should not be precluded. At a minimum, the specification of read, write and execute/search permissions should be supported. Other permissions should neither be required nor should they be precluded as extensions.

(3) New interfaces should be easy to use.

(4) Intermixing use between the existing mechanism and newly defined ACL functions/utilities should provide predictable, well understood results.

Another goal is to be compatible with existing POSIX.1 standards. Current interfaces will continue to exist and will affect the overall ACL. Some users will continue to only use the file permission bits. Existing programs may not be modified to use the ACL interface and may continue to manipulate DAC attributes using current POSIX.1 interfaces. These programs should operate on objects with ACLs in a manner similar to their operation on objects without ACLs. However, complete compatibility between the existing POSIX.1 DAC interfaces and the POSIX.1e ACL interfaces is simply not achievable. For a discussion of these issues, please refer to B.23.1.

The POSIX.1e ACL interfaces should not restrict vendors from providing extensions to the basic ACL mechanism; the POSIX.1e ACL interface should not exclude such extensions.

For the sake of usability and user acceptance, new interfaces should be as simple as possible while maintaining a reasonable level of compatibility with existing POSIX.1 interfaces.

The intermixing of usage between the existing POSIX.1 DAC and the POSIX.1e ACL mechanisms should be well defined and produce reasonable results.

The DAC interfaces described in POSIX.1 are adequate for some needs. The file permission bits defined in POSIX.1 are associated with three classes: owner, group, and other; access for each class is represented by a three-bit field allowing for read, write, and execute/search permissions. The POSIX.1e ACL interfaces extend the POSIX.1 interfaces by defining access control lists (ACLs) in order to provide finer granularity in the control of access to objects. ACLs can provide the ability to allow or deny access for individually-specified users and groups of users. However, implementations which allow processes to modify the process' group membership may not be capable of denying access to users based on groups.

Several methods exist for allowing discretionary access control on objects. These methods include capability lists, profiles, access control lists (ACLs), permission bits, and password DAC mechanisms. ACLs were selected for the POSIX.1e interfaces because they meet the goals stated earlier in this section. ACLs are a straightforward extension of the existing POSIX.1 file permission bits which may be viewed as a limited form of ACL containing only three entries.

The following features are outside the scope of this document:

— Shared ACLs

An ACL is shared if it is associated with more than one object; changes to a shared ACL affect the discretionary access for all objects with which the ACL is associated. Shared ACLs are useful as a single point of control for the specification of DAC attributes for large numbers of objects.

Although the implementation of shared ACLs is not precluded, shared ACLs are not defined in this standard for the following reasons:

- It may be difficult to determine the set of objects sharing an ACL. A user could modify the ACL associated with an object and unintentionally grant access to another object.
- When changing a shared ACL, it may be necessary to produce an audit record for each file system object that is protected by the ACL.
- Any changes to a shared ACL which have an unintended security result affect all objects sharing the ACL.

— Named ACLs

A named ACL is an ACL which exists in the file system space and can be referred to by name. Named ACLs are primarily useful for implementing shared ACLs.

Although the implementation of named ACLs is not precluded, named ACLs are not defined in this standard for the following reasons:

- As file system objects, ACLs themselves may be required to contain discretionary access controls which could require recursive ACLs.
- The owner of a named ACL may not be the owner of the object(s) with which the ACL is associated. The owner of an object could lose control of the DAC attributes associated with that object.

B.23.1 General Overview

POSIX.1 specifies basic DAC interfaces consisting of permissions which specify the access granted to processes in the file owner class, the file group class, and the file other class. These classes correspond to the intuitive notions of the file's owner, members of the file's owning group, and all other users.

B.23.1.1 Extensions to POSIX.1 DAC Interfaces

The specification of the POSIX.1 interfaces provides for two ways to extend discretionary access controls beyond the basic file permission bits:

- An additional access control mechanism may be provided by an implementation, however, the mechanism must only further restrict the access permissions granted by the file permission bits.
- An alternate access control mechanism may be provided by implementation, however, POSIX.1 requires that a *chmod*() function call disable any alternate access control attributes which may be associated with the file.

The POSIX.1e access control interfaces are defined as an additional access control mechanism in order to satisfy the basic goal of working in conjunction with the existing DAC functions and commands; essentially, the ACL interfaces can be viewed as an extension of the base POSIX.1 file permission bits. Also, the POSIX.1e definition of the ACL interfaces only further restrict the access specified by the file permission bits. If the POSIX.1e interfaces were to be defined as an alternate access mechanism, then the POSIX.1e interfaces would have to operate independently of the existing POSIX.1 interfaces with no correlation between the permissions granted by the alternate mechanism and the file permission bits.

B.23.1.2 Extensions to File Classes

POSIX.1 permits that implementation-defined members may be added to the file group class. As such, the ACL entries for individually specified users and groups are defined as members of the file group class. Since the file permission bits for the file group class are defined as the maximum permissions which can be granted to any member of the file group class, then the POSIX.1e interfaces conform to the POSIX.1 definition of an additional access mechanism.

An alternative is to define the additional ACL entries as members of the file other class instead of the file group class. The apparent advantage of extending the file other class is that the permissions granted to the file's owning group would be explicitly specified in the base file permission bits. However, this would not be the case since individually named user entries would be checked prior to the owning group permissions even if the specified user was a member of the owning group.

Refer to B.23.3 for more details on how ACL entries map to the different file classes.

B.23.2 ACL Entry Composition

An ACL entry consists of at least three pieces of information as defined in the standard: the type of ACL entry, the entry tag qualifier, and the access permissions associated with the entry. The standard permits conforming implementations to include additional pieces of information in an ACL entry.

B.23.2.1 ACL Entry Tag Type Field

Seven distinct ACL entry tag types are defined to be the minimum set of tag types which must be supported by a conforming implementation: ACL_USER_OBJ, ACL_GROUP_OBJ, ACL_OTHER, ACL_USER, ACL_GROUP, ACL_MASK, and ACL_UNDEFINED_TAG.

The ACL_USER_OBJ, ACL_GROUP_OBJ, and ACL_OTHER tag type ACL entries are required to exist in all ACLs. If no other entries exist in the ACL, then these entries correspond to the owner, group, and other file permission bits. Since these permission bits can never be removed from a file, the ACL entries corresponding to the permission bits are also required. If an ACL contains any additional ACL entries, then an ACL_MASK entry is also required since it then corresponds to the file group permissions and serves as the maximum permissions that may be granted to the additional ACL entries.

While implementations can define additional tag types, the standard does allow an implementation to require the existence of any additional entries in an ACL. If this were allowed, then an file containing only the file permission bits (i.e., an ACL with only three entries) would not be a valid ACL. This would prevent a strictly conforming application from executing correctly on such an implementation which would violate the goal of providing compatibility with the existing POSIX.1 interfaces.

An additional ACL entry tag type that could be defined is a "user and group" where such entries specify the access permissions for an individual user within a specific group. While such an ACL entry is useful in some environments, it is not required in the standard since it does not appear to provide widely useful functionality. Implementations are not precluded from defining a "user and group" tag type.

Implementations which currently allow "user and group" tag type ACL entries can consider the ACL_USER_OBJ and ACL_USER ACL entry tag types to represent access to a user regardless of group membership, e.g., "user.*". Likewise, ACL_GROUP_OBJ and ACL_GROUP ACL tag types represent group access regardless of user identity, e.g., "*.group", and ACL_OTHER represents anybody in any group, e.g., "*.*".

The names of all ACL entry tag types all begin with the prefix "ACL_" in order to provide consistency in naming with other areas of the POSIX standards. While this may make the use of such names slightly more cumbersome for the programmer, avoiding name conflict through a consistent naming scheme is more important.

POSIX.1e defines two types of ACLs: access and default ACLs. All objects have an access ACL since the POSIX.1 file permission bits are interpreted as a minimal ACL. In addition, a default ACL may be associated with a directory. The rules for ACL entry tag types are the same for both types of ACL. As such, an application can create an ACL and apply it to a file as either an access ACL or a default ACL without changing the ACL structure or any of the ACL data. If POSIX.1e defined ACL entry types which applied to only one type of ACL or if the rules for required ACL entries differed between the types of ACL, then a single ACL could not be applied as both an access and a default ACL.

B.23.2.2 ACL Entry Qualifier Field

The data type of the qualifier field in an ACL entry is specific to the ACL entry tag type. Also, the qualifier field is not extensible for POSIX.1e defined tag types. However, implementations may define the type and structure of the qualifier for entries with implementation-defined tag types. For example, an implementation that wishes to allow the assignment of permissions to an individual user within a specific group could create a tag type, ACL_USER_GROUP, with a qualifier containing the identification of both the user and the group. An implementation could also define a user/time entry which could use the qualifier to identify a process within a specified time of day interval.

If an implementation could extend the POSIX.1e defined ACL entry qualifier fields, then a strictly conforming application might not function as expected when manipulating an ACL with extended qualifier fields. For example, an implementation extends the qualifier field of the ACL_USER entry type to include a time of day (TOD) interval. A strictly conforming application attempts to manipulate an object's ACL which contains two entries for user fred; one entry contains a TOD qualifier for 0800->1800 and one entry has a TOD qualifier for 1800->0800. If the strictly conforming application intends to change the access allowed for user fred, then the application would call *acl_get_entry*() and *acl_get_qualifier*() until it locates an ACL_USER entry for fred and would then update the entry. The application would expect only one ACL_USER entry for fred and would only update one entry; since there are two entries for fred, the resulting access for user fred may not be as desired.

The special qualifier field value, ACL_UNDEFINED_ID, is defined as a value which cannot be used by the implementation as a valid group or user id. This value is used to initialize the qualifier field within a newly created ACL entry to a value which is not a valid group or user id.

B.23.2.3 ACL Entry Permissions Field

ACL entries are required to support read, write, and execute/search permissions for the following reasons:

(1) These permissions allow the abstraction of the POSIX.1 file permission bits as ACL entries.

(2) Existing practice dictates that at least these permissions must be retained.

File permissions in addition to read, write, and execute/search are allowed by an implementation because this would allow finer-grained and extended control of access to objects. For example, an implementation could add "append only" or "delete object allowed" permissions. However, such extended permissions are not required by this standard because such permissions are not universally required.

B.23.2.4 Uniqueness of ACL Entries

The combination of ACL entry tag type and qualifier are required to be unique within an ACL. The requirement for unique ACL entries, in combination with the order in which access is checked, provides a simple and unambiguous model for the specification of access information for an object.

Note that it is possible for the owner of a file to be explicitly named in an ACL_USER entry within the ACL associated with the file. While this entry may appear to conflict with the entry for the file's owner (i.e., the ACL_USER_OBJ entry), the ACL_USER_OBJ entry will be encountered before any ACL_USER entries during the ACL access check algorithm. Thus, in this case the ACL_USER_OBJ entry would uniquely determine the access permissions for the owner of the file; the individual ACL_USER entry for the file's owner would be ignored. The requirement is that the combination of tag type and qualifier must be unique. Also, the ACL_USER_OBJ entry and the ACL_USER entry are quite different semantically even if the ACL_USER entry contains the identity of the file owner.

Likewise, an ACL_GROUP entry with a qualifier id matching the owning group of a file does not conflict with the ACL_GROUP_OBJ entry in the ACL. In such a case, all applicable group entries would be examined to determine if any entry grants the access requested by the process. Both the ACL_GROUP_OBJ entry and the ACL_GROUP entry matching the owning group would be examined and might provide the desired access.

B.23.3 Relationship with File Permission Bits

ACLs expand upon the discretionary access control facility which is already provided by the file permission bits. Although file permission bits do not provide fine granularity DAC, they are sufficient for many uses and are the only mechanism available to existing applications. All existing applications that are security conscious use file permission bits to control access. The relationship between the ACL and the file permission bits must be defined in order to determine the level of compatibility provided to existing programs which manipulate the file permission bits.

Several approaches are possible for handling the interaction of ACLs with file permission bits. Each approach is presented in a separate sub-section with a description of the approach, a list of the advantages, and a list of the disadvantages. Final commentary and a conclusion follow the presentation of the approaches.

B.23.3.1 ACL Always Replaces File Permission Bits (Pure ACL)

In this approach, the file permission bits are no longer consulted for ACL decisions. Instead, each object has an ACL and the ACL completely determines access. File permission bits would be unused in the standard and the interaction between the file permission bits and ACL entries should be implementation-defined. This method would prevent the use of the old access control mechanism in a strictly conforming application.

This approach has the following advantages:

- Reduces complexity because there are no compatibility issues between ACLs and permission bits. Permission bits are no longer used for DAC decisions.
- A single, well defined discretionary access policy is employed.
- Increases security. The old access control mechanism does not provide the proper level of security to meet the requirements of this document.

This approach has the following disadvantages:

- existing applications that use *chmod*() or *stat*() must be examined to see if they are making DAC decisions. This is because *chmod*() and *stat*() update and return, respectively, more than just DAC information.
- existing applications that make DAC decisions must be rewritten to use the new interfaces.
- Compatibility between file permission bits and ACLs is left up the vendors who, realistically, must provide some compatibility with their old implementations. Without standardization the compatibility solutions will be vendor specific and not portable.

B.23.3.2 Owner Selects ACL Or File Permission Bits

In this approach, either the file permission bits or the ACL are consulted for the access control decision on a per object basis. The owner of the object determines whether to use the file permission bits or the ACL. If an ACL is set on a file, then the functions that manipulate file permission bits would return an error. If file permission bits are set on a file, then the ACL manipulation functions would return an error for that file.

This approach has the following advantages:

- If ACLs are never set, then there are no compatibility problems.
- If an access ACL is set on an object or a default ACL set on a directory, then the behavior is like the pure ACL system.

This approach has the following disadvantages:

- Like the previous approach, existing applications that use *chmod*() or *stat*() must be examined to see if they are making DAC decisions.
- Existing applications that make DAC decisions must be rewritten to determine which mechanism is in effect for each object it manages and then use the correct interface.

B.23.3.3 Independent ACL And File Permission Bits (AND)

In this approach, both the file permission bits and the ACL are consulted for the discretionary access control decision. Access is granted if and only if it is granted by both the ACL and the file permission bits.

This approach has the following advantages:

- Calls to *chmod*() have the desired effect from a restrictive point of view; ACL entries can further restrict access.
- The relationship between ACLs and file permission bits is easily defined: to be allowed access both must grant access.

This approach has the following disadvantages:

- To fully utilize the ACL as the effective access control mechanism requires that the file permission bits be set wide-open, i.e. read, write, and execute bits are set for user, group and other.
- In order to grant access, users must be prepared to change both the ACL and the file permission bits.
- An application would have to use *chmod*() and *stat*() to manipulate the file permission bits and the ACL functions to manipulate the ACL entries on a file.

B.23.3.4 Independent ACL And File Permission Bits (OR)

In this approach, both the file permission bits and the ACL are consulted for the discretionary access control decision. Access is granted if it is granted by either the ACL or the file permission bits. The ACL is used to grant access beyond what is set in the file permission bits.

This approach has the following advantage:

- Calls to *chmod*() have the desired effect from a permissive point of view.
- The relationship between ACLs and file permission bits is easily defined: to be allowed access either must grant access.

This approach has the following disadvantages:

- A *chmod(<object>, 0)* call does not deny all access to an object with an ACL.
- In order to deny access, users must be prepared to change both the ACL and the file permission bits.
- An application would have to use *chmod*() and *stat*() to manipulate the file permission bits and the ACL functions to manipulate the ACL entries on a file.

B.23.3.5 File Permission Bits Contained Within ACL Without a Mask

In this approach, only the ACL is consulted for discretionary access control decisions. The file permission bits are logically "mapped" to three base entries in the ACL. Calls to *chmod*() modify the ACL_USER_OBJ, ACL_GROUP_OBJ, and ACL_OTHER entries contained in the ACL. Calls to *stat*() return this information from the ACL.

This approach has the following advantages:

— The mapping of ACL entries to permission bits is straight forward. There is no mask entry that may or may not be there.

— With no additional entries, the semantic meaning of the file permission bits are preserved.

— There is some compatibility between file permission bits and ACLs. Use of *chmod*() to grant access is compatible. Use of *stat*() to return access for the owning group is compatible.

This approach has the following disadvantages:

— *chmod(<object>, 0)* may or may not prevent access to the object depending on the number of ACL entries. With additional entries, the *chmod*() call does not prevent access to the object and this breaks old style file locking.

— *chmod go-rwx <object>* may or may not restrict access only to the owner depending on the number of ACL entries. With additional entries, the *chmod*() call does not give owner only access.

— *creat(<object>, 0600)* may or may not restrict access to the newly created object to the owner. If a non-minimal default ACL exists on the parent directory, then owner only access is not guaranteed.

B.23.3.6 File Permission Bits Contained Within ACL Including a Mask

In this approach, only the ACL is consulted for discretionary access control decisions. The file permission bits are logically "mapped" to entries in the ACL. Logically, the file permission bits are the equivalent of a three entry ACL. Calls to *chmod*() modify the ACL entries corresponding to the file permission bits. Calls to *stat*() return this information from the ACL.

If there are ACL_USER, ACL_GROUP or implementation-defined ACL entries, then an ACL_MASK entry is required and it restricts the permissions that can be granted by these entries. If there is an ACL_MASK entry, then *chmod*() changes the ACL_MASK entry instead of the ACL_GROUP_OBJ entry and *stat*() returns information from the ACL_MASK entry instead of the ACL_GROUP_OBJ entry.

This approach has the following advantages:

— *chmod(<object>, 0)* prevents access to the object. This provides compatibility with the old locking mechanism.

— *chmod go-rwx <object>* restricts access only to the owner. This utility call, especially when used with the `find` utility, is useful for restricting access to objects to the owner.

— The ACL_MASK entry restricts the permissions that are granted via ACL_USER, ACL_GROUP and implementation-defined ACL entries during object creation. For example, without these restrictions, a *creat(<object>, 0600)* would not restrict access of a newly created object to the owner.

This approach has the following disadvantages:

— The mapping between the file group class permission bits is not constant. If the ACL_MASK entry exists, then the bits map to it. Otherwise, the bits map to the ACL_GROUP_OBJ entry. This means that *chmod*() and *stat*() update and return, respectively, different information based on the existence of the ACL_MASK entry. This behavior adds complexity to the ACL mechanism.

— The ACL_MASK entry does not provide complete compatibility with the uses of *chmod*() and *stat*(). *chmod g+rwx <object>* may grant more access than expected due to additional ACL entries.

There are several sub-issues with having an ACL mask. The following sub-sections describe those issues.

(1) Using ACL_GROUP_OBJ as a Mask

The working group considered having the ACL_GROUP_OBJ perform the masking for additional ACL entries.

This approach has the following advantages:

- Removes the five (5) ACL entry to four (4) ACL entry transition problem as described in "Automatic Removal of the ACL_MASK".
- Removes the special cases in *chmod*() for four (4) ACL entries versus five (5) or more ACL entries as described in "Requiring ACL_MASK to be Present".

This approach has the following disadvantages:

- The permission bits associated with the ACL_MASK limit the access granted by additional ACL entries that are added during object creation. There are two solutions if the ACL_MASK is removed. First, simply do not limit the access granted by the additional ACL entries. See section "File Permission Bits Contained Within ACL Including a Mask" for more details on why this solution is not acceptable. The second solution is to modify the additional ACL entries to grant no more access than was specified by the creating process. See B.23.5.1 for more details on why this solution is not acceptable.
- It is not possible to grant an additional ACL entry more access than the owning group. It is possible to solve this by using a special group with no members as the owning group. However, this solution complicates the `setfacl` utility. In the case where an object only grants read access to the owning group and a user wants to add an additional ACL entry that grants read-write access, the `setfacl` utility

would have to add an explicit entry for the owning group, change the owning group to the special group, and add the new ACL entry. This solution adds extreme complexity that will be visible to the user.

- If the file is setgid, then write access is unlikely to be granted by the ACL_GROUP_OBJ entry. This means that additional ACL entries would be unable to be granted write access. However, it is questionable if the owner would want to grant write access to a setgid file.

While using the ACL_GROUP_OBJ entry as the mask reduces the complexity associated with masking additional ACL entries, its benefits do not outweigh the disadvantages in the areas of object creation and usefulness of the ACL_GROUP_OBJ entry itself. Therefore, a separate ACL_MASK entry is defined and the ACL_GROUP_OBJ entry is used only to specify the permissions granted to the owning group.

(2) Requiring ACL_MASK to be Present

The working group considered a strategy to require the ACL_MASK ACL entry to always be present.

Either decision adds complexity to the *chmod*() interface. If the ACL_MASK is required, then *chmod*() will behave differently if there are four (4) ACL entries versus five (5) or more ACL entries. If the ACL_MASK is optional, then *chmod*() will behave differently if the ACL_MASK is present versus if the ACL_MASK is absent.

This approach has the following advantages:

- Requiring the presence of an ACL_MASK ACL entry provides consistency. Consider the following sequence: A user creates an object in a directory without a default ACL. The user examines the ACL and will only see the ACL_USER_OBJ, ACL_GROUP_OBJ and ACL_OTHER entries. The user adds an additional ACL entry. The user examines the ACL and will see the new ACL entry and the ACL_MASK entry, in addition to the ACL_USER_OBJ, ACL_GROUP_OBJ and ACL_OTHER entries. The ACL_MASK entry has suddenly "sprung" into existence.

This approach has the following disadvantages:

- Requiring the presence of an ACL_MASK entry requires mapping four ACL entries (ACL_USER_OBJ, ACL_GROUP_OBJ, ACL_OTHER and ACL_MASK) onto three groups of permission bits if only the base ACL entries are present.

- The ACL_MASK serves no purpose if there are no additional ACL entries. Since it serves no purpose in this case, it should not be required.

The expected use of a system with ACLs includes the use of default ACLs. Therefore, objects without an ACL_MASK ACL entry are expected to be rare, and most users will not see an ACL_MASK entry "spring" into existence. The standard does not require the ACL_MASK entry to be present if there are no ACL_GROUP, ACL_USER or implementation-defined ACL entries present.

(3) Automatic Removal of the ACL_MASK

The working group considered requiring that the ACL_MASK entry automatically be removed when all ACL entries other than ACL_USER_OBJ, ACL_GROUP_OBJ, ACL_OTHER and ACL_MASK were removed.

This approach has the following advantages:

- Requiring automatic removal makes the existence of the ACL_MASK less obvious to the user.

- Requiring automatic removal is simply a clean-up step. The ACL_MASK has performed its function and is no longer needed.

This approach has the following disadvantage:

- Requiring automatic removal of the ACL_MASK and the resultant resetting of the ACL_GROUP_OBJ permission bits leads to execution order specific results (in the absence of automatic recalculation). See below for an example.

If ACL_MASK is explicitly removed, then the permissions of ACL_GROUP_OBJ must be set to reasonable values. The working group considered the following cases:

- Leave ACL_GROUP_OBJ unchanged.

  If the ACL_GROUP_OBJ has more access than the old ACL_MASK, this case could unintentionally grant increased access rights. Since this is a security violation, this case is rejected.

- Set ACL_GROUP_OBJ to the value of ACL_MASK.

  If the ACL_MASK has more access than the old ACL_GROUP_OBJ, this case could unintentionally grant increased access rights. Since this is a security violation, this case is rejected.

- Return an error to the user if an attempt is made to delete ACL_MASK when ACL_MASK and ACL_GROUP_OBJ differ.

  This case was viewed as confusing and was rejected, because deleting an ACL entry should be independent of the ACL_MASK and ACL_GROUP_OBJ interactions. It does force the user to understand the problem and take immediate action, rather than waiting until the inadvertent access reductions from the next case are discovered. Finding out about a problem immediately is generally better than discovering it inadvertently much later.

- Logically AND the ACL_MASK and ACL_GROUP_OBJ together and set ACL_GROUP_OBJ to the result.

This case can lead to inadvertent access reduction (in the absence of automatic recalculation). For example, an object has an ACL with ACL_GROUP_OBJ ACL entry with read-only access and an ACL_USER(fred) entry with read-write access. Deleting the ACL_USER(fred) entry and then adding an ACL_USER(wilma) entry will produce an ACL that does not allow wilma to have write access to the object. However, adding ACL_USER(wilma) followed by deleting ACL_USER(fred) produces the desired affect.

While automatically removing the ACL_MASK when it is no longer needed makes the mask less obvious to the user, its benefits do not outweigh the complexity it adds to the programmatic interface. Therefore, the application must take an explicit action to remove the ACL_MASK entry when it is no longer needed within the ACL.

(4) Migration Path Flag

It is possible to define a flag to indicate whether masking is enabled or disabled for the implementation.

This approach has the following advantages:

- This flag would give individual system administrators the choice of determining the type of operation required for their specific installation.
- The flag would provide a migration path for some applications which use the *chmod*() function for file locking.

This approach has the following disadvantages:

- The existence of a flag would complicate DAC knowledgeable applications. Software vendors would have to provide different versions of the applications for the different environments or will have to modify their applications to work within the different environments.
- The existence of a flag will complicate the utility interfaces defined by this standard when used in a networked environment where some systems have the flag enabled and some systems have the flag cleared.
- The working group is chartered with only producing interfaces. Providing a migration path to a future usage model is beyond the scope of this standard.

Given the complexity involved with providing a migration path flag, this standard does not include such a flag.

B.23.3.7 The Conclusion

Compatibility with the existing DAC interfaces in some form or another is the overriding goal of this section. Most of the approaches considered provided some level of compatibility with the existing DAC interfaces. The file permission bits cannot reflect all the information that can be contained in an ACL. However, the *stat*() function should still reflect a reasonable amount of information regarding the access rights of files and the *chmod*() function should still be reasonably compatible with the previous semantics regarding the update of access information on files. Each approach has compelling advantages and discouraging disadvantages.

- The "ACL Always Replaces File Permission Bits (Pure ACL)" approach was rejected because it provides no compatibility.

- The "Owner Selects ACL Or File Permission Bits" approach was rejected because it requires existing applications that manage DAC to be modified to be used on a system with ACLs.

- The "Independent ACL and File Permission Bits (AND)" approach was rejected because it leads to wide-open file permission bits on systems that make use of ACLs with additional entries.

- The "Independent ACL and File Permission Bits (OR)" approach was rejected because a user of the existing DAC interfaces can be fooled into thinking that an object with additional ACL entries is secure when, in fact, others have access to the object.

- The "File Permission Bits Contained Within ACL Without a Mask" approach was rejected because a user of the existing DAC interfaces can be fooled into thinking that an object with additional ACL entries is secure when, in fact, others have access to the object.

- The "File Permission Bits Contained Within ACL Including a Mask" approach was chosen because it provides the "best" compatibility with the existing DAC interfaces.

B.23.3.8 Altering Permission Bit Mapping

Allowing implementation-defined ACL entries to alter the mapping between file permission bits and ACL entries defined by this standard was considered. If an implementation-defined entry is allowed to modify the permission bits, then it is possible for a strictly conforming POSIX.1e application to fail. Note that a strictly conforming application cannot add the implementation-defined entry to an ACL, but the strictly conforming application may not function properly if it modifies an ACL that contains the implementation-defined ACL entry. Consider the following: an strictly conforming application modifies the ACL_USER_OBJ entry in an ACL that contains an implementation-defined ACL entry. The implementation-defined ACL entry modifies the permission bits. The strictly conforming application expects the middle permission bits to be identical to the permission bits in the ACL_GROUP_OBJ entry. However, the permission bits have been modified by the implementation-defined ACL entry. The strictly conforming application is broken.

B.23.4 Default ACLs

A default ACL is a defined set of ACL entries that are automatically assigned to an object at creation time. There were five major decisions with default ACLs. The following subsections explain the rationale for these decisions.

(1) Why Define Default ACLs?

(2) Types of Default ACLs

(3) Inheritance of Default ACLs During Object Creation

(4) Compulsory versus Non-compulsory ACLs

(5) Default ACL Composition

B.23.4.1 Why Define Default ACLs

Should support for default ACLs be defined by the standard? The following reasons support inclusion of default ACLs in the standard:

(1) ACL use is encouraged in secure systems.

(2) Default ACLs allow the finer granularity of control provided by ACLs to be automatically applied to newly created objects. This control can be either restrictive or permissive.

(3) In a pure ACL environment, it is necessary to provide some initial access rights to a newly created object.

The following reasons support exclusion of default ACLs from the standard:

(1) It is not clear that the benefit of default ACLs outweighs the complexity introduced in object creation and object attribute management. Object creation will have to accommodate the existence of default ACLs in addition to the umask and the object creation mode bits. Either a new set of interfaces has to be created for manipulating default ACLs or the interfaces for access ACL manipulation will have to be modified to accommodate default ACLs.

(2) The default ACL in any form is a new influence on the ACL of a newly created object and cannot be manipulated or worked around by existing applications. Most existing applications will be able to coexist with default ACLs. However, existing applications that make security relevant decisions may not work on a system with default ACLs. See B.23.5 for specific examples.

In general, default ACLs appear to be a useful feature. Several existing ACL implementations have some form of default ACL mechanism. Certainly, default ACLs add complexity to the standard; however, they also add considerable value and should have a well defined standard interface.

B.23.4.2 Types of Default ACLs

Several different types of default ACLs were discussed by the working group. The advantages and disadvantages of each type of default ACL are discussed in the following paragraphs. The final paragraph of this section discusses why a particular type of default ACL was chosen.

(1) System Wide Default ACLs

One specific default ACL is assigned to any object created on the system by any process, in any directory. System wide default ACLs have the following advantages:

— Can only be set by the system administrator who is likely to be security conscious

— Is not complex or difficult to understand and explain

System wide default ACLs have the following disadvantage:

— Limits the specification of the initial discretionary access control on objects to system administrators rather than the user

(2) Per-Process Default ACLs

Each user process defines a default ACL which is assigned to any object created by the process. Per-process default ACLs have the following advantages:

— Models an existing interface, i.e., the umask paradigm

— Allows the user to retain complete control over the configuration of discretionary access

Per-process default ACLs have the following disadvantages:

— Follows a paradigm that is considered to be inadequate for present needs, i.e., the umask paradigm

— Requires the user to be security cognizant at all times; however, a knowledgeable user will only make security relevant decisions with a modest degree of frequency

— Might not be the right default ACL in a shared directory

— Allows the user to set only a single default ACL for all files created

(3) Per-Directory Default ACLs

Each directory is allowed to have a default ACL which is assigned to all objects created in the directory. Newly created subdirectories inherit the default ACL of the parent directory. Per-directory default ACLs have the following advantages:

— Allows the user to set up the hierarchy once

— Prevents the user from having to set a new default ACL as working directories are changed

— Allows system administrators to establish initial default ACLs on users' home directories which will propagate to objects created within the directories

— Allows project administrators to establish initial default ACLs on shared directories which will propagate to objects created within the directories

Per-directory default ACLs have the following disadvantages:

— Propagates the default ACL down through the file system hierarchy in cases where it is not necessary

— An implementation written to conserve disk space may have to implement a default ACL sharing mechanism

— Gives the choice of the default ACL to the directory owner instead of the file creator

The working group recognizes that a per-directory default ACL gives the directory owner control over the default value. However, the directory owner currently has control over at least one attribute of objects created in the directory: specifying the owning group. Also note that the directory owner has control over object creation, deletion, renaming and replacement.

The value added by per-directory default ACLs outweighs the complexity introduced by the mechanism and was, therefore, selected as the default ACL mechanism.

B.23.4.3 Inheritance of Default ACLs During Object Creation

While the working group felt that default ACLs on a per-directory basis provided the best solution, it considered alternatives to simply propagating the default ACL to all newly created objects in a directory. The working group considered two basic schemes for inheritance of ACLs involving the default ACL mechanism:

(1) Inheritance of Default ACLs for All Objects

The first alternative considered was to have all objects created in a directory inherit the default ACL of the directory. The working group felt that this solution provided an ACL inheritance mechanism that was consistent across all objects. This option does not take into account any differing permission requirements for directories as opposed to non-directory objects.

(2) Inheritance of Access ACLs for Directory Objects

The second alternative specified inheritance of the default ACL as the access ACL for all newly-created objects except directories. A newly-created directory would inherit the access ACL of its parent directory as its access ACL instead of inheriting the parent's default ACL. This approach was attractive because it allowed propagation of common properties through a sub-hierarchy which was thought to be the most common case. It further allowed different permissions to be applied to directories and non-directories which was considered a useful feature.

The disadvantages to this approach were the following:

— The implementation would not be consistent across all objects. The semantics for applying initial access control information to a single type of file object would differ from the semantics for all other types of file.

— In the case where a parent directory has no default ACL, counter-intuitive side effects were unavoidable.

- If the access ACL were applied to a newly created directory object only when a default ACL is present, the application of initial access attributes to the directory is determined by an event unrelated to the action of creating the directory, i.e., the presence of a default ACL. This behavior violates the Principle of Least Astonishment.

- If the access ACL were always applied to a newly created directory, the semantics of POSIX.1 are violated. The method for applying initial access attributes to directories no longer would allow the capability to create a minimal ACL, i.e., one corresponding to permission bits, in a manner consistent with the POSIX.1 umask capability.

The working group selected the first mechanism because the ease in which it could be consistently applied. The working group felt that the advantages of the second approach were not sufficiently beneficial to warrant accepting the disadvantages. If a more flexible default ACL mechanism providing some of the advantages of the second alternative is desired, an implementation may include additional default ACLs for this purpose.

B.23.4.4 Compulsory Versus Non-Compulsory Default ACLs

The standard requires a conforming implementation to support a per-directory default ACL mechanism. The working group discussed whether or not default ACLs should be required on every directory.

The following supports requiring default ACLs on every directory:

(1) Allows a consistent ACL policy to be maintained for all newly created objects

(2) minimizes the need for the umask

The following supports the optional use of default ACLs:

(1) Allows users who wish to use only the permission bits to use only the existing DAC mechanism

(2) Allows existing mechanisms to further restrict access on the newly created object, i.e. creat and umask

The working group feels that allowing users to use either default ACLs or the umask interface provides a significant amount of flexibility. Thus, the working group decided to make the use of default ACLs on directories optional.

B.23.4.5 Default ACL Composition

The working group discussed having the same required entries for default and access ACLs or to have no required entries in default ACLs.

The following supports having identical required entries for default and access ACLs:

(1) Supporting optional default ACL entries leads to a more complex object creation algorithm that is difficult to explain.

The following supports having no required entries in default ACLs:

(1) The user has the flexibility to configure the default ACL with the minimum amount of access information that is necessary.

The working group feels that consistency between default ACLs and access ACLs contributes dramatically to the conceptual simplicity of the default ACL mechanism and that the need for simplicity far outweighs the small increase in flexibility provided by optional default ACL entries. Therefore, default ACLs have the same required entries as access ACLs.

Note that default ACLs are optional on individual directories. However, if a directory has a default ACL, then that ACL must contain at least the three required entries for owning user, owning group, and all other users. It may contain additional named user and group entries. If a default ACL contains ACL_USER, ACL_GROUP or implementation-defined ACL entries, then an ACL_MASK entry is also required.

Also note that a default ACL with no entries is not equivalent to no default ACL existing on a directory. A default ACL with no entries is an error and any attempt to associate such a default (or access) ACL on an object will be rejected with an appropriate error code. The appropriate functions (or options on the `setfacl` utility) must be used to completely remove a default ACL from a directory.

B.23.5 Associating an ACL with an Object at Object Creation Time

The following goals guided the working group in determining how ACLs should be assigned on object creation:

- The object creation calls and the *open*() call with the **O_CREAT** flag specify the mode to use when an object is created. The mode provided is the program's way of indicating the access limitations for the object. It was a goal that no access be permitted to the object if it would not traditionally have been granted.

- There are many existing programs that use *creat(filename, 0)* as a locking mechanism. Although this is no longer a recommended way of doing locking, preserving this functionality shall be given high priority.

- The process umask is the user's way of specifying security for newly created objects. It was a goal to preserve this behavior unless it is specifically overridden in a default ACL.

- The access determined by an ACL is *discretionary* access control. But discretion of whom, the creator or the directory owner? Traditionally, discretion has been up to the creator. However, ACLs are often used by projects in shared directories. It was a goal to permit the directory owner to have control, but only within the limits specified by the creator.

- The Principle of Least Astonishment is a guideline that states that changes to existing interfaces should provide a minimal amount of surprise.

The working group considered whether the creating process should be allowed to control the inheritance of default ACLs. If the process controls inheritance, then the process can keep a default ACL from further restricting the permissions. But the creator can achieve this anyway, by changing the ACL after creation. Therefore no additional control for the creator was provided.

The algorithm chosen for determining the mode of a newly-created object is in the body of the standard. The reasons why this algorithm was chosen are:

(1) If there is no default ACL on the parent directory of the created object, the ACL assigned to the object is fully compatible with the access granted to the object in a POSIX.1 system.

(2) The entries of the default ACL are used in place of the equivalent umask bits. Thus, the creator of the default ACL can control the maximum permissions for newly created files in the directory.

If umask were used when a default ACL exists, then the user is likely to set a very permissive umask to permit the full utilization of the default ACL. This permissive umask would be inappropriate in a directory without a default ACL. The chosen solution allows umask and default ACLs to co-exist.

(3) The newly created object has all the ACL_USER and ACL_GROUP ACL entries specified in the default ACL. The ACL_USER_OBJ, ACL_GROUP_OBJ, and ACL_OTHER entries are as close to the ones specified in the default ACL as possible, within the constraints of the creator's mode parameter. If the default ACL contains an ACL_MASK entry, then it is constrained by the creator's mode parameter instead of the ACL_GROUP_OBJ entry. In this case, the newly created object has the ACL_GROUP_OBJ entry as specified in the default ACL.

(4) The overall effect is that the access granted to the newly created object has the granularity specified by the default ACL, while preserving the constraints specified by the object creator.

The only disadvantage recognized by the working group for this algorithm is that the umask is not taken into consideration when creating files in a directory with a default ACL. This solution gives the user little protection against a program that specifies an unwise create mode when creating a file in a directory with an inappropriate default ACL.

Another possible approach is to ignore both the mode parameter of the *creat*() function and the umask value if a default ACL entry exists. This approach was considered because it gives the directory owner complete control over newly created objects in her/his directory. Allowing the directory owner to have control over the permissions of newly created objects is a logical extension. This solution also supports the contention that the directory owner knows how to set up the permissions for newly created objects in a particular hierarchy.

This algorithm was not selected because the directory owner can override the program's advice about the use of a newly created object, i.e., override the create mode. Traditionally, the creator of an object has complete control over the mode of a newly created object. This solution would completely usurp that control from the creator.

The specification of the semantics for applying ACLs on a newly created object is included as part of this standard so that applications can predict reliably the access that will be granted (or more accurately, the maximum access that will be granted) based on a default ACL set by that application. This is simply an extension of the specification of the setting of the file permission bits for newly created files in the POSIX.1 standard without the ACL option.

B.23.5.1 Modification of ACL Entries on Object Creation

The working group considered changing the default ACL mechanism to modify the permissions granted by additional ACL entries that are added during object creation. The permissions would be modified to grant no more access than was specified by the creating process.

This strategy has the following advantage:

- If the permissions of the additional ACL entries are modified as described above, then the mode parameter specified at object creation could be used to remove undesired permissions from all entries in the new object's access ACL.

This strategy was rejected for the following reasons:

- If the permissions of the additional ACL entries are modified as described above, then information that the creator of the default ACL entered is lost. The most common example is that a *creat(file, 0600)* would lose the information in the default ACL for all ACL_USER and ACL_GROUP entries. This represents a potential for considerable information loss.

B.23.6 ACL Access Check Algorithm

The ACL access check algorithm has several important characteristics.

(1) Support for concurrent membership in multiple groups.

If a process belongs to multiple groups, the specific access modes requested are granted if they are granted by the owning group entry or by a matching group entry in the ACL.

(2) Consistency with existing POSIX.1 features.

The *chmod*() and *stat*() functions will continue to operate on the permissions associated with the object's owner, owning group, and other users not matching entries in the ACL.

(3) Relative ordering of algorithm steps.

The relative ordering of the algorithm steps is essential to be able to exclude specific users even if they belong to a group that otherwise may be granted access to the resource.

(4) Support for extensibility.

Implementations that include additional ACL entry tag types or extensions may insert them as appropriate into the relative order of the defined steps in the algorithm.

The rationale for the first of these characteristics is covered in detail below. The issue of interoperability is discussed in detail in B.23.3.

B.23.6.1 Multiple Group Evaluation

The design of supplemental groups in POSIX.1 was intended to provide flexibility in allowing users access to files without requiring separate actions to first change their group identities. The ACL mechanism facilitates that intent by allowing the inclusion of multiple named group entries in the ACL. Since it is possible for a process to match more than one named group entry in the ACL at a time, it is necessary to define the access that is granted by the matched entries.

The following paragraphs discuss the approaches that were considered:

(1) First group-id match. In this approach, the first entry that matches one of the process's groups is used to determine access. Access is granted if the matched entry grants the requested permissions.

This approach does provide a simple solution to the problem, but it does so by putting a burden on the user to order the ACL_GROUP entries correctly to get the desired result. Also, while this is an efficient method to implement, it does dictate implementation details because the ACL entries must be maintained by the system in the order that they were entered by the user.

(2) Intersection of matching entries. In this approach, the permissions of all the entries which match groups of the process are intersected (ANDed) together. Access is granted if the result of the intersection grants the requested permissions.

This approach does provide a slightly complex solution (from a user point of view) to the problem, but it is considered very restrictive. It is difficult to justify that a process that is granted read access through one group and write access through another group should actually get no access.

(3) Union of matching entries. In this approach, the union is taken of the permissions of all the entries which match groups of the process. Access is granted if the result of the union grants the requested permissions.

This approach does provide a slightly complex solution (from a user point of view) to the problem, but it is considered rather permissive. It is not possible to ensure denial of access to all members of a group via a restrictive group entry because members of that group may be allowed access via membership in other groups. It is also possible for a process to be granted more access than is granted by a single entry, e.g., one entry grants read access, one entry grants write access and the process is granted read *and* write access.

(4) Permission match. In this approach, the permissions of all the entries which match groups of the process are compared with the requested access. Access is granted if at least one matched entry grants the requested permissions.

This approach provides a simple solution to the problem that is very similar to the POSIX.1 semantics. In POSIX.1, if a process is in the file group class and the file group class permissions grant at least the requested access, then the process is granted access. In this approach, if a process is in the file group class and the permissions of one of the ACL entries in the file group class grant at least the requested access, then the process is granted access.

One of the goals of the ACL mechanism is to be compatible with POSIX.1. Of the different approaches considered, the "Permission match" approach provides the semantics that most closely match POSIX.1 and is the chosen approach.

B.23.6.2 Multiple User Evaluation

If the effective group ID or any of the supplementary group IDs of a process matches the group ID of an object, then the POSIX.1e access check algorithm uses the permissions associated with the ACL_GROUP_OBJ entry and the permissions associated with any matching ACL_GROUP entries in determining the access which can be granted to the process. However, if the effective user ID of the process matches the user ID of an object owner, then only permissions associated with the ACL_USER_OBJ entry are used to determine the access allowed for the process. No ACL_USER entries are used even if the process matches the qualifier information for one or more entries.

This type of behavior is consistent with the previous POSIX.1 interface since a process could not match multiple user identities yet could match multiple groups.

B.23.7 ACL Functions

B.23.7.1 ACL Storage Management

These issues apply to both access ACLs and default ACLs. The decision to manipulate ACL entries in working storage was made for two reasons: 1) the possibility of unsecure states and 2) the fact that there can be a variable number of ACL entries.

If ACL entries could be manipulated directly, or if ACL entries could be manipulated while the ACL continued to protect the object, unsecure states could arise. This is because the functions which manipulate ACL entries only manipulate single entries. The procedural interfaces we have chosen are not capable of changing several entries in a single autonomous operation. Because of this the possibility exists that a less secure state could arise during the modification of an ACL.

B.23.7.1.1 Allocating ACL Storage

Since an ACL can contain a variable number of ACL entries, mechanisms to allocate and free dynamic memory are required. The working group considered four approaches. The first approach was to have a single function that allocates a specific amount of memory for the ACL. The disadvantage to this approach is that the user must allocate enough storage or an error will occur and new larger working storage will have to be allocated and the ACL entries recreated.

The second approach is to have two functions that allocate space for the ACL. The first function allocates a specific amount of space for the ACL and the second function increases the space allocated by the first function to a specific size.

The third approach is to have a single function that allocates an initial amount of memory. Applications would then provide the address of the pre-allocated ACL storage area to the ACL manipulation functions. The *acl_copy_int*(), *acl_create_entry*(), *acl_from_text*(), *acl_get_fd*(), and *acl_get_file*() functions would manipulate the ACL within the ACL storage area provided by the application and would allocate additional memory as needed.

The fourth approach is to have the routines which work with working storage areas for opaque data types allocate the working storage as needed and then return pointers to descriptors for those areas. Functions which then manipulate the ACL in the working storage area would allocate additional memory for working storage as needed. In addition, a function to allocate storage for an ACL with no entries would be provided.

The final approach has been chosen for inclusion in the standard in order to provide a consistent interface among the various sections of POSIX.1e.

B.23.7.1.2 Copying ACL Storage

The *acl_copy_entry*() function is provided for several reasons: an *acl_entry_t* is a descriptor and cannot be byte copied; an implementation can have extensions and without the function it is not possible for a portable application to copy an entry.

The *acl_copy_entry*() function is also provided to allow an application to copy an entry from one ACL to another ACL. This is useful when the source ACL is a list of "defaults" that the application provides for building ACLs to apply to arbitrary objects.

The *acl_copy_entry*() function allows an application an easy means of copying an ACL entry from one ACL to another ACL. For example, one implementation of an ACL builder application may maintain an ACL "scratch pad" that is used to build ACLs to be applied to objects. The application may provide a means of highlighting specific ACL entries in the "scratch pad" to be copied to the ACL that is being built.

B.23.7.1.3 Freeing ACL Storage

An explicit interface for freeing ACL storage is provided. The working group considered embedding this functionality into the *acl_set_file*() and *acl_set_fd*() interfaces. The disadvantage is that a program wanting to apply a single ACL to multiple files would have to create or read the ACL for each application of the ACL.

B.23.7.2 ACL Entry Manipulation

Interfaces are provided to manipulate ACL entries. There were five major decisions with ACL entry manipulation. The following subsections explain the rationale for these decisions.

B.23.7.2.1 Procedural Versus Data Oriented Interfaces

This standard uses a procedural interface to manipulate ACL entries instead of the traditional UNIX style data oriented interface.

A data oriented interface specification typically defines a small set of primitives to access data objects, e.g. read, write, or commit. The application must be aware of the structure of the data and is responsible for direct manipulation of the data. The advantages of a data oriented interface is that it provides the application a substantial amount of flexibility in accessing and manipulating the data. However, because the application must know the structure of the data, any change in the ordering, size, or type of the data will impact the application.

A procedural interface isolates the application from the structure of the data. The interface consists of a larger set of functions where each function performs one operation on one field within the object. The application manipulates the data items within an object by using a series of functions to get/set each data item and a smaller set of functions to read and write the object. The advantage of a procedural interface is that it allows changes and extensions to the structure of the data without any impact to applications using that data. However, isolating the application from the data structure provides the application with less flexibility in accessing and manipulating the data and exhibit poorer performance.

A data oriented interface has the following advantages:

- consists of a small set of functions.
- can be manipulated by language primitives.
- is consistent with traditional UNIX calls, e.g., *stat*(), *chmod*(), etc.

A procedural interface has the following advantages:

- allows changes/extensions to the data structures without impacting applications.
- contains fewer visible data structures
- supports a move toward object oriented interfaces which tends to encourage more portable code

The advantages of isolating applications from the structure of ACLs and ACL entries are substantial. Thus, a procedural interface was chosen to manipulate access control list information.

We originally did not choose to define a procedural interface for manipulating the permission set within an ACL entry. Our reason was that the application must be aware of the structure of permission sets (bits within a long data type) and should be responsible for manipulating the bits directly. In our original opinion, the ease of direct language manipulation of the permission bits far exceeded any advantage gained in hiding the structure of the information.

During balloting it became clear that procedural interfaces for permission bits had additional advantages. Functions to manipulate permission sets were added later to allow an implementation to have more permissions than could fit in a natural data type (32 bits). While it is somewhat difficult to imagine why more than 32 permissions are needed, it is not good design to preclude such an implementation.

B.23.7.2.2 Automatic Recalculation of the File Group Permission Bits

The initial proposal was to recalculate the **file group permission bits** whenever a new ACL entry is added. The following example illustrates a problem with this approach.

> Consider a file created with a file creation mask of 0 in a directory that contained a fully populated default ACL. This file will have **file group permission bits** of 0, i.e., ---, yet may have named ACL_USER or ACL_GROUP entries specifically granting permissions. (These entries will be effectively ignored during access checking because of the masking effect of the 0 **file group permission bits**.) If the **file group permission bits** are automatically recalculated whenever a new ACL entry is added, the result of adding a ACL_USER entry specifically denying a user access will be to effectively grant access to the previously masked ACL entries.

It seems counter-productive at best to have an entry that denies a user access also grant access to other users. However, there does not exist a technique to allow for the application of a single entry in an ACL and the exclusion of others.

Other proposed alternatives include providing a mechanism in the `setfacl` utility to specifically request recalculation. A problem with this alternative is that typically a user adds an entry to an ACL with the intent of having the new entry affect the access decision. It isn't possible to have one new named ACL_USER or ACL_GROUP entry be guaranteed effective in the access algorithm without recalculating the **file group permission bits** based on all entries.

The final alternative considered by the working group is to provide an explicit interface for recalculating the mask.

B.23.7.2.3 Convenience Functions

The *acl_calc_mask*() function is provided for the convenience of applications. Applications could be required to perform this function, but DAC knowledgeable applications are likely to need it. Therefore, it is better to provide a standard interface.

The *acl_valid*() function is provided as a convenience for applications. Applications could be required to perform this function, however this functionality will likely be used by ACL cognizant applications. Therefore it is better to provide a standard interface for this functionality.

It is possible to merge the *acl_valid*() and *acl_set_∗*() functions together. However, it may be useful for ACL cognizant applications to be able to perform the *acl_valid*() function without having to apply (write out) the ACL to an object. This was seen as particularly useful for interactive tools in dealing with access and default ACLs.

The group considered providing program interfaces for the creation of objects with a specified ACL and other security attributes. The motivation for this is that security-conscious programs may wish to ensure that objects they create have correct ACL and other security attributes throughout their life, from the instant they are created. The group decided not to standardize such interfaces because programs can achieve the security objective by creating the object using existing POSIX.1 interfaces specifying very restrictive permissions and then setting the ACL to the required value.

The *acl_first_entry*() function was added to allow applications to revisit ACL entries previously referenced with *acl_get_entry*(). This is particularly needed by applications which are creating an ACL in working storage and need to revisit a previously created entry.

B.23.7.2.4 Hooks For Sorting

The *acl_valid*() function may change the ordering of ACL entries. This behavior allows an implementation to sort ACL entries before passing them to the *acl_set_∗*() function. This allows a performance improvement to be recognized. Since the *acl_set_∗*() function does not require any specific ordering, the system will likely sort all entries so that it may check for duplicates. If the sorting is performed by the *acl_valid*() function, the system may only need to make one pass through the ACL resulting in an order (N) sort when the *acl_set_**() function is called.

Functions which may add entries to an ACL, or remove them, are also allowed to reorder the entries of an ACL. This permits, but does not require, an implementation to keep an ACL in some implementation specific order.

Note that the standard requires that even implementations that reorder the entries of an ACL do not invalidate any existing ACL entry descriptors that refer to the ACL: these must continue to refer to the same entries even if the implementation reorders the entries.

B.23.7.2.5 Separate Functions for Tag and Permission

A single function (for example, *acl_get_entryinfo*()) could have been provided for retrieving ACL entry fields rather than separate functions. However, the standard provides individual interfaces for retrieving and setting each logical piece of information within an ACL entry. Implementations can add information to an entry and add a separate interface for that implementation-specific information rather than changing the ones specified in this standard.

Implementations are allowed to define additional ACL entry types with arbitrary size qualifier fields. Because of this, *acl_get_qualifier*() cannot simply copy out a user ID or group ID size object. The *acl_get_qualifier*() interface returns a pointer to an independent copy of the qualifier data in the ACL entry. The copy is independent because the ACL entry may be relocated by an *acl_create_entry*() or *acl_delete_entry*() call. When the application is done with the ACL entry, the space needs to be released; hence, the need for for a call to *acl_free*().

B.23.7.3 ACL Manipulation on an Object

Interfaces for manipulating an ACL on an object are provided for reading an ACL into working storage and for writing an ACL to a file. These functions provide a type parameter to allow for implementations which include additional types of default ACLs not defined in the standard. See the rationale for “ACL Storage Management” for additional information.

An earlier version of the draft contained a requirement that modifying an an ACL on an object and removing a default ACL from a directory be implemented as "atomic operations". The specific requirement was that the operations be atomic with respect to the invocation and termination of the function calls and any use of the ACL (access or default ACL). There was also the requirement that changes to an existing access or default ACL could not result in any intermediate state such that both the original ACL and the result ACL were both associated with the target file. While these requirements are certainly necessary, they are requirements upon the implementation, not the functional interface. As such, it is left to the implementation to define and enforce its own atomicity requirements. In addition to not being an interface issue, such atomicity requirements are inherently non-testable. As such, it is unreasonable to require the construction of tests to demonstrate conformance these atomicity requirements. For these reasons, all atomicity requirements were removed from the *acl_delete_def_file*(), *acl_set_fd*(), and *acl_set_file*() functions.

B.23.7.4 ACL Format Translation

There are three formats of an ACL visible to the programmer:

(1) An internal representation that is used by the ACL interfaces.

(2) A self contained data package which can be written to audit logs, stored in databases, or passed to other processes on the same system.

(3) A **NULL** terminated text package (string) that can be displayed to users.

The ACL copy and conversion functions provide the means to translate an ACL among the various ACL representations.

The **NULL** terminated text package may contain a representation of an ACL in either a long text form or a short text form. The following is an example of a valid ACL in the long text form:

```
user::rwx
mask::rwx
user:jon:rwx
user:lynne:r-x
user:dan:---
group::rwx
group:posix:r-x
other::--x
```

The following is a representation of the same ACL in the short text form:

```
u::rwx,m::rwx,u:jon:rwx,u:lynne:r-x,u:dan:---,g::rwx,g:posix:r-x,o::--x
```

The working group considered using the self contained data package as the internal representation of an ACL. The working group rejected this option for the following reasons:

(1) Implies that some implementations would have to translate an internal form into a self contained form on every POSIX.1e compliant ACL operation.

(2) The programmer has to keep track of the size and location of the ACL with every operation that can modify the ACL. The size must be tracked because the ACL size may grow or shrink. The location must be tracked because an ACL may not be able to grow in its present location and would have to be relocated.

B.23.7.5 Function Return Values and Parameters

The acl_get_*() functions can return pointers, descriptors and discrete values. If an acl_get_*() function returns a pointer, then it is returned as the function return value. This is because a NULL pointer is valid indicator for error conditions. If an acl_get_∗() function returns a descriptor or a discrete value, then it is returned as a write-back parameter. This is because there is not a well defined value that can be returned to indicate that an error has occurred.

B.23.7.6 File Descriptor Functions

The working group decided to specify functions that operated via file descriptors in addition to functions that operated via a file name. These functions allow an application to open an object and then pass around a file descriptor to that object instead of both the name and the file descriptor. BSD has found the related fchdir(), fchmod(), fchown() and fchroot() interfaces to be useful.

B.23.8 Header

Values for *acl_perm_t* are defined in the header because no definitions in POSIX.1 were suitable. Those definitions considered in POSIX.1 were:

(1) Definitions in POSIX.1, 5.6.1.2. These definitions refer to the nine permission bits whereas ACL entry permissions have only three values.

(2) Definitions in POSIX.1, 2.9.1. These names, e.g., R_OK, were not appropriate for ACL entry permissions.

B.23.9 Misc Rationale

B.23.9.1 Objects Without Extended ACLs

This standard specifies that each file will always have an ACL associated with the file, but does not require each file to have an extended ACL.

Originally, the provided ACL functions allowed for returning [ENOSYS] if {_POSIX_ACL} was defined and the specified file cannot have an extended ACL. This was subsequently changed because of objections to the overloading of [ENOSYS] to return [ENOTSUP] for the cases where a file cannot have an extended ACL.

A *pathconf*() variable {_POSIX_ACL_EXTENDED} is provided to allow applications to determine if a file can have an extended ACL. This standard does not specify the specific situations where a file cannot have an extended ACL. Examples of possible situations are: CD-ROM file systems, and pre-existing file systems with insufficient space to insert extended ACLs. The *acl_get_fd*() and *acl_get_file*() functions will always return an ACL because each file will always have an ACL associated with the file. The *acl_delete_def_file*(), *acl_set_fd*(), and *acl_set_file*() functions can return [ENOTSUP] if the specified file cannot have an extended ACL.

B.24 Audit

B.24.1 Goals

The goals for the POSIX.1e audit option are:

(1) Support for Portable Audit-generating Applications

   (a) Define standard interfaces for applications to generate audit records.

   (b) Define standard interfaces for applications to request that the system suspend its generation of audit records for the current process.

   (c) Define capabilities for these interfaces.

(2) Support for Portable Audit Post-processing Applications

   (a) Define a standard format for system- and application-generated audit records, as viewed through audit post-processing interfaces.

   (b) Define a minimum set of the POSIX.1e interfaces which shall be reportable in a conforming implementation.

   (c) Define a standard set of record types, corresponding to the reportable POSIX.1e interfaces, and the required content of those record types as viewed through the audit post-processing interfaces.

   (d) Define standard interfaces for reading an audit log and processing the audit records that are read.

(3) Extensibility for Implementation-specific Requirements

   (a) Ensure that standard reading and writing interfaces allow specification of arbitrary data in application-defined audit records.

   (b) Allow for reporting of additional implementation-defined events by conforming implementations.

   (c) Ensure that standard definitions of the content of required auditable events allow for extension by conforming implementations.

   (d) Define standard interfaces for access to implementation-specific audit storage mechanisms (audit logs).

The auditing interfaces specified by this standard are intended to be compatible with the auditing requirements of a number of specifications, including but not limited to the U.S. TCSEC levels C2 and above and the European ITSEC functionality levels F-C2 and above. It should be noted that this compatibility extends– only to the functional specifications; and also that meeting the requirements of this standard would not necessarily be sufficient to meet all of the audit requirements of any of the above specifications.

There was recognition by the working group that it should be possible for a number of differing implementations to be developed all meeting the POSIX.1e audit requirements. Additionally, consideration was given to the fact that implementations may (will) wish to extend the set of audit functions, audit events and audit records in various ways. For these reasons, flexibility in the POSIX.1e audit requirements was a primary goal.

In developing the POSIX.1e audit functions, the working group envisaged two distinct types of auditing applications. First were the class of applications which need to generate their own audit data. These applications, usually trusted, should be able to generate audit data in a standard audit log, rather than simply adding data to an application specific log file. Second were the class of applications that process audit logs. These analysis tools typically read, analyze and produce reports based on the audit data contained in the log. Optimally, these tools should be able to read and analyze audit logs from any POSIX.1e audit conforming application. Currently this goal is only partially met. The POSIX.1e audit option provides functions which could be used to develop a audit analysis tool, however, a common (portable) audit log format is not currently defined by this standard. Note that the POSIX.1e audit option specifies only the functions which an analysis tool would use, not the tool itself. The definition of a portable post-processing utility is left to a later stage, when security administration utilities are standardized.

B.24.1.1 Goal: Support for Portable Audit-Generating Applications

Commonly, portable applications, for example a data base, generate and record application specific audit data. Preferably, this data should be recorded in a system audit log rather than maintaining application-specific log files, or, worse, just ignoring security-relevant events as is common today. It is clearly more desirable for applications to use the standard system auditing mechanism than for each to invent its own.

In support of this goal, POSIX.1e audit provides a set of portable interfaces which an application could use to construct audit records and deliver them to an appropriate destination. In some cases it may be desirable to have these records added directly to the system audit log while in other cases a separate log may be required.

In order to provide maximum flexibility, the ability to support multiple audit logs has been provided. Applications get access to logs (other than write access to the current system audit log) via the POSIX file abstraction: that is, the POSIX.1 *open*() function is used. An additional function, *aud_write*(), is provided to allow records to be added to an audit log by self-auditing applications, since records written will normally have additional data added to them, and may be transmuted into some internal format, by the system in a way which is not consistent with the normal semantics of *write*(). A file descriptor parameter is normally used to tell this *aud_write*() interface which log is the destination, but a special value is defined to identify the system audit log (see “Protecting the Audit Log” below for rationale for this).

Records of security-relevant events, generated by an application, often relate to actions performed by, or on behalf of, a process (ie, acting as a subject), on one or more objects. The record needs to be structured so that the data that relates to the subject, or a particular object, or other aspects of the event, can be related together: for example, if the record contains a UID, it needs to be clear which subject or object it is related to. The standard therefore provides means for an application to build structured audit records, with separate sections for each subject or object. Such records can be quite complex, and it would be inefficient if the application had to build each one from scratch. The standard therefore provides means for the application to alter fields within a record it has constructed, allowing reuse of records.

In general, applications that generate audit records will also perform operations that cause the system to record audit records on their behalf. For example, a data base may open several files in normal course of action. For some applications, these system-generated records may be irrelevant and confusing, because the application itself might generate records that are more precise and informative. Therefore a provision is made to allow these, presumably trustworthy, applications to request that recording of system-generated records be suspended because they will provide their own. To ensure the integrity of the audit log, appropriate privilege is required to request suspension of audit records. Also note that this is a "request" to suspend the generation of audit records; an implementation is free to ignore this request.

B.24.1.2 Goal: Support for Portable Audit Post-Processing Applications

The working group recognized that a practical need for audit analysis tools, applications which read, analyze and formulate audit reports, existed. Additionally, to be of maximum value, these tools must be able to access and analyze audit logs from any conforming implementation. Currently, few audit analysis tools exist, and none of the tools examined by the working group were very sophisticated. It is therefore difficult to determine what functions are required for these analysis tools to function adequately. The working group determined that, at minimum, an analysis tool would need to access (open), read and terminate access (close) to the audit log.

In Draft 14 the working group recognized the need to make audit records available as they are committed to the audit log. The group felt that tools such as intrusion detection programs would require such a feature. The function *aud_tail*() was added to allow an application to request that records be made available to it as they are being written. However, it was later pointed out that the required effects could be obtained without use of a specialized interface: for example, an intrusion detection application could read from the end of the file currently used for the system audit log, using mechanisms similar to tail(1); and it could be told by the administrator (or other software) when the file corresponding to the system audit log gets altered. Accordingly, the interface was removed again. (There was some concern that this might result in records not being delivered for analysis until after a delay due to system buffering, but this was felt to be an implementation matter.)

The working group considered the addition of functions to query (selectively read) the audit log but rejected the idea for several reasons:

1. Understanding of need. The group could not determine what type of query functionality would be required by a portable analysis tool. Lack of market models made the task more difficult.

2. Defined query language. The group was unable to locate an agreed upon standard language for formulating a query. The working group was reluctant to invent a query language for POSIX.1e audit.

3. Extraneous functionality. The working group felt that as long as an analysis tool could access the next sequential record, that an analysis tool could provide its own query capability.

In addition to a set of common functions, a portable analysis tool may need to read and analyze audit logs from various sources. Thus, a portable tool may be dependent upon the definition of a standard audit record format. This standard does define a set of standard audit events, and the required record content for those events; it also defines means by which additional information in those records, and information in other records, can be obtained in a syntactically meaningful way.

Early versions of this standard contained requirements for storage of data in a standard form. This form proved to be unacceptable for most implementations, which have varying requirements for efficient storage of audit data. The working group decided to allow for storage of data in "native format" by default with an option to record data in a "portable format", to be defined. Without this interchange format, analysis of audit data across multiple storage implementations requires the application to do several conversions; from native format to human readable text (e.g., internal to external), gather the data on a single machine and then convert the human readable text to internal format (e.g., external to internal).

Since the portable audit log definition has yet to be developed, a possible goal of support for portable audit post-processing applications is currently satisfied only in part, primarily by defining functional interfaces to audit data.

In addition to the definition of standard functions, POSIX.1e audit also defines a set of standard audit events. These events, based on standard POSIX.1 and POSIX.1e interfaces, define the minimum data elements to be supplied by a conforming implementation when the event occurs (assuming auditing is enabled).

Events generated by standard POSIX.1 operations are defined to ensure that a portable analysis tool has some common ground in any system, although in practice, application-specific analysis tools (using standard interfaces to read application-specific data) will probably be fairly common. By defining the event types in this standard, a consistent mapping across all conforming systems is achieved.

There was some debate on whether to include events related to the relatively small set of POSIX.2 interfaces that are (arguably) security-relevant. However, a POSIX.2 interface is not necessarily built over POSIX.1; conversely, a POSIX.1 system does not necessarily provide POSIX.2 commands and utilities. There is thus no basis for defining POSIX.1e audit events for the POSIX.2 interfaces. The following were also seen to be reasons for excluding these events:

1. If a POSIX.2 implementation is built over POSIX.1, many of the POSIX.2 interfaces are adequately audited by the underlying audit events: eg, chmod(1) is adequately audited by the events for exec(2) of the command and chmod(2).

2. The most important security relevant commands, such as login, are not included in POSIX.2; those that are administrative are generally deferred to the POSIX 1387 working group.

3. In many cases, the commands that are included in POSIX.2 are not the ones that need to be audited. For example, it is not particularly relevant that a user has requested that a file be printed, or a batch job be started; what is relevant is the actual printing or starting of the job, which may or may not occur. POSIX.2 does not define the means by which these latter actions actually occur, any more than it specifies login or administrative interfaces, so it is not possible to standardize audit records for these occurrences.

The working group had debated including commonly known functions such as login, cron, etc to the set of standard events. However, the majority of the working group felt that adding non-POSIX events was not acceptable because (a) while these events were "common" they were not "standard", hence the "common" events were not deemed acceptable for inclusion and (b) systems which did not support these "common" functions would still have to support all the POSIX.1e audit event types. Additionally, there was some variance between the implementations of the "common" events. For these reasons the working group decided to limit the scope of POSIX.1e audit events to the domain of POSIX standards.

The working group debated the set of included events at great length; the goal was to include only those events which were security related and/or critical to the audit log. For example, consideration was given to including AUD_READ in the set of auditable events, however, it was felt that the information deemed desirable would be obtained by auditing the opening of the audit log. The AUD_WRITE event was also debated, with similar results (except that it was decided to audit AUD_WRITE failures). The working group felt that the amount of information derived from events such as these did not justify the potential performance penalty (e.g., auditing each read/write). Consideration was given to making these events optional. The group felt that the concept of "optional" events had little value because portable applications could not depend on the events being supported (because the events were optional) and hence the "optional" events would be of little use.

B.24.1.3 Goal: Extensibility for Implementation-Specific Requirements

It is important to allow applications to generate arbitrary records. Rather than having a single generic record, however, applications are permitted to place information in audit records that, while application-specified, has existing syntax associated with it that allows an analysis program to process the information. For instance, an application refers to a file by pathname, and because there is a standard way to describe a file in an audit record, an analysis program can select records concerning a particular file without knowing anything about the application generating the record that mentions the file.

Similarly, it is important to allow applications to specify arbitrary information in audit records, because not all the items an application needs to specify will be of the sort that can be interpreted in a portable way. The set of audit attributes is extensible to allow this, and additionally includes an explicitly defined opaque data object for application use.

Not all applications will want to use the system audit log; indeed, a particular implementation may not permit such use. So, it is important to allow implementations to provide other audit logs. Because the POSIX file abstraction provides defined interfaces without mandating any particular implementation mechanism, it is appropriate to use this for access to audit logs. Some proposals for this standard specified that audit logs were independent of the normal file systems, having their own set of interfaces (e.g., *aud_open*(), *aud_close*()) however these were not seen to provide any particular advantages.

Apart from the above application-oriented considerations, it is important that implementations be able to extend the set of auditable system interfaces, and to extend the set of data that is reported in audit records for the standard auditable interfaces. They will thus be able to report the occurrence of security relevant events that are beyond the current scope of ratified POSIX.1 standards, and to record additional security information for the standard events.

B.24.2 Scope

The scope of security auditing specifications in POSIX.1e is defined by the above goals. In addition, the following items are specifically excluded:

(1) Administration
Functions and utilities to support security audit administration are excluded. These exclusions include the assignment of audit control parameters to specific users, and pre-selection of which auditable events are to be recorded.

(2) Audit data storage
The definition of formats and organization for permanent audit data storage is not addressed, nor is there any required storage organization for a system's audit log.

(3) Portability/Data interchange
The definition of formats and organization required for a portable audit log and for interchange of audit data are not addressed.

(4) Audit delivery mechanism
The definition of a mechanism for delivering records is not addressed, although the interface to this mechanism, 24.4.40, is included.

Administrative functions are excluded from the POSIX.1e auditing scope, these are the province of POSIX.7.

The specification of criteria for the pre-selection of which audit records should be recorded is deemed to be an administrative issue. It was felt that portable trusted applications could not reasonably make use of interfaces to control pre-selection.

A grouping of event types into *classes* of events for post-processing were excluded from the scope because it was felt that not enough is currently known about post-processing to allow a solid set of post-processing classes to be included in the POSIX.1 standard. The group felt there were two compelling reasons why it was inappropriate to standardize event classes: (1) the grouping of events into classes is inherently arbitrary; while the group could easily agree on a standard set of common events (based on POSIX.1) the grouping of these events into classes differed widely, (2) the definition of classes does not add greatly to application portability because the event type rather than class is what is stored in the audit record.

This standard does not address audit data storage. It is expected that each conforming implementation may have a different form of permanent storage for audit data. Similarly, the issues of interchange of audit data are not addressed. A key problem in the definition of data interchange is that current standards do not address data size issues at all.

This standard does not address the actual mechanism for delivering audit records from a trusted application (or from the operating system itself) to a system's audit log. However, the interfaces that an application (or the operating system) would use to perform the delivery are specified. An actual delivery mechanism might involve spooling daemons, special network protocols, etc.

This standard also does not address the issue of protection of the audit data, that being an implementation's responsibility (see below for further rationale for this).

B.24.3 General Overview

In this standard, the general architecture for audit record processing is that the internal format of audit records is opaque, and functional interfaces are provided both for audit-generating applications to construct audit records (adding, changing and deleting fields) and for audit post-processing applications to analyze records (reading fields). The system manages the working storage used to hold the record; interfaces are provided to create new (empty) records in the working store, to read records from an audit log into the working store, and to write records from working store into an audit log.

An earlier version of this document used explicitly different storage representations for data structures used in reading records and in writing records. Writing records used opaque storage (called an Audit Record Descriptor), whereas reading used a caller-supplied buffer that was implied (but not required) to be a directly accessible storage representation of the portable audit record format. In principle, this would have allowed a processing program to have performed manipulations directly on the record contents, without using the reading interfaces.

A major criticism of this proposal was that it required that all data should be written in a portable format that was biased toward machines that support expanded data types. In abandoning the requirement that all audit data should be stored directly in the portable format, it became impossible to provide this ability. It also became apparent that the defined set of interfaces had become sufficiently complete and efficient that the ability was no longer important.

The original proposal defined audit records as consisting of individual "tokens" where a "token" represented an independent element of a record, for example a *pathname*. To make the token opaque all manipulation of the token (read/write) was done using per-token interfaces. For example *get_pathname_token* and *put_pathname_token* would be required to get (read) and put (write) a specific token. It is easy to see how this style of interface could lead to an excessive number of token types and in turn, an excessive number of interfaces required to manipulate each token type. There was also the possibility of inconsistent use of the tokens by applications performing their own auditing. The concerns regarding efficiency of storage and number of interfaces led to the replacement of the "token based" proposal.

In draft 13, self-auditing applications were required to construct audit records in user-managed storage, because the user (application) knows the size and contents of the record, and there is no point in making the data opaque. Also, the record may be used as a "template", that is the record may be modified and written multiple times without requiring multiple allocate/free operations of system managed storage. However, this proposal was criticized in ballot for not providing either sufficient record structuring capabilities or sufficient support for portable applications; extending the proposal to provide additional structuring would add considerably to the complexity of the data structures applications would have to manipulate (giving problems in some language bindings), and would exacerbate the second criticism. In contrast, system-managed storage was used for reading records, because in many cases the application will rapidly eliminate most records from the analysis, and keeping them in system-managed space saves the cost of converting the whole of each record from an internal to a standard format. Also, programs reading records are likely to be processing many records sequentially, and correspondingly benefit from eliminating application-level storage management overhead.

The current set of interfaces and corresponding data structures have been designed to provide reasonable application support with reasonable efficiency, without an excessive number of interfaces. Data storage representations are not defined. The interfaces deal with opaque structures at the top level, and individual components at a lower level; the latter use 'get item' interfaces, and a 'type length pointer' data structure, thus providing flexible functionality through a small number of interfaces. The interface for application generation of audit records similarly uses 'put_item' interfaces and the 'type length pointer' structure to specify the data to be recorded. Several tradeoffs exist, as described below, and these are not the most efficient interfaces imaginable; merely the most efficient portable interface proposed so far.

One tradeoff exists in the granularity of information access to the audit record. An audit record consisting of individual attributes is the more general interface but also is more inefficient. Structure-based interfaces that put and get information in large chunks are more familiar to programmers but it may be more difficult to validate the attributes; and structures are inconvenient if there are a large number of variable size components (or components with opaque structure that may be variable size).

Another tradeoff is caused by offering only indirect access to the audit record, because the information must be retrieved procedurally. The cost could be minimized by implementing these interfaces as macros and a procedural interface allows an implementation greater flexibility in defining audit log storage and access methods.

B.24.4 Audit Logs and Records

B.24.4.1 Protecting the Audit Log

Of all the data in a secure computing system, the audit log is perhaps the one item which is most important to protect against invalid manipulations EVEN by apparently authorized users. For instance, if an intruder can defeat a system's access control mechanisms, and assume all the rights and powers of an authorized system administrator, it would still be extremely useful to be able to audit the intruder's activities. To any extent possible, the auditing mechanism and the audit log should be protected against external attacks.

The group considered specifying a few possible mechanisms that provide elements of protection against this threat, but decided not to do so. The group took this position because any mechanism that is sufficiently general (not implementation-dependent) to specify in a standard would not, itself, provide significant protection. Only a combination of mechanisms, most of them implementation-dependent and outside the scope of POSIX, can protect a system's audit log to a meaningful degree beyond basic file protection.

If the audit file is protected using the normal filesystem protection mechanisms, the degree of protection increases with the security of the system. Thus in an ACL based system with a single super-user, it could be read/write to superuser only. On a system with the administrative roles divided according to the principle of least privilege, it could be owned by the audit administrator, with read access available also to the security administrator. On a system with MAC controls of disclosure and integrity, it could be owned by audit administrator with a disclosure label making it readable only to security and audit administrators, and an integrity label making it writable only by the system. Of course, these access controls do not prevent the audit subsystem itself from writing to the audit log to record actions of users, even though the users don't have write access to the audit log file.

Thus when audit logs are accessed via the POSIX file abstraction, this standard does not mandate any protection mechanism other than the normal file system access control mechanisms. The exception to this occurs in the case where an application needs to write to the current system audit log. There are two reasons why it would not be appropriate to rely on the usual file protection mechanisms, exercised through *open*(), in this case. Firstly, a self-auditing application should generally not have the ability to open the system audit log for write, since this would confer the ability to corrupt data that was already in the log, for example by writing random data at random positions in the log. Thus in this case an alternative means of accessing the log is needed to ensure its integrity. Secondly, an implementation may not have a fixed mapping between the current system log and a POSIX file: either the log data may be sent to different files at different times (e.g. when the current file reaches a certain size), or the data may not be sent to a medium that is accessible through a POSIX file name. Therefore this standard specifies that the current system log is written without use of *open*(), and uses appropriate privileges as the means to control access to that log.

The working group debated whether self-auditing applications should be permitted to provide all the data of an audit record, some people holding the view that the system should be required to provide some of the data (especially in the record header) in order to protect the integrity of the audit log and provide accountability for application-generated records. However, others held that it is only necessary to protect the integrity of the audit log, and that the application is trusted to create the entire contents of the audit record itself - some even suggested that the application should not even have to be privileged to do this. The final consensus took the 'middle way': that the integrity of the audit log should be protected (by allowing applications to write records without giving them general write access; and by allowing the system to check the format of audit records); and that only 'trusted' applications should be able to write records, the control being provided by use of appropriate privilege. The latter control allows implementations, or even installations, to set their own policy about the degree of trust needed in self-auditing applications, since they can control how widely the privilege to write audit records is distributed.

B.24.4.2 Audit Log and Record Format

The logical audit log is a stream of audit records. That is, an audit log appears to the application program as a sequence of discrete, variable length records. Each record contains a complete description of an audit event: records are intended to be largely independent entities. An important distinction must be made between the "logical" and "physical" descriptions of the audit record. The "logical" appearance of the audit log refers to the appearance of the audit records returned by the functions defined by this standard. The "physical" description of the audit record refers to the audit record as it exists in the audit log, that is how the record would appear if the audit log were read in its raw state. This standard does not define the "physical" view of the audit log. Additionally, this standard does not define the "logical" view of audit records when viewed by interfaces not defined by this standard.

B.24.4.3 Audit Record Contents

The statement above that audit records should be largely independent is an acknowledgement that no audit data can be completely context-independent, and an encouragement that audit records contain enough context to be meaningful for analysis in most circumstances.

Each audit record contains at least a header and a set of subject attributes (the term 'subject attributes' is used in preference to 'process attributes' because a process can also be an object (e.g. when receiving a signal), and also the particular set of attributes reported is that appropriate to the process's role as a subject, as opposed to all the attributes of the process). Most records also contain one or more sets of event specific data, and zero or more sets of object specific information. The header defines a version number (see below), the data format the record is written in, and includes fields for event type, event time, and event status. The event time is compatible with the *timespec* structure in POSIX.1b 1993.

To allow future versions of this standard to extend the audit record format and retain compatibility with previous versions, a version number in each record header identifies the version of the standard the record conforms to. For example, the version number defined by this iteration of the standard may be AUD_STD_1997_1 (the digits implying 1997, POSIX.1) while the version number defined by the first revision of this standard may be AUD_STD_1998_1. Thus, a conforming implementation, by reading the version number will know what audit record definition matches the audit record read. Note that the current defined version identifier AUD_STD_NNNN_N will have to be updated to reflect this iteration of the standard, such as AUD_STD_1997_1.

The format field specifies the format of the data contained in the audit log. Currently, only the format AUD_NATIVE is supported. The AUD_NATIVE format indicates that the audit data contained in the log is written in native machine format. This field is primarily a place holder for future revisions of this standard which are expected to add other formats such as a portable audit format. It is important for the portable application to know what type of data is written in the field, thus the application knows what kind of data to expect (i.e., byte ordering, data type sizes, etc.)

The status value was added to indicate the status of the audit event with some indication greater than success or failure. The following event statuses are currently defined:

AUD_SUCCESS The event completed successfully.

AUD_PRIV_USED The event completed successfully and privilege was exercised. Conforming implementations are not required to report this value (reporting AUD_SUCCESS instead), since not all audit policies require that use of privilege be audited. If the value is reported, however, this does imply that privilege was required, not just that privilege was available and was used. The working group felt this distinction was important because although some implementations may not need to distinguish between a privilege which was used and a privilege which was required, existing practice has shown that for security auditing it is important to report the use of privilege to achieve an operation that would have failed without it.

AUD_FAIL_DAC The event failed because of discretionary access control checks.

AUD_FAIL_MAC The event failed because of mandatory access control checks.

AUD_FAIL_PRIV The event failed because of lack of appropriate privilege. The audit record does not contain an indication of what the appropriate privileges were, though if the POSIX capability option is in use it does indicate the capabilities available to the subject, and other security attributes of the subject and object; thus it would be possible to deduce which capabilities would have been needed to complete the operation.

AUD_FAIL_OTHER The event failed for some reason, none of the above. This includes implementation-defined policy extensions.

Note that implementations are free to extend this list with additional status values. Note also that the standard does not define which of the various AUD_FAIL statuses is to be returned if the event could have failed for more than one reason: if this were specified it would imply that implementations had to perform tests in a certain order, or carry out all tests even if one had already failed, and the working group did not think this a reasonable requirement.

The audit record header includes an identifier, the *audit ID*, for the individual human user accountable for the event: it is a fundamental principle of accountability that each event should identify the human user accountable for it (see below for further rationale related to audit IDs). For system-generated events, if the process initiating an action does so on behalf of a user who is not directly associated with the process (e.g., a server process acting on behalf of a client) the directly accountable user should probably be the one that initiated the server. However, if there is no accountable user (e.g., the server was started automatically at system initiation) then the standard does allow the system to provide a null audit ID. For application-generated records, the standard specifies interfaces that allow a server process to record the audit ID of the client process for which it is acting.

The subject attributes are required to include the process ID and the basic security attributes of the subject: the effective UID and GID; means for reporting other security attributes (e.g., supplementary groups, labels, capabilities) is also provided. The working group considered requiring that all these attributes be present (at least if the relevant POSIX options are implemented) but rejected this because it was not clear that all systems implementing audit would need to provide this information. It is a matter for the policy of the system. Accordingly the standard defines how the information can be provided, and what happens if it is not, and allows implementations to decide on policy.

The object specific information includes fields for the type and name of the object, and object security attributes. Again, some of the security attributes are optional, it being up to the implementation security policy to define whether they are provided. In general, the standard requires that object details be supplied whenever the attributes or data of an object may be accessed or altered; it does not require it otherwise (for example, on a *chdir*()).

The audit events for interfaces that operate on files via file descriptors include the fd among the data reported. There was some feeling that this was in itself not very useful, since the file descriptor is not directly meaningful to an audit administrator, but the audit record for the *open*() call that created the file descriptor is also reportable, and does enable an audit post-processing tool, or audit administrator, to make the link back to a human-readable name.

For records that report changes to subject or object attributes, the standard includes the new attributes, through inclusion of the function arguments. It also requires that details of the relevant subject/object are included; it specifies that if the relevant attribute is included in the details, then the old value shall be given. However, it does not generally require that the relevant attribute must be included in the details. There are several reasons for this: not all security policies require that the old attributes be audited; in some implementations there is no reason for the old attribute to be available to the audit subsystem; for some attributes there could be a significant performance/space impact (e.g. recording 1000-entry ACLs!). Thus the standard always requires the new attribute to be recorded, and permits (but does not require) the old attribute.

B.24.4.3.1 Semantics of Audit Event Types

The standard includes a set of pre-defined system event types with fixed interpretations (corresponding to interfaces defined in POSIX.1). These system event types are defined primarily for use by audit analysis tools such that they can have a base set of defined, standard event types for analysis. It was felt by the working group that a standard means of uniquely identifying these system event types was required to avoid collisions (e.g., various definitions of the same event type); therefore the standard includes a means of identifying the event types themselves, that is, a standard naming of system event types is provided. The event type defines the minimum logical content of the record as it is returned by the POSIX.1e audit functions.

The working group felt that some applications may need to query the list of system event types supported by a system. For example, a interactive audit analysis tool may want to get all the system event types supported on a system, then prompt the user to determine what event types to analyze the audit log for. This type of capability also requires a interface to convert the audit event type from its internal representation (numeric) to text for display purposes, then from text (or numeric-text) to internal format (numeric). To provide this functionality to an analysis tool the following interfaces were defined: *aud_get_all_evid*(), *aud_evid_to_text*(), *aud_evid_from_text*().

Applications also need some defined semantics for audit events. A portable application wishing to generate its own audit records must be able to specify the form and content of the record so that it can convey this information to an audit analysis application. Like system events, application events also require some means of identifying the event type.

The working group debated how best to define the event types. Some iterations of this standard specified the event types as numeric constants (e.g., 1,2, ... nnnn). The working group felt that a portable analysis tool would be most efficient searching for and comparing numeric event type identifiers. For example, an analysis tool searching for records of type AUD_AET_KILL could simply search for records of event type 1. However, the working group felt that the expression of event types as character strings, e.g., "AUD_AET_AUD_OPEN" allowed for easier future expansion. The standard could thus reserve the AUD_AET_ prefix for future use (as opposed to reserving 1-xxx). The former option was proposed in the first ballot of the standard (attracting ballot objections related to extensibility, and the likelihood of applications choosing the same event types); the latter was proposed in the second ballot (attracting ballot objections related to efficiency of processing and storage). Finally, it was decided to adopt a combination, using numeric identifiers for system events and string identifiers for application events. This accomplished several goals:

A. System events can be recorded and processed with maximum efficiency.

B. Applications wishing to do self-auditing were less likely to have audit event type collisions. For example a database could generate records of AET_<MYNAME>_DB as opposed to records of event type 150. The group felt it was far less likely that two applications would choose the same character string.

C. Application event types cannot clash with system event types.

B.24.4.4 Audit Record Data Format

The physical format of an audit record is unspecified - that is, a post-processing application may make no assumptions about the format and location of the header, subject, object and event specific data as it actually exists in the record. Logically, an audit record is a collection of opaque segments (headers, sets of subject attributes, etc) each of which is referred to by a descriptor and accessed only by functions referencing that descriptor.

The segments of a particular type in a record are ordered, so that semantics may be attached to their relative positions; this is likely to be particularly important if a record contains details of more than one object, since these may represent the source and sink of data. Descriptors for the various structures can be obtained either serially or by random access (e.g. to the second set of object attributes).

The segments which comprise a system-generated audit record contain at least the data items defined by this standard and may include additional, implementation-defined data items. The data type of each of the required data items is defined by this standard, as is the ordering of the items. Note that the size and byte-ordering of the data items may vary from system to system. That is, there is no intention that the binary data in the opaque structures is directly portable from system to system.

A header segment must be (logically) included in every record. For system-generated records, the fields of it are set by the system; for application-generated records, the application is required to specify values for certain fields, and may supply more (the system will supply certain fields if the application does not). Similarly, the subject attributes of system-generated records are provided entirely by the system, but for application generated records the application is trusted to provide subject attributes, for example of a client process (again, the system will supply 'default' values, describing the current process, if the application provides none). There was considerable debate in the working group about whether the application could be trusted to supply header and subject attributes; some members felt that the system should always provide the header (except for the event type and status) and subject attributes for the current process. However, the alternative view prevailed, that an application that is trusted to generate audit records (in the system audit log) can also be trusted to do it right.

Although the number and ordering of segments in the record is important, it should be noted that for failed events, some objects and data in the record for the standard event types may be omitted, because this information may be missing from the function or utility invocation.

B.24.4.4.1 Portable Audit Record Format

The current version of the standard does not contain a definition for a portable audit log format. This is currently being investigated for a future iteration of the standard. Earlier drafts of this standard did contain a portable audit log format. However, the standard required that all records be written in this format which proved to be controversial. The rationale contained here defines the reasons why the working group felt a portable format was necessary.

A portable audit log format allows the audit data to be analyzed on systems other than the systems which generated it. Several methods were proposed to place audit records in portable format. One method proposed was to write all audit records in the portable format. This method was rejected because it had the potential to impose performance penalties on those implementations which did not support the data sizes required by the portable format as their "native" data types, in other words, some systems may be required to do size and type conversions on each record written.

To avoid this unnecessary penalty, the data that is returned in the structures is always in the local format. An alternate method proposed was to allow the audit records to be written in native machine format with the conversion to the portable format to be done by some form of audit record filter. These records can then be transferred to other systems.

There are two costs to this approach, however. The first is that each system must be prepared to read the portable format(s) defined. The second is that these records are always translated twice - once on the generating system and once on the system used for analysis.

The portable data formats are not defined in this document. That is, the size of uids, gids, MAC labels, etc. is unspecified at this point. While the elements of a portable audit log can be outlined, the definition of the portable audit log format is not defined.

Auditing by nature is the gathering of data, not the definition of it. Almost all the data types contained in a typical audit record are external to the audit group. True data portability is a problem much larger than the need for a portable audit log. Currently, neither the POSIX.1 standard nor the POSIX.2 standard address data interchange sufficiently to define a portable audit format.

After extensive research and discussion, the audit subgroup has concluded that the portable audit format is a subject that cannot be resolved with the present amount of information obtainable from other internationally recognized standards bodies.

Analysis of the problem revealed the following issues, all of which need to be resolved before a portable audit log format can be developed. The issues are:

(1) Data format (byte ordering)

(2) Data field sizes (very specific! number of bytes or equivalent)

(3) Field mappings (user ID <-> user name, etc.)

(4) Time coherence (time zone, etc.)

(5) Internationalization issues (at least for text strings contained in the file)

(6) Byte size

(7) Field identification and boundaries (how to tell where a record begins and ends)

(8) Naming convention (uniqueness of user, for example user ID plus process ID)

It was decided that the audit log header file needs to contain: an indicator that marks the log as being in POSIX.1e portable format, the version of the standard of the portable format, the data format indicator of the log (XDR, NDR, or ASN1 format), the time zone in which the log was created and any applicable maps required by that machine. There may be several machine identifiers and associated maps, keyed by machine_id. Not much more information can be generated without input from the interchange format group.

The audit subgroup has also yielded the format of the MAC label, ACL, and capabilities associated with the portable audit format to those associated groups. However, they too will be unable to determine the data sizes to be used in a portable interchange format without input from the interchange format subgroup.

B.24.5 Audit Event Types and Event Classes

The distinction between event types and event classes has generated considerable controversy. Two differing proposals were considered. One suggested that grouping types into event classes is arbitrary and may differ from one system to another. Another proposal suggested that event types should belong to a small, fixed set of standard event classes. This proposal also suggested that the event class be recorded in a header, with the event type, thus making it the responsibility of the auditing program to fix the relationship between the two.

Initially the latter proposal was accepted. However, after further reflection, it was decided that recording the event class in a header was not tenable. If an event type belongs to many classes, but only one can be recorded in a header, then the inclusion of such a value might serve to confuse rather than clarify the reason for the audit record. Eventually it proved impossible to reach consensus on how event classes should be standardized; there was also a body of opinion that said it was unnecessary to standardize them, because post-processing applications could group event types into classes at that level. Accordingly, the concept of event class was removed from the standard.

It also turns out to be very hard to define precisely when an event deserves an event type of its own. For instance, are successful and failed **open** calls the same event type? Probably so, because they can be differentiated by the result field in the record header (though looked at another way, that really means that the result field is part of the event type, and so they are two different types).

Are open of a file for reading, and open of a file for read/write, different event types? Though they differ only in one bit of a system call argument, maybe they ought to be different types, because they represent very different abilities being exercised. This example leads to a circular definition of event types: two types should be separate when it would make sense to assign them to separate classes.

It was finally decided to define no more than one event type for each of the POSIX interfaces being audited; in a few cases a single event type was used for several closely related interfaces (e.g. the *exec*() family). The separation of, eg open-read and open-write can then be done by post-processing tools on the basis of information in the record; implementation-specific means could be used to separate these for event pre-selection purposes too (see below).

B.24.6 Selection Criteria

At various times, drafts of this standard have included facilities for both pre-selection and post-selection of audit records: that is, selection of the records that are recorded in the log, and those that are reported from the log to an audit post-processing application. However, the standard does not finally contain any selection facilities. The pre-selection interfaces have been removed because they are seen to be an administrative facility, and therefore out of scope. The post-selection interfaces have been removed on more pragmatic grounds: there was no agreement on what facilities are needed, or how post-selection criteria should be specified. Additionally, the group felt that so long as the next sequential record could always be made available, applications could build selection criteria themselves.

B.24.7 Audit Interfaces

B.24.7.1 Gaining access to the Audit Log

In earlier drafts of this standard, to provide some separation of audit log from file the concept of an *audit_log_descriptor* was conceived. The audit log descriptor provides a level of abstraction above the file descriptor interface. An attempt was made to define a set of interfaces for use in analyzing abstract audit logs, concealing the storage method, location and format of the actual data. In draft 13 (and previous drafts) there were two functions provided to initiate and terminate access to the audit log; *aud_open*() and *aud_close*(). However, this resulted in a need to reinvent a complete I/O package for such objects. Also, it did not succeed in defining any particularly useful interfaces, other than a record-oriented read function.

In draft 13 several balloters objected to the concept of an "audit descriptor". There were two flavors of objection. One type of objection cited existing practice claiming that existing practice (or all that was known to the objector) used files so the abstraction of an audit descriptor was not reflective of current practice. Another type of objection stated that since the descriptor was largely implementation defined that it was of little use to the portable application. In response to these ballot objections, the *aud_open*() and *aud_close*() as well as all concept of "audit descriptor" was deleted. The *aud_open*(), *aud_close*() and "audit descriptor" were replaced by the P1003.1 *open*() and *close*() calls while the audit log descriptor was replaced with a file descriptor. The result of this change was to make the POSIX audit functions more reflective of existing practice.

B.24.7.2 Distinction Between System Audit Log and Audit Log Files

With the removal of the audit descriptor abstraction some semantic differences between the "system audit log" and file-based audit logs (i.e. non-system logs) surfaced. The primary difference being the fact that the system has some a priori knowledge of the system audit log while the file-based audit log may only be known by the application. An example of the difference between the "system log" and file based logs lies in the amount of support which may be provided by the system in ensuring the integrity of the audit records and the audit files. In the case of the system log, the system is responsible for ensuring the integrity of the audit log. For example if an application issues an *aud_write*() call on the system audit log, the system is responsible for ensuring that the audit data is eventually written to a properly formatted audit log. The system is also responsible for proper sequencing of the records and supplying any accessory information necessary to post-process the record (e.g. UID to text representation). When dealing with a file based audit log the system cannot guarantee that the file specified as an "audit log" is in fact properly formatted (i.e. meets the system's requirements for a proper audit log), that the file offset is correct or that any accessory information required for later translation (by *aud_rec_to_text*()) is properly represented in the file. Additionally, if multiple *aud_write*() calls are made to the file based audit log the system has little control over the sequencing of the records. The only method provided by the standard for providing the concept of "next" record is via the POSIX concept of file append. That is if the file based audit logs are opened with the O_APPEND option the system can provide the assurance that the "next" record written is properly placed in the audit log.

B.24.7.3 Read/Write access to the Audit Log

Appropriate privilege is required to write to the system log, but is not normally required to read it. The rationale for this is that the write interface does not require that the log has previously been opened (because the application should not have unrestricted write access to the audit log, but only the ability to request that records be added to the log (subject to an implementation specific pre-selection policy); indeed, it may not even know the name of the file in which the log is stored). However, for the read interface the log must first be opened, and normal system access controls can be applied.

No privilege requirements are placed on implementation-defined audit logs though implementation-defined forms of access control (including privilege) may be applied.

B.24.7.4 Space Allocation

Space allocation for auditing functions is handled by the system throughout, with the user only being required to notify the system when an item is no longer required (by calling *aud_free*()). Functions that create or read in data on behalf of the user automatically allocate space for the data: for example, for records read from audit logs and for text strings created by *aud_id_to_text*(). The only exception is *aud_copy_ext*() which specifically copies into user-created space.

B.24.7.5 Audit Identifiers

The audit ID, an identifier conceptually different from a UID, was introduced as a means of satisfying the requirement for individual accountability. While this requirement can be met in other ways (e.g., unique UIDs) it was felt that the introduction of the audit ID was the best means of meeting the requirement.

In many existing systems, the user has a username and a user ID. Neither of these is appropriate for use as the audit ID, because POSIX.1 does not require that either of these be mapped to an individual human user. Further, the user ID is the basis of the (DAC) authorization policy of the system, which is logically distinct from the accountability policy. In particular, some systems allow aliasing of one user ID to several usernames that all have the same DAC authorizations, or permit several users to share a username; this is incompatible with use of the user ID as an audit ID. An audit ID has its own unique type *aud_id_t*, because only by doing this could an audit file be analyzed on systems of a different type to the one on which it was generated. Some implementations might wish to define mappings between *aud_id_t* values and implementation-defined identifiers, such as personnel numbers; this is not subject to standardization.

Note that there is nothing to stop a particular implementation from implementing user ID and audit ID for each user as the same value, as long as it maintains individual accountability. However, confusion might arise from the existence of two sets of interfaces to the same value. There is no requirement on how the audit ID is assigned, thus it can be administrator or system assigned (in the latter case, perhaps equal to the UID).

Currently, two functions are provided for processing audit IDs. The function, *aud_id_to_text*() is provided to allow an application to convert an audit ID to a string identifying the corresponding individual user. The function, *aud_id_from_text*() is provided to allow an application to convert a string identifying an individual user to an audit ID. The audit option does not define any relationship between the strings handled by these functions and the *pw_name* field obtainable from the function *getpwuid*(). Using these interfaces, an audit post-processing application could provide record-selection facilities that permit an auditor to select records based on the identity of the individual accountable for actions; or could present the identity of the individual responsible for a particular record to the auditor.

A further function, *aud_get_id*() is provided to allow a process that generates records of its own activities to obtain the audit ID of the user accountable for the actions of a client process and include it in such records.

Note that the functions to set, store and allocate audit IDs are not defined by this standard, since these are considered to be administrative and therefore out of scope.

B.24.7.6 Audit Post-Processing Interfaces

B.24.7.6.1 Reading the Audit Log

This standard provides a single read function *aud_read*() which operates with a file descriptor returned via *open*().

The *aud_read*() function returns a pointer to the next sequential record in the audit log. Note that it is up to the underlying implementation to ensure that the next sequential record is returned. Certain events occur on a system for which sequence is important. For example, a parent process forks a child. It is possible that audit records from the child may appear in the *physical* log prior to the record indicating the fork event had occurred. In any case, it is important that the record for the parent's fork is returned prior to any subsequent records for the child (provided, of course, that the implementation-specific pre-selection policy causes the fork event to be recorded). While the records in the internal audit log may not be in the proper logical sequence, the sequence returned by *aud_read*() must reflect the proper sequence.

Note that if an application chooses to write its own audit records to a file-based audit log (e.g. not the system log) it is left largely up to the application to ensure | that the records are properly sequenced. The only mechanism provided by POSIX for maintaining the sequencing of records written to a file-based audit log is via the O_APPEND flag supplied on *open*().

Since the system is not controlling the file-based audit log there may be no additional (system supplied) sequencing information provided.

B.24.7.6.2 Parsing Audit Records

An audit log may contain records in multiple data formats. All data in any given record will be of the same format. The only format currently defined is AUD_NATIVE; previously an AUD_PORTABLE format specifier was also included, but this has been removed in the current draft because of the decision to delay addressing the issue of portable data interchange formats.

A previous draft stated that access to the sets of data within the various sections (headers, subjects, event-specific data and objects) of an audit record and to the individual fields within these sets was sequential, i.e., to get to the nth field required reading all the fields up to that one also. Several objections were made to this claiming that it was both restrictive and inefficient: it prevented the reading of the fields or sets in an arbitrary order and it required the processing of fields or sets that were not needed. To respond to these objections, a third parameter has been added to the aud_get_*() and the aud_get_*_info() functions, where * is one of hdr, subj, event or obj.

For the aud_get_*() functions, this parameter represents the ordinal number of the set being requested in the appropriate section. This allows random access to the sets, while at the same time allowing all of the sets in a section to be processed sequentially.

For the aud_get_*_info() functions, the third parameter represents a field_id, identifying the field being requested; for system-generated records there are defined values of field_id for each item; and the interfaces for construction of application-generated records allow the application to specify the field_id for each item (see below). Thus the field_id allows access to specific fields within the set. Note that field_ids are not necessarily sequential. In addition, two special field_ids, AUD_FIRST_ITEM and AUD_NEXT_ITEM are provided to allow sequential access to the fields within a set. This can be used for rewinding a set. Thus, both random and sequential access to the fields in a set are provided.

Note that the *aud_get_*()* interfaces operate on audit record descriptors as returned by any of *aud_read*(), *aud_init_record*() and *aud_dup_record*(). The decision to use symmetric interfaces allows applications greater latitude in processing a record and allows the implementation to be considerably simplified because separate writing functions are not needed for records that are read from the log as opposed to those that are created from scratch.

As mentioned above, the *aud_read*() function returns a pointer to an opaque structure defining the next sequential record in the audit log. This record is then read in logical pieces: the record header, subject attributes, event-specific information and object attributes. The record segments are read by calls to the following functions:

1. *aud_get_hdr*()
2. *aud_get_hdr_info*()
3. *aud_get_subj*()
4. *aud_get_subj_info*()
5. *aud_get_event*()
6. *aud_get_event_info*()
7. *aud_get_obj*()
8. *aud_get_obj_info*()

aud_get_hdr() returns a descriptor for the header information. *aud_get_hdr_info*() takes the descriptor returned by *aud_get_hdr*(), and returns the data item from within the header of the audit record identified by the field_id. If sequential access is being used, then repeated calls using AUD_NEXT_ITEM as the field_id return the data items from the header in a predefined order.

aud_get_subj() returns a descriptor for a set of subject attributes. *aud_get_subj_info*() takes the descriptor returned by *aud_get_subj*(), and returns the data item from within the subject information of the audit record identified by the field_id. If sequential access is being used, then repeated calls return the data items from the subject attributes in a predefined order.

aud_get_event() returns a descriptor for an opaque data item defining a set of event-specific data from the record. *aud_get_event_info*() takes the descriptor returned by *aud_get_event*() and returns the data item from within the event-specific information identified by the field_id. There are defined items of information to be returned in a defined order for the standard audit event types when sequential access is being used. Repeated calls to *aud_get_info*(), are required to read all items of event specific information.

aud_get_obj() returns a descriptor for an opaque data item defining a set of object attributes. *aud_get_obj_info*() takes the descriptor returned by *aud_get_obj*() and returns the data item from within the object specific information of the audit record identified by the field_id. If sequential access is being used, then repeated calls return data items from the object information segment in a predefined order.

Implementations are free to add additional fields to system audit records. As such, any of the audit record segments defined above may be extended. If the implementation extends an audit record segment, the implementation-defined data items are appended. That is, the implementation-defined data items will be read using AUD_NEXT_ITEM after all the items defined by this standard. Note that this means that an application must issue successive calls to the above interfaces to make sure all data items in a record are read.

B.24.7.6.3 Example of Use

The following describes a brief example of the POSIX.1e audit functions used to read records from an audit log:

```
int            sys_ad1;  /* file descriptor to the audit log */
aud_rec_t      aud_rec1;   /* record descriptor */
aud_hdr_t      aud_hdr;  /* audit record header */
aud_subj_t     aud_subj; /* audit subject info */
aud_event_t    aud_event_info; /* audit event information */
aud_obj_t      aud_obj; /* audit object information */
aud_info_t     aud_info_descr; /* audit info descriptor */

sys_ad1 = open (log, O_RDONLY)   /* Open an audit log */

while ((aud_rec1 = aud_read (sys_ad1)) != (aud_rec_t) NULL)
{

/* Get audit header & header information */

        aud_get_hdr (sys_rd1, 1, &aud_hdr);
        aud_get_hdr_info (aud_hdr, AUD_EVENT_TYPE_ID, &aud_info_descr);

[ repeated calls to aud_get_hdr_info to get all hdr info ]

/* Get audit subject & related information */

        aud_get_subj (sys_rd1, 1, &aud_subj);

/* Get the UID from the subject portion of the record    */

        aud_get_subj_info (aud_subj, AUD_EUID_ID, &aud_info_descr);

[additional calls to aud_get_subj_info for example ... ]

        aud_get_subj_info (aud_subj, AUD_MODE_ID, &aud_info_descr);


/* Get audit object & related information */

        aud_get_obj (sys_rd1, 1, &aud_obj);

[additional calls to aud_get_obj_info for example ... ]

        aud_get_obj_info (aud_obj, AUD_ACL_ID, &aud_info_descr);

/* You could now use the POSIX.1e ACL i/fs to analyze the ACL */


/* Get audit event & related information */

        aud_get_event (sys_rd1, 1, &aud_event_info);
```

```
[additional calls to aud_get_event_info for example ... ]

        aud_get_event_info (aud_event_info, AUD_PATHNAME,
                &aud_info_descr);
}

close (sys_ad1);
```

In the above example, the while loop reads records sequentially from the audit log, referenced by *sys_rd1*. The record is then parsed by a series of calls to; *aud_get_hdr*(), *aud_get_hdr_info*(), *aud_get_subj*(), *aud_get_subj_info*(), *aud_get_obj*(), *aud_get_obj_info*(), *aud_get_event*(), *aud_get_event_info*().

Note that the addition of the field_id allows for (somewhat) random access to the record. In previous versions of this standard, access of this nature was not provided and access to a particular part of the record was sequential. Additionally there had to be some a priori knowledge of the format of the record (i.e. that UID was the 4th field in the record). This problem has been eliminated with the addition of the field_id.

The audit records are processed in logical blocks, the header, subject, object and event information. The *aud_get_∗*() interfaces are used to (logically) extract the corresponding logical block of the audit record so it may be processed by the application. In an implementation, the *aud_get_∗*() may simply position a index to a portion of the audit record. For example a call to *aud_get_hdr*() may simply position a index to the beginning of the audit record header. After the call to *aud_get_∗*(), subsequent calls to *aud_get_∗_info*() are used to extract data fields from the record. For example, repeated calls to *aud_get_hdr_info*() are made to extract the header data items from the audit record.

B.24.7.6.4 Audit Record Conversion

A function is provided to allow audit records to be converted from internal (native) format to human readable format. This function is primarily intended to allow applications to display audit records to a user.

The function *aud_rec_to_text*() converts an audit record, pointed to by an *aud_rec_t*, from internal format to human readable text. The function returns a pointer to the converted record. All space required for the converted record is allocated by the underlying implementation. Aside from the ordering of information in the converted record, the standard does not specify any details of the text; thus the output of the function can be displayed to a user, but cannot be further processed by an application (e.g. adding special formatting). Portable post-processing applications that want to provide formatted text for audit records themselves can do so by using the *aud_get_∗*() and *aud_get_∗_info*() functions to obtain the content of the record, and other POSIX.1 functions to convert each item to text. In draft 13 and 14 there was an attempt to define more details of the output of *aud_rec_to_text*(), but this was widely criticized (for example, it used new-line characters as delimiters, but these were taken to be formatting which was stated to be inappropriate to POSIX.1); therefore these details were withdrawn.

The converse function is not provided by the standard: there is no requirement to be able to take a human readable record and convert it to internal form in order to support either post-processing or self-auditing applications.

B.24.7.6.5 Copying Audit Records

The working group determined that some applications would find it desirable to save audit records. They may be saved for functions such as backup/restore or for applications which are building a database of audit records for later processing. One way to achieve this is just to *aud_read*() records from one log and *aud_write*() them to another. However, this is not very flexible, since the destination has to be an audit log. It is desirable that it be possible to store a record in any user-defined destination. Since the POSIX.1e audit functions use system allocated space to store audit records, a provision needed to be made to copy the audit record from system managed space into user-managed space. Conversely, the ability to move the record back into system managed space and allow it to be processed by the POSIX.1e audit functions was also needed.

The function *aud_copy_ext*() copies an audit record from system managed space to user-managed space. It is the responsibility of the application to ensure that adequate space is reserved for the copied record. To allow the application to determine the space required to hold the copied record, the function *aud_size*() is provided. The *aud_size*() function, accepts a pointer to an audit record in internal format and returns the size required to hold the audit record in user-managed space. Note that the size returned by *aud_size*() may not be reflective of the space allocated for the internal record because pointers or various compression techniques may be used by the underlying implementation to reduce the amount of space required to store audit records.

The function *aud_copy_int*() copies an audit record from user-managed space back to system managed space. This function was provided to allow applications to reprocess audit records that have previously been copied to user space and, maybe, saved. It was suggested that if the POSIX.1e audit functions could be made to operate on the user-managed copy of the record this capability would not be needed. However, because the underlying implementation may use various techniques to compress the size of internally stored records (e.g., pointers) the assumption that the POSIX.1e audit functions could be used on copied records was not valid. The working group did not want to constrain implementations by requiring that the internal and user-managed copies of audit records be identical.

B.24.7.7 Application Auditing Interfaces

B.24.7.7.1 Constructing Audit Records

In draft 12, interfaces were defined that allowed an application to construct an audit record before writing it to an audit log. However, although it was clearly the intent that the application should be able to alter fields in the record, and thus reuse the record, this was not in fact possible. In ballot, this deficiency was widely criticized, as was the efficiency of such interfaces without an ability to reuse records.

In draft 13, a major simplification of the interfaces was proposed. All interfaces for constructing audit records were removed, and instead a data structure approach was proposed; the application constructed the record using rows of type/length/pointer structures to define the data, then passed these structures to *aud_write*(). This too was widely criticized, as being insufficient application support, incompatible with the style of the rest of the standard, and providing insufficient structuring capabilities: there was, for example, no means to indicate that a particular group of data items in the record all related to one object or subject.

The standard has now reverted to interfaces based on those of draft 12, but extended and completed to allow records to be built with all the structure of a system-generated record, and full facilities for altering and reusing records. Thus the objections to both the draft 12 and the draft 13 proposals should be satisfied. Indeed, much of the flexibility of the earlier, token based, proposal has been achieved, without however proposing as many interfaces as that did.

The intent of the supplied interfaces is that an application should be able to implement any reasonable strategy for constructing audit records. For instance, an application is able to include much or little structure information in records: it can specify that most of the data in the record has no defined structure; or it can structure the data according to the subject(s) and object(s) to which it relates, and give meaningful data types for much of the data. Also, an application can chose to

- Create a new *aud_rec_t* for each record it constructs, deleting the *aud_rec_t* when the record has been written to the audit log, or

- Reuse a single *aud_rec_t* for various records, using the various *aud_put_∗*() and *aud_put_∗_info*() interfaces to add information, and using the various *aud_delete_∗*() and *aud_delete_∗_info*() interfaces to remove information, between invocations of *aud_write*().

The *aud_put_∗*() interfaces allow an application to ask the implementation to create new sections (e.g. header, sets of object attributes) in an audit record; the interface returns to the application an identifier (an opaque data item) for the newly created section. The application used this identifier when adding content to the section, and also when it wants to add another new section before the existing one.

The *aud_put_∗_info*() interfaces allow an application to add content to a section created as above. The application tells the i/f an identifier (an integer) for the item that it is adding to the section; it can also give the i/f an identifier for a previously added item before which the new item is to be placed.

B.24.7.7.2 Writing the Audit Log

The ability to write to the system audit log cannot be generally available, because it could provide a malicious user with a means of denying service to other users (by filling up the audit file) or misleading an audit administrator (by seeding the audit log with disinformation). Accordingly, utilities that use the *aud_write*() interface to write to the system audit log must have appropriate privilege and be trusted to use it properly.

The *aud_write*() function accepts an *aud_rec_t* pointing to an audit record which may have been constructed using the interfaces described above, or may have been read from another audit log. Some earlier drafts of the standard did not permit an application calling *aud_write*() to specify all the sections of a record; some did not permit the internals of a record to be structured into sets of related details (e.g. object attributes); some did not permit a record to be read from one log and written to another. All of these earlier restrictions were the subject of ballot objections, leading to the current interfaces.

B.24.7.7.3 Auditing Suspension and Resumption

Any process doing its own auditing may wish to suspend standard auditing of its operations. This is likely to be used mainly by processes auditing themselves at a much finer or coarser granularity than the kernel. For example, a program that scans the filestore periodically and moves to tape any files that have been unused for a long time could audit the movement of the files itself (in a more meaningful way than the kernel); it seems unnecessary to record that it checked the access dates of all files in the system, which would merely clutter the audit log with data. Even standard utilities (with appropriate privilege) might make use of this facility, to provide a higher level view of events than would be given by the kernel. The interface used to request that the system suspend and resume system auditing of the current process is *aud_switch*().

B.24.7.7.4 Error Return Values

If the symbol {_POSIX_AUD} is defined, then the implementation supports the audit option and is required to support the audit functions as described in this standard. If the symbol {_POSIX_AUD} is not defined, then the implementation does not claim conformance to the audit option and the results of an application calling any of the audit functions are not specified within this standard. An alternative is for the audit functions to specify that the error return code [ENOSYS] be returned by the functions if the audit option is not supported. However, in order to remain compliant with the policies of POSIX.1, this standard cannot specify any requirements for implementations that do not support the option.

B.25 Capability

B.25.1 General Overview

Goals

The primary purpose of defining interfaces for a capability mechanism within this standard is to provide for a finer granularity of controlling and granting system capabilities than the traditional super-user model.

The major goals of this standard are to:

(1) Provide a portable means of supporting the assignment of an ability for a process to invoke or perform restricted system services.

(2) Support the implementation of *least privilege* security policies by providing the means to constrain a process by enabling it to invoke only those system capabilities necessary to perform its specific tasks.

The additional goals of this standard are to:

(1) Define a common terminology for addressing the topic of capability.

(2) Define the semantics of how process capabilities are acquired and altered.

(3) Define the system functions and utilities necessary to utilize capabilities.

(4) Provide compatibility with programs that depend upon the set user id on execution and set group id on execution behavior to gain access to system resources.

(5) Provide the means by which an implementation may grant capabilities in order to emulate the traditional super-user.

(6) Allow for extensibility by future implementations.

(7) Define a minimum set of capabilities necessary to support the development and execution of security-relevant programs.

(8) Ensure that there is a mechanism by which capabilities may be transported with their associated files.

It has been pointed out that the term *privilege* has been commonly used for a mechanism that achieves the above stated goals. However, the term *privilege* is also commonly used in the international community to mean something else entirely. It is felt that the confusion that would result from using the term *privilege* would not serve this standard well.

A capability mechanism is a common requirement for most operating systems. Capability controls the availability of particularly important system services to processes that are known to maintain system integrity.

The principle of *least privilege* is a common requirement of security policies, that is, granting to a process only the minimum rights and capabilities necessary to perform a task. The purpose of this principle is to constrain the damage that may arise from a violation of the security policy, e.g., disclosing confidential information or corrupting the integrity of the system. We must emphasize here that the standard does not (nor can it) specify a *least privilege* mechanism—only interfaces that, when used with a correctly defined set of capabilities, could successfully be used to implement a *least privilege* security policy.

An example of the application of the principle of *least privilege* in the commercial environment is the separation of roles in an accounting department. In most firms of any size, the person who records and manages the Accounts Receivable is **NOT** the person who records and manages the Accounts Payable. This is so one person cannot create false bills and then write checks to pay them. A current example in the computer world is the use of the restricted shell (rsh) for computer operator consoles — the operator, who has a great deal of potential access to the entire computer system by virtue of his or her physical access to the machine, can have that access limited to those functions actually required to perform his or her job by the system administrator.

Additional goals 1 and 2 are natural intermediate goals for meeting our major goals. Before a capability mechanism can be defined, a terminology and the basic concepts of capability must be laid out. Once that has been achieved, then the semantics of how capabilities are acquired, manipulated, and controlled need to be defined. Only after this step has been accomplished — deciding what operations are required to provide a capability mechanism — can the next step be taken.

Additional goal 3 is the end result of this effort — the definition of the interfaces that can be used to provide the semantics developed above. The specification of these interfaces is the entire purpose of this effort — to provide a set of tools that can be used by conforming applications to perform those tasks necessary for its functions.

Additional goals 4 and 5 are compatibility goals. The set user id and set group id mechanisms of POSIX.1 continue to function as they have in the past, providing DAC access to objects based upon the owning ids of the executed file. Set uid root functionality may be provided by appropriate use of the file permitted capability set. While our goal is to provide a mechanism that will support implementations intended for high levels of trust, there will be implementations that will still need to support existing setuid root programs, and implementations that will still provide the 'superuser' identity to administrators. While we would like to discourage both of these practices, we understand that current practice is often slow to change and that some existing applications will have to run unmodified on secure machines for at least a transition period.

Goal 6 is a basic goal of all systems — motherhood and apple pie to engineers. All systems need to permit extensibility and flexibility so that unforeseen situations and future improvements do not require an architectural change in order to accommodate them. At some point, every system will need to be completely replaced, but one would like to push that off as long as possible. Implementations will need to provide capabilities not specified here to accommodate various security policies and system functions not part of this standard. Extensibility is therefore an absolute requirement.

Goal 7 is the specification of a standard set of capabilities — is a necessary part of this effort. Trusted applications will need to be able to acquire a certain capability to perform a specific function across all compliant implementations in order to be portable, and that capability will need to have the same meaning across implementations.

Goal 8 was agreed upon primarily to support system backup and restoration operations. This goal does not include the transfer of capabilities from system to system necessarily. Indeed, there is a good argument that requiring that degree of portability adds risk to a system, and that a system administrator should be required to approve every new trusted program before it is assigned capability attributes. As a result we define file capability attributes, but not their actual representation or how they are stored with the file on a tape.

Scope

The scope is the natural result of our goals. In order to support the principle of least privilege, interfaces that provide the means for programs to enable and disable capabilities while running are necessary. In order to support the compatibility goals, there must be a means for programs to pass capabilities to other programs that they execute, and the semantics of that inheritance must therefore be specified to some degree. Because it is programs that are the "trusted" agents on implementations, there must be some method to identify them as trusted — therefore attributes associated with program files must be specified. Finally, a small set of capabilities to be used with the interfaces and utilities in the existing POSIX.1 specifications must be defined so that writers of conforming applications know which capabilities will be available to perform various functions and their appropriate use.

Purpose of a Capability Mechanism

The purpose of a capability mechanism is to provide a finer granularity of control over the access to restricted system services to specific users or processes than that provided by the traditional POSIX.1 "UID 0" access mechanism. A general purpose capability mechanism supports not only the ability to implement the principle of least privilege, but also provides the foundation for building an authorization mechanism to support security administration. The interfaces and concepts presented in this document have been designed to meet these requirements.

Authorization vs Capability

The power to perform an action in a trusted system based on user identity is called an "authorization." Authorizations are generally designed around operational requirements and tasks rather than system services. For example, an authorization to perform backups would be granted to a user. The backup program however, would enable and disable specific capabilities to perform the backup function. A system that supports authorizations simplifies the administrative task of the security officer by eliminating the need to comprehend exactly which capabilities each program requires and how to allocate those capabilities to users.

The establishment of a user identity and a user's authorizations based on that user's identity is presently outside the scope of the POSIX standards. Because of this, the assignment of authorizations to users through a program such as `login` and the use of an authorization mechanism for determining utility capability bracketing is presently undefined, as is the relationship between the authorization mechanism and the capability mechanism used by a program. It is not, however, our intention to preclude any implementation of a user authorization mechanism with this standard.

General Discussion of Capability

Currently, most POSIX.1 and POSIX-like implementations grant all capabilities to a particular user ID – 0 (root). Most of the time, the ability to log in as or to assume this identity is restricted to a small set of users on a system, one of whom is the system administrator. The "root" account has the ability to execute any utility or use any system function regardless of what security restrictions may be involved. Such special rights are necessary to do many administrative tasks, such as system backups and restores, writes into special files, and to operate processes such as line printer daemons and mail handling servers.

In the vast majority of cases, however, a process needs to invoke only a few specific restricted system services or override a single type of access permission in order to accomplish its task. A line printer daemon, for instance, needs to be able to read any file in the system, but does not need the ability to write into them, and the same with a backup program. A login program needs to be able to change its user identity, but it does not need to modify disk quotas, and so forth.

As has been demonstrated numerous times, the requirement that a process be granted the ability to bypass all the security restrictions in a system just to bypass some of them leads to accidents and purposeful misuse. Many times, users do not realize that they are in privileged mode and perform a destructive action (`rm` ∗) without realizing that the system will not stop them in their current state. Other times, a user acquires the ability to become "root" for a perfectly legitimate reason, and then passes it on to other users or applies the special abilities "root" provides in ways not intended by the system administrator.

A capability mechanism provides the means for a system administrator to grant a program the ability to use a restricted system service or bypass specific security checks. For instance, user Joe can run the backup program for an entire network (that can read every file on the network) from the "admin" host. Properly implemented and administered, the capability mechanism could permit Joe to perform his assigned task, but could prevent abuse of the world read access capability such as browsing files normally not accessible to Joe.

Principle of Least Privilege

A process's need for capability access to system resources and functions does not justify giving the process uncontrolled use of capabilities. It is also not appropriate to establish for a process chain (a sequence of programs within a single process) a set of capabilities that remains fixed and active throughout the life of that chain. Rather, the set of active capabilities of a process can be expected to change as the functions of the process change, so that the process has active at any time just those capabilities needed to perform its current function. This is an application of the principle of least privilege, and it applies equally to users and to processes.

Implications of the Principle of Least Privilege

Any capability mechanism will associate with each process a set of capabilities that the process can potentially use, but capabilities should be controlled at the level of granularity of individual programs. The most straightforward way to do that is to associate capability controls with the individual program files. The first requirement implied by the principle of least privilege, *to control capability at the granularity of individual programs*, leads to the assignment of capability attributes to program files; this is the file capability state.

If a program is always executed in a single context, e.g., by a single user to perform a single function, then the specific set of capabilities for that context-program combination applies to all invocations of the program. However, in general a program is executed in varying contexts, e.g., by different users, or on different files, or for different purposes—such as a printer spooling program. Thus we need to be able to change the capabilities of a process as its circumstances change.

This is the second requirement implied by the principle of least privilege: to control use of capability within the context of intended use. It further suggests that process capabilities be divided into two classes: capabilities that are currently active, and capabilities that could be activated. We do this by creating two corresponding kinds of process capability flags: *effective* (indicating that the capability is active) and *permitted* (indicating that the capability could be activated). Thus a process can increase its current set of active capabilities by making *effective* any capability that it is currently *permitted* , and can reduce its active capabilities at any time while retaining the ability to restore them. This ability of a process to adapt its active capabilities to the needs of the moment is referred to in the standard as the "time bounding of capability" and is sometimes also referred to as capability bracketing.

If a process image is instantiated from a program file, its capabilities will be affected by the capability state associated with the file. A program will *exec* a program file to instantiate its successor program in a process chain. Here too the principle of least privilege implies that we adapt the use of capability to the context of use. There are two general ways to do this.

> In the first, *exec*() constrains the maximal extent of capabilities for the process image it instantiates from a program file. In this way the invoking process image can limit the capabilities of the successor process image.

> In the second, *exec*() plays no role in limiting the set of capabilities that the instantiated process image may have; rather, the successor process image sets the capabilities itself, choosing them from the set of capabilities associated with the program file from which it was instantiated, and possibly from a set of capabilities that a predecessor process image had passed on.

In either case, the advantage of passing capabilities along a process chain is that it allows the process to dynamically build up a capability context, rather than limiting its capability context to a single, per–process image state.

Besides providing a capability for a process image to pass capability information to subsequent process images, it may be desirable that a specific process image have capabilities that are not *permitted* to any of its predecessors. We therefore need a way to increase the capabilities of a process based on the program file being *exec*'ed.

Finally, we observe that a process image may wish to pass capabilities to some successor process image through an intermediate third process image that is not itself trusted to properly use the passed capabilities. For example, a process may initiate an untrusted shell that in turn will *exec* a third program file.

B.25.2 Major Features

There must be some method for a process to acquire capability(s) it needs if it is to be able to use it(them) at some point. Because capabilities are security relevant, this method must be restricted to a trusted part of the system, which must grant the ability to use capability based on one or more characteristics of the process. We assert that the characteristic most relevant is the identity of the program or programs that are run within that process.

A result of this assertion, that the identity of programs is the primary characteristic used to assign trust, is the requirement that there be some means to identify a program file as trusted. There are several means available to do this. The first is to embed some form of identification in the program file itself in such a way that the loader can interpret it. This leads to problems, however, in that different installations may have different security policies, and that system administrators may not trust the program developer enough to set the proper capability attributes. The second alternative is to attach capability attributes to the program file. This alternative provides a much larger degree of flexibility, in that system administrators can differ in their trust of a particular program without modifying or altering the actual program itself, and is much more consistent with current practice and methods. As a result, file capability attributes were proposed.

B.25.2.1 Task Bounding of Capability

This standard has the advantage of being flexible enough that a given capability may be bound either for the duration of an executable program or the duration of a single system call. This allows flexibility in the granularity of capability, provides support for backwards compatibility, and allows trusted programs to support capability bracketing. The main advantage in task bounding of capability is that it reduces the chance that program errors will have security-relevant side effects.

B.25.2.2 Capability Inheritance

Trusted programs can perform complicated functions and, as a result, can be very large. The larger and more complicated a program is, however, the harder it is to evaluate for trust and the more difficult it becomes to maintain. In addition, one of the basic tenants of the POSIX.1 operating system is to provide a set of simple utilities that can be executed together or in series to perform more complicated functions. As a result, it is desirable for a trusted program to be able to pass on its capability characteristics to other programs to perform functions it would otherwise have to implement itself.

While one trusted program may want to pass **all** of its capabilities to another, more often the child program only needs a subset of the parent program's capabilities to perform its functions. Also, should the child program be trusted, the parent trusted program may not be aware of how much trust that child program actually has at any given time. Finally, a conforming program CANNOT be trusted to handle implementation-defined capabilities. Therefore, the developer needs to have the ability to restrict what capabilities he or she desires to pass on to the child program, and the system developer and administrator need to have a means of controlling what capabilities they are willing to permit the child program to have.

Since the *exec*() function is the means by which one program invokes another, it must be modified:

- To grant capabilities to programs when they are executed.
- To permit programs to pass capabilities to other programs.
- To restrict which capabilities may be passed from one program to another.

So far, we have provided the basis for program level capabilities. In other words, programs that are granted capabilities using the attributes specified so far have those capabilities during their entire scope of execution. For many systems, program-level capabilities may not provide the level of granularity desired by the security policy. For instance, a program may need to have the capability to write to a system administrative file only during a single call to the *open*() system function. For the remainder of the time the program executes, the capability is available but not required. In order to support implementations that support the concept of least privilege to a finer level of granularity, we need to provide the means by which a program can enable a capability only during the scope of execution for which it is actually required.

In summary, then, the view of the principle of least privilege presented here and the desired functionality described above implies the following:

(1) There is capability state associated with program files as well as with processes.

(2) There are two kinds of process capability attributes: one which defines what capabilities **may** be invoked by the process image, and another that defines what capabilities **are currently** invoked by the process.

(3) There is a way to increase the capabilities of a process that depends on the process image file that it *exec*'s.

(4) There is a way to conditionally transmit capabilities from a process image to its successor image(s).

(5) There is a way to restrict which capabilities may be passed to any particular process image that depends on the process image file.

(6) The *exec*() system function determines the capability attributes of the process it instantiates.

B.25.2.3 Process Capability Flags

A process image acquires capabilities from the set of capabilities attached to the program file from which it is initiated. The *effective* flag determines whether the capability is active for the process. The *permitted* flag determines whether the process may choose to make the capability *effective* . The *inheritable* flag determines whether the process may pass on to its successor process image a conditional right to use a capability. The right must be conditional because the capability may be inappropriate for intermediate image(s). Indication of the successor's appropriate capabilities is reasonably associated with the successor's process image file. In fact, this indication can be made precisely by the *permitted* file capability flag. The determination of the right to use a capability depends on the current process's value of the *inheritable* flag and on the values of the *permitted* and *inheritable* flags of the corresponding file capability. This determination is made by *exec*(). In implementations that depend more heavily on use of the *effective* flag, the *inheritable* flag can be used by a process image to determine the trust associated with its predecessor process image and therefore provide a basis for enforcing its own security policy.

B.25.2.4 File Capability Flags

As we have seen, the principle of least privilege requires that with each program file there is associated the set of capabilities that a process image, instantiated from that file, requires to do any of its functions.

The *inheritable* flag determines which capabilities the resulting process image may pass to subsequent process images and which ones the program may chose to use if the previous program image possessed the capability.

The *permitted* flag determines which capabilities the resulting process image needs to have available in order for the program to function properly, regardless of the capabilities of the previous process image.

The *effective* flag determines which capabilities the resulting process image will possess in its *effective* process set.

The ability to support capability unaware applications on a per executable basis ensures that these programs will continue to function with a limited set of capabilities, thus reducing the risk of unauthorized access to restricted functions. Additionally, the risk of a trojan horse gaining unauthorized access to capabilities is reduced if the inclusion of capabilities into the effective set is automatically limited to a per file basis.

Earlier versions of this standard provided a single *set_effective* flag instead of the *effective* set. The new process *permitted* set was promoted to the *effective* set on *exec*() when this flag was set.

B.25.2.5 The Determination of Process Capability by fork()

This is a simple case. The *fork*() system function is meant to create a new process that is, as much as possible, identical to its parent. Because capability is not an attribute that uniquely identifies a process, such as process ID, the capability state of a child process should be identical to that of its parent immediately after the execution of the *fork*() system function.

B.25.2.6 The Determination of Process Capability by exec()

The *inheritable* and *permitted* capability flags of the program file and the *inheritable* capability flags of the current process together determine the context-dependent set of capabilities *permitted* to the instantiated process. The context–independent set of capabilities that is included in the *permitted* capability set of the program when it is executed is derived from the *permitted* file capability flags associated with the program file. The union of these two sets comprise the set of capabilities that the *exec*() function permits the new process image to use.

The initial state of the *effective* flags of the new process image depends on the *inheritable* flags in the old image and the values of *inheritable*, *permitted*, and *effective* flags of the program file. The justification for selecting the transformation function for process capability state is incorporated throughout the text of this section.

B.25.2.7 Support of the Capability State Attribute on Files

The intent of these interfaces is not to limit the manner in which processes can gain appropriate privilege. Thus, if the value of the pathname variable {_POSIX_CAP_PRESENT} is zero (meaning that the file does not support the POSIX capability state attributes), then it is possible for an implementation to specify other mechanisms. For example, the USL implementation provides both a privilege mechanism and a superuser mechanism.

Certainly, there are implementations that allow files to be exec'ed from file systems that do not support capability attributes (for example, an NFS file system mounted from a system not supporting the capability option). In this case, it is suggested that an implementation treat this file exactly as it would a file without a capability state attribute from a file system that does support capability attributes.

B.25.2.8 Extensions to This Standard

This specification does not preclude providing additional implementation-defined constraints, such as a system-wide configuration variable to further constrain the capability inheritance rules. The value of this variable could be used to act as an additional gating function to permit a single global value to be manipulated by a system security officer to help stop or slow a security breach in progress by preventing any permitted capabilities from being automatically included in every process effective capability set. Additional file capability attributes and file capability flags can also be defined by an implementation. It must be emphasized that such extensions are compliant only if they further constrain (prevent from becoming effective) capability.

B.25.2.9 Process Capability Manipulation

When a process image is instantiated from a program file, its capability flags describe its capability state. As noted earlier, the *effective*, *permitted*, and *inheritable* flags respectively denote which capabilities are active, which may be activated, and which the process image will (conditionally) pass on to its successors. A process should not be permitted to arbitrarily modify these flags, but is restricted according to the following set of rules.

A process can promote to *effective* only those capabilities whose *permitted* flag is set. This lets the process adapt its degree of active capabilities to its current context, and so supports the principle of least privilege. On the other hand, the process can never promote a capability to *effective* if the *permitted* flag is turned off, and can never enable a *permitted* flag that is turned off. Thus the process cannot assume for itself capabilities to which it is not entitled.

To prevent it from accumulating capabilities through inheritance, a process can enable an *inheritable* flag only if the corresponding *permitted* flag is set.

If a process disables a *permitted* flag, the corresponding *effective* flag is automatically disabled. The corresponding *inheritable* flag is not affected, so capabilities can be conditionally transmitted along a process chain whose intermediate processes may themselves have no capabilities. In no other case does changing the value of any flag affect the value of any other flag.

B.25.3 Function Calls Modified for Capability

The standard defines the capabilities required by each of the POSIX.1 functions. However, many implementations included additional functions that should be modified to support the capabilities defined in this standard. While the list presented here is by no means exhaustive, it is included as helpful information for the reader.

Table B-3 – Other System Functions Potentially Affected by Capability Policies

Function
adjtime
bind
chroot
killpg
limit
mincore
mknod
mount
ptrace
readv
reboot
sethostname
settimeofday
shutdown
socket
socketpair
swapon
symlink
syscall
umount
vadvise
vfork
vhangup
writev
sysattr

B.25.4 Capability Header

These types were defined to provide opaqueness and avoid specifying detail that should be left to the implementation. The capabilities defined in this section are limited to those specifically called for in the POSIX.1 standard. Included also are those capabilities defined in POSIX.1e.

B.25.4.1 Rationale for the Selection of Capabilities Defined in the Standard

This section will describe the process that the capability group used to develop the set of capabilities specified in this standard. Enough detail is provided about the process so that an implementor can duplicate it when analyzing an implementation to determine what additional capabilities, if any, are required.

We began the process of defining a capability set for the standard by first developing a set of guidelines to be used. These guidelines are contradictory to a degree, and the group made trade offs between them when discussing each individual capability in order to come up with a minimum set of capabilities that were deemed necessary for the support of conforming applications.

Principles for Determining a Capability Set

Principle #1: A capability should permit the system to exempt a process from a specific security requirement.

In most cases, security requirements found in the function descriptions take the form: "In order for this function to succeed, <requirement>, or the process must possess *appropriate privilege*." A specific example can be found in the POSIX.1 description of the *chown*() function, which states "In order for this function to succeed, the UID associated with the process must match the owner ID of the file, or the process must possess *appropriate privilege*."

This principle is meant to support the principle of least privilege, in that a capability should provide only the minimum rights or authority to perform a specific task.

Principle #2: There should be a minimal overlap between the effects of capabilities.

Capabilities should be defined such that they apply to logically distinct operations, and the granting of a set of capabilities should not, as a side effect, grant an additional capability that is not in that set.

This principle was developed to address the concerns that capabilities should be distinct and unique—no capability or combinations of capabilities should provide the capabilities afforded by another capability. When a system administrator grants one or more capabilities to a specific user or program, they should have some assurance that the recipient is not gaining any additional capabilities.

Principle #3: Insofar as principles #1 and #2 are supported, fewer capabilities are better than more.

When it makes sense to do so, and identical or nearly identical security requirements exist, a single capability should be defined for all those security requirements instead of a separate capability for each individual security requirement.

This principle was defined primarily to support ease of use and ease of administration. If each individual security requirement in an implementation had a unique capability, several hundred capabilities would be required, a management nightmare that would be prone to misunderstanding, confusion and error. If a specific security requirement is especially critical or sensitive, however, it was generally agreed that it should be assigned a unique capability in order to assure positive control over which processes/programs are exempted from the requirement.

Determining the Capability Set

Once the above general principles were agreed to, the group turned to the existing and draft POSIX documents to begin the process of actually developing the set of capabilities included in this standard.

The set of capabilities defined in this document is not intended to be all-inclusive. Implementations may (and probably should) define additional capabilities to support the operation and maintenance of their systems. Finally, it should be emphasized that the development of a capability set is not a cookbook process—implementors must consider their own system security requirements and the design of their own systems when determining what capabilities they will support. Our requirement was to develop a minimum set of capabilities we determined necessary to support conforming POSIX applications.

Step one in the process was to develop a list of security requirements from the POSIX.1, and POSIX.2 documents. This involved searching through the descriptions of the functions and utilities looking for the phrase "appropriate privilege" and also looking for text that implied a security requirement that was not directly stated.

Once we had developed the list of security requirements, or "checks", we grouped sets of identical or nearly identical requirements together, and developed a descriptive name for each individual or group of requirements that remained. When grouping requirements, each case was discussed to ensure that it really did belong to the group, and it was not uncommon for a decision to be re-made as the list developed and additional considerations were brought up.

The last step in the process was to review the entire list. Capabilities were deleted or combined with another capability when it was deemed appropriate to do so with respect to the third principle in B.25.4.1

B.25.4.2 Rationale for DAC Capability Specification

The DAC group defines the extensions to POSIX.6 for a finer granularity of discretionary access control beyond POSIX.1. For systems with {_POSIX_CAP} configured, it is necessary to define the policy override capabilities.

The DAC group initially considered separating DAC overrides into 4 distinct capabilities. These were:

- CAP_DAC_READ
- CAP_DAC_WRITE
- CAP_DAC_SEARCH
- CAP_DAC_EXECUTE.

The CAP_DAC_READ and CAP_DAC_WRITE separation was considered necessary for providing read-only access for a wide range of applications that have no need to write to the objects they are examining. The CAP_DAC_SEARCH and CAP_DAC_EXECUTE capabilities were suggested because it was not necessarily appropriate to group these abilities with the CAP_DAC_READ and CAP_DAC_WRITE capabilities. Also, specification of four separate capabilities maps one-to-one with the existing POSIX.1 features.

The group also considered a single CAP_DAC_OVERRIDE capability, but this granularity was considered insufficient for the following reasons:

- Demonstrated commercial need on other operating systems to support separate CAP_DAC_READ and CAP_DAC_WRITE overrides based on functional requirements. For example, a backup program requires the ability to read all file objects on the system but only requires the ability to write to the backup device. Additionally, this separation provided programmatic support for administrative roles which allow for protection from inadvertent modification of system critical objects.

- Worked examples of trusted systems evaluated at class B2 or higher against the TCSEC on which similar mechanisms were required to meet the System Architecture requirement.

Because the specification of four separate capabilities seemed to be unnecessary, and the specification of a single capability is not sufficient to support commercial requirements, we decided to specify three capabilities and permit implementations to add additional capabilities if appropriate.

In fact, an analysis of the requirements determined that these three capabilities are sufficient to support the principle of least privilege as well as the anticipated commercial demand. Note, however, the specification provides support for implementation defined capabilities where deemed necessary.

The consensus was that applications that required CAP_DAC_READ override would also require CAP_DAC_SEARCH override. Therefore these two capabilities were combined.

B.25.4.3 Rationale for MAC Capability Specification

A MAC policy differs from a DAC policy in that an untrusted process or user does not participate in establishing the access criteria. Rather, the system is responsible for enforcing the policy established by the security officer. As such, the MAC policy can be considered to impose a higher degree of assurance on the protection of an object compared to DAC. Therefore, MAC policy override capabilities must be carefully considered.

The MAC group has established a set of policy overrides that are designed to support sufficient granularity of control to meet the needs of current security standards as well as to meet the needs of future trusted applications, such as databases, multi-level mailers, etc.

CAP_MAC_UPGRADE and CAP_MAC_DOWNGRADE

The MAC group originally considered a single MAC override capability to cover both the upgrade and downgrade cases for manipulating object labels. Although this level of granularity meets the needs of the current TCSEC, more recent security criteria, such as the '91 Compartmented Mode Workstation Evaluation Criteria do require separation of the MAC override capabilities. In addition, the separation of the upgrade and downgrade functions is a common operational requirement. Supporting distinct capabilities is a logical extension of this operational requirement.

CAP_MAC_LOCK

At one time during the writing of this standard, the standard required that a process have MAC write access to a file at the time of a lock operation, or have CAP_MAC_LOCK enabled. These protections were necessary because the set of locks associated with a file are considered to be an object. More specifically, because the data structure which defines the lock on a file can be directly written by processes (by setting locks) and can be directly read by processes (by querying locks), this data structure was deemed a communication channel that must be subject to MAC constraints.

The straightforward application of MAC policy to locks requires that a process have MAC write access to the file prior to setting locks. In a system with only CAP_MAC_WRITE, a process must be trusted to use the override capability appropriately. It can be argued that processes that need to use locks should be trusted enough to use the MAC write override capabilities for this purpose. This approach also has the added feature of minimizing the number capabilities necessary for the MAC policy.

However, the use of CAP_MAC_WRITE to bypass this policy constraint was considered non intuitive and a violation of the principle of least privilege. For example, a process merely wishing to set a read lock on a lower level file simply to read the file, e.g., a password file, would then need to be granted the MAC write capability, despite having no need to write data to the lower level file. Thus in cases such as these, which in actual implementations are likely to be frequent, not only is a powerful capability being used to cover a relatively innocuous activity, but also the use of a write capability to effectively perform a read is confusing. For this reason CAP_MAC_LOCK was originally adopted.

Based on significant ballot objections, this capability was removed and the standard was made mute on the subject of how an implementation handles the channel created by *fcntl* and reading locks.

CAP_MAC_READ and CAP_MAC_WRITE

While the TCSEC does not require separation of the MAC override capability into distinct READ and WRITE capabilities, other security specifications do. In addition MAC is a system enforced policy rather than a discretionary policy, requiring that applications which need only to read an object also have the power to write the object was considered an unwarranted risk. Separation of MAC_READ and MAC_WRITE overrides will encourage application developers to be cautious with their use.

CAP_MAC_RELABEL_SUBJ

The ability of a subject to change its own MAC label is controlled by the CAP_MAC_RELABEL_SUBJ capability. This capability is intended for use by trusted subjects which have the need to modify their label based on some (possibly external) criteria. For example, a trusted server which may need to reset its MAC level prior to executing functions on behalf of a client request. Unlike objects, which tend to have a static label, subjects would need a dynamic label therefore a single capability is more appropriate for subjects.

B.25.4.4 Rationale for Information Labeling Capability Specification

CAP_INF_NOFLOAT_SUBJ and CAP_INF_NOFLOAT_OBJ

These two capabilities are the override capabilities for the Information Label Policy. The INF_NOFLOAT_OBJ capability is necessary to support programs which need to write a shared single file at many information levels. An example of this is the /etc/utmp file which the `login` program writes. Similarly, there are processes which may not wish to allow their information label to float. An example of this would be a server process which must fork off children to perform work in response to a specific request. The INF_NOFLOAT_SUBJ supports these types of processes.

CAP_INF_RELABEL_OBJ and CAP_INF_RELABEL_SUBJ

These capabilities allow processes to explicitly set labels on subjects and objects. As information labels are not an access control policy separate overrides for reading and writing object labels are unnecessary. Rather a single capability is sufficient for applications which need to manipulate information labels on objects.

B.25.5 New Capability Functions

B.25.5.1 Function Naming Scheme

In order to provide for consistency across the sections of this document, a naming scheme for all named entities was adopted. Functions are named with a subsystem identifier—cap_, first, followed by a short name that identifies the type of operation the function performs, then a short name that identifies the data the function operates on. While this scheme generates names that are somewhat longer than are generally customary, it is generally evident from the name of the function what its purpose is and we found it easier to remember them.

B.25.5.2 Allocate, Duplicate, and Release Storage for Capability State

The *cap_init*() function is necessary to create a new object to hold capability attributes. We did not desire to specify the contents and storage requirements of this object in order to permit as many differing implementations as possible. Having provided an allocation function, we need also to provide a free function, *cap_free*(), so that an implementor can release memory and structures associated with a process capability data object. In order to permit the representation to be copied, we defined a duplication function, *cap_dup*().

B.25.5.3 Initialize a Process Capability Data Object

The *cap_clear*() function permits a program to set the representation of the capability state to a known secure state. This has the advantage that a conforming program need not know all the capabilities defined in the implementation to set this "secure" state.

B.25.5.4 Read and Write the Capability Flags of a Process

The *cap_set_proc*() and *cap_get_proc*() functions permit a program to obtain and set the capability state of a process atomically. The atomicity of these functions is significant—the state of a process could possibly change between multiple invocations of a function that deals with only one capability flag at a time.

The *cap_set_proc*() function is an especially security-critical function in any system that implements a capability mechanism, as it is here that the standard requires that the security policy regarding the manipulation of process capability state be applied. The requirement that the capability be permitted to the running program provides the primary means to limit what capabilities any one program can propagate through the system.

B.25.5.5 Get and Set Values of Capability Flags

The *cap_get_flag*(), and *cap_set_flag*() functions provide the standard interface for getting and setting the values of the capability flags. Portable trusted applications will need to manipulate the process capability state on different implementations so that they can perform "time bounding of capabilities" and set what capabilities they want to pass on to programs that they *exec*. The *cap_get_flag*() function permits a conforming application to determine the state of a capability without actually attempting to use it. Without a *get* function, conforming applications could generate numerous unnecessary audit messages attempting to use capabilities not available to the current invocation of the program. The *cap_set_flag*() is the only means by which a conforming application can alter the state of a specific capability.

B.25.5.6 Exporting Capability Data

The *cap_to_text*() and *cap_from_text*() functions translate process capability states between human-readable text and capability data object representations. These functions are necessary to provide a portable means of transferring capability information between systems. Implementations may also use these functions to translate between text and data objects in order to support capability manipulation and display. One possible use is the display of available capabilities using a trusted shell utility, another is the transport of capability information across a network in a form recognizable to all machines.

There are other valid reasons to want to store process capability data objects—for instance, the process capability state could be an important field in certain audit records. Textual data, while easily readable, is not compact. The internal representation of capability state is not guaranteed by this standard to be valid outside of the context in which it exists. For instance, it may contain pointers to strings spread throughout the system-managed space. This was intentional to permit implementors the maximum possible freedom. Because of this, the *cap_copy_ext*() and *cap_copy_int*() functions are provided to convert the internal representation to and from a self-contained binary format that should be more compact than the textual version.

B.25.5.7 Manipulating File Capability Flags

When we developed the set of functions to manipulate file capability flags, we had several goals in mind. First, we wanted the assignment of capability attributes to files to be atomic—there is a reasonable probability that a program file could be executed by another process in the middle of a sequence of non-atomic file attribute operations. Second, we wanted to continue to hide the actual representation of capability attributes in the standard and permit a wide variety of implementations. We feel that the interfaces defined support an implementation where the file capability attributes are stored in the files' inode AND an implementation where the files' capability attributes are stored in a central database maintained by a capability server. Finally, the group as a whole decided to specify procedural interfaces wherever possible instead of data-oriented interfaces in order to better support extensibility and flexibility in the future.

We did not resolve the atomicity problem to the extent we desired, but felt that the correct solution was really outside of our scope. POSIX has no mandatory file locking mechanism, hence, there exists the possibility that file attributes have been altered by a second process between the time the first process has read them and the time it attempts to set them. This is a general problem not limited to file capability state, but includes all file attributes and data. Instead of solving the general problem, we have specified functions that read and write the entire capability state, rather than permit programs direct access to individual capability flags and attributes. This should minimize, but not eliminate, this problem.

B.25.5.8 Read and Write the Capability State of a File

The *cap_get_file*() and *cap_set_file*() functions permit a program to obtain and set the capability state of a file atomically. The atomicity of these functions is significant—the state of a file could change between multiple invocations of a function that deals with only one capability flag at a time. In addition, it keeps device I/O required by the capability function set to these two functions—all the rest can (but are not required to) be memory only operations.

The *cap_set_file*() function is a security-critical function in any system that implements a capability mechanism. We therefore imposed a number of restrictions on the ability of programs to use this function. The requirement that the capability be permitted to the running program provides the means to limit what capabilities any one program can propagate through the system. The requirement to have the CAP_SETFCAP capability effective provides the means to restrict programs that are permitted a capability for other purposes from granting it to programs that the system administrator has not specifically approved. The remaining restriction is that the UID associated with the process be equal to the owner of the file or that the process have the CAP_FOWNER capability effective—this is a standard restriction for all operations dealing with file attributes. The combination of restrictions above are the minimum necessary to prevent the unauthorized propagation of capabilities.

Many times a file is already opened when it is being assigned attributes. Many programs use file-descriptor based functions in order to avoid the performance penalty incurred to perform repeated pathname resolutions. To accommodate this class of applications, we have provided the *cap_set_fd*() and *cap_get_fd*() functions to set and get the capability state of an opened file.

B.25.5.9 Error Return Values

If the symbol {_POSIX_CAP} is defined, then the implementation supports the capability option and is required to support the capability functions as described in this standard. If the symbol {_POSIX_CAP} is not defined, then the implementation does not claim conformance to the capability option and the results of an application calling any of the capability functions are not specified within this standard. An alternative is for the capability functions to specify that the error return code [ENOSYS] be returned by the functions if the capability option is not supported. However, in order to remain compliant with the policies of POSIX.1, this standard cannot specify any requirements for implementations that do not support the option.

B.25.6 Examples of Capability Inheritance and Assignment

B.25.6.1 A User-based Capability Model

The inheritance mechanism provides a method of controlling a process' capabilities based upon the context in which the process is executed. An important part of the context is the identity of the user invoking the process. It is possible to associate capabilities with a user profile which defines a subset of the capabilities available to the trusted programs that a user may execute. Trusted programs may therefore have greater or lesser abilities depending on which user executes them. These user capabilities constitute the *inheritable* capability set on session initialization. A subset of the user capabilities could be selected by utility options to support user roles. The login shell will probably be an untrusted shell, and in itself be incapable of using capability.

It is not possible for a user to alter the set of *inheritable* capabilities within an untrusted shell or program. A user can only modify the set of *inheritable* capabilities by executing a program that gains capabilities either by having *effective* capabilities or by having *permitted* capabilities that have already been set *inheritable*. Programs that have *effective* capabilities may validate a user's authorization to use those capabilities, depending on whether or not the execution of the program could have an adverse impact on the security of the system. This mechanism permits the emulation of a fully privileged user by executing a program that has all capabilities *effective.*

B.25.6.2 A Program-based Capability Model

Instead of forcing every trusted application to perform user authorization checks, it is possible to create a single program that does so, and sets the *inheritable* flag of all capabilities authorized to a user. Program files in this style of implementation would have the *permitted* flags of all the capabilities they require for all their possible functions set. When executed, the program would receive only those capabilities actually authorized to the user, not necessarily the full set that they are capable of using. It is thus possible to provide a trusted shell or user interface program that will assign additional capabilities or disable existing capabilities associated with a user based upon the specific functions to be performed and then invoke one or more programs that are relieved of having to perform a user authorization check.

It is not possible for an executing program to acquire additional capability for itself through the execution of a more trusted program, i.e., through *exec*'ing a more trusted executable file, but only to create a new process image that is more trusted than it is. Since the new process image has, by definition, replaced the old process image, attempts to garner additional capability in this manner will fail.

B.25.7 Capability Worked Examples

This section illustrates the POSIX.1e Capability mechanism by providing both utility and function examples. Included are examples using the POSIX.2 chown utility and POSIX.1 *chown*() function, examples of capability unaware programs, and an illustration of how the capability mechanism defined in this standard can be used to execute shell scripts.

B.25.7.1 CHOWN()

To change the user ID of a file, the *chown*() function imposes the following restrictions:

- A process shall possess an effective user ID equal to the user ID of the file, or its effective capability set shall include the CAP_FOWNER capability.

- If the {_POSIX_CHOWN_RESTRICTED} option is in effect for the file, the process' effective capability set shall include the CAP_CHOWN capability. Thus, to change the user ID of the file, both the CAP_CHOWN and CAP_FOWNER capabilities may be required in the process' effective capability set. If the system implements the MAC option of this standard, the process may also require the CAP_MAC_WRITE capability in the process' effective capability set.

- If the file is a regular file, the set-user-ID (S_ISUID) and set-group-ID (S_ISGID) bits of the file mode shall be cleared upon successful return from *chown*(), unless the call is made by a process whose effective capability set includes the CAP_FSETID capability, in which case, it is implementation defined whether those bits are altered.

In examples 1 through 3 below, the *chown*() executable file is assigned, via the *cap_set_file*() function, an empty effective set, an inheritable capability set that includes:

- CAP_FOWNER
- CAP_CHOWN
- CAP_FSETID

and a permitted capability set with flags set to potentially allow bypassing of DAC and MAC restrictions imposed by the *chown*() function (See 25.2 for capability descriptions.)

- CAP_DAC_READ_SEARCH
- CAP_MAC_READ
- CAP_MAC_WRITE

EXAMPLE 1

If the *chown*() utility is executed by a process that possesses all the above capabilities in its inheritable capability set, then all of these capabilities are included in the resulting process's permitted capability set. When these capabilities are made effective, via the *cap_set_proc*() function, the process may change the user ID of the specified file without regard for mandatory and discretionary access restrictions, file ownership restrictions, or {_POSIX_CHOWN_RESTRICTED} restrictions. Alteration of the set-user-ID and set-group-ID bits of the file mode is implementation defined upon successful return from *chown*().

EXAMPLE 2

If the *chown*() utility is executed by a process that possesses no capabilities in its inheritable capability set, then the resulting process's permitted capability set will not contain the three required capabilities. Therefore, the resulting process shall not possess appropriate capabilities to override any of the *chown*() restrictions described above.

EXAMPLE 3

If the *chown*() utility is executed by a process that possesses only the CAP_CHOWN and CAP_FOWNER capabilities in its inheritable capability set, then the resulting process will possess the CAP_CHOWN and CAP_FOWNER capabilities in its permitted capability set. When these capabilities are made effective, via the *cap_set_proc*() function, the process may change the user ID of the file, regardless of the file's initial user ID, or value of {_POSIX_CHOWN_RESTRICTED}. However, this process must satisfy all mandatory and discretionary access requirements, and the set-user-ID and set-group-ID bits of the file mode shall be cleared upon successful return from *chown*().

EXAMPLE 4

In this example, the file capabilities are initialized as described for examples 1 through 3, above, except that the CAP_FSETID capability is removed from the chown executable file's inheritable capability set, and is assigned to the file's permitted capability set. The process resulting from execution of the chown utility will possess the CAP_FSETID capability as part of its permitted capability set, regardless of the contents of the exec'ing process's inheritable capability set. When the CAP_FSETID capability is made effective, via the *cap_set_proc*() function, alteration of the set-user-ID and set-group-ID bits of the file mode is implementation defined upon successful return from *chown*().

B.25.7.2 Capability Unaware Programs

In this section, we examine the behavior of capability unaware programs. This specification provides support for backwards compatibility of binary executables that depend on traditional UNIX set-user-ID behavior for proper operation. This specification also provides a mechanism for overriding capability on a per executable basis. Additionally, the *permitted* flag provides for a finer granularity of control to enable capabilities based on the *inheritable* flag of the exec'ing process. For all capability unaware programs that require capability, the program file's *effective* flag must be set. This is the only mechanism for enabling capabilities in the *effective* capability set upon execution.

EXAMPLE 1

Suppose an old version of the `mailx` program requires discretionary and mandatory override capabilities to operate correctly on a particular implementation. These capabilities can be enabled via the *effective* capability set regardless of the exec'ing process' *inheritable* capability set. This allows `mailx` to operate on a system supporting {_POSIX_CAP} without modifying the `mailx` source code.

If an administrator desires to control which capabilities become *effective* based on the exec'ing program's *inheritable* capabilities, then the *permitted* flag is used. The *inheritable* flag is ANDed with the *permitted* flag and this result is included in the new process' *effective* flag.

EXAMPLE 2

The `grep` program may have the CAP_DAC_READ_SEARCH capability enabled in the *permitted* capability set, which would then permit the invoker to access all files if and only if the CAP_DAC_READ_SEARCH *inheritable* capability was enabled in the exec'ing process. This would permit a trusted process to *exec* the `grep` program to locate a phrase in a file tree it normally would not have read access to.

B.25.7.3 Shell Script Execution

A shell script can be executed with capability using the capability mechanism defined in this standard. For example, a program stub can be created that can be invoked from the login shell that sets the inheritable capability attributes for those capabilities needed for shell script execution. The *system*() function can then be invoked to execute the shell script file. The capabilities set in the inheritable capability set are then passed through the shell executed by the *system*() function to the individual utilities constituting the shell script. The capabilities available to each utility are then determined by the *exec*() function as described in the capability mechanism.

B.25.7.4 Textual Representation of Capability States

The purpose of this clause is to specify a single, portable format for representing a capability state. This textual representation is intended for use by the *cap_to_text*() function and the `getcap` command to represent the state of an existing capability state object, and by the *cap_from_text*() function and the `setcap` command to translate a textual representation of a capability state into its internal form.

Examples of valid textual capability state specifications include:

No flags for any capabilities defined in the implementation are set:

```
"all="
```

```
"="
```

```
"CAP_CHOWN=
 CAP_DAC_OVERRIDE=
 ...
 <all remaining POSIX-defined capabilities>
 <implementation-defined capability>=
 <implementation-defined capability>=
 ...
 <all remaining implementation-defined capabilities>
"
```

Only the permitted flags for CAP_KILL, CAP_CHOWN, and CAP_DAC_OVERRIDE are set. The remaining flags for the remaining capabilities are all cleared:

```
"CAP_KILL,CAP_CHOWN,CAP_DAC_OVERRIDE=p"

"all=
 CAP_KILL=p CAP_CHOWN=+p-ei
 CAP_DAC_OVERRIDE=p"
```

The inheritable flag for every capability defined by the implementation is set except for the CAP_MAC_* capabilities. The effective flag is set for the CAP_DAC_OVERRIDE capability:

```
"all=i
 CAP_MAC_READ,CAP_MAC_WRITE,CAP_MAC_DOWNGRADE,CAP_MAC_LOCK
 CAP_DAC_OVERRIDE+e"
```

In order to promote the portability of capability state information between implementations, one representation must be specified in this standard. We chose to standardize the textual representation as this promotes not only application portability but user portability as well.

We considered an alternative representation that was flag `set´ oriented, i.e., something that would look like:

```
i=CAP_KILL,CAP_MAC_WRITE
p=all
...
```

however, this was rejected as implying a specific implementation (e.g., implementation of capability data objects as multiple set structures) and potentially being less compact (a privilege having all flags set must be named separately for each flag.) In addition, the requirement in such a representation to name a capability multiple times greatly increases the chances for human error when attempting to specify or interpret the representation.

In general, it is felt that this specification provides implementations with a wide degree of flexibility in how they can represent capability states, while ensuring that they can correctly interpret such states created on other interpretations with a minimum of difficulty and implementation complexity. The same state can be represented in a compact manner or a lengthy manner, depending on the purpose for which it is intended.

B.26 Mandatory Access Control

B.26.1 Goals

The primary goal of adding support for a Mandatory Access Control (MAC) mechanism in the POSIX.1 specification is to provide interfaces to mandatory security policies. A mandatory security policy is a system-enforced access control policy that is outside the control of unprivileged users. Additional goals included are to:

(1) address mandatory access controls that support appropriate, widely recognized criteria, while providing as much flexibility for implementation-specific MAC policies as is practical;

(2) define MAC interfaces for portable, trusted applications and specify MAC restrictions on all other POSIX.1 functions;

(3) preserve the provision for POSIX.1 conforming applications to impose

(4) preserve 100% compatibility to the base POSIX.1 functionality among subjects and objects operating under "single label conditions", i.e., all subjects and objects have an equivalent MAC label;

(5) add no new MAC-specific error messages to existing POSIX.1 and other interface standards, as doing so could interfere with the desire to avoid new covert channels.

The mandatory access control (MAC) interfaces are intended to be compatible with the mandatory access requirements of a number of criteria, particularly compatibility with the U.S. TCSEC levels B1-B3, the European ITSEC functionality levels FB1/FB3, and the U.S. CMW requirements for MAC. It should be noted that compatibility with these criteria extends only to the functionality defined in them, and not to the assurances they may require. Additionally, the interfaces were designed to conform with the requirements for adding "extended security controls" to POSIX-conforming systems, as stated in POSIX.1, section 2.3.1.

There is a recognition that the underlying mechanisms involved can be implemented in a number of different ways that still fulfill the POSIX_MAC requirements. Another consideration is the expectation that POSIX.1 conforming systems will wish to extend the functionality defined in this standard to meet particular, specialized needs. For these reasons, flexibility in the POSIX_MAC requirements while still conforming to the criteria mentioned above, is an important objective.

By defining POSIX.1e interfaces for MAC, it is possible to develop trusted applications which are portable across POSIX_MAC-compliant implementations. Identifying MAC restrictions for other POSIX.1e functions ensures that application developers are made aware of possible changes required for their applications to function in a POSIX_MAC-compliant environment.

MAC is intended to be complete, covering all means of information transmission. Hence for many interfaces (such as *stat*()) MAC read access is required even where ordinary ACL read access is not required in POSIX. This completeness should even cover areas which are not ordinarily regarded as information transmission channels (that is, "covert channels.") A complete analysis of covert channels available through the POSIX interfaces is beyond the scope of this document. Instead, only those cases which have policy implications are discussed here, although we have attempted to avoid introduction of any covert channels in the new interfaces defined by this standard. Hence additional controls needed on reading FIFOs are discussed, but means of controlling the covert channel provided by the process ID returned by *fork*() are not.

No new error codes for existing POSIX.1 interfaces are introduced to minimize the confusion for existing applications. While this confusion cannot be entirely eliminated (in particular because existing error codes can now be returned in situations which would not arise without MAC), avoiding new error values at least ensures existing applications will be able to report errors.

B.26.2 Scope

Section 26 defines and discusses the overall MAC policy and refinements of this overall restriction for the two major current policy areas: files and processes.

It should be noted that the policies in section 26 do not constitute a *formal security policy model* with proven assertions. It is, however, the minimal set of mandatory access restrictions that shall be defined, and serves as a basis for both the trusted interface, and the implementation-defined security policy model.

B.26.2.1 Downgrade and Upgrade

The definitions of downgrade and upgrade are the technically precise ones. They may not be intuitive because downgrade includes incomparable labels. For example, changing *Secret:A* to *Top_Secret:B* is a downgrade.

B.26.2.2 Concepts Not Included

Several concepts that will commonly be implemented by conforming systems have not been treated by this document, many because they have no basis in the POSIX standards upon which this document is currently based. These include:

Process Clearance: There were discussions that each process be given, in addition to its MAC label, a second label called its "clearance." The clearance would serve as an upper bound on certain MAC operations. For example, if the process could request to raise the MAC label of an object, the clearance might limit the label to which it could be raised. However, because there have been no concrete proposals for the process clearance (which should include expected circumstances under which it would be used), and since clearance is normally associated with a user, and users are not included in the base POSIX.1 standard, process clearance is not included in the current MAC proposal.

Range Restrictions: These include various sorts of system-wide, per-file system, per-user, and device MAC range restrictions.

The label testing function, *mac_valid*() is intended to help provide an interface to at least some of these restrictions in a more portable manner. For example, the restrictions may not be a simple range but a more complicated restriction.

System High/Low: A potential function that was rejected was one to return the current "system high" and "system low" labels. Some implementations may not have a simple high and low, but rather a more complex (flexible) notion of "system high and low," for example, a set of high/low ranges.

Devices: Access to devices through device special files is not treated in this document. Often implementations may have special device access rules based on device-specific considerations. Two common examples of such special device access rules are device "ranges" (sets of allowed MAC labels for accessing certain devices), and "public," generally-accessible devices, such as `/dev/null` and `/dev/tty`. Since such device-specific considerations have no basis in POSIX.1, devices as a whole are not addressed in this document.

File Systems: Mounted file systems are not included.

Trusted User Commands:
Commands for both administrators and trusted or partially trusted users have not been included.

Label Translation: POSIX.1 does not address networked systems. Thus, the issue of translating MAC labels into a portable form is not addressed in this standard.

Process Label Functions:

The functions provided as part of this standard to retrieve or set the MAC label associated with a process are limited to the requesting process. That is, no interface is provided whereby a process may specify another process (for example, using a process id) to be the target of the *mac_set_proc*() or *mac_get_proc*() functions. Such mechanisms have been omitted in order to be consistent with the POSIX.1 standard which provides no facilities for processes to manipulate, or be cognizant of, other processes' state information. Note, however, that conforming implementations may choose to provide such functions.

B.26.3 File Object Model

An important part of mandatory access control for files is the seemingly simple assumption that the file attributes and data comprise a logically single data container to which the file MAC label is applied—the "file object." Virtually all MAC function restrictions arise from applying the following two basic policy rules under this assumption:

(FP.1) The MAC label of a file must be *dominated* by the MAC label of a process for the process to read the data or attributes of a file, and

(FP.2) The MAC label of a file must *dominate* the MAC label of a process for the process to write the data or attributes of a file. (Allowed restrictions on this rule are discussed following in *Direct Write-up*).

For example, linking to a file involves altering the link count of that file, and hence MAC write access to the file is required (as well as appropriate restrictions on the directory in which the link is created). This is discussed below in the *link*() example.

MAC restrictions for virtually all file-related functions can be straightforwardly derived from these basic policy assumptions. (See the Policy section for a complete list.)

Two examples:

mkdir()

The *mkdir*() function is used to create a directory, **D.** Apart from actually creating the directory itself, a link name must be placed in the specified parent directory, **PD.** Application of the **FP.1** and **FP.2** yields the MAC restrictions:

(1) The process shall have search access to **PD.** (Search access is an outgrowth of **FP.1.)**

(2) In order to add the link name, the process shall have MAC write access to **PD,** i.e., the MAC label of the process shall be dominated by that of the directory (from **FP.2).**

D is created with the MAC label of the process **(FP.4),** and hence it is correct to leave the file open to the process.

Note that the calls *creat*(), *mkfifo*(), and *open-for-create* are other functions that create files and will have these same MAC restrictions.

link()

The *link*() function is a little more complicated. A new link is to be created in a directory **D** to an existing file **F.** This involves writing the new link name to **F** into **D.** Hence the following MAC rules are applied:

(1) The process shall have search access to the directory **D.**

(2) The process shall also have search access to the file **F,** because the function is implicitly testing for the existence of **F.**

(3) The process shall have MAC write access to **D,** i.e., under **FP.2** the MAC label of the process shall be dominated by that of **D.**

In making a new link to **F,** the link count of **F** must be increased. Hence, the process is implicitly writing into **F,** and:

(4) The process shall have MAC write access to **F,** under **FP.2.**

B.26.4 Direct Write-up

Originally, FP.2 dictated that a process can only open files for writing whose label equals that of the process (“write-at-label”), but that, a POSIX.1e conforming implementation could allow write access under relaxed conditions, in particular, when the MAC label of the file properly dominates that of the process. Because POSIX.1 mandates that additional conditions can only be more restrictive, this was changed to write-up, with write-equal allowed as part of a fully conforming implementation.

The usefulness of allowing open-for-write of higher-label files (“direct write-up”) seemed too small given potential implementation difficulties. For this reason, direct write-up was not required by the standard. However, direct write-up may be a useful feature for the vendor willing to address its implementation problems, and for this reason, along with the reason cited above, the change was made.

Implementations which implement direct write-up will need to consider the impact on return codes and potential covert channels.

Note that the creator of a portable application cannot assume such relaxations are present because they are not required by the standard. Write-at-label must instead be assumed as the rule for MAC write.

In the following discussions, it is generally assumed that write-at-label is the case.

The option of creating objects with MAC labels dominating that of the creating process is allowed, but interfaces to do so are not provided. This facility would be effected by the same set of concerns expressed with regard to direct write-up, hence the more conservative approach. Furthermore, providing an interface for creating an object with a MAC label begs the question of why we don't provide a mechanism for an ACL and a capability set.

B.26.5 Protection of Link Names

As discussed above, in POSIX.1 there really is no such thing as a "filename." This is true both logically and physically, i.e., no name is stored in the file itself. Instead there are only link names to files that are both logically and physically data items within the parent directory.

This proposal takes the most direct interpretation of the protection of link names within a directory: the link names are simply considered data in the directory. This means that the names are protected by the MAC label of the directory that contains them, even when they indicate files or directories at other MAC labels.

A process could determine the link name and hence existence of objects at labels not dominated by the process. However, this cannot be used as a covert channel because the process that defined those names must have had write access to the containing directory, which means that its label equals (or, in some implementations, was dominated by) the label of the directory. More precisely, the covert channel "sender" that creates the link name must be equal to (or, in some implementations, dominated by) the MAC label of the directory, and the "reader" must dominate that label. Hence, because information is at most going to a higher MAC label there is no covert channel.

Since link names may be protected at a lower MAC label than the file to which they point, the user must be careful to choose a name that is adequately protected at the MAC label of the parent directory.

This interpretation is both natural and common for UNIX file systems and underscores that link ("file") names are not a property of the file, but rather of a parent directory.

One of the suggested alternatives is the so-called "name-hiding" model where each link in a directory is considered an object labeled at the label of the file to which it links. This alternative was rejected because it is more complex, and doesn't offer any real improvement over the alternative that was accepted. Access to the link names in a directory must therefore be controlled on a per-link basis.

B.26.6 Pathname Search Access

Files are commonly referenced by a pathname, for example **A/B/F.** If the pathname starts with the "/" character, then the pathname starts at the absolute root of the file system. Otherwise it starts at the current working directory of the process. Even pathnames that contain only one name, e.g., **F,** are still pathnames. Each such reference requires an implicit reading of a sequence of directories, and **FP.1** must be applied to this process. This is called *search access* in this document.

A pathname consists of a sequence of *link* names (**A, B,** and **F** in the previous example) i.e., each name in the pathname is a link name contained in some directory. In other words, the name that most users commonly assume is "attached to" a file is actually the name of a link in a directory, where the link only points to the actual file. There may be many such links with different names to a single file.

In locating a file on behalf of the process, the system is in effect opening the sequence of directories that contain the link names in the pathname, finding (reading) the next link name, and proceeding to the next file or directory named by that link. The following basic constraint is required under **FP.1:**

MAC Search Access

> In order for the system to perform this implicit reading of the directories in the pathname, the process is required to have MAC read access to each directory that contains a link name of the pathname. Specifically, the MAC label of the process must dominate that of the directory. (Note that ACL execute "x" permission is also required, as in standard POSIX.)

For relative pathnames, the current working directory (".") is considered the first, implicit directory in the pathname and is checked first. For absolute pathnames, the absolute root directory is checked first, and, because it is customarily at the lowest MAC label on the system, search access will always proceed from absolute root.

Note that the last element in the pathname is the final link name. Once this final link name is read from the directory in which it resides, search access is considered complete. Hence, by definition the final target element (**F** in the current example) is not itself checked for any MAC access during search access, although it will certainly be checked in the context of specific operations.

Basically, MAC search access determines whether a process can detect the existence of a file, specifically, whether the process can read a directory containing a link to the file.

As a general rule, MAC search access is applied to all pathnames presented in a function. If this succeeds, then other MAC checks follow.

B.26.7 Check-Access-on-Open Only

The MAC policies follow the standing POSIX.1 metaphor that access to the data portion of a file object is checked only when access is requested and not for each data read and write. Subsequently, access to the file is not revoked or changed in mode until the process willingly closes the file.

With this form of access, it is important that the MAC label of a file object not be altered if the alteration would allow information flow to a subject which would have not occurred at the new label. This requirement was originally stated in a **FP.5,** but this was removed when it was pointed out that **FP.5** is really just saying you can not violate **FP.1** or **FP.2.**

There are some conditions (which are rejected in this document) where the label could technically be allowed to change:

— When the file references were write-only and the label was being raised. However, this seems a relatively rare case.

— When the system supported some type of access revocation or recalculation.

— Allow changing the label only when the requesting process is currently referring to the file.

— If all processes currently referencing the file were appropriately privileged, then the label might be allowed to change. The danger here is that the privileged processes may not be aware of the label change.

The application of **FP.1** and **FP.2** to the *mac_set_file*() and *mac_set_fd*() functions takes the simple approach and make the handling implementation-defined as to whether changing of the file label when there are open connections to the file, (other than the calling process in the case of *mac_set_fd*()), are disallowed, even when the processes are privileged, or whether revocation is performed.

B.26.8 Creating Upgraded Directories

An upgraded directory is one whose MAC label properly dominates that of its parent directory.

While in general the operation of **FP.2** and **FP.4** do not allow unprivileged processes to create files or directories at other than the process MAC level, some means of creating multi-label file trees is necessary.

In particular, the ability to create upgraded directories gives a convenient means for organizing a multi-label file tree appropriately, and need not violate any fundamental security constraints. Hence it is appropriate to provide unprivileged processes with some means of doing so; though it has been chosen not to do so as part of this standard.

B.26.9 Objects without MAC labels

This standard specifies that each file will always have a MAC label associated with the file, but does not require each file to have its own unique MAC label.

Originally, the provided MAC functions allowed for returning [ENOSYS] if {_POSIX_MAC} was defined and the specified file did not have its own MAC label. This was subsequently changed because of objections to the overloading of [ENOSYS] to return [ENOTSUP] for the cases where a file does not have its own MAC label.

A *pathconf*() variable {_POSIX_MAC_PRESENT} is provided to allow applications to determine if a file has its own MAC label. This standard does not specify the specific situations where a file does not have its own MAC label. Examples of possible situations are: read only file systems; pre-existing file systems with insufficient space to insert MAC labels; and certain devices such as /dev/null. The *mac_get_file*() and *mac_get_fd*() functions will always return a MAC label because each file will always have a MAC label associated with the file. The *mac_set_file*() and *mac_set_fd*() functions can return [ENOTSUP] if the specified file does not have its own unique MAC label but shares the MAC label of a file system.

B.26.10 Error Return Values

The MAC functions specified in this standard may return one of several errors depending on how the implementation has addressed MAC labeling.

If the symbol {_POSIX_MAC} is defined, then the implementation supports the MAC option and is required to support the MAC functions as described in this standard. If the symbol {_POSIX_MAC} is not defined, then the implementation does not claim conformance to the MAC option and the results of an application calling any of the MAC functions are not specified within this standard. An alternative is for the MAC functions to specify that the error return code [ENOSYS] be returned by the functions if the MAC option is not supported. However, in order to remain compliant with the policies of POSIX.1, this standard cannot specify any requirements for implementations that do not support the option.

The error [ENOTSUP] shall be returned in those cases where the system supports MAC but the particular operation cannot be applied because restrictions imposed by the implementation. For example, if an application attempts to set the MAC label on a file on a system where *sysconf*() indicates that an MAC is supported by the system, but the value that *pathconf*() returns for {_POSIX_MAC_PRESENT} for that file indicates that individual MAC labels are not supported on that file, the application shall receive the [ENOTSUP] error. Therefore, if an application attempts to set the MAC label on a file, it is the application's responsibility to first use *pathconf*() to determine whether the implementation supports MAC labels on that file.

It should be noted that, in general, this standard attempts to avoid adding and defining new errors. However, in the case of [ENOTSUP], the following points were noted: First, the need exists to provide feedback to applications concerning a new error condition. Second, while it is possible to use an existing error code in such cases (for example, ENOSYS), the group felt that this would overload those errors. P1003.1, when consulted, concurred with this view and agreed that the creation of a new error code, in this case, was appropriate. Third, the error [ENOTSUP] is also being used by P1003.4 for roughly the same reasons. Therefore, the consensus of several POSIX working groups is that while adding new errors is generally not recommended, that this case warrants the creation of a new error and that the new error should be [ENOTSUP].

The [EINVAL] error is returned by functions when the MAC label specified in the function call is syntactically incorrect or the MAC label is not permitted on the system because implementation-defined restrictions, (e.g., range restrictions). That is, this error is used to indicate the invalidity of the MAC label specified, independent of whether the operation would have succeeded had it been a valid label.

Although POSIX.1 does not specify precedence for error return values, careful consideration should be given to this matter in the security standard to ensure that covert channel considerations are adequately addressed. Specifically, if an unprivileged application attempts a function for which privileges are required and the implementation returns the EINVAL error in favor of the EPERM error, it may be possible for the application to determine the system's MAC label range restrictions based on whether EINVAL is returned (indicating the label is outside the system's range), or EPERM is returned (indicating the label is valid for the system, but that the application failed the privilege check). Therefore, despite this standard's silence on the issue, it is recommended that when a function could return multiple errors in a particular instance, that the errors be given the following precedence (from most favored to least favored): EPERM, EINVAL, ENOTSUP.

B.26.11 Valid MAC Labels

MAC labels have two forms: internal and external.

The basic MAC label structure defined in this standard *(mac_t)* is a pointer to an opaque data structure. The binary format of that opaque data structure may include such data as a hierarchical classification and non-hierarchical categories. The standard makes no assumptions regarding the underlying representation other than imposing the following constraint: the structure must be an exportable object. That is, the structure is opaque, persistent, and self-contained. The structure can therefore be copied by duplicating the bytes without knowledge of its syntax. Such a copy can be changed without any effect on the original, and the original can be changed without any effect on the copy.

The external format of a label is a text string of undetermined format. Any separator character between fields in the textual representation is implementation-defined. As noted in POSIX.1 section B.2.3.5, the character set used for textual representation of MAC labels is not defined by this standard.

The meaning of a valid MAC label is implementation-defined, as described in *mac_valid*(). A MAC label could be invalid for many reasons, such as:

A. It is malformed, e.g., the label contains a checksum in the opaque type which does not agree with the checksum calculated from the data.

B. It is out of the security level range of the system, e.g., the label refers to a classification or category or combination which is outside the set of valid MAC labels for the system.

C. It is out of the security level range of a process, e.g., the label refers to a classification or category or combination which is outside the set of valid MAC labels for a process.

D. It is outside the representation range, e.g., a system could allow no more than n categories from a universe of m, even though each of the m categories is valid.

Invalid MAC labels may appear for a number of reasons. Examples include: constructing a MAC label in process memory without regard to semantics of the bits, importing a MAC label from a dissimilar system, reading a MAC label previously stored in a file, etc. Note, however, that none of the MAC interfaces defined in this standard will ever return an invalid MAC label.

The *mac_valid*() function is the means for an implementation to communicate to a portable application that the application should not "deal with" certain MAC labels—that they are undefined, disallowed, or some implementation-restricted state. Note however that an implementation may impose additional restrictions on the MAC labels for a particular object or process beyond the system-wide constraints that are addressed by *mac_valid*().

B.26.12 Modification of MAC labels

Unlike some of the other features in this standard, the basic unit of data for mandatory access control (the MAC label) is not usually manipulated. Interfaces and a memory management model to support manipulation of MAC labels were deemed inappropriate, except for the least upper and greatest lower bounds functions discussed below.

B.26.13 Least upper bounds and greatest lower bounds

The function *mac_glb*() is useful for applications that wish to limit their activities to those permitted by both labels. For example, if a user wants to know the maximum classification of data that the user can transmit via a network cleared for MAC label *labelA* to a machine cleared for MAC label *labelB*. Likewise, the *mac_lub*() function allows applications to determine a MAC label which dominates two specified labels.

It is the intent that conforming applications only use these functions, rather than more primitive manipulation of the label structures themselves.

B.26.14 Functions returning MAC labels

Functions which return MAC labels should use a common implementation specific allocation mechanism. For example, *mac_get_file*() allocates space for a MAC label, fills in the MAC label from the requested file system object, and returns a pointer to this space to the caller. The system allocates space because a MAC label could be of variable length in some implementations. Such systems include those which use a sparse matrix representation. If the system did not allocate the space a portable application would have to query the system about the size of a (subject's or object's) MAC label, reserve space for the label, and then call another function to obtain the MAC label. The overhead for systems with a fixed length MAC label is excessive. The use of additional level of indirection in the present interfaces accommodates systems with both fixed and variable sized labels with reasonable efficiency.

The use of an allocator implies the use of a deallocator. The function *mac_free*() frees the storage space allocated by any MAC function which allocated a MAC label.

A function to allow for the translation of an internal label to an alternative external label format was considered and rejected. For example, it is anticipated that some trusted applications will wish to display a short form of the MAC label on a display terminal, perhaps as part of an icon, rather than the entire (possibly very lengthy) external text form. An option considered was to alter the *mac_to_text*() function to include a form argument. Trusted applications could specify the external form of the label desired, e.g., icon, abbreviated, long. The proposal was rejected because the TCSEC, ITSEC, and CMW requirements criteria do not specify alternative external formats. Thus, most implementations do not provide for alternative text labels.

B.26.15 Multi-level directories

Interfaces to create, remove, and scan multi-level directories were considered and actually appeared in earlier drafts, but were removed because a lack of consensus and ballot objections. The basic reason for a multi-level directory mechanism is that certain portions of the filesystem namespace are "well known" and need to be publicly available. The most obvious example is `/tmp`; many applications expect to be able to create files within this directory. However, in a system with MAC, allowing applications at any level to freely create visible files in `/tmp` would be an unacceptable security hole; it allows a trivial means for a Trojan horse program to make great quantities of data visible at lower levels (by encoding the data in file names).

Data at a MAC label higher than that of the multi-level directory may be stored in the multi-level directory by an unprivileged user. However, access to this data will still be governed by the MAC policy.

B.26.15.1 Underlying Mechanism

To overcome this problem, while still allowing applications free access to well known directories, some means of hiding parts of the file system name space is needed. The most direct method, what has been called a "true" multi-level directory, is to implement a new directory structure which allows entries to be truly hidden. Here, for example, *readdir*() would only return entries at the requester's MAC level or lower. While conceptually nice, this is hard to implement properly. For example, compatibility and prevention of a covert channel require lower level processes (at least) to be able to create entries with the same names as pre-existing ones created by higher-level processes. To avoid confusion, the appearance of these names then needs to be altered somehow (for example, by appending a representation of the label) for reference by higher-level processes. To avoid other channels, the apparent size of the directory may need to be altered to prevent visibility of creating and deleting files which might cause the size of the directory to change.

A simpler implementation uses the separation already provided by subdirectories to achieve the goal. References to pathnames such as `/tmp/foo` are "redirected" during pathname resolution to "hidden" subdirectories of `/tmp`, usually to something like

`/tmp/`*LabelRepresentation*/foo

Here, *LabelRepresentation* tends to be a base 64 or hex representation of the binary form of the label. These hidden subdirectories must of course be created somehow, presumably either beforehand by a trusted program or administrator, or as needed by the system.

B.26.15.2 Getting Around The Hiding

Both mechanisms hide part of the file system namespace from applications. There are times when this is not desirable, e.g. when backing up filesystems, or when a user simply wants to get at a lower level file. This is especially pressing with the subdirectory approach, which conceals lower level files just as well as higher level ones. Hence some means of generating a reference to an otherwise invisible object is needed.

Again, two basic approaches have been taken. Either the reference is generated directly by some special pathname:

`/tmp/DON'T*DO*REDIRECTION!!/`*LabelRepresentation*/foo

or it is generated indirectly by setting some process mode which allows using the "real" filename

`/tmp/`*LabelRepresentation*/foo

The "modal" methods are less flexible in allowing redirected and real representations to be mixed, although some of this can be ameliorated by having multiple modes such as

redirect none
redirect "system" directories (`/tmp, /usr/tmp`) only
redirect both system and application (`/usr/spool/mail`, etc.)
directories)

Their interaction with things like symbolic links involves difficulties as well. (Allowing a symbolic link to a file in a hidden directory requires some means of specifying the mode in the symbolic link.)

The "non-modal" special pathname method has the disadvantage of reserving part of the file name space, something which unfortunately there is no precedent for in historical implementations. If the portion reserved, e.g., the pathname component

`DON'T*DO*REDIRECTION!!`

in the (fictitious) implementation above, were not standardized, a portable application would have to abide by every namespace restriction imposed by every implementation.

Finally, there are ways to address these issues without changing the way directories are processed at all. One such mechanism is the "variable symlink", in which a component of the user's environment is use to replace a specified pathname component in the symlink. Thus, if the symlink /tmp contained "/orary/MACLABEL", a process with the environment variable MACLABEL set to "secret" would be directed to "/orary/secret". Other mechanisms, such as an exotic file system type, are also possible.

B.26.16 The Directory Model

The relationships between the MAC label of a directory and its subdirectories and files is often referred to as the "directory model." One of the more common models for POSIX-like systems is for files to equal and for directories to dominate the label of their parent directories. This is sometimes called the "non-decreasing directory" model because MAC labels at most increase as one transverses from the root of a directory tree to its leaves. Multics, for example, used this model.

The following discussion applies only when untrusted processes are allowed to create upgraded directories under one of the schemes above.

This proposal does not absolutely impose the non-decreasing directory model. Neither does it prevent conforming implementations from imposing a non-decreasing restriction. However, the application of the basic MAC restrictions on the processes for accessing and creating the files as simple, labeled data containers leads to the restriction that unprivileged processes (users) can only create non-decreasing directory trees. Privileged processes are not bound by these restrictions and can create files and directories at arbitrary MAC labels.

Implicit in the preceding discussion on upgraded directories is the assumption that trees created by unprivileged processes will be non-decreasing.

The non-decreasing nature of file trees combined with the minor user difficulties of creating upgraded directories (changing login sessions) will tend to group directories according to MAC label. That is, instead of highly intermixed files and directories at various MAC labels, they will tend to be segregated according to MAC label. This is generally a good practice anyway, because the close intermingling of file system elements at different labels tends to be a breeding ground for covert channels and confusion.

Basically, this proposal takes the position that non-decreasing hierarchies are appropriate for unprivileged processes, but that POSIX.1e should not so restrict appropriately privileged processes.

B.26.17 File Tranquillity

The original **FP.5** dealt with file object tranquillity. (Note, this rule was removed as an explicit rule when it was pointed out that it is just a restatement of **FP.1** and **FP.2.)**

FP.5: The MAC label of an object cannot be changed to a new MAC label if the change would allow information flow between a process and an open file object which could not have occurred at the new MAC label.

There are two general ways that a conforming implementation could enforce the file change-level constraint:

Tranquillity

The change request could be denied if there were any open connections to the file (other than the requesting process in the case of the *mac_set_fd*() function).

Readjustment

The change request could be fulfilled if it could be determined that all open connections could have been made in the mode requested after the label was changed. The implementation could either preemptively close the newly-disallowed connections, or attempt to readjust the current access modes of the open connections.

Readjustment can be difficult to implement and is not required by the standard, but is also not precluded by the standard. Since readjustment is not required, this leaves strict tranquillity as the lowest common denominator of conforming implementations. For this reason, portable applications must assume no more than strict tranquillity for maximum portability under the standard.

B.26.18 Process Tranquillity

Requirements for "process tranquillity" do not exist because any process privileged to change its own label is presumed to ensure it does not subsequently cause undesired information flows.

B.26.19 Unnamed Pipes

Unnamed pipes are considered labeled objects. However, because they are not addressable, i.e., cannot be opened, and because MAC is enforced only when objects are opened for access, there are never any actual MAC checks against the label of the pipe. The label will however need to be retrieved in the *mac_get_fd*() function.

The primary rationale for labeling unnamed pipes is so that processes using *mac_get_fd*() (who may not know whether the file descriptor is a pipe) will not see anomalous behavior for pipes.

B.26.20 FIFOs

First-in-first-out (FIFO) data objects have an inherent covert channel in that higher-label readers can affect the state of the object in a manner that can be detected by other (lower-label) readers/writers. For example, a reader/writer at $\mathbf{L_1}$ can write sequences to the FIFO and then determine how much data has been read by a reader at $\mathbf{L_2}$ by reading the FIFO (where $\mathbf{L_2}$ is not dominated by $\mathbf{L_1}$). This constitutes information flow in that is contrary to the basic MAC policy **FP.3**.

FIFOs in POSIX.1 include only FIFO-special files. In order to control the covert channels for these FIFO-special files, the following rule is imposed:

> Unprivileged processes may open FIFO-special files for reading only if the process also has MAC write access to the FIFO, i.e., the process is at the same MAC label as the FIFO-special file.

Hence, unprivileged processes at different MAC labels may not obtain a FIFO between them even if opened such that information may only flow in accordance with **P.**

B.26.21 Inclusion of *mac_set_fd()*

Originally, this function was not included. It was felt that there was too little demonstrated need for the function against potential implementation difficulties. The only mentioned use was by `login`.

One notable implementation difficulty is that it is difficult to find the parent directory (or directories) of a file given only a file descriptor. This makes it difficult for implementations that wish to absolutely enforce the relationship between a file and its parent directory. (Note that the issue of unique parent directory is side-stepped when a pathname is given in that the directory given in the pathname is the one to which various mandatory access controls are applied.)

However, in the interest of consistency with the other POSIX.1e options, it was decided to include the *mac_set_fd*() function.

B.26.22 Inclusion of *mac_size()*

The *mac_size*() function has been provided to allow applications to obtain the size of a MAC label. Applications need to know the size of MAC labels only if they are going to store the MAC label. There is no reason to know the size to use the provided MAC functions. An example of using the *mac_size*() function is a data base system which needs to store a MAC label for each record. It would use the *mac_size*() function to find out the size of the space to allocate and then could byte copy the MAC label to the data base record.

B.26.23 Restrictions on Signals

The following, minimal MAC restriction governs the sending of signals:

> An unprivileged process cannot send signals to another unprivileged process when the signals would result in actions other than an upgrading of information, i.e., the signal is only allowed when the label of the receiver dominates that of the sender.

The general philosophy is to prohibit only those signals that can be repeatedly sent thus causing high-bandwidth covert channels. This affects mainly the *kill*() function.

No additional restrictions are imposed between two processes at the same label or when at least one of the processes is privileged.

B.26.24 Alteration of atime

Many functions require that the file *atime* be marked for update. However, the case where the actions of a process could affect the *atime* of a file whose label does not dominate that of the process presents a potential covert channel. Some implementations can adjust when the *atime* is actually set and thus adequately confine such covert channels, but this is not required by the standard. Instead, the effect on *atime* in such cases is implementation-defined.

B.26.25 Multi-Label Untrusted Process Hierarchies

There are situations where untrusted processes at different MAC labels can have an ancestral relationship. Processes with an ancestral relationship have special opportunities for communicating information, e.g., *wait*, *waitpid* of POSIX.1 section 3.2.1, and when both processes are untrusted and at different MAC labels these opportunities present potential covert channels. There are no MAC restrictions for at least some of the following reasons:

— These situations can only be set up by trusted processes who change their MAC label. It is assumed that a trusted process who changes its label and creates (by *fork*() or *exec*()) untrusted processes will take actions to confine potential covert channels.

— The channels are typically low-bandwidth.

— Restricting all such operations seems like too much imposition for too little gain.

B.26.26 File Status Queries

Following the precedence of IEEE Std 1003.1-1990, no DAC access is required to determine the various status attributes of a file (DAC information, labels, owner, etc.) including all new attributes, such as the MAC label. However, MAC read access is required to prevent potential covert channels.

B.27 Information Labeling

B.27.1 Goals

The primary goal of adding support for an information labeling mechanism in the POSIX.1 specification is to provide interfaces to non-access control related data labeling policies. An information labeling policy, unlike access control related policies (such as mandatory or discretionary access control), provides a means for associating security-relevant information with the data maintained by the system. More specifically, the information labeling mechanism's goals are to:

(1) Address the need for non-access control related mechanisms to implement data labeling policies as specified in existing standards and criteria while providing as much flexibility for implementation-specific information labeling policies as is practical. Specifically, to allow for the variances between existing standards, the interfaces are intended to provide the latitude for implementations to support multiple information label uses. For example: to allow information labels to be applied to subjects and objects by the system, and altered by the system, to record the flow of data between subjects and objects, or to allow information labels to be applied to objects by users, and altered by them on a discretionary basis, to record handling restrictions on the object contents.

(2) The information label interfaces are intended to be compatible with the information label requirements of a number of standards and criteria. In particular, goals include compatibility with the U.S. Compartmented Mode Workstation Information Label requirements, and the European vendor and customer demands, along with DIA document DDS-2600-5502-87 and DIA document DDS-2600-6243-91. Finally, the interfaces were designed to conform with the requirements for adding "extended security controls" to POSIX-conforming systems, as stated in section 2.3.1 of POSIX.1.

There is a recognition that the underlying mechanisms involved can be implemented in a number of different ways that still fulfill the POSIX_INF requirements. Another consideration is the expectation that POSIX.1 conforming systems will wish to extend the functionality defined in this standard to meet particular, specialized needs. For these reasons, flexibility in the POSIX_INF requirements while still conforming to the criteria mentioned above, is an important objective.

(3) Define information labeling interfaces for conforming applications. By so doing, it becomes possible to develop trusted applications which are portable across POSIX_INF-compliant implementations.

(4) Specify information labeling enhancements on other POSIX.1 functions as necessary. Identifying information labeling modifications to other POSIX.1 functions ensures that application developers are made aware of possible changes required for their applications to function in a POSIX_INF-compliant environment.

(5) Address information labeling-related aspects of all forms of data access and transmission visible through the POSIX.1 interfaces. (Please note the distinction made between data and control information, clarified later in this section.) The interface, however, is designed for flexibility: the standard defines the *minimum* functionality that must be provided. Naturally, conforming implementations may choose to perform information labeling on objects, or at times, not required by this standard.

(6) Preserve 100% compatibility with the base POSIX.1 functionality. That is, it is undesirable to require new restrictions on the operation of existing POSIX.1 interfaces, or to require changes to the syntax of existing POSIX interfaces.

(7) Add no new information labeling-specific error messages to existing POSIX.1 interfaces and thus minimize the potential for confusing existing applications. While this potential for confusion cannot be entirely eliminated (in particular because existing error codes can now be returned in situations which would not arise without information labeling present), avoiding new error values at least ensures existing applications will be able to report errors.

B.27.2 Scope

This section examines the information labeling interfaces provided by this standard and explains the overall motivation for including the information labeling interfaces. Rationale and design tradeoffs are presented for the key information label interfaces.

This standard supports a security policy of nondisclosure, primarily through the interfaces defined for discretionary and mandatory access control. In particular, mandatory access control mechanisms implemented using the defined interfaces are expected to conform with the overall intent established in the security standards to which they are targeted. These security standards, (e.g., the TCSEC), normally require policies and mechanisms that protect objects at the level of the most sensitive data that they can contain. Often, however, the *data* contained in objects is actually much less sensitive than indicated by the mandatory access control label associated with that object. In addition, many security policies require that certain non-mandatory access control related information be associated with subjects and objects. Thus, in addition to mandatory access control labeling, this standard provides optional interfaces for data labeling. Use of these interfaces by conforming implementations permit support for a variety of data labeling policies.

B.27.3 Concepts Not Included

Several concepts that will commonly be implemented by conforming systems have not been treated by this document, many because they have no basis in the POSIX standards upon which this document is currently based. These include:

Label Translation: POSIX.1 does not address networked systems. Thus, the translation of information labels into an exportable form is not addressed in this standard.

Process Label Functions: The functions provided as part of this standard to retrieve or set the information label associated with a process are limited to the requesting process. That is, no interface is provided whereby a process may specify another process (for example, using a process id) to be the target of the *inf_get_proc*() or *inf_set_proc*() functions. Such mechanisms have been omitted in order to be consistent with the POSIX.1 standard which provides no facilities for processes to manipulate, or be cognizant of, other processes' state information. Note, however, that conforming implementations may choose to provide such functions.

B.27.4 Data Labeling Policies

There are many instances when security-related information should be associated with subjects and objects even though that information may not, in general, be used for mandatory access control. Such information may include markings that indicate the source of some data, what the data is about, the "trustworthiness" of the data, or anything else about the data other than how it should be protected. This non-mandatory access control related information is represented in an information label that should be associated with data when it is printed or otherwise exported. This specification provides functions to assign initial information labels, combine two information labels, and manipulate information labels.

A sample non-mandatory access control data labeling policy might be one targeted at virus detection. For example, under this policy, programs downloaded from a public bulletin board might be labeled with the marking "suspect-file." If the program contained a virus, and if the *inf_float*() function (discussed below) implemented the Compartmented Mode Workstation (also discussed below) style of floating labels, then it would be easy to track the spread of any infection throughout the system because every file infected by the virus would automatically be stamped with the "suspect-file" marking.

Other examples of non-mandatory access control information that should be associated with data include handling caveats, warning notices, discretionary access control advisories, and release markings. The ability to implement standards-based systems that support these and other non-mandatory access control markings is of great interest to many vendors and users.

One example of existing non-mandatory access control policies this interface is intended to support are those proposed by the European trusted system vendor community. The functionality necessary is that users must be allowed to apply data labels to subjects and objects, and alter them on a discretionary basis, in order to record handling restrictions on the objects' contents.

To provide a data labeling interface that can easily support the existing multiple data labeling policies, the information label interfaces have been carefully generalized to provide a mechanism to support these policies, without attempting to enforce the specifics of any particular policy. The burden of implementing specific policies is left to conforming implementations.

B.27.4.1 General Information Label Policy

Section 27.1.2 of this standard defines a general information labeling policy capable of supporting multiple particular data labeling policies. The information label policy statement consists of:

(1) A broad policy statement

(2) Refinements of this policy for the two major current policy areas: files and processes.

It should be noted that the policies in this section do not constitute a *formal security policy model* with proven assertions. It is, however, the most fundamental set of information label policies that should be defined. The general information label policy is as follows.

> *Information Label Policy*: Each subject (process) and each object that contains data (as opposed to control information) shall have as an attribute an information label at all times.

Information labels are said to "float" as data from one object is introduced to another object. The general information label floating policy is intentionally flexible and can be stated as follows:

> *Information Label Floating Policy:* The implementation-defined policy that determines to what degree information labels associated with data are automatically adjusted as data flows through the system.

The information label float policy is embodied by the *inf_float*() function. This function computes a new information label that is the combination of two information labels passed as arguments. As noted above, the new information label is calculated according to implementation-defined policies.

Note that the information label policy as applied to process functions specifies (in **PI.2**) that when a process with an information label *inf_p1* executes a file with information label *inf_p2*, the information label of the process shall be set to the value returned by *inf_float(inf_p1, inf_p2)*. However, in implementations where the new file executed completely overlays the process' address space, i.e., there is no data transfer from the originally executing process to the newly executing process, the information label of the process after executing the file may be set to *inf_p2*. The central factor in determining whether such an implementation conforms to the information label policy is whether data is transferred: the transfer of control information (such as process id, and various user ids) is inevitable and permissible; the transfer of data is unacceptable.

B.27.4.2 Error Return Values

The information labeling functions specified in this standard may return one of several errors depending on how the implementation has addressed information labeling.

If the symbol {_POSIX_INF} is defined, then the implementation supports the information label option and is required to support the information label functions as described in this standard. If the symbol {_POSIX_INF} is not defined, then the implementation does not claim conformance to the information label option and the results of an application calling any of the information label functions are not specified within this standard. An alternative is for the information label functions to specify that the error return code [ENOSYS] be returned by the functions if the information label option is not supported. However, in order to remain compliant with the policies of POSIX.1, this standard cannot specify any requirements for implementations that do not support the option.

The error [ENOTSUP] shall be returned in those cases where the system supports the information label facility but the particular information label operation cannot be applied because restrictions imposed by the implementation. For example, if an application attempts to set the information label on a file on a system where *sysconf*() indicates that an information label facility is supported by the system, but the value that *pathconf*() returns for {_POSIX_INF_PRESENT} for that file indicates that information labels are not supported on that file, the application shall receive the [ENOTSUP] error. Therefore, if an application attempts to set the information label on a file, it is the application's responsibility to first use *pathconf*() to determine whether the implementation supports information labels on that file.

It should be noted that, in general, this standard attempts to avoid adding and defining new errors. However, in the case of [ENOTSUP], the following points were noted: First, the need exists to provide feedback to applications concerning a new error condition. Second, while it is possible to use an existing error code in such cases (for example, ENOSYS), the group felt that this would overload those errors. P1003.1, when consulted, concurred with this view and agreed that the creation of a new error code, in this case, was appropriate. Third, the error [ENOTSUP] is also being used by P1003.4 for roughly the same reasons. Therefore, the consensus of several POSIX working groups is that while adding new errors is generally not recommended, that this case warrants the creation of a new error and that the new error should be [ENOTSUP].

The [EINVAL] error is returned by functions when the information label specified in the function call is syntactically incorrect or the information label is not permitted on the system because implementation-defined restrictions, (e.g., range restrictions). That is, this error is used to indicate the invalidity of the information label specified, independent of whether the operation would have succeeded had it been a valid label.

Although POSIX.1 does not specify precedence for error return values, careful consideration should be given to this matter in the security standard to ensure that covert channel considerations are adequately addressed. While information labeling is not usually subject to covert channels, in certain cases they may arise. Specifically, if an application that does not possess appropriate privilege attempts a function for which appropriate privilege is required and the implementation returns the EINVAL error in favor of the EPERM error, it may be possible for the application to determine the system's information label range restrictions based on whether EINVAL is returned (indicating the label is outside the system's range), or EPERM is returned (indicating the label is valid for the system, but that the application did not possess appropriate privilege). Therefore, despite this standard's silence on the issue, it is recommended that when a function could return multiple errors in a particular instance, that the errors be given the following precedence (from most favored to least favored): ENOSYS, EPERM, EINVAL, ENOTSUP.

B.27.4.3 Rationale for Pointer Arguments

The functions provided to support information labeling use an opaque data type. Nevertheless, in order to accommodate systems in which the size of an information label may vary (e.g., depending on the actual label encoded or depending on the total set of labels supported), the information label functions operate on pointers. For this reason, the basic information label structure defined in this standard (*inf_t*) is defined to be a pointer to an opaque data structure. In this way, conforming applications need not determine the size of a label prior to requesting an operation that will produce or modify that label. (In some cases, such as *inf_float*(), this would be particularly difficult inasmuch as the resultant information label is not known prior to making the request.) Instead, the system functions themselves are responsible for allocating the space necessary to contain a new label, and a function is provided to applications to free that space when the label is no longer needed.

The tradeoffs between the approach adopted by the information label functions specified in this standard and alternative approaches are many and varied. The structure of the information label function interfaces have been designed to be consistent with those provided by the interfaces supplied in support of the other features included in this standard, and the mandatory access control interfaces in particular. Thus, a more detailed and complete rationale for the adoption of these types of interfaces can be found in the mandatory access control rationale.

B.27.4.4 Rationale for POSIX.1e Functions

The *inf_float*() function is not specified in detail to allow for a range of implementation-defined floating policies. The range of policies would determine the degree to which information labels associated with data are automatically adjusted as data flows through the system. Two explicit floating policies that have been articulated are intended to be supportable in POSIX through the definition of *inf_float*().

The first policy is that articulated as part of the Compartmented Mode Workstation project (see IEEE Transactions on Software Engineering, Vol. 16, No. 6, June 1990, pp 608-618). Under this policy, every data read or write is intended to (potentially) modify the information label of the object being modified through the read or write. In the case of a subject reading an object, the subject's information label would be modified ("floated") to a combination of the information label of the subject before the read, and the information label associated with the object. When a subject writes to an object, the object's information label would be floated to represent the combination of the information label of the object before the write, and the information label associated with the subject. This policy makes a great deal of sense in the case where there are a large number of different information label values, and it is desired to track the flow of data through the system by having the data's information label follow the data. To accommodate this policy, *inf_float*() would always combine its two arguments and return the result. The details of the combination would depend on the semantics of the particular information labels involved.

The second policy makes more sense when there are a relatively small, more static number of information label values. In this policy, the intention is that objects, when created, inherit their creator's information label, but that the information label does not automatically change thereafter. To accommodate this policy, *inf_float*() would be defined such that it floated an information label only one time. In other words, *inf_float*() would return a result other than its second argument only when its second argument is equal to *inf_default*() and its first argument is not *inf_default*().

B.27.4.5 System Floating

Because the *inf_float*() routine takes two labels and returns the result of a float operation, it is not an entirely general function. That is, it cannot base the result of the float operation on any factor other than the two input labels. However, it is possible to imagine other data labeling policies that require different floating rules based on any number of factors (e.g., files involved, or time of day). Support for these peculiar types of policies is not explicitly required in this standard. The main reason for this exclusion is that, of the multiple data labeling polices intended to be supported by this standard, none require such extensions to the *inf_float*() function. Indeed, to the group's knowledge, no known data labeling policy currently used in commercially available systems that would require such extensions presently exists.

The second major reason for the lack of true generality in the floating function was due to technical obstacles. To make the *inf_float*() function more general, additional arguments would be required. The addition of more information used to characterize the two labels involved in floating was discussed. Particular consideration was given to adding type information so that the type of the object with which the information label is associated could be determined. This was to allow the implementation-defined algorithm to act differently based on the types of the objects involved. This addition was rejected because the working group could see no use for it in an external (application level) interface for conforming applications. The group also considered including arguments to identify the specific object being floated. Again, due to lack of motivation, and an inability to devise a useful interface that could be used to identify all POSIX objects that could support ILs, and still be extensible to non-POSIX objects (in a curt acknowledgement of the needs of the real world), this option, too, was dropped.

Note that the *inf_float*() function nevertheless remains a valuable and necessary interface: it allows conforming applications to call a routine which the system provides that is guaranteed to provide a label float operation consistent with the system's data labeling policies. Using the function, trusted applications can perform fine-grained labeling of their own resources.

B.27.4.6 Object Labeling

The objects to which this standard requires information labels be applied include the expected POSIX.1 objects: files. Not included among the objects are processes. As observed in the mandatory access control section, processes may act as objects under certain conditions. For example, when one process sends a signal to another, the former is effectively writing to the latter, and therefore the latter could be considered an object, from the perspective of this function. However, because many data labeling policies consider signals of this type to be a transmission of control information, and therefore not necessarily subject to the information label policies, many data labeling policies do not consider the process to be an object (from the information label perspective) with respect to these functions. Because POSIX.1 does not provide any other functions in which processes act as objects, the information labeling standard does not include processes as objects.

Note that information labels are not required to be applied to directories. Arguments for why they should be are as follows. Directories, like any other type of file, contain arbitrary length strings of process-specified data. This data is, by intent, designed to be communicative to users; that is, it is meaningful information (from the human perspective). Since this is the type of information data labeling policies are intended to label, it would make sense to require that directories be subject to the information label policies.

Alternatively, opposing opinions have been expressed that information labels should not be required to be applied to directories. These arguments are as follows. Directories are not containers of data, but rather are organizers of data containers (such as regular files). As such, the notion that information labels are applied to "data" as opposed to "control information" suggests that information labels may not necessarily be needed on directories. In addition, as with mandatory access control, existing mechanisms and techniques for applying information labels to directories vary widely (directory labeling, directory entry labeling, etc.). Worse yet, directory information labeling must necessarily be closely tied to the multi-level directory implementations used for mandatory access control. As witnessed by the absence of a multi-level directory specification in the mandatory access control section, directory labeling is not an area amenable to standardization at this time.

For the reasons set forth above, information labeling on directories is not required by this standard. Note, however, that conforming implementations may certainly provide that capability.

B.27.5 Initial Information Labels

This standard provides an interface that returns a valid information label that, if applied to a newly created file, will adequately label that file in a manner consistent with the system's information labeling policy. One intended use of this function is by trusted applications that wish to create, maintain, and properly label objects other than system-labeled objects. Examples of process-maintained independently-labeled objects could include: database records, individual mail messages, and so forth. When a process creates an instance of such an object, in order to perform floating as data is written to the object, the object must start with a correct initial information label. However, because these objects reside purely within the process space of the application, or are subcomponents of a larger single system-labeled object, the trusted application must assume responsibility for maintaining the labels on the object, including the initial label. For trusted applications, this initial label may well differ from the process label (especially if the process had floated prior to creating the object). For this reason the *inf_default*() function is provided. (In systems targeted for the CMW requirements, this label is often referred to as "system-low".)

The *inf_default*() function has deliberately been specified in very general terms in order to allow the widest range of implementations to conform to the standard. In particular, the function does not require that each call return the same value; the initial label may vary based on implementation-defined factors (for example, time of day, process id of the calling process, etc.). In addition, it is not guaranteed that the label returned by *inf_default*() will be the same as other system-generated labels at the same time. For example, a process that performs a call to *inf_default*() and immediately creates a new file may well find that the information label applied to the file differs from the information label returned by the call to *inf_default*(). This fact promotes flexibility in meeting this standard without hindering application portability: that the labels returned by *inf_default*() are consistent with the system's information labeling policy when applied to newly-created objects is sufficient for conforming applications to function properly.

Uses to which this flexibility may be put include: systems on which files created at particular times during the day may be more sensitive than files created at other times, systems on which files on particular file systems are labeled differently from those on other file systems, and so forth.

The addition of more information used to characterize the object to receive an initial information label was discussed. Particular consideration was given to adding type information so that the type of the object with which the initial information label is to be associated could be determined. This was to allow the implementation-defined algorithm to act differently based on the type of object to be labeled. This addition was rejected because the working group could see no use for it in an external (user level) interface for conforming applications. Internal (system-specific) initial information labels are not required to use *inf_default*() and therefore can be different based on the object being labeled.

B.27.6 Information Label Validity

Information labels have two forms: internal and external.

The basic information label structure defined in this standard (*inf_t*) is a pointer to an opaque data structure. The binary format of that opaque data structure may include such data as a hierarchical classification, non-hierarchical categories, or non-access control related markings. The standard makes no assumptions regarding the underlying representation or contents of the structure other than imposing the following constraint: the structure must be an exportable object. That is, the structure is opaque, persistent, and self-contained. The structure can therefore be copied by duplicating the bytes without knowledge of its syntax. Such a copy can be changed without any effect on the original, and the original can be changed without any effect on the copy.

The external format of a label is a text string of unspecified format. Any separator characters appearing between the components of an information label are implementation-defined. Note that this standard does not specify the set of legal characters that may be used in the text representation of an information label. Further rationale for this decision can be found in POSIX.1, section B.2.3.5.

The meaning of a valid information label is implementation-defined, as described in *inf_valid*(). An information label could be invalid for a variety of reasons. Some reasons why a label may be invalid on some systems include:

> It is malformed (e.g., the label contains a checksum in the opaque type that does not agree with the checksum calculated from the data).

> It is out of the cleared range of the system (e.g., the label refers to a classification that is outside the set of valid classifications for the system).

> It is outside the representation range (e.g., a system could allow no more than n categories from a universe of m, even though each of the m categories is valid).

> If {_POSIX_MAC} is defined, and the mandatory access control label of a process does not dominate the mandatory access control label associated with all components of an information label, then that information label may be invalid for the process, even though it is valid for other processes

executing on the same system.

Invalid information labels may appear for a great number of reasons. Examples include: constructing an information label in process memory without regard to semantics of the bits, importing an information label from a dissimilar system, etc. Note, however, that combining two information labels (e.g., using *inf_float*()), will calculate an information label that is valid. This is because information labeling, as noted elsewhere in this section, is used for data labeling, not access control. Therefore, if the other security policies implemented in a conforming system permit data to be combined, the information labeling mechanism is obligated to calculate an accurate and valid information label for the combined data.

B.27.7 Control Information

The policy discussion contained in section 27.1.2 specifically notes that the information label of a file applies only to the data portion of the file. That is, manipulation of control information need not result in an information label float operation. This "special" treatment for control information results from a tradeoff between functionality and security. If information labels floated when control information was manipulated (e.g., at file open time, instead of at data transfer time), the information labels associated with subjects and objects would have a tendency to float too often and would lose some of their utility as a mechanism to track the flow of data throughout a system. It can be argued that floating when control information is manipulated would result in more "trustworthy" information labels, however, several groups have expressed interest in favoring functionality over security in this case. It is understood that a conforming implementation may cause the float operation to occur at times in addition to those covered by the specified information labeling policy; such implementations may choose enhanced trustworthiness over security.

B.27.8 Relationship between ILs and Mandatory Access Control Labels

In some systems, such as compartmented mode workstations, there exist certain invariants that hold between ILs and mandatory access control labels. In the case of CMWs, this invariant states that for any specific subject's or object's labels, the access related portion of the information label (e.g., the classification and categories) must be dominated by the mandatory access control label. While this notion is useful for CMWs, it is not generally applicable to all systems that might support the information label interfaces specified in this document. Most notably, some companies that support the fundamental concept of information labels, employ them in a manner such that mandating a relationship between mandatory access control labels and ILs has no meaning. Indeed, there is no requirement in this standard that the mandatory access control option be supported in order to support the IL section.

Note that conforming implementations are always at liberty to enforce additional constraints. Thus a conforming implementation may certainly enforce a relationship between mandatory access control labels and ILs (such as dominance). The silence of this standard on the topic of specific relationships between mandatory access control labels and ILs should not dramatically impact portable applications.

B.27.9 Additional Uses of Information Labeling

The Compartmented Mode Workstation (CMW) security requirements are well known in many parts of the computer security community and have attracted considerable vendor interest. The CMW requirements are documented formally in "Security Requirements for System High and Compartmented Mode Workstations", Defense Intelligence Agency document DDS-2600-5502-87 and are discussed less formally in the June 1990 issue of IEEE Transactions on Software Engineering. Information labeling is a key component of the CMW requirements both for meeting certain data labeling policies that concern non-mandatory access control related information, and to avoid a potential data overclassification problem that may result from use of mandatory access control label-only systems. This section of the rationale will further examine the data overclassification problem as an additional example of the utility of information labels.

According to mandatory access control policy **FP.4**, a newly created file object shall be assigned the mandatory access control label of the creating subject (process). Such a policy is necessary to prevent any subjects with mandatory access control labels dominated by the creator's label from discovering the "fact of existence" of the object, thereby closing a covert channel.

Although the mandatory access control label of a newly created object correctly represents the sensitivity of the object from the standpoint of mandatory access control, it most likely incorrectly represents the actual sensitivity of the data contained in the object. Since the newly-created object contains no data, the sensitivity of the (null) data itself should be considered some system low value.

Another example of the overclassification problem is as follows. Consider a shell process (subject) executing with a mandatory access control label of *mac_p2*. During the lifetime of this shell the user decides to make a copy of another user's file containing data with a sensitivity of *mac_p1* and therefore a mandatory access control label of *mac_p1*. *mac_p2* dominates *mac_p1*, so the copy operation would be permitted by mandatory access control policy **FP.1**. The copy process will be created with a mandatory access control label of *mac_p2* (in accordance with mandatory access control policy **PP.2**), will read the data from the original file and store a copy of the data in a newly created file. In accordance with **FP.4**, the newly created file will have a mandatory access control label of *mac_p2*, even though the original data was only sensitive enough to require protection at the *mac_p1* level.

These overclassification problems can be mitigated with the use of information labels. In particular, an implementation could define *inf_default*() to return an information label of "system low" and *inf_float*() to combine information labels as per the CMW requirements. In such a system the information label of a newly created (empty) object would be system low—an accurate representation of the actual sensitivity of the (null) data contained within the object. Note that this newly created object (and the fact that this object existed) would still be correctly protected by the object's mandatory access control label. When a process reads from a file, the process information label floats with the file information label. When a process writes to a file, the file information label floats with the process information label.

Returning to the copy example, say the information label of the source file is *inf_p1*. The copy process will start with an information label of *inf_p2*, which we assume is system low as defined by *inf_default*() (as will generally be the case). In the model of information label floating described in the paragraph above, when the copy process reads the data from the file to be copied, the copy process' information label will float to the value returned by *inf_float(inf_p1, inf_p2)*, which, because *inf_p2* is system low, will equal *inf_p1*. When the copy process creates and writes the target file, that file will float to *inf_p1* (the copy process' label). Thus the information label of the data in the source file will follow the data as it moves through the system. So, even though the target file has a mandatory access control label that is higher than the mandatory access control label of the source file, the target file's information label is the same as the source file's information label and remains an accurate representation of the actual sensitivity of the data in the file.

Annex F
(informative)
Ballot Instructions

This annex will not appear in the final standard. It is included in the draft to provide instructions for balloting that cannot be separated easily from the main document, as a cover letter might.

It is important that you read this annex, whether you are an official member of the PSSG Balloting Group or not; comments on this draft are welcomed from all interested technical experts.

Summary of Draft 17 Instructions

This is a recirculation on the P1003.1e ballot. The procedure for a recirculation is described in this annex. Because this is a recirculation comments may only be provided concerning sections that have changed, sections affected by those changes, or on rejected comments from the previous ballot.

Send your ballot and/or comments to:

IEEE Standards Office
Computer Society Secretariat
ATTN: PSSG Ballot (Carol Buonfiglio)
P.O. Box 1331
445 Hoes Lane
Piscataway, NJ 08855-1331

It would also be very helpful if you sent us your ballot in machine-readable form. Your official ballot must be returned via mail to the IEEE office; if we receive only the e-mail or diskette version, that version will not count as an official document. However, the online version would be a great help to ballot resolution. Please send your e-mail copies to the following address:

casey@sgi.com

or you may send your files in ASCII format on DOS 3.5 inch formatted diskettes (720Kb or 1.4Mb), or Sun-style QIC-24 cartridge tapes to:

Casey Schaufler
Silicon Graphics
2011 North Shoreline Blvd.
P.O. Box 7311
Mountain View, CA 94039-7311

Background on Balloting Procedures

The Balloting Group consists of approximately eighty technical experts who are members of the IEEE or the IEEE Computer Society; enrollment of individuals in this group has already been closed. There are also a few "parties of interest" who are not members of the IEEE or the Computer Society. Members of the Balloting Group are required to return ballots within the balloting period. Other individuals who may happen to read this draft are also encouraged to submit comments concerning this draft. The only real difference between members of the Balloting Group and other individuals submitting ballots is that *affirmative* ballots are only counted from Balloting Group members who are also IEEE or Computer Society members. (There are minimum requirements for the percentages of ballots returned and for affirmative ballots out of that group.) However, objections and nonbinding comments must be resolved if received from any individual, as follows:

(1) Some objections or comments will result in changes to the standard. This will occur either by the republication of the entire draft or by the publication of a list of changes. The objections/comments are reviewed by a team from the POSIX Security working group, consisting of the Chair, Vice Chair, Technical Editor, and a group of Technical Reviewers. The Chair will act as the Ballot Coordinator. The Technical Reviewers each have subject matter expertise in a particular area and are responsible for objection resolution in one or more sections.

(2) Other objections/comments will not result in changes.

  (a) Some are misunderstandings or cover portions of the document (front matter, informative annexes, rationale, editorial matters, etc.) that are not subject to balloting.

  (b) Others are so vaguely worded that it is impossible to determine what changes would satisfy the objector. These are referred to as *Unresponsive*. (The Technical Reviewers will make a reasonable effort to contact the objector to resolve this and get a newly worded objection.) Further examples of unresponsive submittals are those not marked as either *Objection*, *Comment*, or *Editorial*; those that do not identify the portion of the document that is being objected to (each objection must be separately labeled); those that object to material in a recirculation that has not changed and do not cite an unresolved objection; those that do not provide specific or general guidance on what changes would be required to resolve the objection.

(c) Finally, others are valid technical points, but they would result in decreasing the consensus of the Balloting Group. (This judgment is made based on other ballots and on the experiences of the working group through over seven years of work and fifteen drafts preceding this one.) These are referred to as *Unresolved Objections*. Summaries of unresolved objections and their reasons for rejection are maintained throughout the balloting process and are presented to the IEEE Standards Board when the final draft is offered for approval. Summaries of all unresolved objections and their reason for rejection will also be sent to members of the Balloting Group for their consideration upon a recirculation ballot. (Unresolved objections are not circulated to the ballot group for a re-ballot.) Unresolved objections are only circulated to the balloting group when they are presented by members of the balloting group or by parties of interest. Unsolicited correspondence from outside these two groups may result in draft changes, but are not recirculated to the balloting group members.

Please ensure that you correctly characterize your ballot by providing one of the following:

(1) Your IEEE member number

(2) Your IEEE Computer Society affiliate number

(3) If (1) or (2) don't apply, a statement that you are a "Party of Interest"

Ballot Resolution

The general procedure for resolving ballots is:

(1) The ballots are put online and distributed to the Technical Reviewers.

(2) If a ballot contains an objection, the balloter may be contacted individually by telephone, letter, or e-mail and the corrective action to be taken described (or negotiated). The personal contact will most likely not occur if the objection is very simple and obvious to fix or the balloter cannot be reached after a few reasonable attempts. Repeated failed attempts to elicit a response from a balloter may result in an objection being considered unresponsive, based on the judgment of the Ballot Coordinator. Once all objections in a ballot have been resolved, it becomes an affirmative ballot.

(3) If any objection cannot be resolved, the entire ballot remains negative.

(4) After the ballot resolution period the technical reviewers may chose to either *re-ballot* or *recirculate* the ballot, based on the status of the standard and the number and nature of outstanding (i.e., rejected or unresolved) objections. The ballot group may or may not be reformed at this time. If a *reballot* is chosen, the entire process of balloting begins anew. If a *recirculation* is chosen, only those portions affected by the previous ballot will be under consideration. This ballot falls into this latter category

(5) On a *recirculation* ballot, the list of unresolved objections, along with the ballot resolution group's reasons for rejecting them will be circulated to the existing ballot group along with a copy of the document that clearly indicates all changes that were made during the last ballot period. You have a minimum of ten days (after an appropriate time to ensure the mail got through) to review these two documents and take one of the following actions:

   (a) Do nothing; your ballots will continue to be counted as we have classified them, based on items (3) and (4).

   (b) Explicitly change your negative ballot to affirmative by agreeing to remove all of your unresolved objections.

   (c) Explicitly change your affirmative ballot to negative based on your disapproval of either of the two documents you reviewed. If an issue is not contained in an unresolved objection or is not the result of a change to the document during the last ballot resolution period, it is not allowed. Negative ballots that come in on recirculations cannot be cumulative. They shall repeat any objections that the balloter considers unresolved from the previous recirculation. Ballots that simply say "and all the unresolved objections from last time" will be declared unresponsive. Ballots that are silent will be presumed to fully replace the previous ballot, and all objections not mentioned on the most current ballot will be considered as successfully resolved.

(6) Rather than reissue the entire document, a small number of changes may result in the issuance of a change list rather than the entire document during recirculation.

(7) A copy of all your objections and our resolutions will be mailed to you.

(8) If at the end of a recirculation period there remain greater than seventy-five percent affirmative ballots, and no new objections have been received, a new draft is prepared that incorporates all the changes. This draft and the unresolved objections list go to the IEEE Standards Board for approval. If the changes cause too many ballots to slip back into negative status, another resolution and recirculation cycle begins.

Balloting Guidelines

This section consists of guidelines on how to write and submit the most effective ballot possible. The activity of resolving balloting comments is difficult and time consuming. Poorly constructed comments can make that even worse.

We have found several things that can be done to a ballot that make our job more difficult than it needs to be, and likely will result in a less than optimal response to ballots that do not follow the form below. Thus it is to your advantage, as well as ours, for you to follow these recommendations and requirements.

If a ballot that significantly violates the guidelines described in this section comes to us, we may determine that the ballot is unresponsive.

If we recognize a ballot as "unresponsive," we will try to inform the balloter as soon as possible so he/she can correct it, but it is ultimately the balloter's responsibility to assure the ballot is responsive. Ballots deemed to be "unresponsive" may be ignored in their entirety.

Some general guidelines to follow before you object to something:

(1) Read the Rationale section that applies to the troublesome area. In general there is a matching informative section in the Rationale Annex for each normative section of the standard. This rationale often explains why choices were made and why other alternatives were not chosen.

(2) Read the Scope, section 1, to see what subset of functionality we are trying to achieve. This standard does not attempt to be everything you ever wanted for accomplishing secure software systems. If you feel that an additional area of system interface requires standardization, you are invited to participate in the security working group which is actively involved in determining future work.

(3) Be cognizant of definitions in section 2. We often rely in the document on a precise definition from section 2 which may be slightly different than your expectation.

Typesetting is not particularly useful to us. Also please do not send handwritten ballots. Typewritten (or equivalent) is fine, and if some font information is lost it will be restored by the Technical Editor in any case. You may use any word processor to generate your objections but do not send `[nt]roff` (or any other word processor) input text. Also avoid backslashes, leading periods and apostrophes in your text as they will confuse our word processor during collation and printing of your comments. The ideal ballot is formatted as a "flat ASCII file," without any attempt at reproducing the typography of the draft and without embedded control characters or overstrikes; it is then printed in Courier (or some other typewriter-like) font for paper-mailing to the IEEE Standards Office and simultaneously e-mailed to the Working Group Ballot Coordinator at the following email address.

casey@sgi.com

Don't quote others' ballots. Cite them if you want to refer to another's ballot. If more than one person wants to endorse the same ballot, send just the cover sheets and one copy of the comments and objections. [Note to Institutional Representatives of groups like X/Open, OSF, UI, etc.: this applies to you, too. Please don't duplicate objection text with your members.] Multiple identical copies are easy to deal with, but just increase the paper volume. Multiple almost-identical ballots are a disaster, because we can't tell if they are identical or not, and are likely to miss the subtle differences. Responses of the forms:

— "I agree with the item in <someone>'s ballot, but I'd like to see this done instead"

— "I am familiar with the changes to `foo` in <someone>'s ballot and I would object if this change is [or is not] included" are very useful information to us. If we resolve the objection with the original balloter (the one whose ballot you are referencing), we will also consider yours to be closed, unless you specifically include some text in your objection indicating that should not be done.

Be very careful of “Oh, by the way, this applies <here> too” items, particularly if they are in different sections of the document that are likely to be seen by different reviewers. They are probably going to be missed! Note the problem in the appropriate section, and cite the detailed description if it’s too much trouble to copy it. The reviewers don’t read the whole ballot. They only read the parts that appear in the sections that they have responsibility for reviewing. Particularly where definitions are involved, if the change really belongs in one section but the relevant content is in another, please include two separate comments/objections.

Please consider this a new ballot that should stand on its own. Please do not make backward references to your ballots for the previous draft. Include all the text you want considered here, because the Technical Reviewer will not have your old ballot. (The old section and line numbers won’t match up anyway.) If one of your objections was not accepted exactly as you wanted, it may not be useful to send in the exact text you sent before; read our response to your objection (you will receive these in a separate mailing) and the associated Rationale section and come up with a more compelling (or clearly-stated) justification for the change.

Please be very wary about global statements, such as “all of the arithmetic functions need to be defined more clearly.” Unless you are prepared to cite specific instances of where you want changes made, with reasonably precise replacement language, your ballot will be considered unresponsive.

Ballot Form

The following form is strongly recommended. We would greatly appreciate it if you sent the ballot in electronic form in addition to the required paper copy. Our policy is to handle all ballots online, so if you don’t send it to us that way, we have to type it in manually. See the first page of this Annex for the addresses and media. As you’ll see from the following, formatting a ballot that’s sent to us online is much simpler than a paper-only ballot.

The paper ballot should be page-numbered, and each page should contain the name, e-mail address, and phone number(s) of the objector(s). The electronic copy of the ballot should only have it once, in the beginning. Please leave adequate (at least one inch) margins on both sides.

Don’t format the ballot as a letter or document with its *own* section numbers. These are simply confusing. As shown below, it is best if you cause each objection and comment to have a sequential number that we can refer to amongst ourselves and to you over the phone. Number sequentially from 1 and count objections, comments, and editorial comments the same; don’t number each in its own range.

We recognize three types of responses:

Objection A problem that must be resolved to your satisfaction prior to your casting an "affirmative" vote for the document.

Comment A problem that you might want to be resolved by the reviewer, but which does not in any way affect whether your ballot is negative or positive. Any response concerning the pages preceding page 1 (the Front matter), Rationale text with shaded margins, Annexes, NOTES in the text, footnotes, or examples will be treated as a non-binding comment whether you label it that way or not. (It would help us if you'd label it correctly.)

Editorial A problem that is strictly an editorial oversight and is not of a technical nature. Examples are: typos; misspellings; English syntax or usage errors; appearances of lists or tables; arrangement of sections, clauses, and subclauses (except where the location of information changes the optionality of a feature).

To help us in our processing of your objections and comments, we are requiring that all comments, objections and editorial comments meet the following specific format. (We know that the format defined below contains redundant information but it has become a de facto standard used by many different POSIX standard ballots. It is felt that it is better to continue to use this format with the redundancies rather than to create a new format just for 1003.1e and P1003.2c)

Separate each objection/comment with a line of dashes ("-"), e.g.,

```
--------------------------------------------------------------------------------
```

Precede each objection/comment with two lines of identifying information:

The first line should contain:

@ *<section>*.*<clause>* *<code>* *<seqno>*

where:

@ At-sign in column 1 (which means no @'s in any other column 1's).

<section> The major section (chapter or annex) number or letter in column 3. Use zero for Global or for something, like the frontmatter, that has no section or annex number.

<clause> The clause number (second-level header). Please do not go deeper than these two levels. In the text of your objection or comment, go as deep as you can in describing the location, but this code line uses two levels only.

<code> One of the following lowercase letters, preceded and followed by spaces:

- o Objection.
- c Comment.
- e Editorial Comment.

<seqno> A sequence number, counting all objections and comments in a single range.

The second line should contain:

<seqno>`. Sect` *<sectno>* *<type>*`. page` *<pageno>*`, line` *<lineno>*`:`

where:

<seqno> The sequence number from the preceding line

<sectno> The full section number. (Go as deep as you can in describing the location.)

<type> One of the following key words/phrases, preceded and followed by spaces:

`OBJECTION`

`COMMENT`

`EDITORIAL COMMENT`

<pageno> The page number from the document.

<lineno> The line number or range of line numbers that the object/comment relates to.

For each objection, comment, or editorial comment, you should provide a clear statement of the problem followed by the action required to solve that problem.

`Problem:`

A clear statement of the problem that is observed, sufficient for others to understand the nature of the problem. (Note that you should identify problems by section, page, and line numbers. This may seem redundant, but if you transpose a digit pair, we may get totally lost without a cross-check like this. Use the line number where the problem starts, not just where the section itself starts; we sometimes attempt to sort objections by line numbers to make editing more accurate. If you are referring to a range of lines, please don't say "lines 10xx;" use a real range so we can tell where to stop looking. Please try to include enough context information in the problem statement (such as the name of the function or command) so we can understand it without having the draft in our laps at the time. (It also helps you when we e-mail it back to you.)

`Action:`

A precise statement of the actions to be taken on the document to resolve the objection above, which if taken verbatim will completely remove the objection.

If there is an acceptable range of actions, any of which will resolve the problem for you if taken exactly, please indicate all of them. If we accept any of these, your objection will be considered as resolved.

If the Action section is omitted or is vague in its solution, the objection may be reclassified as a nonbinding comment. The Technical Reviewers, being human, will give more attention to Actions that are well-described than ones that are vague or imprecise. The best ballots of all have very explicit directions to substitute, delete, or add text in a style consistent with the rest of the document, such as:

```
Delete the sentence on lines 101-102:

"The implementation shall not ... or standard error."

On line 245, change "shall not" to "should not".

After line 711, add:

-c  Calculate the mask permissions and update the mask.
```

Some examples of poorly-constructed actions:

```
Remove all features of this command that are not supported by BSD.

Add -i.

Make this command more efficient and reliable.

Use some other flag that isn't so confusing.

I don't understand this section.

Specify a value--I don't care what.
```

Sample Response:

```
Joseph Balloter                  (999)123-4567       page  4 of 17.
EMAIL: jmb@mycomp.com     FAX:   (999)890-1234

----------------------------------------------------------------

@ 1.1 o 23
23. Sect 1.1 OBJECTION. page 7, line 9:

Problem:

The current draft describes one the mechanisms specified in it as
"Least Privilege" which is incorrect.  "Least Privilege" is a
general principle related to access control rather than a mechanism.
In fact, the definition given in the standard (p.  91, l.  274)
calls it a principle rather than a mechanism.

Action:

Replace line 9 with:  "(3)  Enforcement of Least Privilege"

----------------------------------------------------------------

@ 3.1 o 24
24. Sect 3.1 OBJECTION. page 27, line 13:

Problem:

"during process of changing ACL" is vague.
Could be read as the duration from acl_read through acl_write.

Action:
```

Should state "while ACL is being written (acl_write)".

@ 3.3 e 25
25. Sect 3.3.1 EDITORIAL COMMENT. page 29, line 68:

Problem:

The two previous sentences describe the "ACL_USER_OBJ entry" and the "ACL_GROUP_OBJ entry". Line 68 describes "ACL_OTHER_OBJ", the word "entry" should be added for consistency.

Action:

change "ACL_OTHER_OBJ" to "ACL_OTHER_OBJ entry"

Sample Response (continued):

```
Joseph Balloter                 (999)123-4567      page  5 of 17.
EMAIL: jmb@mycomp.com     FAX:  (999)890-1234

----------------------------------------------------------------

@ 4.5 c 26
26. Sect 4.5.1.1 COMMENT. page 92, line 836:

Problem:

There is no introduction to table 4-1.

Action:

Add before line 836  "The aud_ev_info_t structure shall contain at
least the following fields:"

----------------------------------------------------------------

@ 6.5 o 27
27. Sect 6.5.7.2 OBJECTION. page 181, line 449-450:

Problem:

Can this "must" be tested ?
Is this really needed since the format of the label is undefined
and no functions are provided to access the individual components
(so that a comparison could be made). This seems to be a comment
that could just as easily be applied to most other mac functions,
say mac_freelabel for example.

Action:

Suggest either moving this into the MAC introductory section,
striking or changing "must" to "should" or "are advised".

----------------------------------------------------------------
```

Thank you for your cooperation and assistance in this important balloting process.

Lynne M. Ambuel
Chair, POSIX Security Working Group

Identifier Index

Topical Index

Contents

SECTION PAGE

SECTION PAGE

SECTION PAGE

SECTION PAGE

TABLES