

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**

Copyright © 1997 by the Institute of Electrical and Electronics Engineers, Inc
345 East 47th Street,
New York, NY 10017, USA
All rights reserved.

ISBN-xxxx-xxxxx-x

Library of Congress Catalog Number 90-xxxxx

**IEEE Draft P1003.1e, Copyright © IEEE.
All Rights Reserved by IEEE.**

**The IEEE disclaims any responsibility or liability resulting from the
placement and use of this document.**

**This copyrighted document may be downloaded for personal use by one (1)
individual user.**

**No further copying or distribution is permitted without the express written
permission or an appropriate license from the IEEE.**

This is a withdrawn IEEE Standards Draft.

**Permission is hereby granted for IEEE Standards Committee participants to
reproduce this document for purposes of IEEE standardization activities.
Permission is also granted for member bodies and technical committees of
ISO and IEC to reproduce this document for purposes of developing a
national position.**

**Other entities seeking permission to reproduce this document for
standardization or other activities, or to reproduce portions of this
document for these or other uses, must contact the IEEE Standards
Department for the appropriate license.**

Use of information contained in this unapproved draft is at your own risk.

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

XXXXXXX

**WITHDRAWN DRAFT. All Rights Reserved by IEEE.
Preliminary—Subject to Revision.**

WITHDRAWN DRAFT. All Rights Reserved by IEEE.
Preliminary—Subject to Revision.

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 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	

**WITHDRAWN DRAFT. All Rights Reserved by IEEE.
Preliminary—Subject to Revision.**

Information technology—Portable operating system interface for computer environments

2 Section 1: Revisions to the General Section

3 ⇒ **1.1 Scope** *This scope is to be revised and integrated appropriately into the
4 scope when POSIX.1e is approved:*

5 This standard, P1003.1e/D17: October 1997 (POSIX.1e), defines five indepen- %
6 dent, optional sets of interfaces that will be used to implement protection,
7 audit, and control mechanisms. Implementation of any or all of these inter-
8 faces does not ensure the security of the conforming system nor of conforming
9 applications. In addition, implementation of these interfaces does not imply
10 that a conforming system can achieve any class or level of any security evalua-
11 tion criteria. These interfaces will become integrated into the ISO/IEC 9945-1:
12 1990 (System Application Program Interface) standard (POSIX.1) as they are
13 approved and published. The sets of interfaces for implementation are:

- 14 (1) Access Control Lists (ACL)
- 15 (2) Security Auditing
- 16 (3) Capability
- 17 (4) Mandatory Access Controls (MAC)
- 18 (5) Information Labeling (IL)

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

21 ⇒ **1.2 Normative References (POSIX.1: line 39)**

22 The following standards contain provisions that, through references in this
23 text, constitute provisions of this standard. At the time of publication, the edi-
24 tions indicated were valid. All standards are subject to revision, and parties to
25 agreements based on this part of this standard are encouraged to investigate
26 the possibility of applying the most recent editions of the standards listed
27 below. Members of IEC and ISO maintain registers of currently valid Interna-
28 tional Standards.

- 29 (1) ISO/IEC 9945-1: 1990, Information Technology—Portable Operating Sys-
30 tem Interface (POSIX)—Part 1: System Application Program Interface
31 (API) [C Language]
- 32 (2) IEEE Standard for Information Technology—Portable Operating System
33 Interface (POSIX)—Part 2: Shell and Utilities. |
- 34 (3) P1003.2c/D17: October 1997, Draft Standard for Information %
35 Technology—Portable Operating System Interface (POSIX)—Part 2:
36 Shell and Utilities—Amendment #: Protection and Control Utilities

37 ⇒ **1.3.1.3 Conforming Implementation Options (POSIX.1: line 98)** *Insert*
38 *the following options in alphabetic order:*

- 39 {_POSIX_ACL} Access control list option (in 2.9.3) |
- 40 {_POSIX_AUD} Auditing option (in 2.9.3) |
- 41 {_POSIX_CAP} Capability option (in 2.9.3) |
- 42 {_POSIX_MAC} Mandatory access control option (in 2.9.3) |
- 43 {_POSIX_INF} Information label option (in 2.9.3) |

1 **Section 2: Revisions to Terminology and General Requirements**

2 **⇒ 2.2.1 Terminology**

3 **⇒ 2.2.2 General Terms (POSIX.1: lines 89-397)** *Add the following definitions -*
4 *in alphabetical order:*

5 **2.2.2.1 access:** A specific type of interaction between a process and an object
6 that results in the flow of information from one to the other. Possible information
7 flows include the transfer of attributes pertaining to that object, the transfer of
8 data pertaining to that object, or the fact of existence of that object.

9 **2.2.2.2 access acl:** An access control list (ACL) which is used in making discre-
10 tionary access control decisions for an object.

11 **2.2.2.3 access control:** The prevention of unauthorized access to objects by
12 processes and, conversely, the permitting of authorized access to objects by
13 processes.

14 **2.2.2.4 access control list (ACL):** A discretionary access control entity associ-
15 ated with an object, consisting of a list of entries where each entry is an identifier
16 (e.g. user or group of users) coupled with a set of access permissions.

17 **2.2.2.5 access control policy:** A set of rules, part of a security policy, by which
18 a user's authorization to access an object is determined.

19 **2.2.2.6 audit:** The procedure of capturing, storing, analyzing, maintaining and
20 managing data concerning security-relevant activities.

21 **2.2.2.7 auditable event:** An activity which may cause an audit record to be
22 reported in an audit log.

- 23 **2.2.2.8 audit event type:** A field within an audit record that identifies the
24 activity reported by the record and defines the required content of the record.
- 25 **2.2.2.9 audit ID:** An identifier for the user accountable for an audit event.
- 26 **2.2.2.10 audit record:** The discrete unit of data reportable in an audit log on
27 the occurrence of an audit event.
- 28 **2.2.2.11 audit log:** The destination of audit records that are generated and the
29 source of records read by an audit post-processing application.
- 30 **2.2.2.12 availability:** The property of an object or subject being accessible and
31 usable upon demand by an authorized user.
- 32 **2.2.2.13 capability:** An attribute of a process that is included in the determina- %
33 tion of whether or not a process has the appropriate privilege to perform a specific
34 POSIX.1 action where appropriate privilege is required. —
- 35 **2.2.2.14 capability flag:** A per-capability attribute of a file or process that is +
36 used during *exec()* processing in computing the capability of the process executing|
37 that file.
- 38 **2.2.2.15 capability state:** A grouping of all of the flags defined by an implemen-
39 tation for all capabilities defined for the implementation.
- 40 **2.2.2.16 channel:** An information transfer path within a system or a mechanism
41 by which the path is effected.
- 42 **2.2.2.17 confidentiality:** The property that the existence of an object and/or its
43 contents and/or attributes are not made available nor disclosed to unauthorized
44 processes.
- 45 **2.2.2.18 covert channel:** A communications channel that allows a process to
46 transfer information in a manner that violates the system's security policy. Covert
47 channels are typically realized by the exploitation of mechanisms not intended to
48 be used for communication.
- 49 **2.2.2.19 data descriptor:** An internal representation which uniquely identifies
50 a data object.
- 51 **2.2.2.20 default acl:** An ACL which is used in determining the initial discre-
52 tionary access control information for objects.

53 **2.2.2.21 denial of service:** The unauthorized prevention of authorized access to
54 resources or the delaying of time-critical operations.

55 **2.2.2.22 discretionary access control (DAC):** A means of restricting access to
56 objects based on the identity of the user, process, and/or groups to which the
57 objects belong. The controls are discretionary in the sense that a subject with
58 some access permission is capable of passing that permission (perhaps indirectly)
59 on to other subjects.

60 **2.2.2.23 dominate:** An implementation-defined relation between the values of
61 MAC labels or information labels.

62 **2.2.2.24 downgrade:** An operation which changes a MAC label or information
63 label to a value that does not dominate the current label.

64 **2.2.2.25 equivalent:** An implementation-defined relation between the values of
65 MAC labels or of information labels. Two labels are equivalent if each of the labels
66 dominates the other.

67 **2.2.2.26 extended ACL:** An ACL that contains entries in addition to a
68 *minimum ACL*.

69 **2.2.2.27 exportable data:** Opaque data objects for which the data is self-
70 contained and persistent. As a result, they can be copied or stored freely.

71 **2.2.2.28 file group class:** The property of a file indicating access permissions
72 for a process related to the process's group identification.

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

76 If `{_POSIX_ACL}` is defined, then a process is also in the file group class if the pro-
77 cess is not in the file owner class and

- 78 (1) the effective user ID of the process matches the qualifier of one of the
79 `ACL_USER` entries in the ACL associated with the file, or
- 80 (2) the effective group ID or one of the supplementary group IDs of the pro-
81 cess matches the qualifier of one of the `ACL_GROUP` entries in the ACL
82 associated with the file.

83 Other members of the class may be implementation defined.

84 **2.2.2.29 formal security policy model:** A precise statement of a system secu-
85 rity policy.

86 **2.2.2.30 information label:** The representation of a security attribute of a sub-
87 ject or object that applies to the data contained in that subject or object and is not
88 used for mandatory access control.

89 **2.2.2.31 information label floating:** The operation whereby one information
90 label is combined with another information label. The specific algorithm used to
91 define the result of a combination of two labels is implementation defined.

92 **2.2.2.32 information label policy:** The policy that determines how information
93 labels associated with objects and subjects are automatically adjusted as data
94 flows through the system.

95 **2.2.2.33 MAC label:** The representation of a security attribute of a subject or
96 object which represents the sensitivity of the subject or object and is used for
97 mandatory access control decisions.

98 **2.2.2.34 mandatory access control (MAC):** A means of restricting and permitting
99 access to objects based on an implementation-defined security policy using
100 MAC labels and the use of the implementation-defined dominate operator. The
101 restrictions are mandatory in the sense that they are always imposed by the sys-
102 tem.

103 **2.2.2.35 minimum ACL:** An ACL that contains only the required ACL entries. –

104 **2.2.2.36 object:** A passive entity that contains or receives data. Access to an
105 object potentially implies access to the data that it contains.

106 **2.2.2.37 opaque data object:** A data repository whose structure and represen-
107 tation is unspecified. Access to data contained in these objects is possible through-
108 the use of defined programming interfaces.

109 **2.2.2.38 persistent:** A state in which data retains its original meaning as long
110 as the system configuration remains unchanged, even across system reboots.
111 However, any change to the system configuration (such as adding or deleting user
112 IDs and modifying the set of valid labels) may render such data invalid. –

113 **2.2.2.39 principle of least privilege:** A security design principle that states
114 that a process or program be granted only those privileges (e.g., capabilities)
115 necessary to accomplish its legitimate function, and only for the time that such
116 privileges are actually required.

117 **2.2.2.40 query:** Any operation which obtains either data or attributes from a
118 subject or object.

119 **2.2.2.41 read:** A fundamental operation that obtains data from an object or sub-
120 ject.

121 **2.2.2.42 required ACL entries:** The three ACL entries that must exist in every
122 valid ACL. These entries are exactly one entry each for the owning user, the own-
123 ing group, and other users not specifically enumerated in the ACL.

124 **2.2.2.43 security:** The set of measures defined within a system as necessary to
125 adequately protect the information to be processed by the system.

126 **2.2.2.44 security administrator:** An authority responsible for implementing
127 the security policy for a security domain.

128 **2.2.2.45 security attribute:** An attribute associated with subjects or objects
129 which is used to determine access rights to an object by a subject.

130 **2.2.2.46 security domain:** A set of elements, a security policy, a security
131 authority and a set of security-relevant activities in which the set of elements are
132 subject to the security policy, administered by the security authority, for the
133 specified activities.

134 **2.2.2.47 security policy:** The set of laws, rules, and practices that regulate how
135 an organization manages, protects, and distributes sensitive information.

136 **2.2.2.48 security policy model:** A precise presentation of the security policy
137 enforced by a system. +

138 **2.2.2.49 strictly dominate:** A relation between the values of two MAC labels or
139 information labels whereby one label dominates but is not equivalent to the other
140 label.

141 **2.2.2.50 subject:** An active entity that causes information to flow between
142 objects or changes the system state; e.g., a process acting on behalf of a user. |

143 **2.2.2.51 tranquillity:** Property whereby the MAC label of an object can be
144 changed only while it is not being accessed. -

145 **2.2.2.52 upgrade:** An operation that changes the value of a MAC label or infor-
146 mation label to a value that strictly dominates its previous value.

147 **2.2.2.53 user:** Any person who interacts with a computer system. Operations
148 are performed on behalf of the user by one or more processes.

149 **2.2.2.54 write:** A fundamental operation that results only in the flow of information from a subject to an object.

151 ⇒ **2.2.3 Abbreviations (POSIX.1: line 404)**

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

- 153 (1) **POSIX.1:** ISO/IEC 9845-1: 1990: Information Technology—Portable |
154 Operating System Interface (POSIX)—Part 1: System Application Program |
155 Interface (API) [C Language]
- 156 (2) **POSIX.2:** ISO/IEC 9845-1: 1992: Information IEEE Standard for Infor- |
157 mation Technology—Portable Operating System Interface (POSIX)—Part |
158 2: Shell and Utilities
- 159 (3) **POSIX.1e:** IEEE Std 1003.1e/D17: October 1997, Draft Standard for |
160 Information Technology—Portable Operating System Interface |
161 (POSIX)—Protection, Audit and Control Interfaces
- 162 (4) **POSIX.2c:** IEEE Std 1003.2c/D17: October 1997, Draft Standard for |
163 Information Technology—Portable Operating System Interface |
164 (POSIX)—Protection and Control Utilities

165 ⇒ **2.3 General Concepts (POSIX.1: lines 406-498)**

166 ⇒ **2.3.2 file access permissions (POSIX.1: line 413)** *Change this sub-clause to |*
167 “**2.3.2 file access controls**”, and incorporate the concept of “file access permis- |
168 sions” under it along with the following new concepts:

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

172 ⇒ **2.3.2.1 file access permissions (POSIX.1: line 414)** *After the above change |*
173 to section 2.3.2, create a new subsection called 2.3.2.1 and replace the previous |
174 text in POSIX.1 subsection 2.3.2 with the following.

175 This standard defines discretionary file access control on the basis of file per- |
176 mission bits as described below. The additional provisions of section 2.3.2.2 |
177 apply only if {_POSIX_ACL} is defined.

178 The file permission bits of a file contain read, write, and execute/search per- |
179 missions for the file owner class, file group class, and file other class.

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

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

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

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

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

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

199 If the process possesses appropriate privilege:

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

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

206 Otherwise:

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

212 If `{_POSIX_CAP}` is defined, then appropriate privilege includes the following |
213 capabilities: `CAP_DAC_WRITE` for write access, `CAP_DAC_EXECUTE` for exe-
214 cute access, and `CAP_DAC_READ_SEARCH` for read and search access. See +
215 Table 25-5.

216 => **2.3.2.2 access control lists:** Add this as a new concept.

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

223 An ACL entry shall support at a minimum read, write, and execute/search per-
224 missions.

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

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

238 If the process possesses appropriate privilege:

- 239 — If read, write or directory search permission is requested, access
240 is granted.
- 241 — If execute permission is requested, access is granted if execute
242 permission is specified in at least one ACL entry; otherwise,
243 access is denied.

244 Otherwise:

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

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

254 ⇒ **2.3.2.3 mandatory access control:** Add this as a new concept.

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

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

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

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

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

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

282 ⇒ **2.3.2.4 evaluation of file access:** Add this as a new concept.

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

289 ⇒ **2.3.5 file times update: (POSIX.1: line 475)** *Add the following paragraph to*
290 *the concept definition of file times update:*

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

294 ⇒ **2.4 Error Codes** *Add the following items to the error code definitions in alpha–*
295 *betic order.*

296 [ENOTSUP] Operation is not supported. —

297 ⇒ **2.7.2 POSIX.1 Symbols (POSIX.1: Table 2-2)** Insert the following entries in
 298 alphabetical order in Table 2-2:

	Header	Key	Reserved Prefix	Reserved Suffix
299	<sys/acl.h>	1	acl_	
300		2	ACL_	
302	<sys/audit.h>	1	aud_	
303		2	AUD_	
304	<sys/capability.h>	1	cap_	
305		2	CAP_	
306	<sys/inf.h>	1	inf_	
307		2	INF_	
308	<sys/mac.h>	1	mac_	
309		2	MAC_	
310				
311				

312 ⇒ **2.7.3 Headers and Function Prototype (POSIX.1: line 910-927)** Add the
 313 following entries in alphabetical order:

```

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

322     <sys/audit.h> aud_copy_ext(), aud_copy_int(), aud_delete_event(),
323                 aud_delete_event_info(), aud_delete_hdr(), aud_delete_hdr_info(),
324                 aud_delete_obj(), aud_delete_obj_info(), aud_delete_subj(),
325                 aud_delete_subj_info(), aud_dup_record(), aud_evid_from_text(),
326                 aud_evid_to_text(), aud_free(), aud_get_all_evid(),
327                 aud_get_event(), aud_get_event_info(), aud_get_hdr(),
328                 aud_get_hdr_info(), aud_get_id(), aud_get_obj(),
329                 aud_get_obj_info(), aud_get_subj(), aud_get_subj_info(),
330                 aud_id_from_text(), aud_id_to_text(), aud_init_record(),
331                 aud_put_event(), aud_put_event_info(), aud_put_hdr(),
332                 aud_put_hdr_info(), aud_put_obj(), aud_put_obj_info(),
333                 aud_put_subj(), aud_put_subj_info(), aud_read(),
334                 aud_rec_to_text(), aud_size(), aud_switch(), aud_valid(),
335                 aud_write().

336     <sys/capability.h> cap_clear(), cap_copy_ext(), cap_copy_int(), cap_dup(),
337                 cap_free(), cap_from_text(), cap_get_fd(), cap_get_file(),
338                 cap_get_flag(), cap_get_proc(), cap_init(), cap_set_fd(),
339                 cap_set_file(), cap_set_flag(), cap_set_proc(), cap_size(),
340                 cap_to_text().
  
```

WITHDRAWN DRAFT. All Rights Reserved by IEEE.
 Preliminary—Subject to Revision.

```

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

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

```

349 => **2.8.2 Minimum Values (POSIX.1: line 983)** *Insert the following entry in*
 350 *Table 2-3 in alphabetical order:*

351	Name	Description	Value
359	{_POSIX_ACL_ENTRIES_MAX}	The maximum number of entries in an ACL for objects that support ACLs.	16
354			
355		Unspecified	
356		if {_POSIX_ACL} is not	
357		defined.	
358			

360 => **2.8.4 Run-Time Invariant Values (Possibly Indeterminate)
(POSIX.1: line 1023)** *Insert the following entry in Table 2-5 in alphabetical
order:*

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

368 ⇒ **2.8.5 Pathname Variable Values (POSIX.1: line 1044)** Insert the following
369 entries in alphabetical order in Table 2-6:

370 **Table 2-6 - Pathname Variable Values**

372	Name	Description	Minimum Value
379	{_POSIX_ACL_EXTENDED}	A value greater than zero if POSIX extended Access Control Lists are supported on the object; otherwise zero.	Zero
380	{_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
398	{_POSIX_CAP_PRESENT}	A value greater than zero if POSIX File Capability extensions are supported on the object; otherwise zero.	Zero
400	{_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
407	{_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

415 ⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications** |
416 **(POSIX.1: line 1122)** *Insert the following entries in Table 2-10 in alphabetical|*
417 *order:*

418 **Table 2-10 - Compile-Time Symbolic Constants**

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

1

Section 3: Revisions to Process Primitives

2 ⇒ **3.1.1.2 Process Creation — Description (POSIX.1: line 36)** *Insert the fol-*
3 *lowing lines after line 32 in Section 3.1.1.2:*

- 4 (1) If `{_POSIX_ACL}` is defined, the child process shall have its own copy of |
5 any ACL pointers and ACL entry descriptors in the parent, and any ACL |
6 working storage to which they refer. |
7 (2) If `{_POSIX_AUD}` is defined, the child process shall have its own copy of |
8 any audit record descriptors in the parent, and any audit working |
9 storage to which they refer. The audit state of the child, as set by |
10 `aud_switch()`, shall initially be the same as that of the parent; subse- |
11 quent calls to `aud_switch()` in either process shall not affect the audit |
12 state of the other process. |

13 ⇒ **3.1.2.2 Execute a File — Description (POSIX.1: line 153)** *Insert the follow-*
14 *ing at the end of the list of attributes inherited by the new process image on*
15 *exec() following line 153 in Section 3.1.2.2:*

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

17 ⇒ **3.1.2.2 Execute a File — Description (POSIX.1: line 168)** *Insert the follow-*
18 *ing paragraphs after line 168 in section 3.1.2.2:*

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

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

27 If `{_POSIX_CAP}` is defined, the new process image shall not inherit any capa- |
28 bility data objects nor any working storage associated with capabilities in the |
29 previous process image. |

30 If `{_POSIX_CAP}` is defined, the `exec()` functions shall modify the state of each |
31 of the capabilities of the process as follows, where I_1 , E_1 , and P_1 are respec- |
32 tively the inheritable, effective, and permitted flags of the new process image; |
33 I_0 is the inheritable flags of the current process image; and I_f , E_f and P_f are |
34 respectively the inheritable, effective, and permitted flags associated with the |
35 file being executed:

36 $I_1 = I_0$
37 $P_1 = (P_f \&& X) \parallel (I_f \&& I_0)$
38 $E_1 = E_f \&& P_1$

39 where X denotes possible additional implementation-defined restrictions. +
40 If `{_POSIX_INF}` is defined and `{_POSIX_INF_PRESENT}` is in effect for the |
41 file being executed, the information label of the process shall automatically be |
42 set to the same value as returned by `inf_float(file information label, process` |
43 `information label)`. If `{_POSIX_INF}` is defined but `{_POSIX_INF_PRESENT}` |
44 is not in effect for the file being executed, the information label of the process |
45 shall be set in an implementation defined manner.

46 ⇒ **3.3.1.3 Signal Actions — Description (POSIX.1: line 556)** *Insert the fol- |
47 lowing section before line 556:*

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

50 `inf_dominate()` `inf_equal()` `inf_set_fd()` `inf_set_file()`
51 `inf_set_proc()` `inf_size()`

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

54 `mac_dominate()` `mac_equal()` `mac_set_fd()` `mac_set_file()`
55 `mac_set_proc()` `mac_size()`

56 ⇒ **3.3.2.2 Send a Signal to a Process — Description (POSIX.1: line 594)**
57 *Insert the following sentence after the word "privileges":*

58 If `{_POSIX_CAP}` is defined, then appropriate privilege shall include |
59 `CAP_KILL`.

60 \Rightarrow **3.3.2.2 Send a Signal to a Process — Description (POSIX.1: line 616)**
61 *Insert the following after line 616:*

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

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

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

72 (3) If the MAC label of the receiving process dominates the MAC label of the
73 sending process (i.e., the signal is being written up), then it is implemen-
74 tation defined whether the sending process requires appropriate
75 privilege. If `{_POSIX_CAP}` is defined and appropriate privilege is
76 required, then appropriate privilege shall include `CAP_MAC_READ`.

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

81 \Rightarrow **3.3.2.4 Send a Signal to a Process — Errors (POSIX.1: line 625-628)**
82 *Replace lines 625-628 with the following:*

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

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

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

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

1

Section 4: Revisions to Process Environment

2 ⇒ **4.2.2.2 Set User and Group IDs — Description (POSIX.1: line 48)** *Insert*
3 *the following after line 48 of Section 4.2.2.2:*

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

6 ⇒ **4.2.2.2 Set User and Group IDs — Description (POSIX.1: line 52)** *Insert*
7 *the following after line 52 of Section 4.2.2.2:*

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

10 ⇒ **4.2.2.2 Set User and Group IDs — Description (POSIX.1: line 54)** *Insert*
11 *the following after line 54 of Section 4.2.2.2:*

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

14 ⇒ **4.2.2.2 Set User and Group IDs — Description (POSIX.1: line 58)** *Insert*
15 *the following after line 58 of Section 4.2.2.2:*

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

18 ⇒ **4.2.2.2 Set User and Group IDs — Description (POSIX.1: line 61)** *Insert*
19 *the following after line 61 of Section 4.2.2.2:*

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

22 ⇒ **4.2.2.2 Set User and Group IDs — Description (POSIX.1: line 64)** *Insert*
23 *the following after line 64 of Section 4.2.2.2:*

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

26 ⇒ **4.2.2.2 Set User and Group IDs — Description (POSIX.1: line 66)** *Insert*
27 *the following after line 66 of Section 4.2.2.2:*

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

30 ⇒ **4.2.2.2 Set User and Group IDs — Description (POSIX.1: line 69)** *Insert*
31 *the following after line 69 of Section 4.2.2.2:*

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

34 ⇒ **4.8.1.2 Get Configurable System Variables — Description (POSIX.1: line**
35 **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

2 ⇒ 5.3.1.2 Open a File — Description (POSIX.1: lines 192-194) Replace the
 3 sentence beginning “The file permission bits ...”, with the following:

4 If `{_POSIX_ACL}` is defined and `{_POSIX_ACL_EXTENDED}` is in effect for |
 5 the directory in which the file is being created (the “containing directory”) and –
 6 said directory has a default ACL, the following actions shall be performed:

- 7 (1) The default ACL of the containing directory is copied to the access
 8 ACL of the new file.
- 9 (2) Both the `ACL_USER_OBJ` ACL entry permission bits and the file
 10 owner class permission bits of the access ACL are set to the intersec- +
 11 tion of the default ACL’s `ACL_USER_OBJ` permission bits and the file
 12 owner class permission bits in *mode*. The action taken for any
 13 implementation-defined permissions that may be in the
 14 `ACL_USER_OBJ` entry shall be implementation-defined.
- 15 (3) If the default ACL does not contain an `ACL_MASK` entry, both the
 16 `ACL_GROUP_OBJ` ACL entry permission bits and the file group class
 17 permission bits of the access ACL are set to the intersection of the +
 18 default ACL’s `ACL_GROUP_OBJ` permission bits and the file group
 19 class permission bits in *mode*. The action taken for any
 20 implementation-defined permissions that may be in the
 21 `ACL_GROUP_OBJ` entry shall be implementation-defined.
- 22 (4) If the default ACL contains an `ACL_MASK` entry, both the
 23 `ACL_MASK` ACL entry permission bits and the file group class per-
 24 mission bits of the access ACL are set to the intersection of the default+
 25 ACL’s `ACL_MASK` permission bits and the file group class permission
 26 bits in *mode*. The action taken for any implementation-defined per-
 27 missions that may be in the `ACL_MASK` entry shall be
 28 implementation-defined.
- 29 (5) Both the `ACL_OTHER` ACL entry permission bits and the file other
 30 class permission bits of the access ACL are set to the intersection of +
 31 the default ACL’s `ACL_OTHER` permission bits and the file other
 32 class permission bits in *mode*. The action taken for any
 33 implementation-defined permissions that may be in the `ACL_OTHER`
 34 entry shall be implementation-defined.

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

42 If `{_POSIX_ACL}` is not defined, or `{_POSIX_ACL_EXTENDED}` is not in effect
43 for the directory in which the file is being created (the "containing directory"), |
44 or said directory does not have a default ACL, the file permission bits (see
45 5.6.1) shall be set to the value of *mode* except those set in the file mode crea-
46 tion mask of the process (see 5.3.3). In any of these cases (default ACL,
47 implementation-defined default ACL, or file permission bits), access control
48 decisions shall not be made on the newly created file until all access control
49 information has been associated with the file.

50 ⇒ **5.3.1.2 Open a File — Description (POSIX.1: line 197)** *Insert the following*
51 *lines after line 197 in Section 5.3.1.2:*

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

58 ⇒ **5.3.1.2 Open a File — Description (POSIX.1: line 234)** *Insert the following*
59 *sentences after line 234 in Section 5.3.1.2:*

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

63 ⇒ **5.3.1.2 Open a File — Description (POSIX.1: line 240)** *Insert the following*
64 *paragraph after line 240 in Section 5.3.1.2:*

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

69 ⇒ **5.3.4.2 Link a File — Description (POSIX.1: line 331)** *Insert the following*
70 *sentence:*

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

73 ⇒ **5.3.4.2 Link a File — Description (POSIX.1: line 336)** *Insert the following*
74 *paragraph after line 336 in Section 5.3.4.2:*

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

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

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

82 ⇒ **5.3.4.4 Link a File — Errors (POSIX.1: line 347)** *Insert the following after*
83 *the line:*

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

87 ⇒ **5.4.1.2 Make a Directory — Description (POSIX.1: lines 378-380)** *Replace*
88 *the second and third sentences of the paragraph with the following:*

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

- 93 (1) The default ACL of the containing directory is copied to both the
94 access ACL and the default ACL of the new directory.
- 95 (2) Both the `ACL_USER_OBJ` ACL entry permission bits and the file
96 owner class permission bits of the access ACL are set to the intersec- +
97 tion of the default ACL's `ACL_USER_OBJ` permission bits and the file
98 owner class permission bits in *mode*. The action taken for any
99 implementation-defined permissions that may be in the
100 `ACL_USER_OBJ` entry shall be implementation-defined.
- 101 (3) If the default ACL does not contain an `ACL_MASK` entry, both the
102 `ACL_GROUP_OBJ` ACL entry permission bits and the file group class
103 permission bits of the access ACL are set to the intersection of the +

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

108 (4) If the default ACL contains an ACL_MASK entry, both the
109 ACL_MASK ACL entry permission bits and the file group class per-
110 mission bits of the access ACL are set to the intersection of the default+
111 ACL's ACL_MASK permission bits and the file group class permission
112 bits in *mode*. The action taken for any implementation-defined per-
113 missions that may be in the ACL_MASK entry shall be
114 implementation-defined.

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

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

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

136 ⇒ **5.4.1.2 Make a Directory — Description (POSIX.1: line 385)** *Insert the fol-*
137 *lowing paragraphs after line 385 in Section 5.4.1.2:*

138 If `{_POSIX_MAC}` is defined and `{_POSIX_MAC_PRESENT}` is in effect for the|
139 containing directory and the directory is created, the MAC label of the newly
140 created directory shall be equivalent to the MAC label of the calling process.
141 If `{_POSIX_MAC}` is defined, the calling process shall require MAC write |
142 access to the containing directory.

143 ⇒ **5.4.2.2 Make a FIFO Special File — Description (POSIX.1: lines 426-428)**

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

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

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

150 (2) Both the `ACL_USER_OBJ` ACL entry permission bits and the file
151 owner class permission bits of the access ACL are set to the inter-
152 section of the default ACL's `ACL_USER_OBJ` permission bits and
153 the file owner class permission bits in *mode*. The action taken for
154 any implementation-defined permissions that may be in the
155 `ACL_USER_OBJ` entry shall be implementation-defined.

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

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

170 (5) Both the `ACL_OTHER` ACL entry permission bits and the file
171 other class permission bits of the access ACL are set to the inter-
172 section of the default ACL's `ACL_OTHER` permission bits and the
173 file other class permission bits in *mode*. The action taken for any
174 implementation-defined permissions that may be in the
175 `ACL_OTHER` entry shall be implementation-defined.

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

183 If `{_POSIX_ACL}` is not defined or `{_POSIX_ACL_EXTENDED}` is not in effect
184 for the directory in which the file is being created (the "containing directory"),
185 or said directory does not have a default ACL, the file permission bits of the
186 new FIFO are initialized from *mode*. The file permission bits of the *mode*

187 argument are modified by the file creation mask of the process (see 5.3.3).

188 ⇒ **5.4.2.2 Make a FIFO Special File — Description (POSIX.1: lines 432)**
189 *Insert the following paragraphs after line 432 in Section 5.4.2.2:*

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

195 If `{_POSIX_INF}` is defined and `{_POSIX_INF_PRESENT}` is in effect for the %
196 file path, and the special file is created, then the information label of the spe-
197 cial file shall automatically be set to a value which dominates the value
198 returned by `inf_default()`.

199 ⇒ **5.5.1.2 Remove Directory Entries — Description (POSIX.1: line 474)**
200 *Insert the following paragraphs:*

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

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

205 ⇒ **5.5.1.4 Remove Directory Entries — Errors (POSIX.1: line 487)** *Insert*
206 *the following phrase at the end of the line:*

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

209 ⇒ **5.5.2.2 Remove a Directory — Description (POSIX.1: line 520)** *Insert the*
210 *following paragraph after line 520:*

211 If `{_POSIX_MAC}` is defined, the calling process shall have MAC write access |
212 to the parent directory of the directory being removed. If `{_POSIX_MAC}` is |
213 defined, the calling process shall have MAC read access to the parent directory |
214 of the directory being removed.

215 ⇒ **5.5.2.4 Remove a Directory — Errors (POSIX.1: line 532)** *Insert the fol-*
216 *lowing phrase at the end of the line:*

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

220 ⇒ **5.5.3.2 Rename a File — Description (POSIX.1: line 583)** *Insert the follow-*
221 *ing paragraph after line 566:*

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

226 ⇒ **5.6.2.2 Get File Status — Description (POSIX.1: line 726)** *Insert the fol-*
227 *lowing sentence:*

228 If `{_POSIX_ACL}` is defined, and `{_POSIX_ACL_EXTENDED}` is in effect for |
229 the pathname, and the access ACL contains an `ACL_MASK` entry, then the file |
230 group class permission bits represent the `ACL_MASK` access ACL entry file |
231 permission bits. If `{_POSIX_ACL}` is defined, and `{_POSIX_ACL_EXTENDED}` |
232 is in effect for the pathname, and the access ACL does not contain an |
233 `ACL_MASK` entry, then the file group class permission bits represent the |
234 `ACL_GROUP_OBJ` access ACL entry file permission bits.

235 ⇒ **5.6.2.2 Get File Status — Description (POSIX.1: line 727)** *Insert the fol-*
236 *lowing:*

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

240 ⇒ **5.6.2.4 Get File Status — Errors (POSIX.1: line 738)** *Insert the following*
241 *phrase at the end of this line:*

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

243 ⇒ **5.6.4.2 Change File Modes — Description (POSIX.1: line 802)** *Insert the*
244 *following sentence in line 802 of Section 5.6.4.2:*

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

247 ⇒ **5.6.4.2 Change File Modes — Description (POSIX.1: line 804)** *Insert the*
248 *following sentence in line 804:*

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

252 ⇒ **5.6.4.2 Change File Modes — Description (POSIX.1: line 805)** *Insert the*
253 *following paragraph after this line:*

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

- 256 (1) The ACL_USER_OBJ access ACL entry permission bits shall be set equal+
257 to the file owner class permission bits.
- 258 (2) If an ACL_MASK entry is not present in the access ACL, then the +
259 ACL_GROUP_OBJ access ACL entry permission bits shall be set equal to+
260 the file group class permission bits. Otherwise, the ACL_MASK access +
261 ACL entry permission bits shall be set equal to the file group class per-
262 mission bits, and the ACL_GROUP_OBJ access ACL entry permission +
263 bits shall remain unchanged.
- 264 (3) The ACL_OTHER access ACL entry permission bits shall be set equal to +
265 the file other class permission bits.

266 ⇒ **5.6.4.2 Change File Modes — Description (POSIX.1: line 809)** *Insert the*
267 *following sentence after this line:*

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

- 270 ⇒ **5.6.4.2 Change File Modes — Description (POSIX.1: line 811)** *Insert the*
271 *following sentence after line 811 of Section 5.6.4.2:*
- 272 If {_POSIX_MAC} is defined, the calling process shall have MAC write access |
273 to the file.
- 274 ⇒ **5.6.4.2 Change File Modes — Errors (POSIX.1: line 821)** *Insert the follow-*
275 *ing phrase at the end of this line:*
- 276 or {_POSIX_MAC} is defined and MAC write access to the target file is denied. |
- 277 ⇒ **5.6.5.2 Change Owner and Group of a File — Description (POSIX.1: line**
278 **844)** *Insert the following sentence in this line:*
- 279 If {_POSIX_CAP} is defined, then appropriate privilege shall include the |
280 CAP_FOWNER capability.
- 281 ⇒ **5.6.5.2 Change Owner and Group of a File — Description (POSIX.1: line**
282 **847)** *Insert the following sentence after this line:*
- 283 If {_POSIX_CAP} is defined, then appropriate privilege shall include the |
284 CAP_CHOWN capability.
- 285 ⇒ **5.6.5.2 Change Owner and Group of a File — Description (POSIX.1: line**
286 **856)** *Insert the following sentence after the word "altered":*
- 287 If {_POSIX_CAP} is defined, then appropriate privilege shall include the |
288 CAP_FSETID capability.
- 289 ⇒ **5.6.5.2 Change Owner and Group of a File — Description (POSIX.1: line**
290 **858)** *Insert the following paragraph after line 858:*
- 291 If {_POSIX_MAC} is defined, the calling process shall have MAC write access |
292 to the file.

293 ⇒ **5.6.5.4 Change Owner and Group of a File — Errors (POSIX.1: line 868)**
294 *Insert the following phrase at the end of this line:*

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

296 ⇒ **5.6.5.4 Change Owner and Group of a File — Errors (POSIX.1: line 879)** |
297 *Insert the following sentences after this line:* |

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

303 ⇒ **5.6.6.2 Set File Access and Modification Times — Description**
304 (**POSIX.1: line 899**) *Insert the following sentence after this line:*

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

307 ⇒ **5.6.6.2 Set File Access and Modification Times — Description**
308 (**POSIX.1: line 899**) *Insert the following paragraph after this:*

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

311 ⇒ **5.6.6.2 Set File Access and Modification Times — Description**
312 (**POSIX.1: line 903**) *Insert the following sentence after this line:*

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

315 ⇒ **5.6.6.4 Set File Access and Modification Times — Errors (POSIX.1: line**
316 **927)** *Insert the following phrase at the end of this line:*

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

318 ⇒ **5.7.1.3 Get Configurable Pathname Variables — Returns (POSIX.1: line**
319 **965)** Add the following variables to Table 5-2:

§20	Variable	name Value	Notes
322	{_POSIX_ACL_EXTENDED}	{_PC_ACL_EXTENDED}	(7)
323	{_POSIX_ACL_PATH_MAX}	{_PC_ACL_MAX}	(7)
324	{_POSIX_CAP_PRESENT}	{_PC_CAP_PRESENT}	(7)
325	{_POSIX_MAC_PRESENT}	{_PC_MAC_PRESENT}	(7)
326	{_POSIX_INF_PRESENT}	{_PC_INF_PRESENT}	(7)

1 **Section 6: Revisions to Input and Output Primitives**

2 ⇒ **6.1.1.2 Create an Inter-Process Channel — Description (POSIX.1: line**
3 **21)** *Insert the following paragraphs after this line:*

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

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

10 ⇒ **6.4.1.2 Read from a File — Description (POSIX.1: line 158)** *Insert the fol-*
11 *lowing paragraph after this line:*

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

16 ⇒ **6.4.2.2 Write to a File — Description (POSIX.1: line 261)** *Insert the follow-*
17 *ing paragraph after this line:*

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

1 **Section 8: Revisions to C Programming Language Specific Services**

2 **⇒ 8.2.3 Interactions of Other File Type C Functions (POSIX.1: line 345)**

3 *Insert the following sentence after line 345:*

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

1

Section 23: Access Control Lists

2 23.1 General Overview

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

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

- 11 (1) Definition and use of access and default ACLs
- 12 (2) Definition of initial access permissions on object creation
- 13 (3) Specification of the access check algorithm
- 14 (4) Functions to manipulate ACLs.

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

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

32 **23.1.1 ACL Entry Composition**

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

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

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

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

38 A conforming implementation may add implementation-defined pieces of informa-
39 tion to an ACL entry.

40 A conforming ACL implementation shall define the following tag types:

41 — ACL_GROUP: an ACL entry of tag type ACL_GROUP denotes discretion-
42 ary access rights for processes whose effective group ID or any supplemen-
43 tal group IDs match the ACL entry qualifier

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

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

50 — ACL_OTHER: an ACL entry of tag type ACL_OTHER denotes discretion-
51 ary access rights for processes whose attributes do not match any other entry in|
52 the ACL

53 — ACL_USER: an ACL entry of tag type ACL_USER denotes discretion-
54 ary access rights for processes whose effective user ID matches the ACL entry
55 qualifier

56 — ACL_USER_OBJ: an ACL entry of tag type ACL_USER_OBJ denotes dis-
57 cretionary access rights for processes whose effective user ID matches the |
58 user ID of the owner of the file.

59 A conforming implementation may define additional tag types.

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

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

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

72 minimum ACL is called an *extended ACL*.
73 An ACL that contains ACL_GROUP, ACL_USER, or implementation-defined ACL
74 entries in the file group class shall contain exactly one ACL_MASK entry. If an
75 ACL does not contain ACL_GROUP, ACL_USER, or implementation-defined ACL
76 entries in the file group class, then the ACL_MASK entry shall be optional.
77 The qualifier field associated with the POSIX.1e ACL facility defined tag types
78 shall not be extended to contain any implementation-defined information. The
79 qualifier field associated with implementation-defined tag types may contain fully
80 implementation-defined information. The qualifier field shall be unique among
81 all entries of the same POSIX.1e ACL facility defined tag type in a given ACL.
82 For entries of the ACL_USER and ACL_GROUP tag type, the qualifier field shall
83 be present and contain either a user ID or a group ID respectively. The value of
84 the qualifier field in entries of tag types ACL_GROUP_OBJ, ACL_MASK,
85 ACL_OTHER, and ACL_USER_OBJ shall be unspecified.
86 The set of discretionary access permissions shall, at a minimum, include: read,
87 write, and execute/search. Additional permissions may be added and shall be +
88 implementation-defined.

89 23.1.2 Relationship with File Permission Bits

90 ACL interfaces extend the file permission bit interfaces to provide a finer granu- |
91 larity of access control than is possible with permission bits alone. As a superset |
92 of the file permission bit interface, the ACL functionality specified preserves com- |
93 patibility with applications using POSIX.1 interfaces to retrieve and manipulate |
94 access permission bits, e.g., *chmod()*, *creat()*, and *stat()*.
95 The file permission bits shall correspond to three entries in an ACL. The permis- |
96 sions specified by the file owner class permission bits correspond to the permis- |
97 sions associated with the ACL_USER_OBJ entry. The permissions specified by |
98 the file group class permission bits correspond to the permissions associated with |
99 the ACL_GROUP_OBJ entry or the permissions associated with the ACL_MASK |
100 entry if the ACL contains an ACL_MASK entry. The permissions specified by |
101 the file other class permission bits correspond to the permissions associated with the |
102 ACL_OTHER entry.
103 The permissions associated with these ACL entries shall be identical to the per-
104 missions defined for the corresponding file permission bits. Modification of the
105 permissions associated with these ACL entries shall modify the corresponding file
106 permission bits and modification of the file permission bits shall modify the per-
107 missions of the corresponding ACL entries.
108 When the file permissions of an object are modified, e.g. using the *chmod()* func-
109 tion, then:
110 (1) the corresponding permissions associated with the ACL_USER_OBJ
111 entry shall be set equal to each of the file owner class permission bits
112 (2) if the ACL does not contain an ACL_MASK entry, then the corresponding
113 permissions associated with the ACL_GROUP_OBJ entry shall be set

- 114 equal to each of the file group class permission bits
- 115 (3) if the ACL contains an ACL_MASK entry, then the corresponding per-
 116 missions associated with the ACL_MASK entry shall be set equal to each
 117 of the file group class permission bits and the permissions associated
 118 with the ACL_GROUP_OBJ entry shall not be modified.
- 119 (4) the corresponding permissions associated with the ACL_OTHER entry
 120 shall be set equal to each of the file other class permission bits

121 **23.1.3 Default ACLs**

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

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

140 **23.1.4 Associating an ACL with an Object at Object Creation Time**

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

- 144 (1) The *mode* parameter to functions which can create objects may be used |
 145 by an application to specify the maximum discretionary access permis-
 146 sions to be associated with the resulting object. There are four POSIX.1
 147 functions which can be used to create objects: *creat()*, *mkdir()*, *mkfifo()*,
 148 and *open()* (with the O_CREAT flag).
- 149 (2) The default ACL may be used by the owner of a directory to specify the
 150 maximum discretionary access permissions to be associated with objects
 151 created within that directory.

152 The initial access control information is obtained as is specified in 5.3 and 5.4. -
 153 Implementations may provide an additional default ACL that is applied if a

154 default ACL as defined by this standard is not present. Such an implementation-
155 defined default ACL interface may apply different access and/or default ACLs to
156 created objects based upon implementation-defined criteria.

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

159 **23.1.5 ACL Access Check Algorithm**

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

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

167 (1) the ACL_USER_OBJ entry

168 (2) any ACL_USER entries

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

170 (4) the ACL_OTHER entry

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

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

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

181 (1) **If** the effective user ID of the process matches the user ID of the
182 object owner

183 **then**

184 set matched entry to ACL_USER_OBJ entry

185 (2) **else if** the effective user ID of the process matches the user ID
186 specified in any ACL_USER tag type ACL entry,

187 **then**

188 set matched entry to the matching ACL_USER entry

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

193 **then**

```

194     if the requested access modes are granted by at least one entry
195         matched by the effective group ID or any of the supplementary
196         group IDs of the process
197     then
198         set matched entry to a granting entry
199     else
200         access is denied
201     endif
202 (4) else if the requested access modes are granted by the ACL_OTHER |
203     entry of the ACL,
204     then
205         set matched entry to the ACL_OTHER entry
206     endif
207 (5) If the requested access modes are granted by the matched entry
208     then
209         if the matched entry is an ACL_USER_OBJ or ACL_OTHER
210         entry
211         then
212             access is granted
213
214         else if the requested access modes are also granted by the
215         ACL_MASK entry or no ACL_MASK entry exists in the ACL
216         then
217             access is granted
218         else
219             access is denied
220         endif
221     else
222         access is denied
223 endif

```

223 23.1.6 ACL Functions

224 Functional interfaces are defined to manipulate ACLs and ACL entries. The func-
 225 tions provide a portable interface for editing and manipulating the entries within
 226 an ACL and the fields within an ACL entry.

227 Four groups of functions are defined to:

- 228 (1) manage the ACL working storage area
- 229 (2) manipulate ACL entries
- 230 (3) manipulate an ACL on an object
- 231 (4) translate an ACL into different formats.

232 **23.1.6.1 ACL Storage Management**

233 These functions manage the storage areas used to contain working copies of
234 ACLs. An ACL in working storage shall not be used in any access control deci-
235 sions.

- 236 *acl_dup()* Duplicates an ACL in a working storage area
237 *acl_free()* Release the working storage area allocated to an ACL data
238 object
239 *acl_init()* Allocates and initializes an ACL working storage area

240 **23.1.6.2 ACL Entry Manipulation**

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

- 243 (1) Functions that manipulate complete entries in an ACL:
244 *acl_copy_entry()* Copies an ACL entry to another ACL entry
245 *acl_create_entry()* Creates a new entry in an ACL
246 *acl_delete_entry()* Deletes an entry from an ACL
247 *acl_get_entry()* Returns a descriptor to an ACL entry
248 *acl_valid()* Validates an ACL by checking for duplicate, miss-
249 ing, and ill-formed entries
- 250 (2) Functions that manipulate permissions within an ACL entry:
251 *acl_add_perm()* Adds a permission to a given permission set
252 *acl_calc_mask()* Sets the permission granted by the ACL_MASK
253 entry to the maximum permissions granted by the
254 ACL_GROUP, ACL_GROUP_OBJ, ACL_USER and
255 implementation-defined ACL entries
256 *acl_clear_perms()* Clears all permissions from a given permission set
257 *acl_delete_perm()* Deletes a permission from a given permission set
258 *acl_get_permset()* Returns the permissions in a given ACL entry
259 *acl_set_permset()* Sets the permissions in a given ACL entry
- 260 (3) Functions that manipulate the tag type and qualifier in an ACL entry:
261 *acl_get_qualifier()* Returns the qualifier in a given ACL entry
262 *acl_get_tag_type()* Returns the tag type in a given ACL entry
263 *acl_set_qualifier()* Sets the qualifier in a given ACL entry
264 *acl_set_tag_type()* Sets the tag type in a given ACL entry

265 **23.1.6.3 ACL Manipulation on an Object**

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

269	<i>acl_delete_def_file()</i>	Deletes the default ACL associated with an object
270	<i>acl_get_fd()</i>	Reads the contents of an access ACL associated with a file descriptor into working storage
272	<i>acl_get_file()</i>	Reads the contents of an access ACL or default ACL associated with an object into working storage
274	<i>acl_set_fd()</i>	Writes the ACL in working storage to the object associated with a file descriptor as an access ACL
276	<i>acl_set_file()</i>	Writes the ACL in working storage to an object as an access ACL or default ACL
277		

278 **23.1.6.4 ACL Format Translation**

279 The standard defines three different representations for ACLs:

280 *external form* The exportable, contiguous, persistent representation of an
281 ACL in user-managed space

282 *internal form* The internal representation of an ACL in working storage

283 *text form* The structured text representation of an ACL

284 These functions translate an ACL from one representation into another.

285 *acl_copy_ext()* Translates an internal form of an ACL to an external form of
286 an ACL

287 *acl_copy_int()* Translates an external form of an ACL to an internal form of
288 an ACL

289 *acl_from_text()* Translates a text form of an ACL to an internal form of an
290 ACL

291 *acl_size()* Returns the size in bytes required to store the external form
292 of an ACL that is the result of an *acl_copy_ext()*

293 *acl_to_text()* Translates an internal form of an ACL to a text form of an
294 ACL

295 **23.1.7 POSIX.1 Functions Covered by ACLs**

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

	Existing Function	POSIX.1 Section
298	<i>access()</i>	5.6.3
300	<i>chmod()</i>	5.6.4
301	<i>creat()</i>	5.3.2
302	<i>fstat()</i>	5.6.2
303	<i>mkdir()</i>	5.4.1
304	<i>mkfifo()</i>	5.4.2
305	<i>open()</i>	5.3.1
306	<i>stat()</i>	5.6.2
307		
308		

309 **23.2 Header**

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

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

Table 23-1 – ACL Data Types

	Defined Type	Description
315	<i>acl_entry_t</i>	Used as a descriptor for a specific ACL entry in ACL working storage. This data type is non-exportable data.
316	<i>acl_perm_t</i>	Used for individual object access permissions. This data type is exportable data.
317	<i>acl_permset_t</i>	Used for the set of object access permissions. This data type is non-exportable data.
318	<i>acl_t</i>	Used as a pointer to an ACL in ACL working storage. This data type is non-exportable data.
319	<i>acl_tag_t</i>	Used to distinguish different types of ACL entries. This data type is exportable data.
320	<i>acl_type_t</i>	Used to distinguish different types of ACLs (e.g., access, default). This data type is exportable data.
321		
322		
323		
324		
325		
326		
327		
328		
329		
330		
331		
332		
333		
334		

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

337 **23.2.1 *acl_entry_t***

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

341 **23.2.2 acl_perm_t**

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

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

346 **Table 23-2 – acl_perm_t Values**

347	Constant	Description
349	ACL_EXECUTE	ACL execute permission
350	ACL_READ	ACL read permission
351	ACL_WRITE	ACL write permission

352 These constants shall be implementation-defined unique values.

353 **23.2.3 acl_permset_t**

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

357 **23.2.4 acl_t**

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

360 **23.2.5 acl_tag_t**

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

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

364

Table 23-3 – acl_tag_t Values

365

367

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

371
372
373
374
375
376
377
378
379
380
381
382
383
384 These constants shall be implementation-defined unique values.385 **23.2.6 acl_type_t**386 This typedef shall define a data type capable of storing an individual ACL type.
387 Table 23-4 contains *acl_type_t* values for *acl_get_file()* and *acl_set_file()*.

388

Table 23-4 – acl_type_t Values

389

390

391

392

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

393 These constants shall be implementation-defined unique values.

394 **23.2.7 ACL Qualifier**395 Table 23-5 contains the value of undefined user IDs or group IDs for the ACL |
396 qualifier.

397

Table 23-5 – ACL Qualifier Constants

398

399

400

Constant	Description
ACL_UNDEFINED_ID	Undefined ID

401 These constants shall be implementation-defined values.

402 **23.2.8 ACL Entry**

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

405 **Table 23-6 – ACL Entry Constants**

406 Constant	407 Description
408 <code>ACL_FIRST_ENTRY</code>	409 Return the first ACL entry in the ACL.
409 <code>ACL_NEXT_ENTRY</code>	410 Return the next ACL entry in the ACL.

410 These constants shall be implementation-defined values.

411 **23.3 Text Form Representation**

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

416 **23.3.1 Long Text Form for ACLs**

417 The long text form is used for either input or output of ACLs and is defined as fol-
418 lows:

419 $\langle acl_entry \rangle$
420 $[\langle acl_entry \rangle] \dots$

421 Each $\langle acl_entry \rangle$ line shall contain one ACL entry with three required colon-
422 separated fields: an ACL entry tag type, an ACL entry qualifier, and the disre-
423 tionary access permissions. An implementation may define additional colon-
424 separated fields after the required fields. Comments may be included on any
425 $\langle acl_entry \rangle$ line. If a comment starts at the beginning of a line, then the entire
426 line shall be interpreted as a comment.

427 The first field contains the ACL entry tag type. This standard defines the follow-
428 ing ACL entry tag type keywords, one of which shall appear in the first field:

- 429 `user` A user ACL entry specifies the access granted to either the file
430 owner or a specified user.
431 `group` An group ACL entry specifies the access granted to either the file
432 owning group or a specified group.
433 `other` An other ACL entry specifies the access granted to any process
434 that does not match any user, group, or implementation-defined
435 ACL entries.
436 `mask` A mask ACL entry specifies the maximum access which can be
437 granted by any ACL entry except the user entry for the file owner
438 and the other entry.

439 An implementation may define additional ACL entry types.

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

442 *uid* This qualifier specifies a user name or a user ID number.

443 *gid* This qualifier specifies a group name or a group ID number.

444 *empty* This qualifier specifies that no *uid* or *gid* information is to be applied
445 to the ACL entry. An *empty* qualifier shall be represented by an
446 empty string or by white space.

447 An implementation may define additional qualifiers.

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

450 r Read access

451 w Write access

452 x Execute/search access

453 - No access by this ACL entry.

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

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

464 A group entry with an *empty* qualifier shall specify the access granted to the file
465 owning group. A group entry with a *gid* qualifier shall specify the access permis-
466 sions granted to the group name matching the *gid* value. If the *gid* value does not
467 match a group name, then the ACL entry shall specify the access permissions
468 granted to the group ID matching the numeric *gid* value. +

469 The mask and other entries shall contain an *empty* qualifier. An implementa-
470 tion may define additional ACL entry types that use the *empty* qualifier.

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

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

479 White space is permitted in *<acl_entry>* lines as follows: at the start of the line; –
480 immediately before and after a “.” separator; immediately before the first
481 number-sign (#) character; at any point after the first number-sign (#) character.
482 Comments shall have no effect on the discretionary access check of the object with
483 which they are associated. An implementation shall define whether or not com-
484 ments are stored with an ACL.
485 If an implementation allows the colon character “:” to be present in an ACL entry
486 qualifier, then that implementation shall provide a method for distinguishing
487 between a colon character as a field separator in an ACL entry definition and a
488 colon character as a component of the ACL entry qualifier value. –

489 **23.3.2 Short Text Form for ACLs**

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

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

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

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

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

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

504 **23.4 Functions**

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

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

513 **23.4.1 Add a Permission to an ACL Permission Set**

514 Function: *acl_add_perm()*

515 **23.4.1.1 Synopsis**

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

518 **23.4.1.2 Description**

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

523 Any existing descriptors that refer to *permset_d* shall continue to refer to that per-
524 mission set.

525 **23.4.1.3 Returns**

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

528 **23.4.1.4 Errors**

529 If any of the following conditions occur, the *acl_add_perm()* function shall return
530 -1 and set *errno* to the corresponding value:

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

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

534 **23.4.1.5 Cross-References**

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

537 **23.4.2 Calculate the File Group Class Mask**

538 Function: *acl_calc_mask()*

539 **23.4.2.1 Synopsis**

```
540 #include <sys/acl.h>
541 int acl_calc_mask (acl_t *acl_p);
```

542 **23.4.2.2 Description**

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

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

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

559 **23.4.2.3 Returns**

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

562 **23.4.2.4 Errors**

563 If any of the following conditions occur, the *acl_calc_mask()* function shall return
564 -1 and set *errno* to the corresponding value:

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

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

568 **23.4.2.5 Cross-References**

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

570 **23.4.3 Clear All Permissions from an ACL Permission Set**

571 Function: *acl_clear_perms()*

572 **23.4.3.1 Synopsis**

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

575 **23.4.3.2 Description**

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

578 Any existing descriptors that refer to *permset_d* shall continue to refer to that per-
579 mission set.

580 **23.4.3.3 Returns**

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

583 **23.4.3.4 Errors**

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

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

588 **23.4.3.5 Cross-References**

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

591 **23.4.4 Copy an ACL Entry**

592 Function: *acl_copy_entry()*

593 **23.4.4.1 Synopsis**

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

596 **23.4.4.2 Description**

597 The *acl_copy_entry()* function shall copy the contents of the ACL entry indicated
598 by the *src_d* descriptor to the existing ACL entry indicated by the *dest_d* descrip-
599 tor. The *src_d* and *dest_d* descriptors may refer to entries in different ACLs.
600 The *src_d*, *dest_d* and any other ACL entry descriptors that refer to entries in +
601 either ACL shall continue to refer to those entries. The order of all existing
602 entries in both ACLs shall remain unchanged.

603 **23.4.4.3 Returns**

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

606 **23.4.4.4 Errors**

607 If any of the following conditions occur, the *acl_copy_entry()* function shall return
608 -1 and set *errno* to the corresponding value:

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

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

612 **23.4.4.5 Cross-References**

613 *acl_get_entry()*, 23.4.14.

614 **23.4.5 Copy an ACL From System to User Space**

615 Function: *acl_copy_ext()*

616 **23.4.5.1 Synopsis**

```
617 #include <sys/acl.h>
618 ssize_t acl_copy_ext (void *buf_p, acl_t acl, ssize_t size);
```

619 **23.4.5.2 Description**

620 The *acl_copy_ext()* function shall copy an ACL, pointed to by *acl*, from system- |
621 managed space to the user managed space pointed to by *buf_p*. The *size* parame- |
622 ter represents the size in bytes of the buffer pointed to by *buf_p*. The format of the |
623 ACL placed in the user-managed space pointed to by *buf_p* shall be a contiguous, |
624 persistent data item, the format of which is unspecified. It is the responsibility of |
625 the invoker to allocate an area large enough to hold the copied ACL. The size of |
626 the exportable, contiguous, persistent form of the ACL may be obtained by invok- |
627 ing the *acl_size()* function.

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

631 **23.4.5.3 Returns**

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

635 **23.4.5.4 Errors**

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

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

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

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

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

645 **23.4.5.5 Cross-References**

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

647 **23.4.6 Copy an ACL From User to System Space**

648 Function: *acl_copy_int()*

649 **23.4.6.1 Synopsis**

650 #include <sys/acl.h>
651 acl_t *acl_copy_int* (const void **buf_p*);

652 **23.4.6.2 Description**

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

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

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

661 **23.4.6.3 Returns**

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

665 **23.4.6.4 Errors**

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

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

670 [ENOMEM] The ACL working storage requires more memory than is allowed
671 by the hardware or system-imposed memory management con-
672 straints.

673 **23.4.6.5 Cross-References**

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

675 **23.4.7 Create a New ACL Entry**

676 Function: *acl_create_entry()*

677 **23.4.7.1 Synopsis**

```
678 #include <sys/acl.h>
679 int acl_create_entry (acl_t *acl_p, acl_entry_t *entry_p);
```

680 **23.4.7.2 Description**

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

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

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

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

697 **23.4.7.3 Returns**

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

700 **23.4.7.4 Errors**

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

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

704 [ENOMEM] The ACL working storage requires more memory than is allowed
705 by the hardware or system-imposed memory management con-
706 straints.

707 **23.4.7.5 Cross-References**

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

709 **23.4.8 Delete a Default ACL by Filename**

710 Function: *acl_delete_def_file()*

711 **23.4.8.1 Synopsis**

```
712 #include <sys/acl.h>
713 int acl_delete_def_file (const char *path_p);
```

714 **23.4.8.2 Description**

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

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

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

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

729 **23.4.8.3 Returns**

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

732 **23.4.8.4 Errors**

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

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

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

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

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

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

747 Argument *path_p* does not refer to a directory.

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

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

752 **23.4.8.5 Cross-References**

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

754 **23.4.9 Delete an ACL Entry**

755 Function: *acl_delete_entry()*

756 **23.4.9.1 Synopsis**

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

759 **23.4.9.2 Description**

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

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

767 **23.4.9.3 Returns**

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

770 **23.4.9.4 Errors**

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

773 [EINVAL] Argument *acl* does not point to a valid ACL. Argument *entry_d* +
774 is not a valid descriptor for an ACL entry in *acl*.
775 [ENOSYS] This function is not supported by the implementation.

776 **23.4.9.5 Cross-References**

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

778 **23.4.10 Delete Permissions from an ACL Permission Set**

779 Function: *acl_delete_perm()*

780 **23.4.10.1 Synopsis**

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

783 **23.4.10.2 Description**

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

788 Any existing descriptors that refer to *permset_d* shall continue to refer to that per-
789 mission set.

790 **23.4.10.3 Returns**

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

793 **23.4.10.4 Errors**

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

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

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

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

800 **23.4.10.5 Cross-References**

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

803 **23.4.11 Duplicate an ACL**

804 Function: *acl_dup()*

805 **23.4.11.1 Synopsis**

```
806 #include <sys/acl.h>
807 acl_t acl_dup (acl_t acl);
```

808 **23.4.11.2 Description**

809 The *acl_dup()* function returns a pointer to a copy of the ACL pointed to by argu-
810 ment *acl*.

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

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

816 **23.4.11.3 Returns**

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

820 **23.4.11.4 Errors**

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

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

824 [ENOMEM] The ACL working storage requires more memory than is allowed
825 by the hardware or system-imposed memory management con-
826 straints.

827 **23.4.11.5 Cross-References**

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

829 **23.4.12 Release Memory Allocated to an ACL Data Object**

830 Function: *acl_free()*

831 **23.4.12.1 Synopsis**

```
832 #include <sys/acl.h>
833 int acl_free (void *obj_p);
```

834 **23.4.12.2 Description**

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

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

843 **23.4.12.3 Returns**

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

846 **23.4.12.4 Errors**

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

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

850 **23.4.12.5 Cross-References**

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

854 **23.4.13 Create an ACL from Text**

855 Function: *acl_from_text()*

856 **23.4.13.1 Synopsis**

857 #include <sys/acl.h>
858 acl_t *acl_from_text* (const char **buf_p*);

859 **23.4.13.2 Description**

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

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

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

870 **23.4.13.3 Returns**

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

874 **23.4.13.4 Errors**

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

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

878 **[ENOMEM]** The ACL working storage requires more memory than is allowed
879 by the hardware or system-imposed memory management con-
880 straints.

881 **23.4.13.5 Cross-References**

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

883 **23.4.14 Get an ACL Entry**

884 Function: *acl_get_entry()*

885 **23.4.14.1 Synopsis**

```
886 #include <sys/acl.h>
887 int acl_get_entry (acl_t acl,
888                    int entry_id,
889                    acl_entry_t *entry_p);
```

890 **23.4.14.2 Description**

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

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

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

910 **23.4.14.3 Returns**

911 If the function successfully obtains an ACL entry, the function shall return a
912 value of 1. If the ACL has no ACL entries, the function shall return a value of |
913 zero. If the value of *entry_id* is ACL_NEXT_ENTRY and the last ACL entry in
914 the ACL has already been returned by a previous call to *acl_get_entry()*, the func- +
915 tion shall return a value of zero until a successful call with *entry_id* of +
916 ACL_FIRST_ENTRY is made. Otherwise, a value of -1 shall be returned and
917 *errno* shall be set to indicate the error.

918 **23.4.14.4 Errors**

919 If any of the following conditions occur, the *acl_get_entry()* function shall return
920 -1 and set *errno* to the corresponding value:

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

923 **23.4.14.5 Cross-References**

924 *acl_calc_mask()*, 23.4.2; *acl_copy_int()*, 23.4.6; *acl_create_entry()*, 23.4.7;
925 *acl_delete_entry()*, 23.4.9; *acl_dup()*, 23.4.11; *acl_from_text()*, 23.4.13;
926 *acl_get_fd()*, 23.4.15; *acl_get_file()*, 23.4.16; *acl_init()*, 23.4.20; *acl_set_fd()*,
927 23.4.21; *acl_set_file()*, 23.4.22; *acl_valid()*, 23.4.28.

928 **23.4.15 Get an ACL by File Descriptor**

929 Function: *acl_get_fd()*

930 **23.4.15.1 Synopsis**

931 #include <sys/acl.h>
932 acl_t *acl_get_fd* (int *fd*);

933 **23.4.15.2 Description**

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

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

941 The ACL in the working storage is an independent copy of the ACL associated
942 with the object referred to by *fd*. The ACL in the working storage shall not partici-
943 pate in any access control decisions.

944 **23.4.15.3 Returns**

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

948 **23.4.15.4 Errors**

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

- 951 [**EACCES**] If `{_POSIX_MAC}` is defined, MAC read access to the object is |
952 denied.
953 [**EBADF**] The `fd` argument is not a valid file descriptor.
954 [**ENOMEM**] The ACL working storage requires more memory than is allowed |
955 by the hardware or system-imposed memory management con-
956 straints.

957 **23.4.15.5 Cross-References**

958 `acl_free()`, 23.4.12; `acl_get_entry()`, 23.4.14; `acl_get_file()`, 23.4.16; `acl_set_fd()`, +
959 23.4.21.

960 **23.4.16 Get an ACL by Filename**

961 Function: `acl_get_file()`

962 **23.4.16.1 Synopsis**

963 `#include <sys/acl.h>`
964 `acl_t acl_get_file (const char *path_p, acl_type_t type);`

965 **23.4.16.2 Description**

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

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

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

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

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

984 **23.4.16.3 Returns**

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

988 **23.4.16.4 Errors**

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

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

994 If *{_POSIX_MAC}* is defined, MAC read access to the object is |
995 denied.

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

998 [EINVAL] Argument *type* is not *ACL_TYPE_ACCESS*,
999 *ACL_TYPE_DEFAULT*, or a valid implementation-defined
1000 value.

1001 [ENAMETOOLONG]

1002 The length of the *path_p* argument exceeds *{PATH_MAX}*, or a
1003 pathname component is longer than *{NAME_MAX}* while
1004 *{POSIX_NO_TRUNC}* is in effect.

1005 [ENOENT] The named object does not exist or the *path_p* argument points
1006 to an empty string.

1007 [ENOMEM] The ACL working storage requires more memory than is allowed
1008 by the hardware or system-imposed memory management con-
1009 straints.

1010 [ENOTDIR] A component of the path prefix is not a directory.

1011 **23.4.16.5 Cross-References**

1012 *acl_delete_def_file()*, 23.4.8; *acl_free()*, 23.4.12; *acl_get_entry()*, 23.4.14; +
1013 *acl_get_fd()*, 23.4.15; *acl_set_file()*, 23.4.22.

1014 **23.4.17 Retrieve the Permission Set from an ACL Entry**

1015 Function: *acl_get_permset()*

1016 **23.4.17.1 Synopsis**

1017 `#include <sys/acl.h>`
1018 `int acl_get_permset (acl_entry_t entry_d, acl_permset_t *permset_p);`

1019 **23.4.17.2 Description**

1020 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. +

1024 Any ACL entry descriptors that refer to the entry referred to by *entry_d* shall continue to refer to those entries.

1026 **23.4.17.3 Returns**

1027 Upon successful completion, the function shall return a value of zero. Otherwise,
1028 a value of -1 shall be returned and *errno* shall be set to indicate the error.

1029 **23.4.17.4 Errors**

1030 If any of the following conditions occur, the *acl_get_permset()* function shall
1031 return -1 and set *errno* to the corresponding value:

1032 [EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry. -

1033 **23.4.17.5 Cross-References**

1034 *acl_add_perm()*, 23.4.1; *acl_clear_perms()*, 23.4.3; *acl_delete_perm()*, 23.4.10;
1035 *acl_set_permset()*, 23.4.23.

1036 **23.4.18 Get ACL Entry Qualifier**

1037 Function: *acl_get_qualifier()*

1038 **23.4.18.1 Synopsis**

```
1039 #include <sys/acl.h>
1040 void *acl_get_qualifier (acl_entry_t entry_d);
```

1041 **23.4.18.2 Description**

1042 The *acl_get_qualifier()* function retrieves the qualifier of the tag for the ACL entry
1043 indicated by the argument *entry_d* into working storage and returns a pointer to
1044 that storage.

1045 If the value of the tag type in the ACL entry referred to by *entry_d* is ACL_USER,
1046 then the value returned by *acl_get_qualifier()* shall be a pointer to type *uid_t*. If
1047 the value of the tag type in the ACL entry referred to by *entry_d* is ACL_GROUP,
1048 then the value returned by *acl_get_qualifier()* shall be a pointer to type *gid_t*. If
1049 the value of the tag type in the ACL entry referred to by *entry_d* is
1050 implementation-defined, then the value returned by *acl_get_qualifier()* shall be a
1051 pointer to an implementation-defined type. If the value of the tag type in the ACL
1052 entry referred to by *entry_d* is ACL_UNDEFINED_TAG, ACL_USER_OBJ,
1053 ACL_GROUP_OBJ, ACL_OTHER, ACL_MASK, or an implementation-defined
1054 value for which a qualifier is not supported, then *acl_get_qualifier()* shall return a
1055 value of (*void* *)NULL and the function shall fail. Subsequent operations using
1056 the returned pointer shall operate on an independent copy of the qualifier in
1057 working storage.

1058 This function may cause memory to be allocated. The caller should free any
1059 releaseable memory, when the new qualifier is no longer required, by calling
1060 *acl_free()* with the *void** as an argument.

1061 The argument *entry_d* and any other ACL entry descriptors that refer to entries
1062 within the ACL containing the entry referred to by *entry_d* shall continue to refer
1063 to those entries. The order of all existing entries in the ACL containing the entry
1064 referred to by *entry_d* shall remain unchanged.

1065 **23.4.18.3 Returns**

1066 Upon successful completion, the function shall return a pointer to the tag qualifier
1067 that was retrieved into ACL working storage. Otherwise, a value of (*void* *)NULL
1068 shall be returned and *errno* shall be set to indicate the error.

1069 **23.4.18.4 Errors**

1070 If any of the following conditions occur, the *acl_get_qualifier()* function shall
1071 return a value of (*void* *)NULL and set *errno* to the corresponding value:

1072 [EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry.

1073 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.

1076 [ENOMEM] The value to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

1079 **23.4.18.5 Cross-References**

1080 *acl_create_entry()*, 23.4.7; *acl_free()*, 23.4.12; *acl_get_entry()*, 23.4.14;
 1081 *acl_get_tag_type()*, 23.4.19; *acl_set_qualifier()*, 23.4.24; *acl_set_tag_type()*, 23.4.25.

1082 **23.4.19 Get ACL Entry Tag Type**

1083 Function: *acl_get_tag_type()*

1084 **23.4.19.1 Synopsis**

1085 `#include <sys/acl.h>`

1086 `int acl_get_tag_type (acl_entry_t entry_d, acl_tag_t *tag_type_p);`

1087 **23.4.19.2 Description**

1088 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*.

1092 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.

1095 **23.4.19.3 Returns**

1096 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.

1100 **23.4.19.4 Errors**

1101 If any of the following conditions occur, the *acl_get_tag_type()* function shall return -1 and set *errno* to the corresponding value:

1103 [EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry. —

1104 **23.4.19.5 Cross-References**

1105 *acl_create_entry()*, 23.4.7; *acl_get_entry()*, 23.4.14; *acl_get_qualifier()*, 23.4.18;
1106 *acl_set_qualifier()*, 23.4.24; *acl_set_tag_type()*, 23.4.25.

1107 **23.4.20 Initialize ACL Working Storage**

1108 Function: *acl_init()*

1109 **23.4.20.1 Synopsis**

1110 `#include <sys/acl.h>`
1111 `acl_t acl_init (int count);`

1112 **23.4.20.2 Description**

1113 The *acl_init()* function allocates and initializes working storage for an ACL of at
1114 least *count* ACL entries. A pointer to the working storage is returned. The work-
1115 ing storage allocated to contain the ACL is freed by a call to *acl_free()*. When the —
1116 area is first allocated, it shall contain an ACL that contains no ACL entries. The
1117 initial state of any implementation-defined attributes of the ACL shall be
1118 implementation-defined.

1119 This function may cause memory to be allocated. The caller should free any
1120 releaseable memory, when the new ACL is no longer required, by calling
1121 *acl_free()* with the *(void *)acl_t* as an argument.

1122 **23.4.20.3 Returns**

1123 Upon successful completion, this function shall return a pointer to the working
1124 storage. Otherwise, a value of *(acl_t)NULL* shall be returned and *errno* shall be
1125 set to indicate the error.

1126 **23.4.20.4 Errors**

1127 If any of the following conditions occur, the *acl_init()* function shall return a value
1128 of *(acl_t)NULL* and set *errno* to the corresponding value:

1129 [EINVAL] The value of *count* is less than zero.

1130 [ENOMEM] The *acl_t* to be returned requires more memory than is allowed
1131 by the hardware or system-imposed memory management con-
1132 straints. —

1133 **23.4.20.5 Cross-References**

1134 *acl_free()*, 23.4.12.

1135 **23.4.21 Set an ACL by File Descriptor**

1136 Function: *acl_set_fd()*

1137 **23.4.21.1 Synopsis**

```
1138 #include <sys/acl.h>
1139 int acl_set_fd (int fd, acl_t acl);
```

1140 **23.4.21.2 Description**

1141 The *acl_set_fd()* function associates an access ACL with the object referred to by
1142 *fd*. The effective user ID of the process must match the owner of the object or the |
1143 process must have appropriate privilege to set the access ACL on the object. If |
1144 *{_POSIX_CAP}* is defined, then appropriate privilege shall include |
1145 CAP_FOWNER. In addition, if *{_POSIX_MAC}* is defined, then the process must |
1146 have MAC write access to the object.

1147 The *acl_set_fd()* function will succeed only if the ACL referred to by *acl* is valid as +
1148 defined by the *acl_valid()* function.

1149 Upon successful completion, *acl_set_fd()* shall set the access ACL of the object -
1150 referred to by argument *fd* to the ACL contained in the argument *acl*. The object's
1151 previous access ACL shall no longer be in effect. The invocation of this function
1152 may result in changes to the object's file permission bits. If *acl_set_fd()* is unsuc- +
1153 cessful, the access ACL and the file permission bits of the object referred to by +
1154 argument *fd* shall not be changed.

1155 The ordering of entries within the ACL referred to by *acl* may be changed in some
1156 implementation-defined manner.

1157 Existing ACL entry descriptors that refer to entries within the ACL referred to by
1158 *acl* shall continue to refer to those entries. Existing ACL pointers that refer to the
1159 ACL referred to by *acl* shall continue to refer to the ACL.

1160 **23.4.21.3 Returns**

1161 Upon successful completion, the function shall return a value of zero. Otherwise,
1162 a value of *-1* shall be returned and *errno* shall be set to indicate the error.

1163 **23.4.21.4 Errors**

1164 If any of the following conditions occur, the *acl_set_fd()* function shall return *-1*
1165 and set *errno* to the corresponding value:

1166	[EACCES]	If <code>{_POSIX_MAC}</code> is defined, MAC write access to the object is denied.	 —
1168	[EBADF]	The <code>fd</code> argument is not a valid file descriptor.	
1169	[EINVAL]	Argument <code>acl</code> does not point to a valid ACL. The function <code>acl_valid()</code> may be used to determine what errors are in the ACL.	+ —
1172		<code>fpathconf()</code> indicates that <code>{_POSIX_ACL_EXTENDED}</code> is in effect for the object referenced by the argument <code>fd</code> , but the ACL has more entries than the value returned by <code>fpathconf()</code> for <code>{_POSIX_ACL_PATH_MAX}</code> for the object.	+ + + +
1176	[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.	— —
1179	[EPERM]	The process does not have appropriate privilege to perform the operation to set the ACL.	
1181	[EROFS]	This function requires modification of a file system which is currently read-only.	

1183 **23.4.21.5 Cross-References**

1184 `acl_delete_def_file()`, 23.4.8; `acl_get_entry()`, 23.4.14; `acl_get_fd()`, 23.4.15;
 1185 `acl_get_file()`, 23.4.16; `acl_set_file()`, 23.4.22; `acl_valid()`, 23.4.28.

1186 **23.4.22 Set an ACL by Filename**

1187 Function: `acl_set_file()`

1188 **23.4.22.1 Synopsis**

```
1189 #include <sys/acl.h>
1190 int acl_set_file (const char *path_p, acl_type_t type, acl_t acl);
```

1191 **23.4.22.2 Description**

1192 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.

1200 The value of the argument *type* is used to indicate whether the access ACL or the
1201 default ACL associated with *path_p* is being set. If *type* is `ACL_TYPE_ACCESS`,
1202 then the access ACL shall be set. If *type* is `ACL_TYPE_DEFAULT`, then the
1203 default ACL shall be set. If the argument *type* specifies a type of ACL that cannot
1204 be associated with *path_p*, then the function shall fail.

1205 The *acl_set_file()* function will succeed only if the access or default ACL is valid as
1206 defined by the *acl_valid()* function.

1207 If `{_POSIX_ACL_EXTENDED}` is not in effect for *path_p*, then the function shall
1208 fail if:

- 1209 (1) the value of *type* is `ACL_TYPE_DEFAULT`, or
- 1210 (2) the value of *type* is `ACL_TYPE_ACCESS` and *acl* is not a minimum ACL.

1211 If the value of *type* is `ACL_TYPE_ACCESS` or `ACL_TYPE_DEFAULT`, then the
1212 function shall fail if the number of entries in *acl* is greater than the value *path-*
1213 *conf()* returns for `{_POSIX_ACL_PATH_MAX}` for *path_p*. |

1214 Upon successful completion, *acl_set_file()* shall set the access ACL or the default –
1215 ACL, as indicated by *type_d*, of the object *path_p* to the ACL contained in the
1216 argument *acl*. The object’s previous access ACL or default ACL, as indicated by
1217 *type_d*, shall no longer be in effect. The invocation of this function may result in
1218 changes to the object’s file permission bits. If *acl_set_file()* is unsuccessful, the +
1219 access ACL, the default ACL, and the file permission bits of the object referred to +
1220 by argument *path_p* shall not be changed.

1221 The ordering of entries within the ACL referred to by *acl* may be changed in some
1222 implementation-defined manner.

1223 Existing ACL entry descriptors that refer to entries within the ACL referred to by
1224 *acl* shall continue to refer to those entries. Existing ACL pointers that refer to the
1225 ACL referred to by *acl* shall continue to refer to the ACL.

1226 **23.4.22.3 Returns**

1227 Upon successful completion, the function shall return a value of zero. Otherwise,
1228 a value of `-1` shall be returned and *errno* shall be set to indicate the error.

1229 **23.4.22.4 Errors**

1230 If any of the following conditions occur, the *acl_set_file()* function shall return `-1`
1231 and set *errno* to the corresponding value:

1232 [`EACCES`] Search permission is denied for a component of the path prefix
1233 or the object exists and the process does not have appropriate
1234 access rights.
1235 If `{_POSIX_MAC}` is defined, MAC write access to *path_p* is |
1236 denied.

1237		Argument <i>type</i> specifies a type of ACL that cannot be associated with <i>path_p</i> .	-
1239	[EINVAL]	Argument <i>acl</i> does not point to a valid ACL. The function <i>acl_valid()</i> may be used to determine what errors are in the ACL.	
1242		Argument <i>type</i> is not <code>ACL_TYPE_ACCESS</code> , <code>ACL_TYPE_DEFAULT</code> , or a valid implementation-defined value.	+
1245		<i>pathconf()</i> indicates that <code>{_POSIX_ACL_EXTENDED}</code> is in effect for the object referenced by the argument <i>path_p</i> , but the ACL has more entries than the value returned by <i>pathconf()</i> for <code>{_POSIX_ACL_PATH_MAX}</code> for the object.	+
1249	[ENAMETOOLONG]	The length of the <i>path_p</i> argument exceeds <code>{PATH_MAX}</code> , or a pathname component is longer than <code>{NAME_MAX}</code> while <code>{POSIX_NO_TRUNC}</code> is in effect.	
1253	[ENOENT]	The named object does not exist or the <i>path_p</i> argument points to an empty string.	
1255	[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.	-
1258	[ENOTDIR]	A component of the path prefix is not a directory.	
1259	[EPERM]	The process does not have appropriate privilege to perform the operation to set the ACL.	
1261	[EROFS]	This function requires modification of a file system which is currently read-only.	

1263 23.4.22.5 Cross-References

1264 *acl_delete_def_file()*, 23.4.8; *acl_get_entry()*, 23.4.14; *acl_get_fd()*, 23.4.15;
 1265 *acl_get_file()*, 23.4.16; *acl_set_fd()*, 23.4.21; *acl_valid()*, 23.4.28.

1266 23.4.23 Set the Permissions in an ACL Entry

1267 Function: *acl_set_permset()*

1268 23.4.23.1 Synopsis

```
1269 #include <sys/acl.h>
1270 int acl_set_permset (acl_entry_t entry_d, acl_permset_t permset_d);
```

1271 **23.4.23.2 Description**

1272 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*.
—

1275 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.
—

1279 **23.4.23.3 Returns**

1280 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.
—

1282 **23.4.23.4 Errors**

1283 If any of the following conditions occur, the *acl_set_permset()* function shall return -1 and set *errno* to the corresponding value:
—

1285 [EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry.
—

1286 Argument *permset_d* is not a valid descriptor for a permission set within an ACL entry.
—

1288 Argument *permset_d* contains values which are not valid *acl_permset_t* values.
—

1290 **23.4.23.5 Cross-References**

1291 *acl_add_perm()*, 23.4.1; *acl_clear_perms()*, 23.4.3; *acl_delete_perm()*, 23.4.10;
1292 *acl_get_permset()*, 23.4.17.
—

1293 **23.4.24 Set ACL Entry Tag Qualifier**

1294 Function: *acl_set_qualifier()*

1295 **23.4.24.1 Synopsis**

```
1296 #include <sys/acl.h>
1297 int acl_set_qualifier (acl_entry_t entry_d,
1298                         const void *tag_qualifier_p);  
—
```

1299 **23.4.24.2 Description**

1300 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*.
+
—

1303 If the value of the tag type in the ACL entry referred to by *entry_d* is ACL_USER,
1304 then the value referred to by *tag_qualifier_p* shall be of type *uid_t*. If the value of
1305 the tag type in the ACL entry referred to by *entry_d* is ACL_GROUP, then the
1306 value referred to by *tag_qualifier_p* shall be of type *gid_t*. If the value of the tag
1307 type in the ACL entry referred to by *entry_d* is ACL_UNDEFINED_TAG,
1308 ACL_USER_OBJ, ACL_GROUP_OBJ, ACL_OTHER or ACL_MASK, then
1309 *acl_set_qualifier()* shall return an error. If the value of the tag type in the ACL
1310 entry referred to by *entry_d* is an implementation-defined value, then the value
1311 referred to by *tag_qualifier_p* shall be implementation-defined.
1312 Any ACL entry descriptors that refer to the entry referred to by *entry_d* shall con--
1313 tinue to refer to that entry. This function may cause memory to be allocated. The
1314 caller should free any releaseable memory, when the ACL is no longer required,
1315 by calling *acl_free()* with a pointer to the ACL as an argument.

1316 **23.4.24.3 Returns**

1317 Upon successful completion, the function shall return a value of zero. Otherwise,
1318 a value of -1 shall be returned and *errno* shall be set to indicate the error.

1319 **23.4.24.4 Errors**

1320 If any of the following conditions occur, the *acl_set_qualifier()* function shall
1321 return -1 and set *errno* to the corresponding value:

1322 [EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry.

1323 The tag type of the ACL entry referred to by the argument
1324 *entry_d* is not ACL_USER, ACL_GROUP, nor a valid
1325 implementation-defined value.

1326 The value pointed to by the argument *tag_qualifier_p* is not
1327 valid.

1328 [ENOMEM] The *acl_set_qualifier()* function is unable to allocate the memory
1329 required for an ACL tag qualifier.

1330 **23.4.24.5 Cross-References**

1331 *acl_get_qualifier()*, 23.4.18.

1332 **23.4.25 Set ACL Entry Tag Type**

1333 Function: *acl_set_tag_type()*

1334 **23.4.25.1 Synopsis**

```
1335 #include <sys/acl.h>
1336 int acl_set_tag_type (acl_entry_t entry_d, acl_tag_t tag_type);
```

1337 **23.4.25.2 Description**

1338 The *acl_set_tag_type()* function shall set the tag type for the ACL entry referred to
1339 by the argument *entry_d* to the value of the argument *tag_type*. —

1340 Any ACL entry descriptors that refer to the entry referred to by *entry_d* shall con-
1341 tinue to refer to that entry.

1342 **23.4.25.3 Returns**

1343 Upon successful completion, the function shall return a value of zero. Otherwise,
1344 a value of -1 shall be returned and *errno* shall be set to indicate the error.

1345 **23.4.25.4 Errors**

1346 If any of the following conditions occur, the *acl_set_tag_type()* function shall
1347 return -1 and set *errno* to the corresponding value:

1348 [EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry.

1349 Argument *tag_type* is not a valid tag type. —

1350 **23.4.25.5 Cross-References**

1351 *acl_get_tag_type()*, 23.4.19.

1352 **23.4.26 Get the Size of an ACL**

1353 Function: *acl_size()*

1354 **23.4.26.1 Synopsis**

```
1355 #include <sys/acl.h>
1356 ssize_t acl_size (acl_t acl);
```

1357 **23.4.26.2 Description**

1358 The *acl_size()* function shall return the size, in bytes, of the buffer required to
1359 hold the exportable, contiguous, persistent form of the ACL pointed to by argu-
1360 ment *acl*, when converted by *acl_copy_ext()*.

1361 Any existing ACL entry descriptors that refer to entries in *acl* shall continue to —
1362 refer to the same entries. Any existing ACL pointers that refer to the ACL

1363 referred to by *acl* shall continue to refer to the ACL. The order of ACL entries
1364 within *acl* shall remain unchanged.

1365 **23.4.26.3 Returns**

1366 Upon successful completion, the *acl_size()* function shall return the size in bytes
1367 of the contiguous, persistent form of the ACL. Otherwise, a value of (ssize_t) -1
1368 shall be returned and *errno* shall be set to indicate the error.

1369 **23.4.26.4 Errors**

1370 If any of the following conditions occur, the *acl_size()* function shall return
1371 (ssize_t) -1 and set *errno* to the corresponding value:

1372 [EINVAL] Argument *acl* does not point to a valid ACL. –

1373 **23.4.26.5 Cross-References**

1374 *acl_copy_ext()*, 23.4.5.

1375 **23.4.27 Convert an ACL to Text**

1376 Function: *acl_to_text()*

1377 **23.4.27.1 Synopsis**

1378 `#include <sys/acl.h>`
1379 `char *acl_to_text (acl_t acl, ssize_t *len_p);`

1380 **23.4.27.2 Description**

1381 The *acl_to_text()* function translates the ACL pointed to by argument *acl* into a
1382 **NULL** terminated character string. If the pointer *len_p* is not **NULL**, then the –
1383 function shall return the length of the string (not including the **NULL** terminator)
1384 in the location pointed to by *len_p*. The format of the text string returned by –
1385 *acl_to_text()* shall be the long text form defined in 23.3.1. |

1386 This function allocates any memory necessary to contain the string and returns a +
1387 pointer to the string. The caller should free any releaseable memory, when the +
1388 new string is no longer required, by calling *acl_free()* with the (*void **)*char* as an +
1389 argument.

1390 Any existing ACL entry descriptors that refer to entries in *acl* shall continue to
1391 refer to the same entries. Any existing ACL pointers that refer to the ACL
1392 referred to by *acl* shall continue to refer to the ACL. The order of ACL entries
1393 within *acl* shall remain unchanged.

1394 **23.4.27.3 Returns**

1395 Upon successful completion, the function shall return a pointer to the long text
1396 form of an ACL. Otherwise, a value of `(char *)NULL` shall be returned and `errno`
1397 shall be set to indicate the error.

1398 **23.4.27.4 Errors**

1399 If any of the following conditions occur, the `acl_to_text()` function shall return a
1400 value of `(char *)NULL` and set `errno` to the corresponding value:

1401 [EINVAL] Argument *acl* does not point to a valid ACL.

1402 The ACL denoted by *acl* contains one or more improperly formed
1403 ACL entries, or for some other reason cannot be translated into a
1404 text form of an ACL.

1405 [ENOMEM] The character string to be returned requires more memory than
1406 is allowed by the hardware or system-imposed memory manage-
1407 ment constraints. —

1408 **23.4.27.5 Cross-References**

1409 `acl_free()`, 23.4.12; `acl_from_text()`. 23.4.13.

1410 **23.4.28 Validate an ACL**

1411 Function: `acl_valid()`

1412 **23.4.28.1 Synopsis**

```
1413 #include <sys/acl.h>
1414 int acl_valid (acl_t acl);
```

1415 **23.4.28.2 Description**

1416 The `acl_valid()` function checks the ACL referred to by the argument *acl* for vali-
1417 dity.

1418 The three required entries (ACL_USER_OBJ, ACL_GROUP_OBJ, and
1419 ACL_OTHER) shall exist exactly once in the ACL. If the ACL contains any
1420 ACL_USER, ACL_GROUP, or any implementation-defined entries in the file
1421 group class, then one ACL_MASK entry shall also be required. The ACL shall
1422 contain at most one ACL_MASK entry.

1423 The qualifier field shall be unique among all entries of the same POSIX.1e ACL
1424 facility defined tag type. The tag type field shall contain valid values including
1425 any implementation-defined values. Validation of the values of the qualifier field
1426 is implementation-defined.

1427 The ordering of entries within the ACL referred to by *acl* may be changed in some
1428 implementation-defined manner.

1429 Existing ACL entry descriptors that refer to entries within the ACL referred to by
1430 *acl* shall continue to refer to those entries. Existing ACL pointers that refer to the
1431 ACL referred to by *acl* shall continue to refer to the ACL.

1432 If multiple errors occur in the ACL, the order of detection of the errors and, as a
1433 result, the ACL entry descriptor returned by *acl_valid()* shall be implementation-
1434 defined.

1435 **23.4.28.3 Returns**

1436 Upon successful completion, the function shall return a value of zero. Otherwise,
1437 a value of -1 shall be returned and *errno* shall be set to indicate the error.

1438 **23.4.28.4 Errors**

1439 If any of the following conditions occur, the *acl_valid()* function shall return -1
1440 and set *errno* to the corresponding value:

1441 [EINVAL] Argument *acl* does not point to a valid ACL.

1442 One or more of the required ACL entries is not present in *acl*.

1443 The ACL contains entries that are not unique. —

1444 **23.4.28.5 Cross-References**

1445 *acl_get_entry()*, 23.4.14; *acl_get_fd()*, 23.4.15; *acl_get_file()*, 23.4.16; *acl_init()*,
1446 23.4.20; *acl_set_fd()*, 23.4.21; *acl_set_file()*, 23.4.22.

Section 24: Audit

2 24.1 General Overview

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

- 5 (1) Interfaces for a conforming application to construct and write records to
6 an audit log and control the auditing of the current process
- 7 (2) Interfaces for reading an audit log and manipulating audit records
- 8 (3) The definition of a standard set of events, based on the POSIX.1 function
9 interfaces, that shall be reportable in conforming implementations
- 10 (4) The definition of the contents of audit records.

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

13 Support for the interfaces defined in this section is optional but shall be provided
14 if the symbol `_POSIX_AUD` is defined.

15 24.1.1 Audit Logs

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

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

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

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

35 **24.1.2 Audit Records**

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

40 Audit records are generated in two ways:

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

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

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

53 An audit record that is generated by an application, or an auditable event that
54 occurs in a system conforming to the POSIX.1 audit option, may or may not actu-
55 ally be reported to a conforming application. This standard specifies that these
56 events shall be reportable on a conforming implementation, but not that they
57 always be reported. The record will be reported only if `{_POSIX_AUD}` was
58 defined at the time the event occurred and was defined at the time the event com-
59 pleted. The results are indeterminate if `{_POSIX_AUD}` was not defined through
60 the lifetime of the event. There may also be other implementation-specific con-
61 trols on the events that are actually reported (in particular, a conforming imple-
62 mentation may have some configurable selectivity of the events that are reported).

63 **24.1.2.1 Audit Record Contents**

64 Although there is no requirement on how the system stores an audit record, logi-
65 cally it appears to the post-processing application, and to a self-auditing applica-
66 tion constructing a record, to have several parts:

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

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

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

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

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

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

99 **24.1.3 Audit Interfaces**

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

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

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

109 (1) Construct audit records

110 (2) Write audit records
111 (3) Control system auditing of the current process
112 (4) Read audit records
113 (5) Analyze an audit record
114 (6) Save audit records in user-managed store and return them to system
115 managed store.

116 The following sections provide an overview of those functions.

117 **24.1.3.1 Accessing an Audit Log**

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

120 **24.1.3.2 Constructing Audit Records**

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

123 *aud_init_record()* Get access to an unused audit record in working store.
124 *aud_dup_record()* Create a duplicate of an existing audit record in working
125 store.

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

128 *aud_put_hdr()* Add an empty header to an audit record
129 *aud_put_subj()* Add an empty set of subject attributes to an audit record
130 *aud_put_event()* Add an empty set of event-specific data to an audit record
131 *aud_put_obj()* Add an empty set of object attributes to an audit record

132 And data can be added to each type of section:

133 *aud_put_hdr_info()* Add a data item to a header in an audit record
134 *aud_put_subj_info()* Add a data item to a set of subject attributes in an audit
135 record
136 *aud_put_event_info()* Add a data item to a set of event-specific data in an audit
137 record
138 *aud_put_obj_info()* Add a data item to a set of object attributes in an audit
139 record.

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

141 *aud_delete_hdr_info()* Delete a data item from a header in an audit record
142 *aud_delete_subj_info()* Delete a data item from a set of subject attributes in an
143 audit record

148 And whole sections can be deleted too:

149 *aud_delete_hdr()* Delete a header from an audit record

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

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

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

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

155 *aud_get_id()* Get the audit ID of a process with a specified process ID.
156 This allows, for example, a server process to include the

157 audit ID of a client in a record it generates.
158 A function is provided to check the validity of an audit record:

159 *aud.validate()* Validates an audit record by checking

160 header.

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

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.

171 24.1.3.4 Controlling System Auditing

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

181 24.1.3.5 Reading Audit Records

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

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

188 24.1.3.6 Analyzing an Audit Record

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

191 *aud_get_hdr()* Get the descriptor for a header from an audit record.

192 *aud_get_hdr_info()* Get an item from within a header of an audit record.

193 *aud_get_subj()* Get the descriptor for a subject attribute set from

194 audit record.

aud_get_event_info() Get an item from within a set of event-specific data from an audit record.

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

210 aud_rec_to_text() Convert an entire audit record into human-readable text.

211 *aud_evid_to_text()* Map a numeric identifier for a system audit event to a text
212 string.

213 *aud_evid_from_text()* Map a text string, representing an system audit event type,
214 to a numeric audit event.

215 *aud id to text()* Map an audit ID to text identifying an individual user.

216 aud_id_from_text() Map text identifying an individual user to an audit ID.

217 *aud_get_all_evid()* Get a list of all system generated audit event types currently
218 reportable on the system. This interface retrieves both

220 24.1.3.7 Storing Audit Records

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

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

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

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

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

236 Finally, an interface is provided to allow an application to free any memory allo-
237 cated by the various audit functions:

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

240 24.1.4 Summary of POSIX.1 System Interface Impact

241 When {_POSIX_AUD} is defined, there is no impact on the interface syntax of any
242 POSIX.1 function, nor on the function semantics defined by POSIX.1. However,
243 use of some POSIX.1 functions may cause audit records to be reported, see section
244 24.2.1.1, below.

245 24.2 Audit Record Content

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

252 **24.2.1 Auditable Interfaces and Event Types**

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

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

258 **24.2.1.1 Auditing at the System Interface**

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

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

267 **Table 24-1 – Interfaces and Corresponding Audit Events**

268	Interface	Event Type
270	<i>aud_switch()</i>	AUD_AET_AUD_SWITCH
271	<i>aud_write()</i>	AUD_AET_AUD_WRITE
272	<i>chdir()</i>	AUD_AET_CHDIR
273	<i>chmod()</i>	AUD_AET_CHMOD
274	<i>chown()</i>	AUD_AET_CHOWN
275	<i>creat()</i>	AUD_AET_CREAT
276	<i>dup()</i>	AUD_AET_DUP
277	<i>dup2()</i>	AUD_AET_DUP
278	<i>exec()</i>	AUD_AET_EXEC
279	<i>execl()</i>	AUD_AET_EXEC
280	<i>execlp()</i>	AUD_AET_EXEC
281	<i>execv()</i>	AUD_AET_EXEC
282	<i>execvp()</i>	AUD_AET_EXEC
283	<i>execle()</i>	AUD_AET_EXEC
284	<i>execve()</i>	AUD_AET_EXEC
285	<i>_exit()</i>	AUD_AET_EXIT
286	<i>fork()</i>	AUD_AET_FORK
287	<i>kill()</i>	AUD_AET_KILL
288	<i>link()</i>	AUD_AET_LINK
289	<i>mkdir()</i>	AUD_AET_MKDIR
290	<i>mkfifo()</i>	AUD_AET_MKFIFO
291	<i>open()</i>	AUD_AET_OPEN
292	<i>opendir()</i>	AUD_AET_OPEN
293	<i>pipe()</i>	AUD_AET_PIPE
294	<i>rename()</i>	AUD_AET_RENAME
295	<i>rmdir()</i>	AUD_AET_RMDIR
296	<i>setgid()</i>	AUD_AET_SETGID
297	<i>setuid()</i>	AUD_AET_SETUID
298	<i>unlink()</i>	AUD_AET_UNLINK
299	<i>utime()</i>	AUD_AET_UTIME

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

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

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

305	Interface	Event Type
307	<i>acl_delete_def_file()</i>	AUD_AET_ACL_DELETE_DEF_FILE
308	<i>acl_set_fd()</i>	AUD_AET_ACL_SET_FD
309	<i>acl_set_file()</i>	AUD_AET_ACL_SET_FILE

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

§12	Interface	Event Type
313	<i>cap_set_fd()</i>	AUD_AET_CAP_SET_FD
314	<i>cap_set_file()</i>	AUD_AET_CAP_SET_FILE
315	<i>cap_set_proc()</i>	AUD_AET_CAP_SET_PROC

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

§18	Interface	Event Type
319	<i>inf_set_fd()</i>	AUD_AET_INF_SET_FD
320	<i>inf_set_file()</i>	AUD_AET_INF_SET_FILE
321	<i>inf_set_proc()</i>	AUD_AET_INF_SET_PROC

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

§24	Interface	Event Type
325	<i>mac_set_fd()</i>	AUD_AET_MAC_SET_FD
326	<i>mac_set_file()</i>	AUD_AET_MAC_SET_FILE
327	<i>mac_set_proc()</i>	AUD_AET_MAC_SET_PROC

328 Event types recording use of other system interfaces shall be implementation-defined; a complete set of such events shall be obtainable through the 329 *aud_get_all_evid()* interface.

331 24.2.1.2 Auditing by Applications

332 No specific types are defined for auditing by applications. The event types used 333 by applications are character strings (to reduce the chances of different applica- 334 tions using the same types and ensure they do not clash with the integer event 335 types used for system-generated events) and applications are free to add their 336 own audit event types. Applications which generate their own audit records will 337 use the *aud_write()* function passing the event type in the record header.

338 24.2.2 Audit Event Types and Record Content

339 This clause defines the minimum required content of audit records for each of the 340 standard event types. The required contents of the header is the same for all 341 records, and is defined in *aud_get_hdr_info()*; the required content of the set of 342 subject attributes is similar for all records, and is defined in *aud_get_subj_info()*; 343 the required contents of a set of object attributes is defined in *aud_get_obj_info()*. 344 This section defines the required minimum content for the set of items specific to 345 each event, and the required minimum object attribute sets for each event. A con- 346 forming implementation may include additional items in the required header, set 347 of subject attributes, set of event-specific items, and object attribute sets, or may 348 add additional sets, but the required content must be reported before these 349 implementation-specific additions.

350 A header, subject attribute set, set of event-specific items, and object attribute set 351 from an audit record are not C-language structures; each is a separate logical sec- 352 tion within the record, with components accessed using the *aud_get_*_info()* 353 interfaces described below. An argument *item_id* of these interfaces identifies the 354 component to access; a value for this argument for each component is defined in

355 the tables below.

356 Unless otherwise specified, event-specific data contains the argument values
357 requested for the operation. If the argument is not available (for example, if the
358 caller supplied a **NULL** or invalid pointer for a pathname), the
359 *aud_get_event_info()* function shall return an *aud_info_t* structure with a zero *len*
360 member. *Pathname* values reported as arguments may be the exact values
361 passed as arguments, or may be expanded by the implementation to full path-
362 names.

363 **24.2.2.1 AUD_AET_ACL_DELETE_DEF_FILE**

364 This event will be encountered only if `{_POSIX_ACL}` was defined when the audit |
365 log was generated.

366 Calls on *aud_get_event_info()* for the audit record of an
367 `AUD_AET_ACL_DELETE_DEF_FILE` event shall return *aud_info_t* structures
368 for the following event-specific items, with *aud_info_type* members as specified:

360	Type	Description	item_id
371	<code>AUD_TYPE_STRING</code>	Pathname	<code>AUD_PATHNAME</code>

372 The *Pathname* contains the value passed as an argument to the
373 *acl_delete_def_file()* function.

374 If the call succeeded a set of object attributes shall also be available from the
375 record, describing the object affected; if an ACL is reported in the set of object
376 attributes it shall contain the ACL before the event. If the call failed due to
377 access controls, and a set of object attributes is still available from the record, it
378 shall describe the object at which the failure occurred. Otherwise it is unspecified
379 whether a set of object attributes is available, or what object is defined by such a
380 set.

381 **24.2.2.2 AUD_AET_ACL_SET_FD**

382 This event will be encountered only if `{_POSIX_ACL}` was defined when the audit |
383 log was generated.

384 Calls on *aud_get_event_info()* for the audit record of an `AUD_AET_ACL_SET_FD`
385 event shall return *aud_info_t* structures for the following event-specific items,
386 with *aud_info_type* members as specified:

387	Type	Description	item_id
389	<code>AUD_TYPE_INT</code>	File desc	<code>AUD_FILE_ID</code>
390	<code>AUD_TYPE_ACL_TYPE</code>	ACL type	<code>AUD_ACL_TYPE</code>
391	<code>AUD_TYPE_ACL</code>	ACL	<code>AUD_ACL</code>

392 The *File desc*, *ACL type*, and *ACL* contain the values passed as arguments to the
393 *acl_set_fd()* function.

394 If the call succeeded a set of object attributes shall also be available from the
395 record, describing the object affected; if an ACL is reported in the set of object
396 attributes it shall contain the ACL before the event. If the call failed due to

397 access controls, and a set of object attributes is still available from the record, it
398 shall describe the object at which the failure occurred. Otherwise it is unspecified
399 whether a set of object attributes is available, or what object is defined by such a
400 set.

401 **24.2.2.3 AUD_AET_ACL_SET_FILE**

402 This event will be encountered only if {_POSIX_ACL} was defined when the audit |
403 log was generated.

404 Calls on *aud_get_event_info()* for the audit record of an
405 AUD_AET_ACL_SET_FILE event shall return *aud_info_t* structures for the fol-
406 lowing event-specific items, with *aud_info_type* members as specified:

407	Type	Description	item_id
409	AUD_TYPE_STRING	Pathname	AUD_PATHNAME
410	AUD_TYPE_ACL_TYPE	ACL type	AUD_ACL_TYPE
411	AUD_TYPE_ACL	ACL	AUD_ACL

412 The *Pathname*, *ACL type*, and *ACL* contain the values passed as arguments to
413 the *acl_set_file()* function.

414 If the call succeeded a set of object attributes shall also be available from the
415 record, describing the object affected; if an ACL is reported in the set of object
416 attributes it shall contain the ACL before the event. If the call failed due to
417 access controls, and a set of object attributes is still available from the record, it
418 shall describe the object at which the failure occurred. Otherwise it is unspecified
419 whether a set of object attributes is available, or what object is defined by such a
420 set.

421 **24.2.2.4 AUD_AET_AUD_SWITCH**

422 Calls on *aud_get_event_info()* for the audit record of an
423 AUD_AET_AUD_SWITCH event shall return *aud_info_t* structures for the follow-
424 ing event-specific items, with *aud_info_type* members as specified:

425	Type	Description	item_id
427	AUD_TYPE_AUD_STATE	Audit state	AUD_AUDIT_STATE

428 The *Audit state* contains the value passed as an argument to the *aud_switch()*
429 function: AUD_STATE_ON, AUD_STATE_OFF or AUD_STATE_QUERY.

430 **24.2.2.5 AUD_AET_AUD_WRITE**

431 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_AUD_WRITE
432 event are not required to report any event-specific data. This event is required to
433 be reportable only if an attempt to use *aud_write()*, to write a record to the sys-
434 tem audit log, fails (e.g. due to lack of appropriate privilege). The header of the
435 record shall give details of the attempt to use *aud_write()*, and the set of subject
436 attributes shall relate to the caller of *aud_write()*; that is, the record is not
437 required to contain data from the record that the application tried to write to the

438 system audit log.

439 **24.2.2.6 AUD_AET_CAP_SET_FD**

440 This event will be encountered only if {_POSIX_CAP} was defined when the audit |
441 log was generated.

442 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_CAP_SET_FD
443 event shall return *aud_info_t* structures for the following event-specific items,
444 with *aud_info_type* members as specified:

	Type	Description	item_id
447	AUD_TYPE_INT	File desc	AUD_FILE_ID
448	AUD_TYPE_CAP	Capability state	AUD_CAP

449 The *File desc* and *Capability state* contain the values passed as arguments to the
450 *cap_set_fd()* function.

451 If the call succeeded a set of object attributes shall also be available from the
452 record, describing the object affected; if a file capability state is reported in the set
453 of object attributes it shall contain the file capability state before the event. If the
454 call failed due to access controls, and a set of object attributes is still available
455 from the record, it shall describe the object at which the failure occurred. Other-
456 wise it is unspecified whether a set of object attributes is available, or what object
457 is defined by such a set.

458 **24.2.2.7 AUD_AET_CAP_SET_FILE**

459 This event will be encountered only if {_POSIX_CAP} was defined when the audit |
460 log was generated.

461 Calls on *aud_get_event_info()* for the audit record of an
462 AUD_AET_CAP_SET_FILE event shall return *aud_info_t* structures for the fol-
463 lowing event-specific items, with *aud_info_type* members as specified:

	Type	Description	item_id
466	AUD_TYPE_STRING	Pathname	AUD_PATHNAME
467	AUD_TYPE_CAP	Capability state	AUD_CAP

468 The *Pathname* and *Capability state* contain the values passed as arguments to the
469 *cap_set_file()* function.

470 If the call succeeded a set of object attributes shall also be available from the
471 record, describing the object affected. If a file capability state is reported in the
472 set of object attributes it shall contain the file capability state before the event. If
473 the call failed due to access controls, and a set of object attributes is still available
474 from the record, it shall describe the object at which the failure occurred. Other-
475 wise it is unspecified whether a set of object attributes is available, or what object
476 is defined by such a set.

477 **24.2.2.8 AUD_AET_CAP_SET_PROC**

478 This event will be encountered only if {_POSIX_CAP} was defined when the audit |
479 log was generated.

480 Calls on *aud_get_event_info()* for the audit record of an |
481 AUD_AET_CAP_SET_PROC event shall return *aud_info_t* structures for the fol- |
482 lowing event-specific items, with *aud_info_type* members as specified:

483 Type	Description	item_id
485 AUD_TYPE_CAP	Capability state	AUD_CAP

486 The *Capability state* records the value passed as an argument to the |
487 *cap_set_proc()* function. If a capability state is reported in the set of subject attri- |
488 butes in the record, this shall record the process capability state of the process |
489 before the event.

490 **24.2.2.9 AUD_AET_CHDIR**

491 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_CHDIR event |
492 shall return *aud_info_t* structures for the following event-specific items, with |
493 *aud_info_type* members as specified:

494 Type	Description	item_id
496 AUD_TYPE_STRING	Pathname	AUD_PATHNAME

497 The *Pathname* contains the value passed as an argument to the *chdir()* function.

498 **24.2.2.10 AUD_AET_CHMOD**

499 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_CHMOD event |
500 shall return *aud_info_t* structures for the following event-specific items, with |
501 *aud_info_type* members as specified:

502 Type	Description	item_id
504 AUD_TYPE_STRING	Pathname	AUD_PATHNAME
505 AUD_TYPE_MODE	Mode	AUD_MODE

506 The *Pathname* and *Mode* contain the values passed as arguments to the *chmod()* |
507 function.

508 If the call succeeded a set of object attributes shall also be available from the |
509 record, describing the object affected; if a mode is reported in the set of object |
510 attributes it shall contain the mode before the event. If the call failed due to |
511 access controls, and a set of object attributes is still available from the record, it |
512 shall describe the object at which the failure occurred. Otherwise it is unspecified |
513 whether a set of object attributes is available, or what object is defined by such a |
514 set.

515 **24.2.2.11 AUD_AET_CHOWN**

516 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_CHOWN event
517 shall return *aud_info_t* structures for the following event-specific items, with
518 *aud_info_type* members as specified:

520	Type	Description	item_id
521	AUD_TYPE_STRING	Pathname	AUD_PATHNAME
522	AUD_TYPE_UID	Owner	AUD_UID
523	AUD_TYPE_GID	Group	AUD_GID

524 The *Pathname*, *Owner*, and *Group* contain the values passed as arguments to the
525 *chown()* function.

526 If the call succeeded a set of object attributes shall also be available from the
527 record, describing the object affected; if an owner and group are reported in the
528 set of object attributes they shall contain the object owner and group before the
529 event. If the call failed due to access controls, and a set of object attributes is still
530 available from the record, it shall describe the object at which the failure
531 occurred. Otherwise it is unspecified whether a set of object attributes is avail-
532 able, or what object is defined by such a set.

533 **24.2.2.12 AUD_AET_CREAT**

534 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_CREAT event
535 shall return *aud_info_t* structures for the following event-specific items, with
536 *aud_info_type* members as specified:

537	Type	Description	item_id
539	AUD_TYPE_STRING	Pathname	AUD_PATHNAME
540	AUD_TYPE_MODE	Mode	AUD_MODE
541	AUD_TYPE_INT	Return value (file descriptor)	AUD_RETURN_ID

542 The *Pathname* and *Mode* contain the values passed as arguments to the *creat()*
543 function.

544 If the call succeeded a set of object attributes shall also be available from the
545 record, describing the object created. If the call failed due to access controls, and
546 a set of object attributes is still available from the record, it shall describe the
547 object at which the failure occurred. Otherwise it is unspecified whether a set of
548 object attributes is available, or what object is defined by such a set.

549 **24.2.2.13 AUD_AET_DUP**

550 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_DUP event
551 shall return *aud_info_t* structures for the following event-specific items, with
552 *aud_info_type* members as specified:

553	Type	Description	item_id
555	AUD_TYPE_INT	File descriptor	AUD_FILE_ID
556	AUD_TYPE_INT	Return value (file descriptor)	AUD_RETURN_ID

557 This event is recorded for any of the functions *dup()*, *dup2()*, or *fcntl()* with com-
558 mand *F_DUPFD*.

559 The *File descriptor* contains the value passed as the first argument to the func-
560 tion.

561 24.2.2.14 AUD_AET_EXEC

562 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_EXEC event
563 shall return *aud_info_t* structures for the following event-specific items, with
564 *aud_info_type* members as specified:

565	Type	Description	item_id
567	AUD_TYPE_STRING	Pathname	AUD_PATHNAME
568	AUD_TYPE_STRING_ARRAY	Command-args (Records <i>arg0...argn</i>)	AUD_CMD_ARGS
569	AUD_TYPE_STRING_ARRAY	Env_args (Records <i>envp</i>)	AUD_ENVP
570	AUD_TYPE_UID	Effective UID	AUD_UID_ID
571	AUD_TYPE_GID	Effective GID	AUD_GID_ID
572	AUD_TYPE_CAP	Process capability state	AUD_CAP
573			

574 This event is recorded for any of the functions *exec()*, *execel()*, *execelp()*, *execv()*,
575 *execvp()*, *execle()*, or *execve()*.

576 The *Pathname* contains the value passed as an argument to the function.

577 An implementation may choose not to report the value of *Command_args*. If this
578 is the case, or the arrays pointed to by the argument contained any invalid
579 pointers, the *aud_get_event_info()* function shall return an *aud_info_t* with a zero
580 *aud_info_length* member.

581 For calls other than *execle()* and *execve()*, the *aud_get_event_info()* function may
582 return an *aud_info_t* with a zero *aud_info_length* member for *Env_args*. For *exec-
583 le()* and *execve()* an implementation may choose not to report the value of
584 *Env_args*. If this is the case, or the arrays pointed to by the arguments contained
585 any invalid pointers, the *aud_get_event_info()* function shall return an *aud_info_t*
586 with a zero *aud_info_length* member.

587 The *Effective UID* and *GID* are those in effect after the call to *exec()*. The values
588 previous to the call to *exec()* are reportable in the record's subject attributes. The
589 *aud_info_length* member of the *aud_info_t* reporting these values may be zero
590 length if the effective UID and GID of the process are the same before and after
591 the *exec()*.

592 If *{_POSIX_CAP}* was in effect when the record was generated, then the *process
593 capability state* in the event-specific data shall record the state at the end of the
594 call, and if a process capability state is reported in the subject attributes in the
595 audit record, it shall be that at the start of the call. If *{_POSIX_CAP}* was not in
596 effect when the record was generated, the *aud_get_event_info()* function shall
597 return an *aud_info_t* with a zero *aud_info_length* member for the process capabil-
598 ity state.

599 If the call succeeded a set of object attributes shall also be available from the
600 record, describing the object executed. If the call failed due to access controls, and
601 a set of object attributes is still available from the record, it shall describe the
602 object at which the failure occurred. Otherwise it is unspecified whether a set of
603 object attributes is available, or what object is defined by such a set.

604 **24.2.2.15 AUD_AET_EXIT**

605 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_EXIT event
606 shall return *aud_info_t* structures for the following event-specific items, with
607 *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_INT	Exit code	AUD_EXIT_CODE

611 The *Exit code* contains the value passed as an argument to the *_exit()* function.

612 **24.2.2.16 AUD_AET_FORK**

613 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_FORK event
614 shall return *aud_info_t* structures for the following event-specific items, with
615 *aud_info_type* members as specified:

Type	Description	item_id
AUD_TYPE_PID	Return value	AUD_RETURN_ID

619 The audit record shall be reportable on behalf of the parent, when the *Return*
620 *value* shall be the child's process ID, thus the parent's process ID is recorded in
621 the record header, and the child's is the return value. A conforming implementa-
622 tion may also report a record for the child process; in this case the *Return value*
623 shall be zero. No events that are reported for the child shall be reported before
624 the parent's AUD_AET_FORK record.

625 **24.2.2.17 AUD_AET_INF_SET_FD**

626 This event will be encountered only if `{_POSIX_INF}` was defined when the audit |
627 log was generated.

628 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_INF_SET_FD
629 event shall return *aud_info_t* structures for the following event-specific items,
630 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

635 The *File desc* and *Label* contain the values passed as arguments to the
636 *inf_set_fd()* function.

637 If the call succeeded a set of object attributes shall also be available from the
638 record, describing the object affected; if an information label is reported in the set
639 of object attributes it shall contain the information label before the event. If the

640 call failed due to access controls, and a set of object attributes is still available
641 from the record, it shall describe the object at which the failure occurred. Other-
642 wise it is unspecified whether a set of object attributes is available, or what object
643 is defined by such a set.

644 **24.2.2.18 AUD_AET_INF_SET_FILE**

645 This event will be encountered only if {_POSIX_INF} was defined when the audit |
646 log was generated.

647 Calls on *aud_get_event_info()* for the audit record of an
648 AUD_AET_INF_SET_FILE event shall return *aud_info_t* structures for the fol-
649 lowing event-specific items, with *aud_info_type* members as specified:

650	Type	Description	item_id
652	AUD_TYPE_STRING	Pathname	AUD_PATHNAME
653	AUD_TYPE_INF	Label	AUD_INF_LBL

654 The *Pathname* and *Label* contain the values passed as arguments to the
655 *inf_set_file()* function.

656 If the call succeeded a set of object attributes shall also be available from the
657 record, describing the object affected; if an information label is reported in the set
658 of object attributes it shall contain the information label before the event. If the
659 call failed due to access controls, and a set of object attributes is still available
660 from the record, it shall describe the object at which the failure occurred. Other-
661 wise it is unspecified whether a set of object attributes is available, or what object
662 is defined by such a set.

663 **24.2.2.19 AUD_AET_INF_SET_PROC**

664 This event will be encountered only if {_POSIX_INF} was defined when the audit |
665 log was generated.

666 Calls on *aud_get_event_info()* for the audit record of an
667 AUD_AET_INF_SET_PROC event shall return *aud_info_t* structures for the fol-
668 lowing event-specific items, with *aud_info_type* members as specified:

669	Type	Description	item_id
671	AUD_TYPE_INF	Label	AUD_INF_LBL

672 The *Label* contains the value passed as an argument to the *inf_set_proc()* func-
673 tion. If an information label is reported in the record header it shall contain the
674 process's information label before the event.

675 **24.2.2.20 AUD_AET_KILL**

676 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_KILL event
677 shall return *aud_info_t* structures for the following event-specific items, with
678 *aud_info_type* members as specified:

	<u>Type</u>	<u>Description</u>	<u>item_id</u>
681	AUD_TYPE_PID	Pid	AUD_PID
682	AUD_TYPE_INT	Signal Number	AUD_SIG

683 The *Pid* and *Signal Number* shall record the values passed as arguments to the
684 *kill()* function.

685 If the call succeeded, or if the call failed because of access control restrictions, sets
686 of object attributes shall also be available from the record, one describing each
687 object to which the signal was directed. In addition, following the content nor-
688 mally required from each set of object attributes, there shall also be available
689 from each an item:

	<u>Type</u>	<u>Description</u>	<u>item_id</u>
692	AUD_TYPE_AUD_STATUS	The audit status of the event	AUD_STATUS

693 recording whether the signal was successfully sent to that object. If the call failed
694 for reasons other than access control, it is not defined whether any sets of object
695 attributes are available.

696 **24.2.2.21 AUD_AET_LINK**

697 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_LINK event
698 shall return *aud_info_t* structures for the following event-specific items, with
699 *aud_info_type* members as specified:

	<u>Type</u>	<u>Description</u>	<u>item_id</u>
702	AUD_TYPE_STRING	Path1	AUD_PATHNAME
703	AUD_TYPE_STRING	Path2	AUD_LINKNAME

704 The *Path1* and *Path2* contain the values passed as arguments to the *link()* func-
705 tion. *Path1* contains the pathname of the existing file, *Path2* contains the path-
706 name of the new directory entry to be created.

707 If the call succeeded a set of object attributes shall also be available from the
708 record, describing the file to which the link is made. If the call failed due to
709 access controls, and a set of object attributes is still available from the record, it
710 shall describe the object at which the failure occurred. Otherwise it is unspecified
711 whether a set of object attributes is available, or what object is defined by such a
712 set.

713 **24.2.2.22 AUD_AET_MAC_SET_FD**

714 This event will be encountered only if {_POSIX_MAC} was defined when the audit
715 log was generated.

716 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_MAC_SET_FD
717 event shall return *aud_info_t* structures for the following event-specific items,
718 with *aud_info_type* members as specified:

	<u>Type</u>	<u>Description</u>	<u>item_id</u>
721	AUD_TYPE_INT	File desc	AUD_FILE_ID
722	AUD_TYPE_MAC	Label	AUD_MAC_LBL

723 The *File desc* and *Label* contain the values passed as arguments to the
724 *mac_set_fd()* call.

725 If the call succeeded a set of object attributes shall also be available from the
726 record, describing the object affected; if a MAC label is reported in the set of object
727 attributes it shall contain the MAC label before the event. If the call failed due to
728 access controls, and a set of object attributes is still available from the record, it
729 shall describe the object at which the failure occurred. Otherwise it is unspecified
730 whether a set of object attributes is available, or what object is defined by such a
731 set.

732 **24.2.2.23 AUD_AET_MAC_SET_FILE**

733 This event will be encountered only if `{_POSIX_MAC}` was defined when the audit
734 log was generated.

735 Calls on *aud_get_event_info()* for the audit record of an
736 AUD_AET_MAC_SET_FILE event shall return *aud_info_t* structures for the fol-
737 lowing event-specific items, with *aud_info_type* members as specified:

	<u>Type</u>	<u>Description</u>	<u>item_id</u>
740	AUD_TYPE_STRING	Pathname	AUD_PATHNAME
741	AUD_TYPE_MAC	Label	AUD_MAC_LBL

742 The *Pathname* and *Label* contain the values passed as arguments to the
743 *mac_set_file()* call.

744 If the call succeeded a set of object attributes shall also be available from the
745 record, describing the object affected; if a MAC label is reported in the set of object
746 attributes it shall contain the MAC label before the event. If the call failed due to
747 access controls, and a set of object attributes is still available from the record, it
748 shall describe the object at which the failure occurred. Otherwise it is unspecified
749 whether a set of object attributes is available, or what object is defined by such a
750 set.

751 **24.2.2.24 AUD_AET_MAC_SET_PROC**

752 This event will be encountered only if `{_POSIX_MAC}` was defined when the audit
753 log was generated.

754 Calls on *aud_get_event_info()* for the audit record of an
755 AUD_AET_MAC_SET_PROC event shall return *aud_info_t* structures for the fol-
756 lowing event-specific items, with *aud_info_type* members as specified:

	<u>Type</u>	<u>Description</u>	<u>item_id</u>
759	AUD_TYPE_MAC	Label	AUD_MAC_LBL

760 The *Label* contains the value passed as an argument to the *mac_set_proc()* func-
761 tion. If a MAC label is reported in the record header it shall contain the process

762 MAC label before the event.

763 **24.2.2.25 AUD_AET_MKDIR**

764 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_MKDIR event
765 shall return *aud_info_t* structures for the following event-specific items, with
766 *aud_info_type* members as specified:

768	Type	Description	item_id
769	AUD_TYPE_STRING	Pathname	AUD_PATHNAME
770	AUD_TYPE_MODE	Mode	AUD_MODE

771 The *Pathname* and *Mode* contain the values passed as arguments to the *mkdir()*
772 function.

773 If the call succeeded a set of object attributes shall also be available from the
774 record, describing the object created. If the call failed due to access controls, and
775 a set of object attributes is still available from the record, it shall describe the
776 object at which the failure occurred. Otherwise it is unspecified whether a set of
777 object attributes is available, or what object is defined by such a set.

778 **24.2.2.26 AUD_AET_MKFIFO**

779 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_MKFIFO event
780 shall return *aud_info_t* structures for the following event-specific items,
781 with *aud_info_type* members as specified:

782	Type	Description	item_id
784	AUD_TYPE_STRING	Pathname	AUD_PATHNAME
785	AUD_TYPE_MODE	Mode	AUD_MODE

786 The *Pathname* and *Mode* contain the values passed as arguments to the *mkfifo()*
787 function.

788 If the call succeeded a set of object attributes shall also be available from the
789 record, describing the object created. If the call failed due to access controls, and
790 a set of object attributes is still available from the record, it shall describe the
791 object at which the failure occurred. Otherwise it is unspecified whether a set of
792 object attributes is available, or what object is defined by such a set.

793 **24.2.2.27 AUD_AET_OPEN**

794 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_OPEN event
795 shall return *aud_info_t* structures for the following event-specific items, with
796 *aud_info_type* members as specified:

798	Type	Description	item_id
799	AUD_TYPE_STRING	Pathname	AUD_PATHNAME
800	AUD_TYPE_INT	Oflag	AUD_OFLAG
801	AUD_TYPE_MODE	Mode	AUD_MODE
802	AUD_TYPE_INT	Return value (file descriptor)	AUD_RETURN_ID

803 The *Pathname*, *Oflag* and *Mode* contain the values passed as arguments to the
804 *open()* function. If the *O_CREAT* flag is not set in *Oflag*, the *aud_get_event_info()*
805 function shall return an *aud_info_t* with a zero *aud_info_length* field if an attempt
806 is made to read *Mode*.

807 If the call succeeded a set of object attributes shall also be available from the
808 record, describing the object opened. If the call failed due to access controls, and a
809 set of object attributes is still available from the record, it shall describe the object
810 at which the failure occurred. Otherwise it is unspecified whether a set of object
811 attributes is available, or what object is defined by such a set.

812 24.2.2.28 AUD_AET_PIPE

813 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_PIPE event
814 shall return *aud_info_t* structures for the following event-specific items, with
815 *aud_info_type* members as specified:

816	Type	Description	item_id
818	AUD_TYPE_INT	Read file descriptor	AUD_RD_FILE_ID
819	AUD_TYPE_INT	Write file descriptor	AUD_WR_FILE_ID

820 If the call succeeded, the *File descriptors* shall contain the values returned to the
821 caller. Otherwise, the *aud_get_event_info()* function shall return *aud_info_t*
822 structures with zero *aud_info_length* members for these items.

823 24.2.2.29 AUD_AET_RENAME

824 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_RENAME event
825 shall return *aud_info_t* structures for the following event-specific items,
826 with *aud_info_type* members as specified:

828	Type	Description	item_id
829	AUD_TYPE_STRING	Old pathname	AUD_OLD_PATHNAME
830	AUD_TYPE_STRING	New pathname	AUD_NEW_PATHNAME

831 The *pathnames* contain the values passed as arguments to the *rename()* call.

832 If the call succeeded a set of object attributes shall also be available from the
833 record, describing the object renamed; the name reported in the set of object attri-
834 butes shall contain the name before the event. If the call failed due to access con-
835 trols, and a set of object attributes is still available from the record, it shall
836 describe the object at which the failure occurred. Otherwise it is unspecified
837 whether a set of object attributes is available, or what object is defined by such a
838 set.

839 24.2.2.30 AUD_AET_RMDIR

840 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_RMDIR event
841 shall return *aud_info_t* structures for the following event-specific items, with
842 *aud_info_type* members as specified:

	<u>Type</u>	<u>Description</u>	<u>item_id</u>
844	AUD_TYPE_STRING	Pathname	AUD_PATHNAME
845	The <i>pathname</i> contains the value passed as an argument to the <i>rmdir()</i> call.		
846	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.		
847			
848			
849			
850			
851			
852	24.2.2.31 AUD_AET_SETGID		
853	Calls on <i>aud_get_event_info()</i> for the audit record of an AUD_AET_SETGID event shall return <i>aud_info_t</i> structures for the following event-specific items, with <i>aud_info_type</i> members as specified:		
854			
855			
856			
857			
858	<u>Type</u>	<u>Description</u>	<u>item_id</u>
	AUD_TYPE_GID	gid	AUD_GID
859	The <i>gid</i> contains the value passed as an argument. The value before the call is reportable in the subject attributes.		
860			
861	24.2.2.32 AUD_AET_SETUID		
862	Calls on <i>aud_get_event_info()</i> for the audit record of an AUD_AET_SETUID event shall return <i>aud_info_t</i> structures for the following event-specific items, with <i>aud_info_type</i> members as specified:		
863			
864			
865			
866			
867	<u>Type</u>	<u>Description</u>	<u>item_id</u>
	AUD_TYPE_UID	uid	AUD_UID
868	The <i>uid</i> contains the value passed as an argument. The value before the call is reportable in the subject attributes.		
869			
870	24.2.2.33 AUD_AET_UNLINK		
871	Calls on <i>aud_get_event_info()</i> for the audit record of an AUD_AET_UNLINK event shall return <i>aud_info_t</i> structures for the following event-specific items, with <i>aud_info_type</i> members as specified:		
872			
873			
874			
875			
876	<u>Type</u>	<u>Description</u>	<u>item_id</u>
	AUD_TYPE_STRING	Pathname	AUD_PATHNAME
877	The <i>Pathname</i> contains the value passed as an argument to the <i>unlink()</i> function.		
878	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.		
879			
880			
881			
882			

883 **24.2.2.34 AUD_AET_UTIME**

884 Calls on *aud_get_event_info()* for the audit record of an AUD_AET_UTIME event
885 shall return *aud_info_t* structures for the following event-specific items, with
886 *aud_info_type* members as specified:

887	Type	Description	item_id
889	AUD_TYPE_STRING	Pathname	AUD_PATHNAME
890	AUD_TYPE_TIME	Access time	AUD_ATIME
891	AUD_TYPE_TIME	Modification time	AUD_MTIME

892 The *Pathname* contains the value passed as an argument to the *utime()* function.
893 The *Access time* and *Modification time* contain the values from the *timebuf* struc-
894 ture passed as an argument.

895 If the call succeeded a set of object attributes shall also be available from the
896 record, describing the object affected. If the call failed due to access controls, and
897 a set of object attributes is still available from the record, it shall describe the
898 object at which the failure occurred. Otherwise it is unspecified whether a set of
899 object attributes is available, or what object is defined by such a set.

900 **24.3 Header**

901 Some of the data types used by the audit functions are not defined as part of this
902 standard, but shall be implementation-defined. If *{_POSIX_AUD}* is defined,
903 these types shall be defined in the header *<sys/audit.h>*, which contains
904 definitions for at least the types shown in the following table.

Table 24-2 – Audit Data Types

Defined Type	Description
<i>aud_evinfo_t</i>	Used to access the set of event-specific data within an audit record. This data type is non-exportable data.
<i>aud_hdr_t</i>	Used to access the header of an audit record. This data type is non-exportable data.
<i>aud_id_t</i>	Item in an audit record header used to provide individual accountability for the audit event. This data type is exportable data.
<i>aud_info_t</i>	Defines the type, size and location of various items from an audit record. This data type is non-exportable data.
<i>aud_obj_t</i>	Used to access an object attribute set within an audit record. This data type is non-exportable data.
<i>aud_obj_type_t</i>	Item in an object attribute set that defines the type of the object. This data type is exportable data.
<i>aud_rec_t</i>	A pointer to an opaque audit record. This data type is non-exportable data.
<i>aud_state_t</i>	Controls whether system-generated records are auditable for a process. This data type is exportable data.
<i>aud_status_t</i>	Item in an audit record header giving the success/failure status of the audit event. This data type is exportable data.
<i>aud_subj_t</i>	Used to access the subject attribute set within an audit record. This data type is non-exportable data.
<i>aud_time_t</i>	The time of an audit event. This data type is exportable data.

941 Further details of these types are given below.

942 In addition, the header <sys/audit.h> shall define the following constants:

- 943 • All the AUD_AET_* constants defined in section 24.2.2, for the POSIX-defined event types
- 944
- 945 • All the constants defined in sections 24.2.2, 24.4.17, 24.4.19, 24.4.22, and
- 946 24.4.24, (including AUD_FIRST_ITEM and AUD_NEXT_ITEM) for the
- 947 *item_id* arguments that can be supplied to the *aud_get_*_info()* functions
- 948
- The following miscellaneous constants:

Table 24-3 – Other Constants

Constant	Description
AUD_SYSTEM_LOG	Value of the <i>filedes</i> argument for <i>aud_write()</i> .
AUD_NATIVE	Value of the format item in a record header.
AUD_LAST_ITEM	Value of the <i>position</i> argument for the <i>aud_put_*_info()</i> functions.
AUD_STD_NNNN_N	Value
957	of the version item in a record header.
958	The NNNN_N in AUD_STD_NNNN_N is merely a placeholder for the year
959	(e.g., 1997) this standard is approved and standard (e.g., _1 implying
960	POSIX.1) it is placed into.

961 Further constants are identified in the rest of this section.

962 **24.3.1 aud_evinfo_t**

963 This typedef shall define an opaque, implementation-defined descriptor for the set
964 of event-specific data in an audit record. The internal structure of an
965 *aud_evinfo_t* is unspecified.

966 **24.3.2 aud_hdr_t**

967 This typedef shall define an opaque, implementation-defined descriptor for an
968 audit record header. The internal structure of an *aud_hdr_t* is unspecified.

969 **24.3.3 aud_id_t**

970 The *aud_id_t* obtainable from an audit record header is an implementation-
971 defined typedef for holding a value which uniquely identifies a user.

972 **24.3.4 aud_info_t**

973 The *aud_info_t* structure defines the type, length and location of some data from
974 an audit record. The *aud_info_t* structure shall contain at least the following
975 members:

976 **Table 24-4 – *aud_info_t* members**

978	Defined Type	Name	Description
979	<i>int</i>	<i>aud_info_type</i>	The type of the data
980	<i>size_t</i>	<i>aud_info_length</i>	The length of the data
981	<i>void</i> *	<i>aud_info_p</i>	Pointer to the data

982 The *aud_info_type* member may be used to interpret the data referenced by the
983 *aud_info_p* member. Values for *aud_info_type* shall be defined in the header
984 `<sys/audit.h>`. At least the following values of *aud_info_type* shall be defined,
985 and shall have the specified interpretation:

Table 24-5 – Values for *aud_info_type* Member

	<i>Value of aud_info_type</i>	<i>Interpretation of aud_info_p</i>
987		
989	AUD_TYPE_ACL	<i>acl_t*</i>
990	AUD_TYPE_ACL_TYPE	<i>acl_type_t*</i>
991	AUD_TYPE_AUD_ID	<i>aud_id_t*</i>
992	AUD_TYPE_AUD_OBJ_TYPE	<i>aud_obj_type_t*</i>
993	AUD_TYPE_AUD_STATE	<i>aud_state_t*</i>
994	AUD_TYPE_AUD_STATUS	<i>aud_status_t*</i>
995	AUD_TYPE_AUD_TIME	<i>aud_time_t*</i>
996	AUD_TYPE_CAP	<i>cap_t*</i>
997	AUD_TYPE_CHAR	<i>char*</i>
998	AUD_TYPE_GID	<i>gid_t*</i>
999	AUD_TYPE_INF	<i>inf_t*</i>
1000	AUD_TYPE_INT	<i>int*</i>
1001	AUD_TYPE_LONG	<i>long*</i>
1002	AUD_TYPE_MAC	<i>mac_t*</i>
1003	AUD_TYPE_MODE	<i>mode_t*</i>
1004	AUD_TYPE_OPAQUE	<i>void*</i>
1005	AUD_TYPE_PID	<i>pid_t*</i>
1006	AUD_TYPE_SHORT	<i>short*</i>
1007	AUD_TYPE_STRING	<i>char*</i> , pointing to a null terminated character string
1008	AUD_TYPE_STRING_ARRAY	<i>char**</i>
1009	AUD_TYPE_TIME	<i>time_t*</i>
1010	AUD_TYPE_UID	<i>uid_t*</i>
1011		
1012	With the exception of AUD_TYPE_STRING and AUD_TYPE_OPAQUE, <i>aud_info_p</i> should be interpreted as a pointer to zero or more items of the type specified. In the case of AUD_TYPE_STRING, <i>aud_info_p</i> should be interpreted as a (<i>char *</i>) value. For AUD_TYPE_OPAQUE <i>aud_info_p</i> is interpreted as a pointer to zero or more bytes of opaque data.	
1013		
1014		
1015		
1016		
1017	A conforming implementation may define further values for <i>aud_info_type</i> , that can be treated in the same way as AUD_TYPE_OPAQUE.	
1018		
1019	In all cases, the <i>aud_info_length</i> member gives the length, in bytes, of the data to which <i>aud_info_p</i> points.	
1020		

1021 **24.3.5 aud_obj_t**

1022 This typedef shall define an opaque, implementation-defined descriptor for a set
 1023 of object attributes in an audit record. The internal structure of an *aud_obj_t* is
 1024 unspecified.

1025 **24.3.6 aud_obj_type_t**

1026 The *aud_obj_type_t* obtainable from an object attribute set indicates the object
1027 type. This data type shall support a unique value for each of the object types for
1028 which object attribute sets can be generated in the implementation. The imple-
1029 mentation shall define in <sys/audit.h> at least the following unique values:

1030 **Table 24-6 – *aud_obj_type_t* Values**

1032	Defined Type	Description
1033	AUD_OBJ_BLOCK_DEV	Block device
1034	AUD_OBJ_CHAR_DEV	Character device
1035	AUD_OBJ_DIR	Directory
1036	AUD_OBJ_FIFO	FIFO object
1037	AUD_OBJ_FILE	Regular file
1038	AUD_OBJ_PROC	Process object

1039 **24.3.7 aud_rec_t**

1040 This typedef shall define a pointer to an opaque data item capable of holding a
1041 specific audit record, the format and storage of which are unspecified. Thus, an
1042 application cannot depend on performing normal byte-copy operations on the data
1043 item to which an *aud_rec_t* points.

1044 **24.3.8 aud_state_t**

1045 An *aud_state_t* describes whether system events are being audited for a process.
1046 An implementation shall define in <sys/audit.h> at least the following unique
1047 values for this type:

1048 **Table 24-7 – *aud_state_t* Values**

1049	Defined Type	Description
1051	AUD_STATE_OFF	System events not audited
1052	AUD_STATE_ON	System events audited
1053	AUD_STATE_QUERY	Enquiry value for <i>aud_switch()</i>

1054 **24.3.9 aud_status_t**

1055 The *aud_status_t* item obtainable from an audit record header indicates the
1056 status of the event. This data type shall define in <sys/audit.h> at least the
1057 following unique values for this type:

1058

Table 24-8 – *aud_status_t* Values

1059	Defined Type	Description
1061	AUD_FAIL_PRIV	The event failed because the process did not have appropriate privilege (see below).
1062	AUD_FAIL_DAC	The event failed because of DAC access checks.
1063	AUD_FAIL_MAC	The event failed because of MAC access checks.
1064	AUD_FAIL_OTHER	The event failed for some reason not included in other AUD_FAIL_* values.
1065	AUD_PRIV_USED	The event completed successfully; appropriate privilege was used (see below).
1066	AUD_SUCCESS	The event completed successfully.
1067		
1068		
1069		
1070		The value AUD_PRIV_USED indicates that the operation succeeded, but would not have done so if the process had not had appropriate privilege.
1071		–
1072		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.
1073		
1074		
1075		
1076		A conforming implementation may add additional status values.

1077 **24.3.10 aud_subj_t**

1078 This typedef shall define an opaque, implementation-defined descriptor for the set
 1079 of subject attributes in an audit record. The internal structure of an *aud_subj_t* is
 1080 unspecified.

1081 **24.3.11 aud_time_t**

1082 An *aud_time_t* structure specifies a single time value and shall include at least
 1083 the following members:

1084

Table 24-9 – *aud_time_t* Members

1085	Defined Type	Name	Description
1087	<i>time_t</i>	sec	Seconds
1088	<i>long</i>	nsec	Nanoseconds
1089			The <i>nsec</i> 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).
1090			
1091			
1092			

1093 **24.4 Functions**

1094 The functions in this section comprise the set of services that permit a process to
1095 construct, write, read and analyze audit records. Support for the audit facility
1096 functions described in this section is optional. If the symbol `{_POSIX_AUD}` is
1097 defined the implementation supports the audit option and all of the audit func-
1098 tions shall be implemented as described in this section. If `{_POSIX_AUD}` is not
1099 defined, the result of calling any of these functions is unspecified.

1100 The error [ENOTSUP] shall be returned in those cases where the system supports
1101 the audit facility but the particular audit operation cannot be applied because of
1102 restrictions imposed by the implementation.

1103 **24.4.1 Copy an Audit Record From System to User Space**

1104 Function: *aud_copy_ext()*

1105 **24.4.1.1 Synopsis**

```
1106 #include <sys/audit.h>
1107 ssize_t aud_copy_ext (void *aud_rec_ext_p, aud_rec_t aud_rec_int,
1108                         ssize_t size);
```

1109 **24.4.1.2 Description**

1110 The *aud_copy_ext()* function shall copy an audit record, pointed to by *aud_rec_int*,
1111 from system-managed space to user-managed space (pointed to by *aud_rec_ext_p*).
1112 The *size* argument represents the size in bytes of the buffer pointed to by the
1113 *aud_rec_ext_p* argument.

1114 The *aud_copy_ext()* function will do any conversions necessary to convert the
1115 record from internal format. The audit record returned by *aud_copy_ext()* will be
1116 a contiguous, persistent data item. It is the responsibility of the user to allocate a
1117 record buffer large enough to hold the copied record. The size of the buffer needed
1118 can be obtained by a call to the *aud_size()* function.

1119 The *aud_copy_ext()* call shall not affect the record pointed to by *aud_rec_int*.

1120 It is the responsibility of the user to release any space required to store the con-
1121 verted record.

1122 **24.4.1.3 Returns**

1123 Upon successful completion, the *aud_copy_ext()* function returns the size of the
1124 converted record placed in *aud_rec_ext_p*. Otherwise, a value of ((*ssize_t*)−1) shall
1125 be returned and *errno* shall be set to indicate the error.

1126 **24.4.1.4 Errors**

1127 If any of the following conditions occur, the *aud_copy_ext()* function shall return
1128 ((*ssize_t*)−1) and set *errno* to the corresponding value:

1129 [EINVAL] The value for the *aud_rec_int* argument is invalid.

1130 The *size* argument is zero or negative.

1131 [ERANGE] The *size* argument is greater than zero but smaller than the
1132 length of the audit record.

1133 **24.4.1.5 Cross-References**

1134 *aud_copy_int()*, 24.4.2; *aud_size()*, 24.4.38; *aud_valid()*, 24.4.40.

1135 **24.4.2 Copy an Audit Record From User to System Space**

1136 Function: *aud_copy_int()*

1137 **24.4.2.1 Synopsis**

```
1138 #include <sys/audit.h>
1139 aud_rec_t aud_copy_int (const void *aud_rec_ext_p);
```

1140 **24.4.2.2 Description**

1141 The *aud_copy_int()* function shall copy an audit record, pointed to by
1142 *aud_rec_ext_p*, from user-managed space to system-managed space. Upon success-
1143 ful completion, the function shall return an *aud_rec_t* pointing to the internal ver-
1144 sion of the audit record.

1145 Once copied to system-managed space, the record can be manipulated by the
1146 *aud_get_**() functions, and other functions that manipulate audit records.

1147 The record pointed to by *aud_rec_ext_p* must have been obtained from a previous,
1148 successful call to *aud_copy_ext()* for this function to work successfully.

1149 This function may cause memory to be allocated. The caller should free any
1150 releasable memory, when the new record is no longer required, by calling
1151 *aud_free()* with the (*void**)*aud_rec_t* as an argument.

1152 The *aud_copy_int()* call shall not affect the record pointed to by *aud_rec_ext_p*.

1153 **24.4.2.3 Returns**

1154 Upon successful completion, the *aud_copy_int()* function returns an audit record
1155 pointer set to point to the internal version of the audit record. Otherwise, a value
1156 of (*aud_rec_t*)**NULL** shall be returned, the caller shall not have to free any releas-
1157 able memory, and *errno* shall be set to indicate the error.

1158 **24.4.2.4 Errors**

1159 If any of the following conditions occur, the *aud_copy_int()* function shall return
1160 (*aud_rec_t*)**NULL** and set *errno* to the corresponding value:

1161 [EINVAL] The value of the *aud_rec_ext_p* argument is invalid.

1162 [ENOMEM] The function requires more memory than is allowed by the
1163 hardware or system-imposed memory management constraints. –

1164 **24.4.2.5 Cross-References**

1165 *aud_copy_ext()*, 24.4.1; *aud_free()*, 24.4.14; *aud_get_event()*, 24.4.16;
1166 *aud_get_hdr()*, 24.4.18; *aud_get_obj()*, 24.4.21; *aud_get_subj()*. 24.4.23.

1167 **24.4.3 Delete Set of Event-specific Data from a Record**

1168 Function: *aud_delete_event()*

1169 **24.4.3.1 Synopsis**

```
1170 #include <sys/audit.h>
1171 int aud_delete_event (aud_evinfo_t aud_event_d);
```

1172 **24.4.3.2 Description**

1173 The *aud_delete_event()* function deletes a set of event-specific data from an audit
1174 record, including any data items within the set. The set to be deleted is defined
1175 by the *aud_event_d* descriptor. Upon successful execution, the set of event-specific
1176 data shall no longer be accessible, and the *aud_event_d* descriptor shall become
1177 undefined.

1178 Calls to this function shall not affect the status of descriptors for any other set of
1179 data in this or any other audit record.

1180 **24.4.3.3 Returns**

1181 Upon successful completion, the *aud_delete_event()* function returns 0. Otherwise,
1182 it returns a value of -1 and *errno* is set to indicate the error. The audit
1183 record shall not be changed if the return value is -1.

1184 **24.4.3.4 Errors**

1185 If any of the following conditions occur, the *aud_delete_event()* function shall
1186 return -1 and set *errno* to the corresponding value:

1187 [EINVAL] Argument *aud_event_d* is not a valid descriptor for a set of
1188 event-specific data within an audit record. –

1189 **24.4.3.5 Cross-References**

1190 *aud_delete_event_info()*, 24.4.4; *aud_init_record()*, 24.4.27; *aud_put_event()*,
1191 24.4.28; *aud_valid()*, 24.4.40; *aud_write()*, 24.4.41.

1192 **24.4.4 Delete Item from Set of Event-specific Data**

1193 Function: *aud_delete_event_info()*

1194 **24.4.4.1 Synopsis**

```
1195 #include <sys/audit.h>
1196 int aud_delete_event_info (aud_evinfo_t aud_event_d,
1197                           int item_id);
```

1198 **24.4.4.2 Description**

1199 The *aud_delete_event_info()* function deletes a data item from a set of event-
1200 specific data in an audit record. Upon successful execution of
1201 *aud_delete_event_info()*, the item defined by *item_id* shall no longer be accessible
1202 in the set of event-specific data defined by *aud_event_d*.

1203 The value of *item_id* specifies an item within the set of event-specific data. For
1204 system-generated records, the items available are dependent upon the *event type*
1205 of the audit record being examined; for each POSIX-defined event type the
1206 minimum set of items that shall be available, together with values of *item_id* to
1207 access them, are specified in section 24.2.2. For application-generated records,
1208 the values of *item_id* match the calls on *aud_put_event_info()* that put the items
1209 into the set of event-specific data.

1210 Calls to this function shall not affect the status of descriptors for any other data
1211 item in this or any other audit record.

1212 **24.4.4.3 Returns**

1213 Upon successful completion, the *aud_delete_event_info()* function returns 0. Otherwise,
1214 it returns a value of -1 and *errno* is set to indicate the error. The audit
1215 record shall not be changed if the return value is -1.

1216 **24.4.4.4 Errors**

1217 If any of the following conditions occur, the *aud_delete_event_info()* function shall
1218 return -1 and set *errno* to the corresponding value:

1219 [EINVAL] Argument *aud_event_d* is not a valid descriptor for a set of
1220 event-specific data within an audit record.

1221 Argument *item_id* does not reference a valid data item within
1222 *aud_event_d*. —

1223 **24.4.4.5 Cross-References**

1224 *aud_delete_event()*, 24.4.3; *aud_init_record()*, 24.4.27; *aud_put_event()*, 24.4.28;
1225 *aud_put_event_info()*, 24.4.29; *aud_valid()*, 24.4.40; *aud_write()*, 24.4.41.

1226 **24.4.5 Delete Header from an Audit Record**

1227 Function: *aud_delete_hdr()*

1228 **24.4.5.1 Synopsis**

1229 `#include <sys/audit.h>`
1230 `int aud_delete_hdr (aud_hdr_t aud_hdr_d);`

1231 **24.4.5.2 Description**

1232 The *aud_delete_hdr()* function deletes a header from an audit record, including
1233 any data items within the header. The header to be deleted is defined by the
1234 *aud_hdr_d* descriptor. Upon successful execution, the header shall no longer be
1235 accessible in the record, and the *aud_hdr_d* descriptor shall become undefined.

1236 Calls to this function shall not affect the status of descriptors for any other set of
1237 data in this or any other audit record.

1238 **24.4.5.3 Returns**

1239 Upon successful completion, the *aud_delete_hdr()* function returns 0. Otherwise,
1240 it returns a value of -1 and *errno* is set to indicate the error. The audit record
1241 shall not be changed if the return value is -1.

1242 **24.4.5.4 Errors**

1243 If any of the following conditions occur, the *aud_delete_hdr()* function shall return
1244 -1 and set *errno* to the corresponding value:

1245 [EINVAL] Argument *aud_hdr_d* is not a valid descriptor for a header
1246 within an audit record.

1247 **24.4.5.5 Cross-References**

1248 *aud_delete_hdr_info()*, 24.4.6; *aud_init_record()*, 24.4.27; *aud_put_hdr()*, 24.4.30;
1249 *aud_valid()*, 24.4.40; *aud_write()*, 24.4.41.

1250 **24.4.6 Delete Item from Audit Record Header**

1251 Function: *aud_delete_hdr_info()*

1252 **24.4.6.1 Synopsis**

```
1253 #include <sys/audit.h>
1254 int aud_delete_hdr_info (aud_hdr_t aud_hdr_d,
1255                         int item_id);
```

1256 **24.4.6.2 Description**

1257 The *aud_delete_hdr_info()* function deletes a data item from a header in an audit
1258 record. Upon successful execution of *aud_delete_hdr_info()*, the item defined by
1259 *item_id* shall no longer be accessible in the header defined by *aud_hdr_d*.

1260 The value of *item_id* specifies an item within the audit record header. For records
1261 read from an audit log, the minimum set of items that shall be available from the
1262 first header, together with values of *item_id* to access them, are specified in sec-
1263 tion 24.4.19. For application-generated records the values of *item_id* match the
1264 calls on *aud_put_hdr_info()* that put the items into the header.

1265 Calls to this function shall not affect the status of descriptors for any other data
1266 item in this or any other audit record.

1267 **24.4.6.3 Returns**

1268 Upon successful completion, the *aud_delete_hdr_info()* function returns 0. Other-
1269 wise, it returns a value of -1 and *errno* is set to indicate the error. The audit
1270 record shall not be changed if the return value is -1.

1271 **24.4.6.4 Errors**

1272 If any of the following conditions occur, the *aud_delete_hdr_info()* function shall
1273 return -1 and set *errno* to the corresponding value:

1274 [EINVAL] Argument *aud_hdr_d* is not a valid descriptor for a header
1275 within an audit record.

1276 Argument *item_id* does not reference a valid data item within
1277 *aud_hdr_d*. —

1278 **24.4.6.5 Cross-References**

1279 *aud_delete_hdr()*, 24.4.5; *aud_init_record()*, 24.4.27; *aud_put_hdr()*, 24.4.30;
1280 *aud_put_hdr_info()*, 24.4.31; *aud_valid()*, 24.4.40; *aud_write()*, 24.4.41.

1281 **24.4.7 Delete Set of Object Attributes from a Record**

1282 Function: *aud_delete_obj()*

1283 **24.4.7.1 Synopsis**

```
1284 #include <sys/audit.h>
1285 int aud_delete_obj (aud_obj_t aud_obj_d);
```

1286 **24.4.7.2 Description**

1287 The *aud_delete_obj()* function deletes a set of object attributes from an audit
1288 record, including any data items within the set. The set to be deleted is defined
1289 by the *aud_obj_d* descriptor. Upon successful execution, the set of object attri-
1290 butes shall no longer be accessible in the record, and the *aud_obj_d* descriptor
1291 shall become undefined.

1292 Calls to this function shall not affect the status of descriptors for any other set of
1293 data in this or any other audit record.

1294 **24.4.7.3 Returns**

1295 Upon successful completion, the *aud_delete_obj()* function returns 0. Otherwise,
1296 it returns a value of -1 and *errno* is set to indicate the error. The audit record
1297 shall not be changed if the return value is -1.

1298 **24.4.7.4 Errors**

1299 If any of the following conditions occur, the *aud_delete_obj()* function shall return
1300 -1 and set *errno* to the corresponding value:

1301 [EINVAL] Argument *aud_obj_d* is not a valid descriptor for a set of object
1302 attributes within an audit record.

1303 **24.4.7.5 Cross-References**

1304 *aud_delete_obj_info()*, 24.4.8; *aud_init_record()*, 24.4.27; *aud_put_obj()*, 24.4.32;
1305 *aud_valid()*, 24.4.40; *aud_write()*, 24.4.41.

1306 **24.4.8 Delete Item from Set of Object Attributes**

1307 Function: *aud_delete_obj_info()*

1308 **24.4.8.1 Synopsis**

```
1309 #include <sys/audit.h>
1310 int aud_delete_obj_info (aud_obj_t aud_obj_d,
1311                         int item_id);
```

1312 **24.4.8.2 Description**

1313 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*.

1317 The value of *item_id* specifies an item within the set of object attributes. For
1318 system-generated records, the minimum set of items that shall be available,
1319 together with values of *item_id* to access them, are specified in section 24.4.22.
1320 For application-generated records, the values of *item_id* match the calls on
1321 *aud_put_obj_info()* that put the items into the set of object attributes.

1322 Calls to this function shall not affect the status of descriptors for any other data
1323 item in this or any other audit record.

1324 **24.4.8.3 Returns**

1325 Upon successful completion, the *aud_delete_obj_info()* function returns 0. Otherwise,
1326 it returns a value of -1 and *errno* is set to indicate the error. The audit
1327 record shall not be changed if the return value is -1.

1328 **24.4.8.4 Errors**

1329 If any of the following conditions occur, the *aud_delete_obj_info()* function shall
1330 return -1 and set *errno* to the corresponding value:

1331 [EINVAL] Argument *aud_obj_d* is not a valid descriptor for a set of object
1332 attributes within an audit record.

1333 Argument *item_id* does not reference a valid data item within
1334 *aud_obj_d*.
—

1335 **24.4.8.5 Cross-References**

1336 *aud_delete_obj()*, 24.4.7; *aud_init_record()*, 24.4.27; *aud_put_obj()*, 24.4.32;
1337 *aud_put_obj_info()*, 24.4.33; *aud_valid()*, 24.4.40; *aud_write()*, 24.4.41.

1338 **24.4.9 Delete Set of Subject Attributes from a Record**

1339 Function: *aud_delete_subj()*

1340 **24.4.9.1 Synopsis**

```
1341 #include <sys/audit.h>
1342 int aud_delete_subj (aud_subj_t aud_subj_d);
```

1343 **24.4.9.2 Description**

1344 The *aud_delete_subj()* function deletes a set of subject attributes from an audit
1345 record, including any data items within the set. The set to be deleted is defined
1346 by the *aud_subj_d* descriptor. Upon successful execution, the set of subject attri-
1347 butes shall no longer be accessible in the record, and the *aud_subj_d* descriptor
1348 shall become undefined.

1349 Calls to this function shall not affect the status of descriptors for any other set of
1350 data in this or any other audit record.

1351 **24.4.9.3 Returns**

1352 Upon successful completion, the *aud_delete_subj()* function returns 0. Otherwise,
1353 it returns a value of -1 and *errno* is set to indicate the error. The audit record
1354 shall not be changed if the return value is -1.

1355 **24.4.9.4 Errors**

1356 If any of the following conditions occur, the *aud_delete_subj()* function shall
1357 return -1 and set *errno* to the corresponding value:

1358 [EINVAL] Argument *aud_subj_d* is not a valid descriptor for a set of sub-
1359 ject attributes within an audit record.

1360 **24.4.9.5 Cross-References**

1361 *aud_delete_subj_info()*, 24.4.10; *aud_init_record()*, 24.4.27; *aud_put_subj()*,
1362 24.4.34; *aud_valid()*, 24.4.40; *aud_write()*, 24.4.41.

1363 **24.4.10 Delete Item from Set of Subject Attributes**

1364 Function: *aud_delete_subj_info()*

1365 **24.4.10.1 Synopsis**

```
1366 #include <sys/audit.h>
1367 int aud_delete_subj_info (aud_subj_t aud_subj_d,
1368                           int item_id);
```

1369 **24.4.10.2 Description**

1370 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
1371 item defined by *item_id* shall no longer be accessible in the set of subject attributes
1372 defined by *aud_subj_d*.
1373
1374 The value of *item_id* specifies an item within the set of subject attributes. For
1375 system-generated records, the minimum set of items that shall be available,
1376 together with values of *item_id* to access them, are specified in section 24.4.24.
1377 For application-generated records, the values of *item_id* match the calls on
1378 *aud_put_subj_info()* that put the items into the set of subject attributes.
1379 Calls to this function shall not affect the status of descriptors for any other data
1380 item in this or any other audit record.

1381 **24.4.10.3 Returns**

1382 Upon successful completion, the *aud_delete_subj_info()* function returns 0. Otherwise,
1383 it returns a value of -1 and *errno* is set to indicate the error. The audit
1384 record shall not be changed if the return value is -1.

1385 **24.4.10.4 Errors**

1386 If any of the following conditions occur, the *aud_delete_subj_info()* function shall
1387 return -1 and set *errno* to the corresponding value:

1388 [EINVAL] Argument *aud_subj_d* is not a valid descriptor for a set of sub-
1389 ject attributes within an audit record.
1390 Argument *item_id* does not reference a valid data item within
1391 *aud_subj_d*.

1392 **24.4.10.5 Cross-References**

1393 *aud_delete_subj()*, 24.4.9; *aud_init_record()*, 24.4.27; *aud_put_subj()*, 24.4.34;
1394 *aud_put_subj_info()*, 24.4.35; *aud_valid()*, 24.4.40; *aud_write()*, 24.4.41.

1395 **24.4.11 Duplicate an Audit Record**

1396 Function: *aud_dup_record()*

1397 **24.4.11.1 Synopsis**

```
1398 #include <sys/audit.h>
1399 aud_rec_t aud_dup_record (aud_rec_t ar);
```

1400 **24.4.11.2 Description**

1401 The *aud_dup_record()* function creates a duplicate of the audit record pointed to
1402 by argument *ar*. The duplicate shall be independent of the original record; subse-
1403 quent operations on either shall not affect the other. Upon successful execution,
1404 the *aud_dup_record()* function returns a pointer to the duplicate record.

1405 Any existing descriptors that refer to *ar* shall continue to refer to that record.
1406 Calls to *aud_dup_record()* shall not affect the status of any existing records.

1407 This function may cause memory to be allocated. The caller should free any
1408 releasable memory, when the new record is no longer required, by calling
1409 *aud_free()* with the *(void*)aud_rec_t* as an argument.

1410 **24.4.11.3 Returns**

1411 Upon successful completion, the *aud_dup_record()* function returns an *aud_rec_t*
1412 pointing to the new record. Otherwise, a value of *(aud_rec_t)NULL* shall be
1413 returned, the caller shall not have to free any releasable memory, and *errno* is set
1414 to indicate the error.

1415 **24.4.11.4 Errors**

1416 If any of the following conditions occur, the *aud_dup_record()* function shall
1417 return *(aud_rec_t)NULL* and set *errno* to the corresponding value:

1418 [EINVAL] Argument *ar* does not point to a valid audit record.

1419 [ENOMEM] The function requires more memory than is allowed by the
1420 hardware or system-imposed memory management constraints. –

1421 **24.4.11.5 Cross-References**

1422 *aud_free()*, 24.4.14; *aud_init_record()*, 24.4.27; *aud_valid()*, 24.4.40; *aud_write()*,
1423 24.4.41.

1424 **24.4.12 Map Text to Event Type**

1425 Function: *aud_evid_from_text()*

1426 **24.4.12.1 Synopsis**

```
1427 #include <sys/audit.h>
1428 int aud_evid_from_text (const char *text);
```

1429 **24.4.12.2 Description**

1430 The *aud_evid_from_text()* function returns the audit event type of the system
1431 audit event identified by the string pointed to by *text*. The means by which this
1432 information is obtained is unspecified.

1433 **24.4.12.3 Returns**

1434 Upon successful completion, the *aud_evid_from_text()* function returns the event
1435 type associated with *text*. On error, or if the requested entry is not found a value
1436 of -1 is returned and *errno* is set to indicate the error.

1437 **24.4.12.4 Errors**

1438 If any of the following conditions occur, the *aud_evid_from_text()* function shall
1439 return a value of -1 and set *errno* to the corresponding value:

1440 [EINVAL] The *text* argument does not identify a valid system audit event
1441 type. —

1442 **24.4.12.5 Cross-References**

1443 *aud_evid_to_text()*, 24.4.13; *aud_get_hdr_info()*, 24.4.19; *aud_put_hdr_info()*,
1444 24.4.31.

1445 **24.4.13 Map Event Type to Text**

1446 Function: *aud_evid_to_text()*

1447 **24.4.13.1 Synopsis**

```
1448 #include <sys/audit.h>
1449 char *aud_evid_to_text (int event_type, ssize_t *aud_info_length);
```

1450 **24.4.13.2 Description**

1451 The *aud_evid_to_text()* function shall transform the system audit *event_type* into
1452 a human-readable, null terminated character string identifying an event type.
1453 The means by which this information is obtained is unspecified. The function
1454 shall return the address of the string, and set the location pointed to by
1455 *aud_info_length* to the length of the string (not including the null terminator).

1456 This function may cause memory to be allocated. The caller should free any
1457 releasable memory when the string is no longer required, by calling the *aud_free()*
1458 function with the string address (cast to a (*void* *)) as an argument.

1459 **24.4.13.3 Returns**

1460 Upon successful completion, the *aud_evid_to_text()* function returns a pointer to a
1461 string containing the event name associated with *event_type*. On error, or if the
1462 requested entry is not found, *(char *)NULL* is returned, the caller shall not have
1463 to free any releasable memory, and *errno* is set to indicate the error.

1464 **24.4.13.4 Errors**

1465 If any of the following conditions occur, the *aud_evid_to_text()* function shall
1466 return *(char *)NULL* and set *errno* to the corresponding value:

1467 [EINVAL] The *event_type* argument does not contain a valid system audit
1468 event type.

1469 [ENOMEM] The string to be returned requires more memory than is allowed
1470 by the hardware or system-imposed memory management con-
1471 straints.

1472 **24.4.13.5 Cross-References**

1473 *aud_evid_from_text();* 24.4.12. *aud_get_hdr_info(),* 24.4.19; *aud_put_hdr_info(),*
1474 24.4.31.

1475 **24.4.14 Release Memory Allocated to an Audit Data Object**

1476 Function: *aud_free()*

1477 **24.4.14.1 Synopsis**

```
1478 #include <sys/audit.h>
1479 int aud_free (void *obj_p);
```

1480 **24.4.14.2 Description**

1481 The function *aud_free()* shall free any releasable memory currently allocated to
1482 the item identified by *obj_p*. This may identify an audit record (i.e., be a
1483 *(void*)aud_rec_t*) or a pointer to a string or event list allocated by one of the audit
1484 functions.

1485 If the item identified by *obj_p* is an *aud_rec_t*, the *aud_rec_t* and any existing
1486 descriptors and *aud_info_t* items that refer to parts of the audit record shall
1487 become undefined. If it is a string (*char**), then use of the *char** shall become
1488 undefined.

1489 **24.4.14.3 Returns**

1490 Upon successful completion, the *aud_free()* function returns 0. Otherwise, a
1491 value of -1 shall be returned and *errno* shall be set to indicate the error, and the
1492 memory shall not be freed.

1493 **24.4.14.4 Errors**

1494 If any of the following conditions occur, the *aud_free()* function shall return -1
1495 and set *errno* to the corresponding value:

1496 [EINVAL] The *obj_p* argument does not identify an audit record, string or
1497 event list allocated by one of the audit functions. —

1498 **24.4.14.5 Cross-References**

1499 *aud_copy_int()*, 24.4.2; *aud_dup_record()*, 24.4.11; *aud_get_all_evid()*, 24.4.15;
1500 *aud_get_event()*, 24.4.16; *aud_get_event_info()*, 24.4.17; *aud_get_hdr()*, 24.4.18;
1501 *aud_get_hdr_info()*, 24.4.19; *aud_get_obj()*, 24.4.21; *aud_get_obj_info()*, 24.4.22;
1502 *aud_get_subj()*, 24.4.23; *aud_get_subj_info()*, 24.4.24; *aud_id_to_text()*, 24.4.26;
1503 *aud_init_record()*, 24.4.27; *aud_read()*, 24.4.36; *aud_rec_to_text()*, 24.4.37;
1504 *aud_valid()*, 24.4.40.

1505 **24.4.15 Get All Audit Event Types**

1506 Function: *aud_get_all_evid()*

1507 **24.4.15.1 Synopsis**

```
1508 #include <sys/audit.h>
1509 int *aud_get_all_evid (void)
```

1510 **24.4.15.2 Description**

1511 The *aud_get_all_evid()* function returns the list of event types for system-
1512 generated events currently reportable on a conforming implementation. Each
1513 event type is a non-negative integer; the list is terminated by a negative value.
1514 The means by which this information is obtained is unspecified. These event
1515 types can be converted into textual format by the *aud_evid_to_text()* function.

1516 This function may cause memory to be allocated. The caller should free any
1517 releasable memory when the event list is no longer required, by calling the
1518 *aud_free()* function with the event list address (cast to a *void**) as an argument.

1519 **24.4.15.3 Returns**

1520 Upon successful completion, the *aud_get_all_evid()* function returns a pointer to a
1521 list of the system-generated event types currently reportable on a conforming
1522 implementation. Otherwise, *(int *)NULL* is returned, the caller shall not have to
1523 free any releasable memory, and *errno* is set to indicate the error.

1524 **24.4.15.4 Errors**

1525 If any of the following conditions occur, the *aud_get_all_evid()* function shall
1526 return *(int *)NULL* and set *errno* to the corresponding value:

1527 [ENOMEM] The event types to be returned require more memory than is
1528 allowed by the hardware or system-imposed memory manage-
1529 ment constraints. —

1530 **24.4.15.5 Cross-References**

1531 *aud_free()*, 24.4.14; *aud_evid_from_text()*, 24.4.12; *aud_evid_to_text()*, 24.4.13.

1532 **24.4.16 Get Audit Record Event-specific Data Descriptor**

1533 Function: *aud_get_event()*

1534 **24.4.16.1 Synopsis**

```
1535 #include <sys/audit.h>
1536 int aud_get_event (aud_rec_t ar,
1537                     int index,
1538                     aud_evinfo_t *aud_event_p);
```

1539 **24.4.16.2 Description**

1540 The *aud_get_event()* function returns a descriptor to a set of event-specific data
1541 from an audit record. The function accepts an audit record pointer *ar* returned
1542 from a previously successful call to *aud_read()*, *aud_init_record()* or
1543 *aud_dup_record()*. If *aud_event_p* is not **NULL**, then upon successful execution
1544 the *aud_get_event()* function shall return a descriptor via *aud_event_p* for the set
1545 of event-specific data identified by *index*. The descriptor returned by this call can
1546 then be used in subsequent calls on *aud_get_event_info()* to extract the data items
1547 from the set of event-specific data from the audit record. If *aud_event_p* is **NULL**,
1548 then the value of the *index* argument is ignored and the function just returns a
1549 value as described below.

1550 Calls to *aud_get_event()* shall not affect the status of any other existing descrip-
1551 tors. Calls on the various *aud_get_**() functions can be interleaved without affect-
1552 ing each other.

1553 This function may cause memory to be allocated. The caller should free any
1554 releasable memory, when the record is no longer required, by calling *aud_free()*
1555 with the *(void*)aud_rec_t* as an argument.

1556 A descriptor for the first set of event-specific data in the record is obtained by sup-
1557 plying an *index* of 1. While the standard does not require more than one set of
1558 event-specific data to be present in a record, an implementation or application
1559 may add additional sets that can be read by supplying values of *index* that are
1560 greater than 1.

1561 **24.4.16.3 Returns**

1562 Upon successful completion, the *aud_get_event()* function returns a non-negative
1563 value. This value indicates the number of sets of event-specific data in the record.
1564 In the event of failure the *aud_get_event()* function returns a value of -1, the
1565 caller shall not have to free any releasable memory, and *errno* is set to indicate
1566 the error. The *aud_evinfo_t* referenced by *aud_event_p* shall not be affected if the
1567 return value is -1.

1568 **24.4.16.4 Errors**

1569 If any of the following conditions occur, the *aud_get_event()* function shall return
1570 -1 and set *errno* to the corresponding value:

1571 [EINVAL] Argument *ar* does not point to a valid audit record.

1572 Argument *index* does not identify a valid set of event-specific
1573 data in the record.

1574 [ENOMEM] The function requires more memory than is allowed by the
1575 hardware or system-imposed memory management constraints. –

1576 **24.4.16.5 Cross-References**

1577 *aud_free()*, 24.4.14; *aud_get_event_info()*, 24.4.17; *aud_put_event()*, 24.4.28;
1578 *aud_read()*, 24.4.36; *aud_valid()*, 24.4.40.

1579 **24.4.17 Examine Audit Record Event-specific Data**

1580 Function: *aud_get_event_info()*

1581 **24.4.17.1 Synopsis**

```
1582 #include <sys/audit.h>
1583 int aud_get_event_info (aud_evinfo_t aud_event_d,
1584                         int item_id,
1585                         aud_info_t *aud_event_info_p);
```

1586 **24.4.17.2 Description**

1587 The *aud_get_event_info()* function returns a data item from within a set of event-
1588 specific data. The set of event-specific data within an audit record to be examined
1589 is identified by *aud_event_d* which was obtained from a previous successful call to
1590 *aud_get_event()* or *aud_put_event()*. If *aud_event_info_p* is not **NULL**, then upon
1591 successful execution the *aud_get_event_info()* function shall return via
1592 *aud_event_info_p* an *aud_info_t* for the data identified by *item_id*. If
1593 *aud_event_info_p* is **NULL**, then the value of the *item_id* argument is ignored, and
1594 the function just returns a value as described in the Returns section below.

1595 The value of *item_id* may specify a named item within the set of event-specific
1596 data, or may specify the ‘first’ item or the ‘next’ item. The named items available
1597 are dependent upon the *event type* of the audit record being examined; for each
1598 POSIX-defined event type the minimum set of items that shall be available,
1599 together with values of *item_id* to access them, are specified in section 24.2.2.

1600 If *item_id* is **AUD_FIRST_ITEM**, then this specifies the first item of event-specific
1601 data in the set. A call of *aud_get_event_info()* with *item_id* set to
1602 **AUD_NEXT_ITEM** shall return the item that follows the previous one read; for
1603 POSIX-defined events, the required items are returned in the order they are
1604 defined for each event type in section 24.2.2; implementations may report addi-
1605 tional items after the required items. If **AUD_NEXT_ITEM** is used when there
1606 has not been a previous successful call of this function for this set of event infor-
1607 mation, the effect is unspecified.

1608 Any existing descriptors shall not be affected by use of this function. Calls on the
1609 various *aud_get_**() functions can be interleaved without affecting each other.

1610 This function may cause memory to be allocated. The caller should free any
1611 releasable memory, when the record containing *aud_event_d* is no longer
1612 required, by calling *aud_free()* with the *aud_rec_t* for the record (cast to a *(void*)*)
1613 as an argument.

1614 **24.4.17.3 Returns**

1615 Upon successful completion, the *aud_get_event_info()* function returns a non-
1616 negative value. This value indicates the number of items of event-specific data in
1617 the set.

1618 In the event of failure the *aud_get_event_info()* function returns a value of -1, the
1619 caller shall not have to free any releasable memory, and *errno* is set to indicate
1620 the error. The *aud_info_t* referenced by *aud_event_info_p* shall not be affected if
1621 the return value is -1.

1622 **24.4.17.4 Errors**

1623 If any of the following conditions occur, the *aud_get_event_info()* function shall
1624 return -1 and set *errno* to the corresponding value:

1625 [EINVAL] Argument *aud_event_d* is not a valid descriptor for a set of
1626 event-specific data within an audit record.
1627 Argument *item_id* does not identify a valid item from the set of
1628 event-specific data.
1629 [ENOMEM] The function requires more memory than is allowed by the
1630 hardware or system-imposed memory management constraints. –

1631 **24.4.17.5 Cross-References**

1632 *aud_free()*, 24.4.14; *aud_get_event()*, 24.4.16; *aud_put_event_info()*, 24.4.29;
1633 *aud_read()*, 24.4.36; *aud_valid()*, 24.4.40.

1634 **24.4.18 Get an Audit Record Header Descriptor**

1635 Function: *aud_get_hdr()*

1636 **24.4.18.1 Synopsis**

```
1637 #include <sys/audit.h>
1638 int aud_get_hdr (aud_rec_t ar,
1639                   int index,
1640                   aud_hdr_t *aud_hdr_p);
```

1641 **24.4.18.2 Description**

1642 The *aud_get_hdr()* function returns a descriptor to the header of an audit record.
1643 The function accepts an audit record pointer *ar* returned from a previously suc-
1644 cessful call to *aud_read()*, *aud_init_record()* or *aud_dup_record()*. If *aud_hdr_p*
1645 is not **NULL**, then upon successful execution the *aud_get_hdr()* function shall
1646 return a descriptor via *aud_hdr_p* for the header identified by *index*. The descrip-
1647 tor returned by this call can then be used in subsequent calls to
1648 *aud_get_hdr_info()* to extract the data from the audit record header. If
1649 *aud_hdr_p* is **NULL**, then the value of the *index* argument is ignored, and the
1650 function just returns a value as described below.

1651 Calls to *aud_get_hdr()* shall not affect the status of any other existing descriptors.
1652 Calls on the various *aud_get_**() functions can be interleaved without affecting
1653 each other.

1654 This function may cause memory to be allocated. The caller should free any
1655 releasable memory, when the record is no longer required, by calling *aud_free()*
1656 with the (*void**)*aud_rec_t* as an argument.

1657 A descriptor for the first header in the record is obtained by supplying an *index* of
1658 1. While the standard does not require more than one header to be present in a
1659 record, an implementation or application may add additional headers that can be
1660 read by supplying values of *index* that are greater than 1.

1661 **24.4.18.3 Returns**

1662 Upon successful completion, the *aud_get_hdr()* function returns a non-negative
1663 value. This value indicates the number of headers in the record.

1664 In the event of failure the *aud_get_hdr()* function returns a value of -1, the caller
1665 shall not have to free any releasable memory, and *errno* is set to indicate the
1666 error. The *aud_hdr_t* referenced by *aud_hdr_p* shall not be affected if the return
1667 value is -1.

1668 **24.4.18.4 Errors**

1669 If any of the following conditions occur, the *aud_get_hdr()* function shall return -1
1670 and set *errno* to the corresponding value:

1671 [EINVAL] Argument *ar* does not point to a valid audit record.

1672 Argument *index* does not identify a valid header in the record.

1673 [ENOMEM] The function requires more memory than is allowed by the
1674 hardware or system-imposed memory management constraints. –

1675 **24.4.18.5 Cross-References**

1676 *aud_free()*, 24.4.14; *aud_get_hdr_info()*, 24.4.19; *aud_put_hdr()*, 24.4.30;
1677 *aud_read()*, 24.4.36; *aud_valid()*, 24.4.40.

1678 **24.4.19 Examine an Audit Record Header**

1679 Function: *aud_get_hdr_info()*

1680 **24.4.19.1 Synopsis**

```
1681 #include <sys/audit.h>
1682 int aud_get_hdr_info (aud_hdr_t aud_hdr_d,
1683                      int item_id,
1684                      aud_info_t *aud_hdr_info_p);
```

1685 **24.4.19.2 Description**

1686 The *aud_get_hdr_info()* function returns a data item from within a header of an
1687 audit record. The audit record header to be examined is identified by *aud_hdr_d*
1688 which was obtained from a previous successful call to *aud_get_hdr()* or
1689 *aud_put_hdr()*. If *aud_hdr_info_p* is not **NULL**, then upon successful execution
1690 the *aud_get_hdr_info()* function shall return via *aud_hdr_info_p* an *aud_info_t* for
1691 the item of event-specific data identified by *item_id*. If *aud_hdr_info_p* is **NULL**,
1692 then the value of the *item_id* argument is ignored, and the function just returns a
1693 value as described in the Returns section below.

1694 The value of *item_id* may specify a named item within the set of header data, or
1695 may specify the ‘first’ item or the ‘next’ item.

1696 The minimum set of named items to be available from the first header of an audit
1697 record is specified in the table below, together with values of *item_id* to access the
1698 items. If a record contains more than one header, the contents of the second and
1699 subsequent headers is not specified by this standard.

1700 **Table 24-10 – aud_hdr_info_p Values**

1702	Type	Description	item_id	Notes
1703	AUD_TYPE_SHORT	The format of the audit record	AUD_FORMAT	(1)
1704	AUD_TYPE_SHORT	The version number of the record	AUD_VERSION	(2)
1705	AUD_TYPE_AUD_ID	The audit ID of the process	AUD_AUD_ID	(3)
1706	AUD_TYPE_INT or AUD_TYPE_STRING	The event type of the record	AUD_EVENT_TYPE	(4)
1708	AUD_TYPE_AUD_TIME	The time the event occurred	AUD_TIME	
1709	AUD_TYPE_AUD_STATUS	The audit status of the event	AUD_STATUS	
1710	AUD_TYPE_INT	Value returned for event (<i>errno</i>)	AUD_ERRNO	(5)

1711 Notes on the table:

1712 (1) Only one value is currently defined for the *format* item: AUD_NATIVE.
1713 All data in any given record shall be in the same format.

1714 (2) The *version* item provides a means of identifying the version of the
1715 POSIX.1e audit option to which the audit record conforms. Conforming
1716 applications can make use of the *version* to provide for backward compa-
1717 tibility or to ignore records which they are not prepared to handle.

1718 Currently only one value for *version* is defined: AUD_STD_NNNN_N. |
1719 This identifies records which conform to the initial version of this stan-
1720 dard. Further revisions of this standard may define additional values for
1721 the header version. The NNNN_N is merely a placeholder for the year |
1722 (e.g., 1997) this standard is approved and standard (e.g., _1 implying
1723 POSIX.1) it is placed into.

1724 (3) If the record is not associated with any accountable user (e.g., it was
1725 recorded before a user had completed authentication), then the
1726 *aud_get_hdr_info()* function shall return an *aud_info_t* with a zero
1727 *aud_info_length* member.

1728 (4) The event type is an integer if this is a system-generated event, or a
1729 string if it is an application-generated event.

1730 (5) For system-generated events, the return value reported contains the
1731 *errno* on return from the function audited; if the operation succeeded (as
1732 shown by the *status*), this value is undefined. For application-generated
1733 records there may be no *errno* reported.

1734 If *item_id* is AUD_FIRST_ITEM, then this specifies the first of the items of infor-
1735 mation from the header. A call of *aud_get_hdr_info()* with *item_id* set to
1736 AUD_NEXT_ITEM shall return the item that follows the previous one read; for
1737 the POSIX-defined header, the required items are returned in the order they are

1738 defined in the table above; implementations may report additional items after the
1739 required items. If AUD_NEXT_ITEM is used when there has not been a previous
1740 successful call of this function for this header, the effect is unspecified.

1741 This function may cause memory to be allocated. The caller should free any
1742 releasable memory, when the record containing *aud_hdr_d* is no longer required,
1743 by calling *aud_free()* with the *aud_rec_t* for the record (cast to a (*void**)) as an
1744 argument.

1745 Any existing descriptors shall not be affected by use of this function. Calls on the
1746 various *aud_get_**() functions can be interleaved without affecting each other.

1747 **24.4.19.3 Returns**

1748 Upon successful completion, the *aud_get_hdr_info()* function returns a non-
1749 negative value. This value indicates the number of items of header information in
1750 the set.

1751 In the event of failure the *aud_get_hdr_info()* function returns a value of -1, the
1752 caller shall not have to free any releasable memory, and *errno* is set to indicate
1753 the error. The *aud_info_t* referenced by *aud_hdr_info_p* shall not be affected if
1754 the return value is -1.

1755 **24.4.19.4 Errors**

1756 If any of the following conditions occur, the *aud_get_hdr_info()* function shall
1757 return -1 and set *errno* to the corresponding value:

1758 [EINVAL] Argument *aud_hdr_d* is not a valid descriptor for an audit
1759 record header within an audit record.

1760 Argument *item_id* does not identify a valid item from the
1761 header.

1762 [ENOMEM] The function requires more memory than is allowed by the
1763 hardware or system-imposed memory management constraints. –

1764 **24.4.19.5 Cross-References**

1765 *aud_free()*, 24.4.14; *aud_get_hdr()*, 24.4.18; *aud_put_hdr_info()*, 24.4.31;
1766 *aud_read()*, 24.4.36; *aud_valid()*, 24.4.40.

1767 **24.4.20 Get a Process Audit ID**

1768 Function: *aud_get_id()*

1769 **24.4.20.1 Synopsis**

```
1770 #include <sys/audit.h>
1771 #include <sys/types.h>
1772 aud_id_t aud_get_id (pid_t pid);
```

1773 **24.4.20.2 Description**

1774 The *aud_get_id()* function returns the audit ID of the user who is accountable for
1775 auditable actions of the existing process identified by *pid*.
1776 It is unspecified whether appropriate privilege is required to use this function.

1777 **24.4.20.3 Returns**

1778 Upon successful completion, the *aud_get_id()* function returns the audit ID of the
1779 nominated process. Otherwise, a value of ((*aud_id_t*)−1) is returned and *errno* is
1780 set to indicate the error.

1781 **24.4.20.4 Errors**

1782 If any of the following conditions occur, the *aud_get_id()* function shall return a
1783 value of ((*aud_id_t*)−1) and set *errno* to the corresponding value:

1784 [EINVAL] The value of the *pid_t* argument is invalid. —

1785 **24.4.20.5 Cross-References**

1786 *aud_id_to_text()*, 24.4.26; *aud_put_hdr_info()*, 24.4.31; *aud_write()*, 24.4.41.

1787 **24.4.21 Get an Audit Record Object Descriptor**

1788 Function: *aud_get_obj()*

1789 **24.4.21.1 Synopsis**

```
1790 #include <sys/audit.h>
1791 int aud_get_obj (aud_rec_t ar,
1792                   int index,
1793                   aud_obj_t *aud_obj_p);
```

1794 **24.4.21.2 Description**

1795 The *aud_get_obj()* function returns a descriptor to a set of object attributes from
1796 an audit record. The function accepts an audit record pointer *ar* returned from a
1797 previously successful call to *aud_read()*, *aud_init_record()* or *aud_dup_record()*.
1798 If *aud_obj_p* is not **NULL**, then upon successful execution the *aud_get_obj()* func-
1799 tion shall return a descriptor via *aud_obj_p* for the set of object data identified by

1800 *index*. The descriptor returned by this call can then be used in subsequent calls
1801 to *aud_get_obj_info()* to extract the object data for that object. If *aud_obj_p* is
1802 **NULL**, then the function just returns a value as described below.

1803 Calls to *aud_get_obj()* shall not affect the status of any other existing descriptors.
1804 Calls on the various *aud_get_**() functions can be interleaved without affecting
1805 each other.

1806 This function may cause memory to be allocated. The caller should free any
1807 releasable memory, when the record is no longer required, by calling *aud_free()*
1808 with the (*void**)*aud_rec_t* as an argument.

1809 A descriptor for the first set of object attributes in the record is obtained by sup-
1810 plying an *index* of 1. Any additional sets can be read by supplying values of *index*
1811 that are greater than 1.

1812 **24.4.21.3 Returns**

1813 Upon successful completion, the *aud_get_obj()* function returns a non-negative
1814 value. This value indicates the number of sets of object attributes in the record.

1815 In the event of failure the *aud_get_obj()* function returns a value of -1, the caller
1816 shall not have to free any releasable memory, and *errno* is set to indicate the
1817 error. The *aud_obj_t* referenced by *aud_obj_p* shall not be affected if the return
1818 value is -1.

1819 **24.4.21.4 Errors**

1820 If any of the following conditions occur, the *aud_get_obj()* function shall return -1
1821 and set *errno* to the corresponding value:

1822 [EINVAL] Argument *ar* does not point to a valid audit record.

1823 Argument *index* does not identify a valid set of object attributes
1824 in the record.

1825 [ENOMEM] The function requires more memory than is allowed by the
1826 hardware or system-imposed memory management constraints. –

1827 **24.4.21.5 Cross-References**

1828 *aud_free()*, 24.4.14; *aud_get_obj_info()*, 24.4.22; *aud_put_obj()*, 24.4.32;
1829 *aud_read()*, 24.4.36; *aud_valid()*, 24.4.40.

1830 **24.4.22 Examine Audit Record Object Data**

1831 Function: *aud_get_obj_info()*

1832 **24.4.22.1 Synopsis**

```
1833 #include <sys/audit.h>
1834 int aud_get_obj_info (aud_obj_t aud_obj_d,
1835                     int item_id,
1836                     aud_info_t *aud_obj_info_p);
```

1837 **24.4.22.2 Description**

1838 The *aud_get_obj_info()* function returns a data item from within a set of object
1839 data. For system-generated events recording use of an interface that changes
1840 object attributes, the attributes reported are those at the start of the event. The
1841 set of object data to be examined is identified by *aud_obj_d* which was obtained
1842 from a previous successful call to *aud_get_obj()* or *aud_put_obj()*. If
1843 *aud_obj_info_p* is not **NULL**, then upon successful execution the
1844 *aud_get_obj_info()* function shall return via *aud_obj_info_p* an *aud_info_t* for the
1845 object attribute identified by *item_id*. If *aud_obj_info_p* is **NULL**, then the value
1846 of the *item_id* argument is ignored, and the function just returns a value as
1847 described in the Returns section below.

1848 The value of *item_id* may specify a named item within the set of object data or
1849 may specify the ‘first’ item or the ‘next’ item.

1850 The minimum set of named items that shall be available for system generated
1851 events that are required to report object attributes is specified in the table below,
1852 together with values of *item_id* to access them:

1853 **Table 24-11 – aud_obj_info_p Values**

1854	Type	Description	item_id	Notes
1856	AUD_TYPE_AUD_OBJ_TYPE	The type of the object	AUD_TYPE	
1857	AUD_TYPE_UID	The user ID of the object owner	AUD_UID	(1)
1858	AUD_TYPE_GID	The group ID of the object owner	AUD_GID	(2)
1859	AUD_TYPE_MODE	The mode bits of the object	AUD_MODE	(3)
1860	AUD_TYPE_STRING	The name of the object	AUD_NAME	(4)
1861	AUD_TYPE_ACL	The ACL of the object	AUD_ACL	(5)
1862	AUD_TYPE_MAC	The MAC label of the object	AUD_MAC_LBL	(6)
1863	AUD_TYPE_INF	The information label of the object	AUD_INF_LBL	(7)
1864	AUD_TYPE_CAP	The capability set of the object	AUD_CAP	+

1865 Notes on the table:

- 1866 (1) For a process object, the object owner is the effective UID of the process.
- 1867 (2) For a process object, the object group is the effective GID of the process.
- 1868 (3) For a process object, the *aud_get_obj_info()* function may return an
1869 *aud_info_t* with a zero *aud_info_length* member for the mode bits.
- 1870 (4) This item contains the name of the object, which shall provide sufficient
1871 information to identify the object.
- 1872 (5) This item contains an *acl_t* recording the ACL of the object at the start of
1873 the event. If {_POSIX_ACL} was not defined at that time, or the object |

1874 does not have a POSIX.1e conformant ACL, the *aud_get_obj_info()* func-
1875 tion shall return an *aud_info_t* with a zero *aud_info_length* member.

1876 (6) This item contains a *mac_t* recording the MAC label of the object at the
1877 start of the event. If *{_POSIX_MAC}* was not defined at that time, the |
1878 *aud_get_obj_info()* function shall return an *aud_info_t* with a zero |
1879 *aud_info_length* member.

1880 (7) This item contains an *inf_t* recording the information label of the object
1881 at the start of the event. If *{_POSIX_INF}* was not defined at that time, |
1882 the *aud_get_obj_info()* function shall return an *aud_info_t* with a zero |
1883 +
1884 (8) This item contains a *cap_t* recording the capability set of the object at the+
1885 start of the event. If *{_POSIX_CAP}* was not in effect at that time or if the+
1886 object does not have a POSIX.1e conformant capability set, the +
1887 *aud_get_obj_info()* function shall return an *aud_info_t* with a zero +
1888 *aud_info_length* member.
1889 If *item_id* is AUD_FIRST_ITEM, this specifies the first of the items of information
1890 from the set. A call of *aud_get_obj_info()* with *item_id* set to AUD_NEXT_ITEM
1891 shall return the item that follows the previous one read; for system-generated
1892 events that are required to report object attributes, the required items are
1893 returned in the order they are defined in the table above; implementations may
1894 report additional items after the required items. If AUD_NEXT_ITEM is used
1895 when there has not been a previous successful call of this function for this set of
1896 object attributes, the effect is unspecified.
1897 Only the object type and object owner items are required. The other specified
1898 items are optional. If an item is not available, the function *aud_get_obj_info()*
1899 shall return a *aud_info_t* with a zero *aud_info_length* member.
1900 Any existing descriptors shall not be affected by use of this function. Calls on the
1901 various *aud_get_**() functions can be interleaved without affecting each other.
1902 This function may cause memory to be allocated. The caller should free any
1903 releasable memory, when the record containing *aud_obj_d* is no longer required,
1904 by calling *aud_free()* with the *aud_rec_t* for the record (cast to a (*void**)) as an
1905 argument.

1906 24.4.22.3 Returns

1907 Upon successful completion, the *aud_get_obj_info()* function returns a non-
1908 negative value. This value indicates the number of items of object attributes in
1909 the set.
1910 In the event of failure the *aud_get_obj_info()* function returns a value of -1, the
1911 caller shall not have to free any releasable memory, and *errno* is set to indicate
1912 the error. The *aud_info_t* referenced by *aud_obj_info_p* shall not be affected if the
1913 return value is -1.

1914 **24.4.22.4 Errors**

1915 If any of the following conditions occur, the *aud_get_obj_info()* function shall
1916 return -1 and set *errno* to the corresponding value: If any of the following condi-
1917 tions occur, this function will fail and set *errno* to one of the following values:

1918 [EINVAL] Argument *aud_obj_d* is not a valid descriptor for a set of object
1919 attributes within an audit record.

1920 Argument *item_id* does not identify a valid item from the set of
1921 object attributes.

1922 [ENOMEM] The function requires more memory than is allowed by the
1923 hardware or system-imposed memory management constraints. –

1924 **24.4.22.5 Cross-References**

1925 *aud_free()*, 24.4.14; *aud_get_obj()*, 24.4.21; *aud_put_obj_info()*, 24.4.33;
1926 *aud_read()*, 24.4.36; *aud_valid()*, 24.4.40.

1927 **24.4.23 Get an Audit Record Subject Descriptor**

1928 Function: *aud_get_subj()*

1929 **24.4.23.1 Synopsis**

```
1930 #include <sys/audit.h>
1931 int aud_get_subj (aud_rec_t ar,
1932                    int index,
1933                    aud_subj_t *aud_subj_p);
```

1934 **24.4.23.2 Description**

1935 The *aud_get_subj()* function returns a descriptor to a set of subject attributes
1936 from an audit record. The function accepts an audit record pointer *ar* returned
1937 from a previously successful call to *aud_read()*, *aud_init_record()* or
1938 *aud_dup_record()*. If *aud_subj_p* is not **NULL**, then upon successful execution the
1939 *aud_get_subj()* function shall return a descriptor via *aud_subj_p* for the set of
1940 subject attributes identified by *index*. The descriptor returned by this call can
1941 then be used in subsequent calls to *aud_get_subj_info()* to extract the attributes
1942 for that process. If *aud_subj_p* is **NULL**, then the function just returns a value as
1943 described below.

1944 Calls to *aud_get_subj()* shall not affect the status of any other existing descrip-
1945 tors. Calls on the various *aud_get_**() functions can be interleaved without affect-
1946 ing each other.

1947 This function may cause memory to be allocated. The caller should free any
1948 releasable memory, when the record is no longer required, by calling *aud_free()*
1949 with the *(void*)aud_rec_t* as an argument.

1950 A descriptor for the first set of subject attributes in the record is obtained by sup-
1951 plying an *index* of 1. While the standard does not require more than one set of
1952 subject attributes to be present in a record, an implementation or application may
1953 add additional sets that can be read by supplying values of *index* that are greater
1954 than 1.

1955 **24.4.23.3 Returns**

1956 Upon successful completion, the *aud_get_subj()* function returns a non-negative
1957 value. This value indicates the number of sets of subject attributes in the record.

1958 In the event of failure the *aud_get_subj()* function returns a value of -1, the caller
1959 shall not have to free any releasable memory, and *errno* is set to indicate the
1960 error. The *aud_subj_t* referenced by *aud_subj_p* shall not be affected if the return
1961 value is -1.

1962 **24.4.23.4 Errors**

1963 If any of the following conditions occur, the *aud_get_subj()* function shall return
1964 -1 and set *errno* to the corresponding value:

1965 [EINVAL] Argument *ar* does not point to a valid audit record.

1966 Argument *index* does not identify a valid set of subject attributes
1967 in the record.

1968 [ENOMEM] The function requires more memory than is allowed by the
1969 hardware or system-imposed memory management constraints. –

1970 **24.4.23.5 Cross-References**

1971 *aud_free()*, 24.4.14; *aud_get_subj_info()*, 24.4.24; *aud_put_subj()* 24.4.34;
1972 *aud_read()*, 24.4.36; *aud_valid()*, 24.4.40.

1973 **24.4.24 Examine Audit Record Subject Data**

1974 Function: *aud_get_subj_info()*

1975 **24.4.24.1 Synopsis**

```
1976 #include <sys/audit.h>
1977 int aud_get_subj_info (aud_subj_t aud_subj_d,
1978                      int item_id,
1979                      aud_info_t *aud_subj_info_p);
```

1980 **24.4.24.2 Description**

1981 The *aud_get_subj_info()* function returns a data item from within a set of subject
 1982 attributes in an audit record. For system-generated events recording use of an
 1983 interface that changes subject attributes, the attributes reported are those at the
 1984 start of the event. The set of attributes to be examined is identified by
 1985 *aud_subj_d* which was obtained from a previous successful call to *aud_get_subj()*
 1986 or *aud_put_subj()*. If *aud_subj_info_p* is not **NULL**, then upon successful execu-
 1987 tion the *aud_get_subj_info()* function shall return via *aud_subj_info_p* an
 1988 *aud_info_t* for the attribute identified by *item_id*. If *aud_subj_info_p* is **NULL**,
 1989 then the value of the *item_id* argument is ignored, and the function just returns a
 1990 value as described in the Returns section below.

1991 The value of *item_id* may specify a named item within the set of subject attri-
 1992 butes, or may specify the ‘first’ item or the ‘next’ item. The minimum set of
 1993 named items that shall be available from system-generated records is specified in
 1994 the table below, together with values of *item_id* to access them:

1995 **Table 24-12 – aud_subj_info_p Values**

1996	Type	Description	item_id	Notes
1998	AUD_TYPE_PID	The process ID	AUD_PID	
1999	AUD_TYPE_UID	The effective user ID	AUD_EUID	
2000	AUD_TYPE_GID	The effective group ID	AUD_EGID	
2001	AUD_TYPE_GID	The supplementary group IDs	AUD_SGIDS	(1)
2002	AUD_TYPE_UID	The real user ID	AUD_RUID	
2003	AUD_TYPE_GID	The real group ID	AUD_RGID	
2004	AUD_TYPE_MAC	The process MAC label	AUD_MAC_LBL	(2)
2005	AUD_TYPE_INF	The process information label	AUD_INF_LBL	(3)
2006	AUD_TYPE_CAP	The process capability state	AUD_CAP	(4)

2007 Notes on the table:

- 2008 (1) The number of supplementary groups can be calculated from the
 2009 *aud_info_length* member of the *aud_info_t*.
- 2010 (2) If *{_POSIX_MAC}* was not defined at that time, the *aud_get_subj_info()* |
 2011 function shall return an *aud_info_t* with a zero *aud_info_length* member.
- 2012 (3) If *{_POSIX_INF}* was not defined at that time, the *aud_get_subj_info()* |
 2013 function shall return an *aud_info_t* with a zero *aud_info_length* member.
- 2014 (4) If *{_POSIX_CAP}* was not defined at that time, the *aud_get_subj_info()* |
 2015 function shall return an *aud_info_t* with a zero *aud_info_length* member.

2016 If *item_id* is AUD_FIRST_ITEM, then this specifies the first of the items from the
 2017 set of subject attributes. A call of *aud_get_subj_info()* with *item_id* set to
 2018 AUD_NEXT_ITEM shall return the item that follows the previous one read; for
 2019 system-generated records, the required items are returned in the order they are
 2020 defined in the table above; implementations may report additional items after the
 2021 required items. If AUD_NEXT_ITEM is used when there has not been a previous
 2022 successful call of this function for this set of subject attributes, the effect is
 2023 unspecified.

2024 For system-generated records, the first three items are required; the MAC label,
2025 information label and capability state are required if the relevant POSIX options
2026 are in effect; the other specified items are optional. If an item is not available, the
2027 function *aud_get_subj_info()* shall return an *aud_info_t* with a zero
2028 *aud_info_length* member.

2029 Any existing descriptors shall not be affected by use of this function. Calls on the
2030 various *aud_get_**() functions can be interleaved without affecting each other.

2031 This function may cause memory to be allocated. The caller should free any
2032 releasable memory, when the record containing *aud_subj_d* is no longer required,
2033 by calling *aud_free()* with the *aud_rec_t* for the record (cast to a (*char**)) as an
2034 argument.

2035 **24.4.24.3 Returns**

2036 Upon successful completion, the *aud_get_subj_info()* function returns a non-
2037 negative value. This value indicates the number of items of subject attributes in
2038 the set.

2039 In the event of failure the *aud_get_subj_info()* function returns a value of -1, the
2040 caller shall not have to free any releasable memory, and *errno* is set to indicate
2041 the error. The *aud_info_t* referenced by *aud_subj_info_p* shall not be affected if
2042 the return value is -1.

2043 **24.4.24.4 Errors**

2044 If any of the following conditions occur, the *aud_get_subj_info()* function shall
2045 return -1 and set *errno* to the corresponding value:

2046 [EINVAL] Argument *aud_subj_d* is not a valid descriptor for a set of sub-
2047 ject attributes within an audit record.

2048 Argument *item_id* does not identify a valid item from the set of
2049 subject attributes.

2050 [ENOMEM] The function requires more memory than is allowed by the
2051 hardware or system-imposed memory management constraints. –

2052 **24.4.24.5 Cross-References**

2053 *aud_free()*, 24.4.14; *aud_get_subj()*, 24.4.23; *aud_put_subj_info()*, 24.4.35;
2054 *aud_read()*, 24.4.36; *aud_valid()*, 24.4.40.

2055 **24.4.25 Map Text to Audit ID**

2056 Function: *aud_id_from_text()*

2057 **24.4.25.1 Synopsis**

```
2058 #include <sys/audit.h>
2059 aud_id_t aud_id_from_text (const char *text_p);
```

2060 **24.4.25.2 Description**

2061 The *aud_id_from_text()* function returns the audit ID identified by the string
2062 pointed to by *text_p*. The means by which this information is obtained is
2063 unspecified.

2064 **24.4.25.3 Returns**

2065 Upon successful completion, the *aud_id_from_text()* function returns the audit ID
2066 associated with *text_p*. On error, or if the requested entry is not found, a value of
2067 $((aud_id_t)-1)$ is returned and *errno* is set to indicate the error.

2068 **24.4.25.4 Errors**

2069 If any of the following conditions occur, the *aud_id_from_text()* function shall
2070 return a value of $((aud_id_t)-1)$ and set *errno* to the corresponding value:

2071 [EINVAL] The *text_p* argument does not identify a valid user. —

2072 **24.4.25.5 Cross-References**

2073 *aud_get_hdr_info()*, 24.4.19; *aud_id_to_text()*, 24.4.26; *aud_put_hdr_info()*,
2074 24.4.31.

2075 **24.4.26 Map Audit ID to Text**

2076 Function: *aud_id_to_text()*

2077 **24.4.26.1 Synopsis**

```
2078 #include <sys/audit.h>
2079 char *aud_id_to_text (aud_id_t audit_ID, ssize_t *len_p);
```

2080 **24.4.26.2 Description**

2081 The *aud_id_to_text()* function transforms the *audit_ID* into a human-readable,
2082 null terminated character string. The means by which this information is
2083 obtained is unspecified. Upon successful completion, the function shall return the

2084 address of the string, and set the location pointed to by *len_p* to the length of the
2085 string (not including the null terminator).

2086 This function may cause memory to be allocated. The caller should free any
2087 releasable memory when the text form of *audit_ID* is no longer required, by cal-
2088 ling *aud_free()* with the string address (cast to a (*void**)) as an argument.

2089 **24.4.26.3 Returns**

2090 Upon successful completion, the *aud_id_to_text()* function returns a pointer to a
2091 string identifying the user associated with *audit_ID*. On error, or if the requested
2092 entry is not found, the caller shall not have to free any releasable memory,
2093 (*char**)**NULL** is returned, the location pointed to by *len_p* is not changed, and
2094 *errno* is set to indicate the error.

2095 **24.4.26.4 Errors**

2096 If any of the following conditions occur, the *aud_id_to_text()* function shall return
2097 (*char **)**NULL** and set *errno* to the corresponding value:

2098 [EINVAL] The *audit_ID* argument does not contain a valid audit identifier.

2099 [ENOMEM] The function requires more memory than is allowed by the
2100 hardware or system-imposed memory management constraints. –

2101 **24.4.26.5 Cross-References**

2102 *aud_free()*, 24.4.14; *aud_get_hdr_info()*, 24.4.19; *aud_id_from_text()*, 24.4.25;
2103 *aud_put_hdr_info()*, 24.4.31.

2104 **24.4.27 Create a New Audit Record**

2105 Function: *aud_init_record()*

2106 **24.4.27.1 Synopsis**

```
2107 #include <sys/audit.h>
2108 aud_rec_t aud_init_record (void);
```

2109 **24.4.27.2 Description**

2110 The *aud_init_record()* function returns a pointer to an audit record that is other-
2111 wise not in use. The record shall contain no headers or sets of subject, event-
2112 specific, or object information.

2113 Upon successful execution of the *aud_init_record()* function, the pointer returned
2114 can be used in subsequent calls to the *aud_put_**() functions to add information to
2115 the record, and in other functions that manipulate audit records, and the record
2116 can be written to an audit log by a call of *aud_write()*.

2117 Calls to *aud_init_record()* shall not affect the status of any existing records.
2118 This function may cause memory to be allocated. The caller should free any
2119 releasable memory, when the record is no longer required, by calling *aud_free()*
2120 with the *(void*)aud_rec_t* as an argument.

2121 **24.4.27.3 Returns**

2122 Upon successful completion, the *aud_init_record()* function returns an *aud_rec_t*
2123 pointing to the new record. Otherwise, a value of *(aud_rec_t)NULL* shall be
2124 returned, the caller shall not have to free any releasable memory, and *errno* is set
2125 to indicate the error.

2126 **24.4.27.4 Errors**

2127 If any of the following conditions occur, the *aud_init_record()* function shall
2128 return *(aud_rec_t)NULL* and set *errno* to the corresponding value:

2129 [ENOMEM] The function requires more memory than is allowed by the
2130 hardware or system-imposed memory management constraints. –

2131 **24.4.27.5 Cross-References**

2132 *aud_dup_record()*, 24.4.11; *aud_free()*, 24.4.14; *aud_put_event()*, 24.4.28;
2133 *aud_put_hdr()*, 24.4.30; *aud_put_obj()*, 24.4.32; *aud_put_subj()*, 24.4.34;
2134 *aud_write()*, 24.4.41.

2135 **24.4.28 Add Set of Event-specific Data to Audit Record**

2136 Function: *aud_put_event()*

2137 **24.4.28.1 Synopsis**

```
2138 #include <sys/audit.h>
2139 int aud_put_event (aud_rec_t ar,
2140                     const aud_evinfo_t *next_p,
2141                     aud_evinfo_t *new_p);
```

2142 **24.4.28.2 Description**

2143 The *aud_put_event()* function creates a new set of event-specific data, containing
2144 no data items, in an audit record, and returns a descriptor to the set. The func-
2145 tion accepts an audit record pointer *ar*, and puts the new set of event-specific data
2146 logically before the existing set *next_p* in the record. If *next_p* is **NULL**, then the
2147 new set shall be logically the last in the record.

2148 Upon successful execution the *aud_put_event()* function shall return via *new_p* a
2149 descriptor for the new set of event-specific data. The descriptor returned by this
2150 call can then be used in subsequent calls to *aud_put_event_info()* to add data to

2151 this set of event-specific data in the audit record.
2152 Calls to *aud_put_event()* shall not affect the status of any existing descriptors for
2153 this or any other audit record. Calls on the various *aud_put_**() functions can be
2154 interleaved without affecting each other.
2155 This function may cause memory to be allocated. The caller should free any
2156 releasable memory, when the record is no longer required, by calling *aud_free()*
2157 with the (*void**)*aud_rec_t* as an argument.

2158 **24.4.28.3 Returns**

2159 Upon successful completion, the *aud_put_event()* function returns 0. Otherwise, a
2160 value of -1 shall be returned, the caller shall not have to free any releasable
2161 memory, and *errno* is set to indicate the error. The audit record referenced by *ar*
2162 shall not be affected if the return value is -1.

2163 **24.4.28.4 Errors**

2164 If any of the following conditions occur, the *aud_put_event()* function shall return
2165 -1 and set *errno* to the corresponding value:

2166 [EINVAL] Argument *ar* does not point to a valid audit record.
2167 Argument *next_p* is neither **NULL** nor does it indicate an exist-
2168 ing set of event-specific data in record *ar*.
2169 [ENOMEM] The function requires more memory than is allowed by the
2170 hardware or system-imposed memory management constraints. –

2171 **24.4.28.5 Cross-References**

2172 *aud_free()*, 24.4.14; *aud_delete_event()*, 24.4.3; *aud_get_event()*, 24.4.16;
2173 *aud_init_record()*, 24.4.27; *aud_put_event_info()*, 24.4.29; *aud_valid()*, 24.4.40;
2174 *aud_write()*, 24.4.41.

2175 **24.4.29 Add Item to Set of Event-specific Data**

2176 Function: *aud_put_event_info()*

2177 **24.4.29.1 Synopsis**

```
2178 #include <sys/audit.h>
2179 int aud_put_event_info (aud_evinfo_t aud_event_d,
2180                         int position,
2181                         int item_id,
2182                         const aud_info_t *aud_event_info_p);
```

2183 **24.4.29.2 Description**

2184 The *aud_put_event_info()* function adds a data item to a set of event-specific data
2185 within an audit record. The function accepts a descriptor for a set of event-
2186 specific data *aud_event_d* in an audit record, and puts into the set of event-
2187 specific data the item with type, size and address defined in the structure refer-
2188 enced by *aud_event_info_p*. The item shall subsequently be identifiable by
2189 *item_id* in calls to functions as the record is manipulated, including after being
2190 written to and read back from an audit log; no item identifiable by *item_id* shall
2191 already exist in the set of event-specific information.

2192 The *position* argument shall specify either

- 2193 — the *item_id* of an item that already exists in the set of event-specific data;
2194 in this case the new data item shall be placed logically before the existing
2195 item
- 2196 — AUD_LAST_ITEM; in this case the new item shall be logically the last in
2197 the set.

2198 After the call of *aud_put_event_info()*, the caller can continue to manipulate the
2199 data item indicated by the *aud_info_t*, and the *aud_info_t* itself, and changes to
2200 them shall not affect the record unless they are used in a further call to
2201 *aud_put_*_info()*.

2202 Calls to *aud_put_event_info()* shall not affect the status of any other existing
2203 descriptors for this or any other audit record. Calls on the various
2204 *aud_put_*_info()* functions can be interleaved without affecting each other.

2205 This function may cause memory to be allocated. The caller should free any
2206 releasable memory, when the record is no longer required, by calling *aud_free()*
2207 with the (*void**)*aud_rec_t* as an argument.

2208 **24.4.29.3 Returns**

2209 Upon successful completion, the *aud_put_event_info()* function returns 0. Other-
2210 wise, it returns a value of -1, the caller shall not have to free any releasable
2211 memory, and *errno* is set to indicate the error. The set of event-specific data refer-
2212 enced by *aud_event_d* shall not be affected if the return value is -1.

2213 **24.4.29.4 Errors**

2214 If any of the following conditions occur, the *aud_put_event_info()* function shall
2215 return -1 and set *errno* to the corresponding value:

- 2216 [EINVAL] Argument *aud_event_d* is not a valid descriptor for a set of
2217 event-specific data within an audit record.
- 2218 Argument *position* is not AUD_LAST_ITEM and does not iden-
2219 tify a valid item from the set of event-specific data.
- 2220 The value of the *aud_info_type* field of the structure referenced
2221 by *aud_event_info_p* is invalid.

2222 An item with identifier *item_id* already exists in the set of
2223 event-specific data.
2224 The argument *item_id* is equal to AUD_FIRST_ITEM,
2225 AUD_NEXT_ITEM, or AUD_LAST_ITEM.
2226 [ENOMEM] The function requires more memory than is allowed by the
2227 hardware or system-imposed memory management constraints. –

2228 **24.4.29.5 Cross-References**

2229 *aud_delete_event_info()*, 24.4.4; *aud_free()*, 24.4.14; *aud_get_event_info()*, 24.4.17;
2230 *aud_put_event()*, 24.4.28; *aud_valid()*, 24.4.40; *aud_write()*, 24.4.41.

2231 **24.4.30 Add Header to Audit Record**

2232 Function: *aud_put_hdr()*

2233 **24.4.30.1 Synopsis**

```
2234 #include <sys/audit.h>
2235 int aud_put_hdr (aud_rec_t ar,
2236                   const aud_hdr_t *next_p,
2237                   aud_hdr_t *new_p);
```

2238 **24.4.30.2 Description**

2239 The *aud_put_hdr()* function creates a new header, containing no data items, in an
2240 audit record, and returns a descriptor to the header. The function accepts an
2241 audit record pointer *ar*, and puts the new header logically before the existing
2242 header *next_p* in the record. If *next_p* is NULL, then the new header shall be logi-
2243 cally the last in the record.

2244 Upon successful execution the *aud_put_hdr()* function shall return via *new_p* a
2245 descriptor for the new header. The descriptor returned by this call can then be
2246 used in subsequent calls to *aud_put_hdr_info()* to add data to this header in the
2247 audit record.

2248 Calls to *aud_put_hdr()* shall not affect the status of any existing descriptors for
2249 this or any other audit record. Calls on the various *aud_put_**() functions can be
2250 interleaved without affecting each other.

2251 This function may cause memory to be allocated. The caller should free any
2252 releasable memory, when the record is no longer required, by calling *aud_free()*
2253 with the (*void**)*aud_rec_t* as an argument.

2254 **24.4.30.3 Returns**

2255 Upon successful completion, the *aud_put_hdr()* function returns 0. Otherwise, a
2256 value of -1 shall be returned, the caller shall not have to free any releasable
2257 memory, and *errno* is set to indicate the error. The audit record referenced by *ar*
2258 shall not be affected if the return value is -1.

2259 **24.4.30.4 Errors**

2260 If any of the following conditions occur, the *aud_put_hdr()* function shall return -1
2261 and set *errno* to the corresponding value:

2262 [EINVAL] Argument *ar* does not point to a valid audit record.

2263 Argument *next_p* is neither **NULL** nor does it indicate an exist-
2264 ing header in record *ar*.

2265 [ENOMEM] The function requires more memory than is allowed by the
2266 hardware or system-imposed memory management constraints. –

2267 **24.4.30.5 Cross-References**

2268 *aud_delete_hdr()*, 24.4.5; *aud_free()*, 24.4.14; *aud_get_hdr()*, 24.4.18;
2269 *aud_init_record()*, 24.4.27; *aud_put_hdr_info()*, 24.4.31; *aud_valid()*, 24.4.40;
2270 *aud_write()*, 24.4.41.

2271 **24.4.31 Add Item to Audit Record Header**

2272 Function: *aud_put_hdr_info()*

2273 **24.4.31.1 Synopsis**

```
2274 #include <sys/audit.h>
2275 int aud_put_hdr_info (aud_hdr_t aud_hdr_d,
2276                      int position,
2277                      int item_id,
2278                      const aud_info_t *aud_hdr_info_p);
```

2279 **24.4.31.2 Description**

2280 The *aud_put_hdr_info()* function adds a data item to a header within an audit
2281 record. The function accepts a descriptor for a header *aud_hdr_d* in an audit
2282 record, and puts into the header the item with type, size and address defined in
2283 the structure referenced by *aud_hdr_info_p*. The item shall subsequently be
2284 identifiable by *item_id* in calls to functions as the record is manipulated, including
2285 after being written to and read back from an audit log; no item identifiable by
2286 *item_id* shall already exist in the header.

2287 The *position* argument shall specify either

2288 — the *item_id* of an item that already exists in the header; in this case the
2289 new data item shall be placed logically before the existing item
2290 — AUD_LAST_ITEM; in this case the new item shall be logically the last in
2291 the header.
2292 After the call of *aud_put_hdr_info()*, the caller can continue to manipulate the
2293 data item indicated by the *aud_info_t*, and the *aud_info_t*, and changes to them
2294 shall not affect the record unless they are used in a further call to
2295 *aud_put_*_info()*.
2296 Calls to *aud_put_hdr_info()* shall not affect the status of any other existing
2297 descriptors for this or any other audit record. Calls on the various
2298 *aud_put_*_info()* functions can be interleaved without affecting each other.
2299 This function may cause memory to be allocated. The caller should free any
2300 releasable memory, when the record is no longer required, by calling *aud_free()*
2301 with the *(void*)aud_rec_t* as an argument.

2302 **24.4.31.3 Returns**

2303 Upon successful completion, the *aud_put_hdr_info()* function returns 0. Otherwise,
2304 it returns a value of -1, the caller shall not have to free any releasable
2305 memory, and *errno* is set to indicate the error. The header referenced by
2306 *aud_hdr_d* shall not be affected if the return value is -1.

2307 **24.4.31.4 Errors**

2308 If any of the following conditions occur, the *aud_put_hdr_info()* function shall
2309 return -1 and set *errno* to the corresponding value:

2310 [EINVAL] Argument *aud_hdr_d* is not a valid descriptor for a header
2311 within an audit record.
2312 Argument *position* is not AUD_LAST_ITEM and does not identify a valid item from the header.
2314 The value of the *aud_info_type* field of the structure referenced
2315 by *aud_hdr_info_p* is invalid.
2316 An item with identifier *item_id* already exists in the header.
2317 The argument *item_id* is equal to AUD_FIRST_ITEM,
2318 AUD_NEXT_ITEM, or AUD_LAST_ITEM.
2319 [ENOMEM] The function requires more memory than is allowed by the
2320 hardware or system-imposed memory management constraints. —

2321 **24.4.31.5 Cross-References**

2322 *aud_delete_hdr_info()*, 24.4.6; *aud_free()*, 24.4.14; *aud_get_hdr_info()*, 24.4.20;
2323 *aud_put_hdr()*, 24.4.30; *aud_valid()*, 24.4.40; *aud_write()*, 24.4.41.

2324 **24.4.32 Add Set of Object Attributes to Audit Record**

2325 Function: *aud_put_obj()*

2326 **24.4.32.1 Synopsis**

```
2327 #include <sys/audit.h>
2328 int aud_put_obj (aud_rec_t ar,
2329                   const aud_obj_t *next_p,
2330                   aud_obj_t *new_p);
```

2331 **24.4.32.2 Description**

2332 The *aud_put_obj()* function creates a new set of object attributes, containing no
2333 data items, in an audit record, and returns a descriptor to the set. The function
2334 accepts an audit record pointer *ar*, and puts the new set of object attributes logically
2335 before the existing set *next_p* in the record. If *next_p* is **NULL**, then the new
2336 set shall be logically the last in the record.

2337 Upon successful execution the *aud_put_obj()* function shall return via *new_p* a
2338 descriptor for the new set of object attributes. The descriptor returned by this call
2339 can then be used in subsequent calls to *aud_put_obj_info()* to add data to this set
2340 of object attributes in the audit record.

2341 Calls to *aud_put_obj()* shall not affect the status of any existing descriptors for
2342 this or any other audit record. Calls on the various *aud_put_**() functions can be
2343 interleaved without affecting each other.

2344 This function may cause memory to be allocated. The caller should free any
2345 releasable memory, when the record is no longer required, by calling *aud_free()*
2346 with the (*void**)*aud_rec_t* as an argument.

2347 **24.4.32.3 Returns**

2348 Upon successful completion, the *aud_put_obj()* function returns 0. Otherwise, a
2349 value of -1 shall be returned, the caller shall not have to free any releasable
2350 memory, and *errno* is set to indicate the error. The audit record referenced by *ar*
2351 shall not be affected if the return value is -1.

2352 **24.4.32.4 Errors**

2353 If any of the following conditions occur, the *aud_put_obj()* function shall return -1
2354 and set *errno* to the corresponding value:

2355 [EINVAL] Argument *ar* does not point to a valid audit record.

2356 Argument *next_p* is neither **NULL** nor does it indicate an existing
2357 set of object attributes in record *ar*.

2358 [ENOMEM] The function requires more memory than is allowed by the
2359 hardware or system-imposed memory management constraints. –

2360 **24.4.32.5 Cross-References**

2361 *aud_delete_obj()*, 24.4.8; *aud_free()*, 24.4.14; *aud_get_obj()*, 24.4.21;
2362 *aud_init_record()*, 24.4.27; *aud_put_obj_info()*, 24.4.33; *aud_valid()*, 24.4.40;
2363 *aud_write()*, 24.4.41.

2364 **24.4.33 Add Item to Set of Object Attributes**

2365 Function: *aud_put_obj_info()*

2366 **24.4.33.1 Synopsis**

```
2367 #include <sys/audit.h>
2368 int aud_put_obj_info (aud_obj_t aud_obj_d,
2369                         int position,
2370                         int item_id,
2371                         const aud_info_t *aud_obj_info_p);
```

2372 **24.4.33.2 Description**

2373 The *aud_put_obj_info()* function adds a data item to a set of object attributes
2374 within an audit record. The function accepts a descriptor for a set of object attri-
2375 butes *aud_obj_d* in an audit record, and puts into the set of object attributes the
2376 item with type, size and address defined in the structure referenced by
2377 *aud_obj_info_p*. The item shall subsequently be identifiable by *item_id* in calls to
2378 functions as the record is manipulated, including after being written to and read
2379 back from an audit log; no item identifiable by *item_id* shall already exist in the
2380 set of object attributes.

2381 The *position* argument shall specify either

- 2382 — the *item_id* of an item that already exists in the set of object attributes; in
2383 this case the new data item shall be placed logically before the existing
2384 item
- 2385 — AUD_LAST_ITEM; in this case the new item shall be logically the last in
2386 the set.

2387 After the call of *aud_put_obj_info()*, the caller can continue to manipulate the
2388 data item indicated by the *aud_info_t*, and the *aud_info_t*, and changes to them
2389 shall not affect the record unless they are used in a further call to
2390 *aud_put_*_info()*.

2391 Calls to *aud_put_obj_info()* shall not affect the status of any other existing
2392 descriptors for this or any other audit record. Calls on the various
2393 *aud_put_*_info()* functions can be interleaved without affecting each other.

2394 This function may cause memory to be allocated. The caller should free any
2395 releasable memory, when the record is no longer required, by calling *aud_free()*
2396 with the (*void**)*aud_rec_t* as an argument.

2397 **24.4.33.3 Returns**

2398 Upon successful completion, the *aud_put_obj_info()* function returns 0. Otherwise,
2399 it returns a value of -1, the caller shall not have to free any releasable
2400 memory, and *errno* is set to indicate the error. The set of object attributes refer-
2401 enced by *aud_obj_d* shall not be affected if the return value is -1.

2402 **24.4.33.4 Errors**

2403 If any of the following conditions occur, the *aud_put_obj_info()* function shall
2404 return -1 and set *errno* to the corresponding value:

- 2405 [EINVAL] Argument *aud_obj_d* is not a valid descriptor for a set of object
2406 attributes within an audit record.
- 2407 Argument *position* is not AUD_LAST_ITEM and does not iden-
2408 tify a valid item from the set of object attributes.
- 2409 The value of the *aud_info_type* field of the structure referenced
2410 by *aud_obj_info_p* is invalid.
- 2411 An item with identifier *item_id* already exists in the set of object
2412 attributes.
- 2413 The argument *item_id* is equal to AUD_FIRST_ITEM,
2414 AUD_NEXT_ITEM, or AUD_LAST_ITEM.
- 2415 [ENOMEM] The function requires more memory than is allowed by the
2416 hardware or system-imposed memory management constraints. –

2417 **24.4.33.5 Cross-References**

2418 *aud_delete_obj_info()*, 24.4.8; *aud_free()*, 24.4.14; *aud_get_obj_info()*, 24.4.22;
2419 *aud_put_obj()*, 24.4.32; *aud_valid()*, 24.4.40; *aud_write()*, 24.4.41.

2420 **24.4.34 Add Set of Subject Attributes to Audit Record**

2421 Function: *aud_put_subj()*

2422 **24.4.34.1 Synopsis**

```
2423 #include <sys/audit.h>
2424 int aud_put_subj (aud_rec_t ar,
2425                     const aud_subj_t *next_p,
2426                     aud_subj_t *new_p);
```

2427 **24.4.34.2 Description**

2428 The *aud_put_subj()* function creates a new set of subject attributes, containing no
2429 data items, in an audit record, and returns a descriptor to the set. The function
2430 accepts an audit record pointer *ar*, and puts the new set of subject attributes logically
2431 before the existing set *next_p* in the record. If *next_p* is **NULL**, then the new
2432 set shall be logically the last in the record.

2433 Upon successful execution the *aud_put_subj()* function shall return via *new_p* a
2434 descriptor for the new set of subject attributes. The descriptor returned by this
2435 call can then be used in subsequent calls to *aud_put_subj_info()* to add data to
2436 this set of subject attributes in the audit record.

2437 Calls to *aud_put_subj()* shall not affect the status of any existing descriptors for
2438 this or any other audit record. Calls on the various *aud_put_**() functions can be
2439 interleaved without affecting each other.

2440 This function may cause memory to be allocated. The caller should free any
2441 releasable memory, when the record is no longer required, by calling *aud_free()*
2442 with the (*void**)*aud_rec_t* as an argument.

2443 **24.4.34.3 Returns**

2444 Upon successful completion, the *aud_put_subj()* function returns 0. Otherwise, a
2445 value of -1 shall be returned, the caller shall not have to free any releasable
2446 memory, and *errno* is set to indicate the error. The audit record referenced by *ar*
2447 shall not be affected if the return value is -1.

2448 **24.4.34.4 Errors**

2449 If any of the following conditions occur, the *aud_put_subj()* function shall return
2450 -1 and set *errno* to the corresponding value:

2451 [EINVAL] Argument *ar* does not point to a valid audit record.

2452 Argument *next_p* is neither **NULL** nor does it indicate an exist-
2453 ing set of subject attributes in record *ar*.

2454 [ENOMEM] The function requires more memory than is allowed by the
2455 hardware or system-imposed memory management constraints. –

2456 **24.4.34.5 Cross-References**

2457 *aud_delete_subj()*, 24.4.9; *aud_free()*, 24.4.14; *aud_get_subj()*, 24.4.23;
2458 *aud_init_record()*, 24.4.27; *aud_put_subj_info()*, 24.4.35; *aud_valid()*, 24.4.40;
2459 *aud_write()*, 24.4.41.

2460 **24.4.35 Add Item to Set of Subject Attributes**

2461 Function: *aud_put_subj_info()*

2462 **24.4.35.1 Synopsis**

```
2463 #include <sys/audit.h>
2464 int aud_put_subj_info (aud_subj_t aud_subj_d,
2465                         int position,
2466                         int item_id,
2467                         const aud_info_t *aud_subj_info_p);
```

2468 **24.4.35.2 Description**

2469 The *aud_put_subj_info()* function adds a data item to a set of subject attributes
2470 within an audit record. The function accepts a descriptor for a set of subject attributes
2471 *aud_subj_d* in an audit record, and puts into the set of subject attributes
2472 the item with type, size and address defined in the structure referenced by
2473 *aud_subj_info_p*. The item shall subsequently be identifiable by *item_id* in calls
2474 to functions as the record is manipulated, including after being written to and
2475 read back from an audit log; no item identifiable by *item_id* shall already exist in
2476 the set of subject attributes.

2477 The *position* argument shall specify either

- 2478 — the *item_id* of an item that already exists in the set of subject attributes; in
2479 this case the new data item shall be placed logically before the existing
2480 item
- 2481 — AUD_LAST_ITEM; in this case the new item shall be logically the last in
2482 the set.

2483 After the call of *aud_put_subj_info()*, the caller can continue to manipulate the
2484 data item indicated by the *aud_info_t*, and the *aud_info_t*, and changes to them
2485 shall not affect the record unless they are used in a further call to
2486 *aud_put_*_info()*.

2487 Calls to *aud_put_subj_info()* shall not affect the status of any other existing
2488 descriptors for this or any other audit record. Calls on the various
2489 *aud_put_*_info()* functions can be interleaved without affecting each other.

2490 This function may cause memory to be allocated. The caller should free any
2491 releasable memory, when the record is no longer required, by calling *aud_free()*
2492 with the (*void**)*aud_rec_t* as an argument.

2493 **24.4.35.3 Returns**

2494 Upon successful completion, the *aud_put_subj_info()* function returns 0. Otherwise,
2495 it returns a value of -1, the caller shall not have to free any releasable
2496 memory, and *errno* is set to indicate the error. The set of subject attributes refer-
2497 enced by *aud_subj_d* shall not be affected if the return value is -1.

2498 **24.4.35.4 Errors**

2499 If any of the following conditions occur, the *aud_put_subj_info()* function shall
2500 return -1 and set *errno* to the corresponding value:

- 2501 [EINVAL] Argument *aud_subj_d* is not a valid descriptor for a set of sub-
2502 ject attributes within an audit record.
- 2503 Argument *position* is not AUD_LAST_ITEM and does not iden-
2504 tify a valid item from the set of subject attributes.
- 2505 The value of the *aud_info_type* field of the structure referenced
2506 by *aud_subj_info_p* is invalid.
- 2507 An item with identifier *item_id* already exists in the set of sub-
2508 ject attributes.
- 2509 The argument *item_id* is equal to AUD_FIRST_ITEM,
2510 AUD_NEXT_ITEM, or AUD_LAST_ITEM.
- 2511 [ENOMEM] The function requires more memory than is allowed by the
2512 hardware or system-imposed memory management constraints. –

2513 **24.4.35.5 Cross-References**

2514 *aud_delete_subj_info()*, 24.4.10; *aud_free()*, 24.4.14; *aud_get_subj_info()*, 24.4.24;
2515 *aud_put_subj()*, 24.4.34; *aud_valid()*, 24.4.40; *aud_write()*, 24.4.41.

2516 **24.4.36 Read an Audit Record**

2517 Function: *aud_read()*

2518 **24.4.36.1 Synopsis**

```
2519 #include <sys/audit.h>
2520 aud_rec_t aud_read (int filedes);
```

2521 **24.4.36.2 Description**

2522 This function attempts to read an audit record from the current file offset of the
2523 file identified by *filedes*. If the function successfully reads an audit record, the file
2524 offset shall be incremented such that a further call of the function will operate on
2525 the next audit record in the log. If the file contains records that were written to
2526 the system audit log, it is left to the implementation to provide any sequencing
2527 information required to ensure that successive calls of *aud_read()* each obtain the
2528 “next” available record that was written to the log. If the file contains records
2529 that were written to a file, the ordering of the records depends on the position of
2530 the file offset at the time *aud_write()* was called. If no more records are in the
2531 file, a value of zero is returned. In other cases, if a call is unsuccessful, the effect
2532 of further calls is unspecified.

2533 Upon successful completion, the function returns an audit record pointer,
2534 *aud_rec_t*, identifying the audit record. The format of the audit record is
2535 unspecified, but the *aud_rec_t* can be supplied as an input argument to functions
2536 such as the *aud_get_**() functions .
2537 Any existing audit record pointers that refer to records from the audit log shall
2538 continue to refer to those records.
2539 This function may cause memory to be allocated. The caller should free any
2540 releasable memory allocated by this function (and by other functions that are
2541 used to process the record), when the caller is finished with the record, by a call to
2542 *aud_free()* with the (*void**)*aud_rec_t* as an argument.
2543 If *{_POSIX_INF}* is defined, and *{_POSIX_INF_PRESENT}* is in effect for the file |
2544 designated by *filedes*, then the information label of the process shall automatically
2545 be set to an implementation-defined value which shall be the same as the
2546 value returned by *inf_float(file information label, process information label)*.

2547 **24.4.36.3 Returns**

2548 Upon successful completion, the *aud_read()* function returns an *aud_rec_t* pointing
2549 to the record. If there are no more records in the audit log, the caller shall not
2550 have to free any releasable memory, and the function returns a value of
2551 (*aud_rec_t*) 0. Otherwise, a value of (*aud_rec_t*) -1 is returned, the caller shall not
2552 have to free any releasable memory, and *errno* is set to indicate the error.

2553 **24.4.36.4 Errors**

2554 If any of the following conditions occur, the *aud_read()* function shall return a
2555 value of -1 and set *errno* to the corresponding value:

- 2556 [EAGAIN] The *O_NONBLOCK* flag is set for the file descriptor *filedes* and
2557 the process would be delayed in the read operation.
- 2558 [EBADF] The *filedes* argument is not a valid file descriptor open for reading.
- 2560 [EINTR] The operation was interrupted by a signal, and no data was
2561 transferred.
- 2562 [EINVAL] The value of the *filedes* argument does not identify an audit log
2563 positioned at a valid audit record.
- 2564 The header of the next record in the audit log identified by
2565 *filedes* indicates the record has an AUD_FORMAT or
2566 AUD_VERSION that is not supported by the implementation.
- 2567 [ENOMEM] The function requires more memory than is allowed by the
2568 hardware or system-imposed memory management constraints. –

2569 **24.4.36.5 Cross-References**

2570 *aud_free()*, 24.4.14; *aud_get_event()*, 24.4.16; *aud_get_hdr()*, 24.4.18;
2571 *aud_get_obj()*, 24.4.21; *aud_get_subj()*, 24.4.23; *aud_rec_to_text()*, 24.4.37.

2572 **24.4.37 Convert an Audit Record to Text**

2573 Function: *aud_rec_to_text()*

2574 **24.4.37.1 Synopsis**

2575 `#include <sys/audit.h>`
2576 `char *aud_rec_to_text (aud_rec_t ar, ssize_t *len_p);`

2577 **24.4.37.2 Description**

2578 The *aud_rec_to_text()* function transforms the audit record identified by *ar* into a
2579 human-readable, null terminated character string. The function shall return the
2580 address of the string and, if *len_p* is not **NULL**, set the location pointed to by *len_p*
2581 to the length of the string (not including the null terminator).

2582 The text string produced by *aud_rec_to_text()* shall contain a text form of the vari-
2583 ous sections of the audit record; the record header(s) shall be given first, followed
2584 by any set(s) of subject attributes, followed by any set(s) of event specific informa-
2585 tion, followed by any set(s) of object attributes. Items within each section shall be
2586 given in the order they would be returned by the *aud_get_**() functions. Other
2587 than this, the form of the text string is unspecified by this standard.

2588 This function may cause memory to be allocated. The caller should free any
2589 releasable memory when the text form of the record is no longer required, by cal-
2590 ling *aud_free()* with the string address (cast to a (*void**)) as an argument.

2591 **24.4.37.3 Returns**

2592 Upon successful completion, the *aud_rec_to_text()* function returns a pointer to
2593 the text record. Otherwise, a value of **NULL** shall be returned, the caller shall not
2594 have to free any releasable memory, and *errno* shall be set to indicate the error.

2595 **24.4.37.4 Errors**

2596 If any of the following conditions occur, the *aud_rec_to_text()* function shall return
2597 a value of **NULL** and set *errno* to the corresponding value:

2598 **[EINVAL]** The value of the *ar* argument does not identify a valid audit
2599 record.

2600 **[ENOMEM]** The text to be returned requires more memory than is allowed
2601 by the hardware or system-imposed memory management con-
2602 straints.

2603 **24.4.37.5 Cross-References**

2604 *aud_free()*, 24.4.14; *aud_read()*, 24.4.36; *aud_valid()*, 24.4.40.

2605 **24.4.38 Get the Size of an Audit Record**

2606 Function: *aud_size()*

2607 **24.4.38.1 Synopsis**

```
2608 #include <sys/audit.h>
2609 ssize_t aud_size (aud_rec_t ar);
```

2610 **24.4.38.2 Description**

2611 The *aud_size()* function returns the total length (in bytes) that the audit record
2612 identified by *ar* would use when converted by *aud_copy_ext()*. The audit record *ar*
2613 will have been obtained by a previous, successful call to the *aud_read()*,
2614 *aud_init_record()* or *aud_dup_record()* function. The *aud_size()* function is used
2615 to ascertain the buffer size required to copy an audit record (via *aud_copy_ext()*)
2616 into user-allocated space.

2617 **24.4.38.3 Returns**

2618 Upon successful completion, the *aud_size()* function returns the length of the
2619 audit record.

2620 In the event of failure the *aud_size()* function returns a value of *-1* and *errno* is
2621 set to indicate the error.

2622 **24.4.38.4 Errors**

2623 If any of the following conditions occur, the *aud_size()* function shall return *-1*
2624 and set *errno* to the corresponding value:

2625 [EINVAL] The value of the *ar* argument does not identify a valid audit
2626 record.

2627 **24.4.38.5 Cross-References**

2628 *aud_copy_ext()*, 24.4.1; *aud_dup_record()*, 24.4.11; *aud_init_record()*, 24.4.27;
2629 *aud_read()*, 24.4.36; *aud_valid()*, 24.4.40.

2630 **24.4.39 Control the Generation of Audit Records**

2631 Function: *aud_switch()*

2632 **24.4.39.1 Synopsis**

```
2633 #include <sys/audit.h>
2634 aud_state_t aud_switch (aud_state_t aud_state);
```

2635 **24.4.39.2 Description**

2636 The *aud_switch()* function requests that recording of system-generated audit
2637 records for the current process be suspended (using AUD_STATE_OFF) or
2638 resumed (using AUD_STATE_ON), or enquires about the current state (using
2639 AUD_STATE_QUERY). A request to set the state is advisory and may be ignored
2640 either wholly or partially if the auditing policy of the system prohibits the suspen-
2641 sion of process auditing. A request to suspend auditing does not affect auditing
2642 performed by the *aud_write()* function.

2643 The current state of this switch is inherited by a child if the process calls the
2644 *fork()* function.

2645 Appropriate privilege is required to use this function. If *{_POSIX_CAP}* is defined,
2646 then appropriate privilege is provided by the CAP_AUDIT_CONTROL capability.

2647 **24.4.39.3 Returns**

2648 Upon successful completion, the *aud_switch()* function returns the value of the
2649 audit state for the calling process at the start of the call. Otherwise, a value of
2650 ((*aud_state_t*)−1) is returned and no change shall be made to the calling process's
2651 audit state.

2652 **24.4.39.4 Errors**

2653 If any of the following conditions occur, the *aud_switch()* function shall return a
2654 value of ((*aud_state_t*)−1) and set *errno* to the corresponding value:

2655 [EINVAL] The value of the *aud_state* argument is invalid. —

2656 [EPERM] The process does not have appropriate privileges to call this
2657 function.

2658 **24.4.39.5 Cross-References**

2659 *aud_write()*, 24.4.41.

2660 **24.4.40 Validate an Audit Record**

2661 Function: *aud_valid()*

2662 **24.4.40.1 Synopsis**

```
2663 #include <sys/audit.h>
2664 int aud_valid (aud_rec_t ar);
```

2665 **24.4.40.2 Description**

2666 The *aud_valid()* function checks the audit record referred to by the argument *ar*
2667 for validity.

2668 The audit record *ar* shall have been created by a previous call to
2669 *aud_init_record()*, *aud_copy_int()* or *aud_dup_record()*, or shall have been read
2670 from an audit log by *aud_read()*. The record shall contain at least one header,
2671 and the first or only header shall contain at least the following items:

- 2672 • The event type for the event (identified by a *item_id* of
2673 AUD_EVENT_TYPE_ID). The corresponding *aud_info_t* shall have its
2674 *aud_info_type* member equal to AUD_TYPE_STRING or AUD_TYPE_INT.
- 2675 • The audit status for the event (identified by a *item_id* of
2676 AUD_STATUS_ID). The corresponding *aud_info_t* shall have its
2677 *aud_info_type* member equal to AUD_TYPE_AUD_STATUS.

2678 Calls to *aud_valid()* shall not affect the status of any existing descriptors for this
2679 or any other audit record.

2680 **24.4.40.3 Returns**

2681 Upon successful completion, the function shall return a value of zero. Otherwise,
2682 a value of -1 shall be returned and *errno* shall be set to indicate the error.

2683 **24.4.40.4 Errors**

2684 If any of the following conditions occur, the *aud_valid()* function shall return -1
2685 and set *errno* to the corresponding value:

2686 [EINVAL] Argument *ar* does not point to an *aud_rec_t* structure as recognized
2687 by the implementation.

2688 One or more of the required entries is not present. —

2689 **24.4.41 Write an Audit Record**

2690 Function: *aud_write()*

2691 **24.4.41.1 Synopsis**

```
2692 #include <sys/audit.h>
2693 int aud_write (int filedes, aud_rec_t ar);
```

2694 **24.4.41.2 Description**

2695 The *aud_write()* function writes an application-specific audit record to an audit
2696 log. Upon successful completion the audit record identified by *aud_rec_t* shall be
2697 written into the audit log file identified by *filedes*; if *filedes* is equal to
2698 AUD_SYSTEM_LOG then the record shall be written to the system audit log. If
2699 *filedes* is not equal to AUD_SYSTEM_LOG then the record shall be written at the
2700 position in the file defined for the POSIX *write()* interface.

2701 The record *ar* shall be a valid audit record, as defined by the *aud_valid()* function.
2702 The *aud_write()* call shall not alter the record *ar*; after the call of *aud_write()*, the
2703 caller can continue to manipulate the record, and changes to it shall not affect the
2704 record reportable from this call of *aud_write()*.

2705 If the first or only header in the record does not contain an item with *item_id* set
2706 to AUD_TIME_ID, then the time reported by a later call on *aud_get_hdr_info()*
2707 for the AUD_TIME_ID field of this header shall be the time at which the
2708 *aud_write()* function was executed. If the header does not contain items with
2709 *item_ids* set to AUD_FORMAT_ID and AUD_VERSION_ID, then the values of
2710 these fields reported for this record shall be the same as those that would be
2711 reported for records generated by the system at the time the *aud_write()* function
2712 was called. If the header does not contain an item with *item_id* set to
2713 AUD_AUD_ID, then the audit ID reported by a later call on *aud_get_hdr_info()*
2714 for the AUD_AUD_ID field of this header shall be the audit ID of the user
2715 accountable for the current process.

2716 The application may include in the record one or more sets of subject attributes. If
2717 the application is auditing an action performed on behalf of a client process, the
2718 first set of subject attributes should describe the client, and the header should
2719 include the client's audit ID in an item with *item_id* set to AUD_AUD_ID and
2720 *aud_info_type* field AUD_TYPE_AUD_ID. If the application is writing a record
2721 that was read from another log, the record will already contain one or more sets of
2722 subject attributes. If the record does not contain any sets of subject attributes,
2723 then later calls to *aud_get_subj()* and *aud_get_subj_info()* for this record shall
2724 report one set of subject attributes, containing details of the process that invoked
2725 *aud_write()*.

2726 If the record has been constructed by the application, later reading of the record
2727 using *aud_read()*, *aud_get_**() and the *aud_get_*_info()* functions shall report the
2728 items from the record *ar* in the logical order specified by the *aud_put_**() and
2729 *aud_put_*_info()* calls used to construct the record. The content of the record,

2730 reported by calls to the *aud_read()*, *aud_get_**() and the *aud_get_*_info()* func-
2731 tions, shall be the content at the time *aud_write()* was invoked.

2732 If *{_POSIX_INF}* is defined, and *{_POSIX_INF_PRESENT}* is in effect for the log |
2733 designated by *filedes*, then the information label of the log shall automatically be |
2734 set to an implementation-defined value which should be the same as the value |
2735 returned by *inf_float(process information label, log information label)*.

2736 Appropriate privilege is required to use *aud_write()* to write to the system audit |
2737 log. If *{_POSIX_CAP}* is defined then appropriate privilege is provided by the |
2738 *CAP_AUDIT_WRITE* capability.

2739 **24.4.41.3 Returns**

2740 Upon successful completion, the *aud_write()* function returns a value of 0 and the
2741 specified record is written to the specified audit log. Otherwise, a value of -1 is
2742 returned and *errno* is set to indicate the error, and the specified record is not writ-
2743 ten to the specified audit log.

2744 **24.4.41.4 Errors**

2745 If any of the following conditions occur, the *aud_write()* function shall return -1
2746 and set *errno* to the corresponding value:

2747 [EBADF]	The value of the <i>filedes</i> argument is not a valid file descriptor 2748 open for writing and is not AUD_SYSTEM_LOG.
2749 [EINTR]	The operation was interrupted by a signal, and no data was 2750 transferred.
2751 [EINVAL]	The value of the <i>ar</i> argument does not identify a valid audit 2752 record. 2753 The audit record identified by <i>ar</i> does not contain the required 2754 header data.
2755 [EPERM]	The process does not have appropriate privilege to perform the 2756 requested operation.

2757 **24.4.41.5 Cross-References**

2758 *aud_dup_record()*, 24.4.11; *aud_get_id()*, 24.4.20; *aud_init_record()*, 24.4.27;
2759 *aud_read()*, 24.4.36; *aud_switch()*, 24.4.39; *aud_valid()*, 24.4.40.

Section 25: Capabilities

2 25.1 General Overview

3 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
4 process to be enabled or disabled, and for a set of these capabilities to be passed
5 on to the next program associated with a process. This specification also
6 identifies a minimum set of capabilities required for the support of portable
7 security-relevant programs, and specifies the circumstances in POSIX.1 under
8 which these capabilities shall be used. Support for the interfaces defined in this
9 section is optional, but shall be provided if the symbol `{_POSIX_CAP}` is defined.
10

11 POSIX.1 specifies that certain actions require a process to possess appropriate
12 privilege in order to complete those actions. This section specifies the names of
13 the capabilities which constitute appropriate privilege to perform those actions on
14 a system that supports the POSIX Capability Option.
15

15 This section describes a set of interfaces by which *capabilities* may be associated
16 with a process and the method by which a process's *capabilities* are derived.
17 Specific capabilities of a process that exec's a particular file may be revoked,
18 inherited from the previous process image, or granted to the process, depending
19 on the value(s) of the file *capability state* of the file and the process *capability*
20 *state* of the previous process image.
21

21 The set of interfaces defined by this standard provide the means to support the
22 *principle of least privilege*. Note, however, it does not require that a conforming
23 implementation actually enforce a least privilege (least capability) security policy.
24 The capability related interfaces and semantics specified in this standard permit
25 individual capabilities to be defined down to a per-function level and permit them
26 to be granted or denied to the granularity of an individual process image. They
27 also permit a process image to control the effectiveness of the capabilities
28 assigned to it during its execution. These capabilities are necessary, but not
29 sufficient, for the implementation of a least privilege security policy. Implementa-
30 tions may extend the capability interfaces such that use of and/or access to capa-
31 bilities by programs are further constrained.
32

32 This section also defines a minimal number of capabilities that shall be supported
33 by conforming implementations. Implementations may define additional capabili-
34 ties that affect the behavior of POSIX defined and/or other system functions.

35 **25.1.1 Major Features**

36 **25.1.1.1 Task Bounding of Capability**

37 Another major characteristic of the capability interfaces is that capabilities may
38 be bounded in the extent of code they are effective over. That is, they can be
39 enabled for only as long as they are actually needed to perform a task (or tasks),
40 and then disabled. The extent of code that could exercise a particular capability
41 can be bounded both at the program level and within a particular program.

42 At the program level, a process may be assigned or denied specific capabilities by
43 setting the capability flags and attributes associated with the program file. When
44 the file is executed, these flags and attributes are examined by *exec()*. The *exec()*
45 function then modifies the capability state of the process in a specific manner
46 according to those flags and attributes. In this way, a process may gain additional
47 capabilities by executing certain programs, or it may lose capabilities it currently
48 possesses.

49 Within itself, a process image may enable, disable, or permanently revoke its
50 capabilities. For example, a process modifies the effective flag of a given capability|
51 to either enable or disable that capability. This flag shall be set in order for the
52 capability to be available for use. A process image permanently, i.e., for the dura-|
53 tion of that process image, revokes a given capability by resetting both the effec-|
54 tive and permitted flags for that capability. More information on these two flags|
55 is provided in section 25.1.1.4 below. |

56 **25.1.1.2 Capability Inheritance**

57 Following the *exec()* of a program, the capabilities that have their permitted flags|
58 set in the new process image depend on the capability states of both the previous|
59 process image and the exec'd program file. Each capability marked as permitted|
60 may have been forced to be set by the program file or inherited from the previous|
61 image (if the capability attributes of the program file allow the inheritance).|

62 Inheritance permits a process image to request that all, some or none of its capa-
63 bilities be passed to the next process image, subject to restrictions set by the sys-
64 tem security policy. For example, a backup program may *exec()* the pax utility,
65 granting it the capabilities required to read all files in a file system (providing it
66 is allowed to inherit those capabilities). However, the same backup utility may
67 *exec()* other utilities to which it does not pass any capabilities.

68 **25.1.1.3 Capability Flags**

69 The capability flags defined by this standard are permitted, effective, and inherit-
70 able. These flags apply to each capability separately, and together their values
71 determine a capability state. Capability states shall be assignable to at least two
72 entities: processes and files. Implementations may define additional flags for
73 capabilities and may provide for the assignment of capability states to additional
74 entities.

75 **25.1.1.4 Capability Flags and Processes**

76 The capability state is the attribute of a process which contains the value of all of
77 the process's capability flags. A conforming implementation shall support the
78 assignment of a capability state to processes. When the process permitted flag for
79 a capability is set, a process shall be able to set all its flags defined by this stan-
80 dard for that capability. A process shall be able to clear any flag for any of its
81 capabilities regardless of the state of the permitted flags. A process can exercise a
82 particular capability only when that capability's process effective flag is set. The
83 process effective flag shall be the only flag considered by system functions when
84 determining if the process possesses appropriate privilege. The process inherit-
85 able flag is used by the *exec()* function when determining the capability flags of
86 the new process image. A capability may be passed from one process image to the
87 next through an *exec()* only if the inheritable flag of that capability is set. This
88 inheritance may or may not actually occur, depending on the capability state of
89 the file as described in the next section. The new process image may also acquire
90 capabilities based upon the capability state of the file used to create the new pro-
91 cess image, as defined in section 3.1.2.2.

92 **25.1.1.5 Capability Flags and Files**

93 The capability state is the attribute of a file which contains the value of all of the
94 file's capability flags. A conforming implementation shall support the assignment
95 of capability states to files. The purpose of assigning capability states to files is to
96 provide the *exec()* function with information regarding the capabilities that any
97 process image created with the program in the file is capable of dealing with and
98 have been granted by some authority to use.

99 If *pathconf()* indicates that `{_POSIX_CAP_PRESENT}` is not in effect for a file,
100 then the capability state of that file shall be implementation-defined.

101 **25.1.1.6 File System Support of Capability**

102 The capability state of a process after an *exec()* of a file for which the value of the
103 pathname variable `{_POSIX_CAP_PRESENT}` is zero shall be implementa-
104 tion-defined.

105 **25.1.1.7 Application**

106 The POSIX.1 functions listed in Table 25-1 are affected by the capability func-
107 tionality defined in this standard.

Table 25-1 – POSIX.1 Functions Covered by Capability Policies

109 110	Existing Function	POSIX.1 Section
112	chmod	5.6.4
113	chown	5.6.5
114	creat	5.3.2
115	exec	3.1.2
116	fpathconf	5.7.1
117	fstat	5.6.2
118	kill	3.3.2
119	link	5.3.4
120	mkdir	5.4.1
121	mkfifo	5.4.2
122	open	5.3.1
123	pathconf	5.7.1
124	read	6.4.1
125	rename	5.5.3
126	rmdir	5.5.2
127	setgid	4.2.2
128	setuid	4.2.2
129	stat	5.6.2
130	unlink	5.5.1
131	utime	5.6.6
132	write	6.4.2
133 134	New Function	POSIX.1e Synopsis
136	acl_delete_def_fd	Delete a Default ACL by File Descriptor
137	acl_delete_def_file	Delete a Default ACL by Filename
138	acl_get_fd	Get an ACL by File Descriptor
139	acl_get_file	Get an ACL by Filename
140	acl_set_fd	Set an ACL by File Descriptor
141	acl_set_file	Set an ACL Filename
142	aud_switch	Control the Generation of Audit Records
143	aud_write	write an application-generated record to an audit log
144	inf_get_fd	Get the Information Label of a File Identified by File Descriptor
145	inf_get_file	Get the Information Label of a File Identified by File Pathname
146	inf_set_fd	Set the Information Label of a File Identified by File Descriptor
147	inf_set_file	Set the Information Label of a File Identified by File Pathname
148	inf_set_proc	Set the Process Information Label
149	mac_get_fd	Get the Label of a File Designated by File Descriptor
150	mac_get_file	Get the Label of a File Designated by File Pathname
151	mac_set_fd	Set the Label of a File Designated by File Descriptor
152	mac_set_file	Set the Label of a File
153	mac_set_proc	Set the Process Label
154	cap_get_fd	Get the Capability State of an Open File
155	cap_get_file	Read the Capability State of a File
156	cap_set_fd	Set the Capability State of an Open File
157	cap_set_file	Write the Capability State of a File

WITHDRAWN DRAFT. All Rights Reserved by IEEE.
Preliminary—Subject to Revision.

158 **25.1.2 Capability Functions**

159 Functional interfaces are defined to manipulate capability states, to assign them
160 to files and processes and to obtain them for files and processes. These functions
161 comprise a set of interfaces that permit portable programs to manipulate their
162 own capability state and a minimal set of interfaces to manipulate the capability
163 state of files.

164 Four groups of functions are defined to:

- 165 (1) manage the working storage area used by capability states
166 (2) manipulate the capability flags within a capability state
167 (3) manipulate (read and write) a capability state on a file or process
168 (4) translate a capability state into different formats.

169 **25.1.2.1 Capability Data Object Storage Management**

170 The capabilities associated with a file or process are never edited directly.
171 Rather, a working storage area is allocated to contain a representation of the
172 capability state. Capabilities are edited and manipulated only within this working
173 storage area. Once the editing of the capability state is completed, the updated
174 capability state is used to replace the capability state associated with the file or
175 process.

176 Working storage is allocated as needed by the capability manipulation functions.
177 The *cap_init()* and *cap_dup()* functions also allow the application to allocate
178 working storage for the creation of a new capability state. The working storage
179 area may be released by the application once the capability state is no longer
180 needed by use of the *cap_free()* function.

- 181 *cap_dup()* Duplicates a capability state in a working storage area
182 *cap_free()* Releases working storage area previously allocated by capability
183 manipulation functions
184 *cap_init()* Allocates and initializes working storage area for a capability
185 state.

186 **25.1.2.2 Capability Data Object Manipulation**

187 These functions manipulate capability state only in working storage not associ-
188 ated with file or process.

- 189 *cap_get_flag()* Obtain the value of a specific flag for a specific capability.
190 *cap_set_flag()* Sets the value of a specific flag for a specific capability.
191 *cap_clear()* Initializes or resets a capability state such that all flags for
192 all capabilities are cleared.

193 **25.1.2.3 Capability Manipulation on an Object**

194 These functions read the capability state of a file or process into working storage
195 and write the capability state in working storage to a file or process.

196	<i>cap_get_fd()</i>	Reads the capability state associated with a file descriptor 197 into working storage.
198	<i>cap_get_file()</i>	Reads the capability state associated with a file into work- 199 ing storage.
200	<i>cap_get_proc()</i>	Reads the capability state associated with the calling pro- 201 cess into working storage.
202	<i>cap_set_fd()</i>	Writes the capability state in working storage to the object 203 associated with a file descriptor.
204	<i>cap_set_file()</i>	Writes the capability state in working storage to a file.
205	<i>cap_set_proc()</i>	Sets the process capability state of the calling process to a 206 capability state in working storage.

207 **25.1.2.4 Capability State Format Translation**

208 This standard defines three different representations for a capability state: -

209	<i>external form</i>	The exportable, contiguous, persistent representation of a 210 capability state in user-managed space.
211	<i>internal form</i>	The internal representation of a capability state in working 212 storage managed by the capability functions.
213	<i>text form</i>	The structured text representation of a capability state.

214 These functions translate a capability state from one representation into another. +

215	<i>cap_copy_ext()</i>	Translates an internal form of a capability state to the exter- 216 nal form of a capability state.
217	<i>cap_copy_int()</i>	Translates the external form of a capability state to the inter- 218 nal form of a capability state.
219	<i>cap_from_text()</i>	Translates a text form of a capability state to the internal 220 form of a capability state.
221	<i>cap_size()</i>	Returns the size in bytes required to store the external form 222 of a capability state that is the result of an <i>cap_copy_ext()</i> .
223	<i>cap_to_text()</i>	Translates an internal form of a capability state to the text 224 form of a capability state.

225 **25.2 Header**

226 Some of the data types used by the capability functions are not defined as part of
227 this standard, but shall be implementation-defined. These types shall be defined
228 in the header <sys/capability.h>, which contains definitions for at least the
229 types shown in the following table.

230 **Table 25-2 – Capability Data Types**

231	Defined Type	Description
235	cap_flag_t	Used to identify capability flags. This data type is exportable data.
238	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.
240	cap_flag_value_t	Used to specify the value of capability flags. This data type is exportable data.
244	cap_value_t	Used to identify capabilities. This data type is exportable data.

246 The symbolic constants specified in the remainder of this section shall be defined
247 in the header <sys/capability.h>.

248 Table 25-3 contains *cap_flag_t* values for the *cap_get_flag()* and *cap_set_flag()*
249 functions.

250 **Table 25-3 – cap_flag_t Values**

251	Constant	Description
254	CAP_EFFECTIVE	Specifies the effective flag.
256	CAP_INHERITABLE	Specifies the inheritable flag.
258	CAP_PERMITTED	Specifies the permitted flag.

259 Table 25-4 contains *cap_flag_value_t* values for the *cap_get_flag()* and
260 *cap_set_flag()* functions.

261 **Table 25-4 – cap_flag_value_t Values**

262	Constant	Description
264	CAP_CLEAR	The flag is cleared/disabled.
266	CAP_SET	The flag is set/enabled.

268 Table 25-5 through Table 25-8 contains *cap_value_t* values for *cap_get_flag()* and
269 *cap_set_flag()*. Note that the description of each capability specifies exactly what
270 restriction the capability is intended to affect. Possession of a capability that
271 overrides one restriction should not imply that any other restrictions are overriden.
272 For example, possession of the CAP_DAC_OVERRIDE capability should not
273 imply that a process can read files with MAC labels that dominate that of the pro-
274 cess, nor should it override any restrictions that the file owner ID match the user
275 ID of the process.

276 If the {_POSIX_CAP} system configuration option is defined, the implementation

277 shall define at least the following set of *cap_value_t* values:

Table 25-5 – cap_value_t Values		
	Constant	Description
278	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 <i>chown()</i> function shall be equal to either the group ID or one of the supplementary group IDs of the calling process.
279	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.
280	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.
281	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.

326	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.
327		
328		
329		
330		
331		
332		
333		
334		
335		
336		
337		
359	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 <i>chown()</i> .
340		
341		
342		
343		
344		
345		
346		
347		
348		
349		
350		
351		
352		
353		
354		
355		
356		
357		
369	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.
360		
361		
362		
363		
364		
360	CAP_LINK_DIR	This capability overrides the restriction that a process cannot create or delete a hard link to a directory.
367		
368		
369		

370	CAP_SETFCAP	This capability shall override the restriction that a process cannot set the file capability state of a file.
373		
374		
375		
391	CAP_SETGID	This capability shall override the restriction in the <i>setgid()</i> 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 <i>{_POSIX_SAVED_IDS}</i> 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.
378		
379		
380		
381		
382		
383		
384		
385		
386		
387		
388		
389		
390		
392	CAP_SETUID	This capability shall override the restriction in the <i>setuid()</i> 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 <i>{_POSIX_SAVED_IDS}</i> is defined, then this capability also overrides any restrictions on setting the saved set-user-ID.
393		
394		
395		
396		
397		
398		
399		
400		
401		
402		
403		

405 If the *{_POSIX_MAC}* system configuration option is defined, the implementation –
 406 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
407	CAP_MAC_DOWNGRADE	This capability shall override the restriction that no process may downgrade the MAC label of a file.
408	CAP_MAC_READ	This capability shall override mandatory read access restrictions when accessing objects.
409	CAP_MAC_RELABEL_SUBJ	This capability shall override the restriction that a process may not modify its own MAC label.
410	CAP_MAC_UPGRADE	This capability shall override the restriction that no process may upgrade the MAC label of a file.
411	CAP_MAC_WRITE	This capability shall override mandatory write access restrictions when accessing objects.
412		
413		
414		
415		
416		
417		
418		
419		
420		
421		
422		
423		
424		
425		
426		
427		
428		
429		
430		
431		
432		
433		

435 If the {_POSIX_INF} system configuration option is defined, the implementation
 436 shall define at least the following set of *cap_value_t* values:

437 **Table 25-7 – cap_value_t Values for Information Labels**

438	Constant	Description
439	CAP_INF_NOFLOAT_OBJ	
440		This capability shall override the requirement that an object's information label shall automatically float when a write operation is per- formed by a process.
441		
442		
443		
444		
445		
446	CAP_INF_NOFLOAT_SUBJ	
447		This capability shall override the requirement that a pro- cess' information label shall automatically float when a read or execute operation is performed on an object.
448		
449		
450		
451		
452		
453	CAP_INF_RELABEL_OBJ	
454		This capability shall override the restriction against chang- ing the information label of an object.
455		
456		
457		
458	CAP_INF_RELABEL_SUBJ	
459		This capability shall override the restriction that a process may not modify its own infor- mation label in violation of the information labeling pol- icy.
460		
461		
462		
463		
464		

466 If the {_POSIX_AUD} system configuration option is defined, the implementation
 467 shall define at least the following set of *cap_value_t* values:

468 **Table 25-8 – cap_value_t Values for Audit**

469	Constant	Description
470	CAP_AUDIT_CONTROL	
471		This capability shall override the restriction that a process cannot modify audit control parameters.
472		
473		
474		
475	CAP_AUDIT_WRITE	
476		This capability shall override the restriction that a process cannot write data into the system audit trail.
477		
478		
479		

481 The symbolic constants defined in this section shall be implementation-defined
 482 unique values.

483 **25.3 Text Form Representation**

484 The text form of a capability state shall consist of one or more *clauses* con-
485 tained within a single, **NULL**-terminated character string. *Clauses* are
486 separated by whitespace characters. Each valid *clause* identifies a capability or a
487 set of capabilities, an *op* (operation), and one or more *flags* that the operation
488 applies to:

489 *clause* [SEP *clause*]...

490 where *clause* has the following format:

491 [*caplist*] *actionlist*

492 and SEP is “:” or any whitespace character.

493 *caplist* has the following format:

494 *capability_name*[,*capability_name*]... .

495 *actionlist* has the following format:

496 *op* [*flags*] [*op* [*flags*]]...

497 *op* is one of “=”, “-”, or “+”.

498 *flags* is a token consisting of one or more alphabetic characters.

499 The string shall be interpreted in order, e.g., the *op* specified in a later *clause* -
500 shall supplant or modify *op* that apply to the same capabilities in an earlier -
501 *clause*.

502 The *capability_name* symbols shall specify which capability or capabilities the
503 *clause* is to operate on. The symbols to be used are those defined in the
504 *capability.h* header file for each capability, e.g., “CAP_FOWNER”,
505 “CAP_SETUID”, etc. More than one *capability_name* may be specified in a
506 *clause* by separating them with a comma. A *capability_name* consisting of the
507 string “all” shall be equivalent to a list containing every capability defined by
508 the implementation. *Capability_names* are case insensitive on input, and the
509 case used for output shall be implementation defined.

510 The *flags* symbols e, i and p shall represent the *effective*, *inheritable* and *per-*
511 *mitted* capability flags, respectively. All lowercase characters for use as *flags*
512 symbols are reserved for use by future versions of this standard. Implementations
513 may define uppercase characters for *flags* to represent implementation-defined
514 flags.

515 If multiple *actionlists* are grouped with a single *caplist* in the grammar, each
516 *actionlist* shall be applied in the order specified with that *caplist*. The *op* symbols
517 shall represent the operation performed, as follows:

518 + If *flags* is not specified and *caplist* contains one or more capability
519 names, the + operation shall not change the capability state; else,
520 if *caplist* is not specified, this shall be considered an error; otherwise
521 if *caplist* is specified as “all”, the capability flags represented by *flags*
522 for all capabilities defined for the target by the implementation shall be
523 set; otherwise,
524 the flags specified in *flags* for all the capabilities specified in *caplist*
525 shall be set.

526 - If *flags* is not specified and *caplist* contains one or more capability
527 names, the - operation shall not change the target capability state; else,
528 if *caplist* is not specified or is specified as “all”, the capability flags
529 represented by *flags* for all capabilities defined by the implementation
530 shall be cleared; otherwise,
531 the capability flags specified in *flags* for all the capabilities specified in
532 *caplist* shall be cleared.

534 = Clear all the capability flags for the capabilities specified in *caplist*, or,
535 if no *caplist* is specified, clear all capability flags for all capabilities
536 defined by the implementation, then:
537 if *flags* is not specified, the = operation shall make no further
538 modification to the target capability state; else,
539 if *caplist* is not specified or is specified as “all”, the capability flags
540 represented by *flags* shall be set for all capabilities defined for the tar-
541 get by the implementation; otherwise,
542 the capability flags represented by *flags* shall be set for all the capabili-
543 ties specified in *caplist* in the target capability state.

544 25.3.1 Grammar

545 The grammar and lexical conventions in this subclause describe the syntax for the
546 textual representation of capability state. The general conventions for this style of
547 grammar are described in POSIX.2, "Grammar Conventions", 2.1.2. A valid *capa-*
548 *bility state* can be represented as the nonterminal symbol *capability state* in the
549 grammar. The formal syntax description in this grammar shall take precedence
550 over the textual descriptions in this clause.

551 The lexical processing shall be based on single characters except for capability
552 name recognition. Implementations need not allow whitespace characters within
553 the single argument being processed.

```

554      %start      capability_state
555      %%
556      capability_state : clause
557          | capability_state clause
558          ;
559
560      clause       : actionlist
561          | caplist actionlist
562          ;
563
564      caplist      : capability_name
565          | caplist ',' capability_name
566          ;
567
568      actionlist   : action
569          | actionlist action
570          ;
571
572      action       : op
573          | op flaglist
574          ;
575
576      op           : ``+``
577          | ``-``
578          | ``=``
579
580      flaglist     : flag
581          | flaglist flag
582
583      flag         : ``e``
584          | ``i``
585          | ``p``
586
587
588
589

```

582 25.4 Functions

583 The functions in this section comprise the set of services that permit a process
 584 image to acquire, manipulate, and pass capabilities on to new process images they
 585 create. Support for the capability facility functions identified in this section is
 586 optional. If the symbol `{_POSIX_CAP}` is defined, the implementation supports
 587 the capability option and all of the capability functions shall be implemented as
 588 described in this section. If `{_POSIX_CAP}` is not defined, the result of calling any
 589 of these functions is unspecified.

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

593 **25.4.1 Initialize a Capability State in Working Storage**

594 Function: *cap_clear()*

595 **25.4.1.1 Synopsis**

```
596 #include <sys/capability.h>
597 int cap_clear (cap_t cap_p);
```

598 **25.4.1.2 Description**

599 The function *cap_clear()* shall initialize the capability state in working storage
600 identified by *cap_p* so that all capability flags for all capabilities defined in the
601 implementation shall be cleared.

602 **25.4.1.3 Returns**

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

605 **25.4.1.4 Errors**

606 If any of the following conditions occur, the *cap_clear()* function shall return -1
607 and set *errno* to the corresponding value:

608 [EINVAL] The value of the *cap_p* argument does not refer to a capability state in working storage. —

610 **25.4.1.5 Cross-References**

611 *cap_init()*, 25.4.11; *cap_set_flag()*, 25.4.14. |

612 **25.4.2 Copy a Capability State From System to User Space**

613 Function: *cap_copy_ext()*

614 **25.4.2.1 Synopsis**

```
615 #include <sys/capability.h>
616 ssize_t cap_copy_ext (void *ext_p, cap_t cap_p, ssize_t size) |
```

617 **25.4.2.2 Description**

618 The *cap_copy_ext()* function shall copy a capability state in working storage,
619 identified by *cap_p*, from system managed space to user-managed space (pointed
620 to by *ext_p*) and returns the length of the resulting data record. The *size* parameter
621 represents the maximum size, in bytes, of the resulting data record. The

622 *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.

627 **25.4.2.3 Returns**

628 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.

631 **25.4.2.4 Errors**

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

634 [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.

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

639 **25.4.2.5 Cross-References**

640 *cap_copy_int()* 25.4.3.

641 **25.4.3 Copy a Capability State From User to System Space**

642 Function: *cap_copy_int()*

643 **25.4.3.1 Synopsis**

```
644 #include <sys/capability.h>
645 cap_t cap_copy_int (const void *ext_p)
```

646 **25.4.3.2 Description**

647 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.

655 Note that the record pointed to by *ext_p* must have been obtained from a previous,
656 successful call to *cap_copy_ext()* for this function to work successfully.

657 This function may cause memory to be allocated. The caller should free any
658 releasable memory, when the capability state in working storage is no longer
659 required, by calling *cap_free()* with the *cap_t* as an argument.

660 **25.4.3.3 Returns**

661 Upon successful completion, the *cap_copy_int()* function returns a pointer to the
662 newly created capability state in working storage. Otherwise, a value of
663 **(*cap_t*)NULL** shall be returned and *errno* shall be set to indicate the error.

664 **25.4.3.4 Errors**

665 If any of the following conditions occur, the *cap_copy_int()* function shall return
666 **(*cap_t*)NULL** and set *errno* to the corresponding value:

667 [EINVAL] The value of the *ext_p* argument does not refer to a capability
668 data record as defined in section 25.3.

669 [ENOMEM] The capability state to be returned requires more memory than
670 is allowed by the hardware or system-imposed memory manage-
671 ment constraints.

672 **25.4.3.5 Cross-References**

673 *cap_copy_ext()*, 25.4.2; *cap_free()*, 25.4.5; *cap_init()*, 25.4.11.

674 **25.4.4 Duplicate a Capability State in Working Storage**

675 Function: *cap_dup()*

676 **25.4.4.1 Synopsis**

```
677 #include <sys/capability.h>
678 cap_t cap_dup (cap_t cap_p);
```

679 **25.4.4.2 Description**

680 The *cap_dup()* function returns a duplicate capability state in working storage
681 given the source object *cap_p*, allocating any memory necessary, and returning a
682 pointer to the newly created capability state. Once duplicated, no operations on
683 either capability state shall affect the other in any way.

684 This function may cause memory to be allocated. The caller should free any
685 releasable memory, when the capability state in working storage is no longer
686 required, by calling *cap_free()* with the *cap_t* as an argument.

687 **25.4.4.3 Returns**

688 Upon successful completion, the *cap_dup()* function returns a pointer to the newly
689 created capability state in working storage. Otherwise, a value of (*cap_t*)NULL |
690 shall be returned and *errno* shall be set to indicate the error.

691 **25.4.4.4 Errors**

692 If any of the following conditions occur, the *cap_dup()* function shall return |
693 (*cap_t*)NULL and set *errno* to the corresponding value:

694 [EINVAL] The value of the *cap_p* argument does not refer to a capability
695 state in working storage. —

696 [ENOMEM] The capability state to be returned requires more memory
697 than is allowed by the hardware or system-imposed memory
698 management constraints. —

699 **25.4.4.5 Cross-References**

700 *cap_free()*, 25.4.5. |

701 **25.4.5 Release Memory Allocated to a Capability State in Working
702 Storage**

703 Function: *cap_free()*

704 **25.4.5.1 Synopsis**

705 `#include <sys/capability.h>`
706 `int cap_free (void *obj_d);` |

707 **25.4.5.2 Description**

708 The function *cap_free()* shall free any releasable memory currently allocated to
709 the capability state in working storage identified by *obj_d*. The *obj_d* argument %
710 may identify either a *cap_t* entity, or a *char ** entity allocated by the *cap_to_text()*
711 function.

712 **25.4.5.3 Returns**

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

715 **25.4.5.4 Errors**

716 If any of the following conditions occur, the *cap_free()* function shall return -1 and
717 set *errno* to the corresponding value:

718 [EINVAL] The value of the *obj_d* argument does not refer to memory
719 recognized as releasable by the implementation. —

720 **25.4.5.5 References**

721 *cap_copy_int()*, 25.4.3; *cap_dup()*, 25.4.4; *cap_from_text()*, 25.4.6; *cap_get_fd()*, %
722 25.4.7; *cap_get_file()*, 25.4.8; *cap_get_proc()*, 25.4.10; *cap_init()*, 25.4.11; %
723 *cap_to_text()*, 25.4.17. |

724 **25.4.6 Convert Text to a Capability State in Working Storage**

725 Function: *cap_from_text()*

726 **25.4.6.1 Synopsis**

727 `#include <sys/capability.h>`
728 `cap_t cap_from_text (const char *buf_p);` |

729 **25.4.6.2 Description**

730 This function shall allocate and initialize a capability state in working storage. It
731 shall then set the contents of this newly-created capability state to the state
732 represented by the human-readable, null terminated character string pointed to
733 by *buf_p*. It shall then return a pointer to the newly created capability state.

734 This function may cause memory to be allocated. The caller should free any
735 releasable memory, when the capability state in working storage is no longer
736 required, by calling *cap_free()* with the *cap_t* as an argument.

737 The function shall recognize and correctly parse any string that meets the
738 specification in 25.3. The function shall return an error if the implementation can
739 not parse the contents of the string pointed to by *buf_p* or does not recognize any
740 *capability_name* or flag character as valid. The function shall also return an error
741 if any flag is both set and cleared within a single clause.

742 **25.4.6.3 Returns**

743 Upon successful completion, a non-NULL value is returned. Otherwise, a value of
744 (*cap_t*)NULL shall be returned and *errno* shall be set to indicate the error.

745 **25.4.6.4 Errors**

746 If any of the following conditions occur, the *cap_from_text()* function shall return |
747 (*cap_t*)**NULL** and set *errno* to the corresponding value:

748 [EINVAL] The *buf_p* argument does not refer to a character string as
749 defined in section 25.3, the string pointed to by *buf_p* is not
750 parseable by the function, the text string contains a
751 *capability_name* or a flag character that the implementation
752 does not recognize as valid.

753 [ENOMEM] The capability state to be returned requires more memory
754 than is allowed by the hardware or system-imposed memory
755 management constraints. —

756 **25.4.6.5 Cross-References**

757 *cap_to_text()*, 25.4.17; *cap_free()*, 25.4.5; *cap_init()*, 25.4.11; *cap_set_flag()*, |
758 25.4.14.

759 **25.4.7 Get the Capability State of an Open File**

760 Function: *cap_get_fd()*

761 **25.4.7.1 Synopsis**

762 #include <sys/capability.h>
763 *cap_t cap_get_fd (int fd);* |

764 **25.4.7.2 Description**

765 The function *cap_get_fd()* shall allocate a capability state in working storage and
766 set it to represent the capability state of the file open on the descriptor *fd*, then
767 return a pointer to the newly created capability state.

768 A process can get the capability state of any regular file for which the process has |
769 a valid file descriptor. If the file open on the descriptor *fd* is not a regular file,
770 then *cap_get_fd()* shall return an error. If {POSIX_CAP_PRESENT} is not in |
771 effect for the file, then the results of *cap_get_fd()* shall be implementation-defined. |

772 If {POSIX_MAC} is defined, the process must also have mandatory access control
773 read access to the file. —

774 This function may cause memory to be allocated. The caller should free any
775 releasable memory, when the capability state in working storage is no longer
776 required, by calling *cap_free()* with the *cap_t* as an argument.

777 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.

781 25.4.7.4 Errors

782 If any of the following conditions occur, the `cap_get_fd()` function shall return
783 `(cap_t)NULL` and set `errno` to the corresponding value:

784	[EACCES]	If the <code>{_POSIX_MAC}</code> system configuration option is enabled, MAC read access to the file is denied.
786	[EBADF]	The <i>fd</i> argument is not a valid open file descriptor. +
787	[EINVAL]	The file open on <i>fd</i> is not a regular file.
788	[ENOMEM]	The capability state to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints. -
789		
790		

791 25.4.7.5 Cross-References

792 *cap_init()*, 25.4.11; *cap_free()*, 25.4.5; *cap_get_file()*, 25.4.8; *cap_set_fd()*, 25.4.12.

793 25.4.8 Get the Capability State of a File

794 Function: *cap_get_file()*

795 25.4.8.1 Synopsis

```
796 #include <sys/capability.h>
797 cap_t cap_get_file (const char *path_p);
```

798 25.4.8.2 Description

799 The function *cap_get_file()* shall allocate a capability state in working storage and
800 set it to be equal to the capability state of the pathname pointed to by *path_p*,
801 then return a pointer for the newly created capability state in working storage.

802 A process can get the capability state of any regular file for which the process has +
803 search access to the path specified. If the file pointed to by *path_p* is not a regular+
804 file, then *cap_get_file()* shall return an error. If *{_POSIX_CAP_PRESENT}* is not +
805 in effect for the file, then the results of *cap_get_file()* shall be implementation- +
806 defined. +

807 If `{_POSIX_MAC}` is defined, the process must also have MAC read access to the
808 file.

809 This function may cause memory to be allocated. The caller should free any
810 releasable memory, when the capability state in working storage is no longer
811 required, by calling *cap_free()* with the *cap_t* as an argument.

812 **25.4.8.3 Returns**

813 Upon successful completion, this function returns a non-**NULL** value. Otherwise,
814 a value of (*cap_t*)**NULL** shall be returned and *errno* shall be set to indicate the
815 error.

816 **25.4.8.4 Errors**

817 If any of the following conditions occur, the *cap_get_file()* function shall return
818 (*cap_t*)**NULL** and set *errno* to the corresponding value:

819 [EACCES]	Search permission is denied for a component of the path prefix, or, if { <u>POSIX_MAC</u> } is defined, MAC read access to the file <i>path_p</i> is denied.
822 [EINVAL]	The file pointed to by <i>path_p</i> is not a regular file.
823 [ENAMETOOLONG]	The length of the <i>path_p</i> argument exceeds { <u>PATH_MAX</u> }, or a pathname component is longer than { <u>NAME_MAX</u> } while { <u>POSIX_NO_TRUNC</u> } is in effect.
826 [ENOENT]	The named file does not exist or the <i>path_p</i> argument points to an empty string.
828 [ENOMEM]	The capability state to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.
831 [ENOTDIR]	A component of the path prefix is not a directory.

832 **25.4.8.5 Cross-References**

833 *cap_free()*, 25.4.5; *cap_init()*, 25.4.11; *cap_set_file()*, 25.4.13; *cap_get_fd()*, 25.4.7.

834 **25.4.9 Get the Value of a Capability Flag**

835 Function: *cap_get_flag()*

836 **25.4.9.1 Synopsis**

```
837 #include <sys/capability.h>
838 int cap_get_flag (cap_t cap_p,
839                   cap_value_t cap,
840                   cap_flag_t flag,
841                   cap_flag_value_t *value_p);
```

842 **25.4.9.2 Description**

843 The function *cap_get_flag()* shall obtain the current value of the capability flag
844 *flag* of the capability *cap* from the capability state in working storage identified by
845 *cap_p* and place it into the location pointed to by *value_p*.

846 **25.4.9.3 Returns**

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

849 **25.4.9.4 Errors**

850 If any of the following conditions occur, the *cap_get_flag()* function shall return -1
851 and set *errno* to the corresponding value:

852 [EINVAL] At least one of the values of the *cap_p*, *cap*, *flag* and *value_p*
853 arguments does not refer to the corresponding entity. —

854 **25.4.9.5 Cross-References**

855 *cap_set_flag()*, 25.4.14. |

856 **25.4.10 Obtain the Current Process Capability State**

857 Function: *cap_get_proc()*

858 **25.4.10.1 Synopsis**

859 #include <sys/capability.h>
860 cap_t cap_get_proc (void); |

861 **25.4.10.2 Description**

862 The function *cap_get_proc()* shall allocate a capability state in working storage,
863 set its state to that of the calling process, and return a pointer to the newly
864 created capability state.

865 This function may cause memory to be allocated. The caller should free any
866 releasable memory, when the capability state in working storage is no longer
867 required, by calling *cap_free()* with the *cap_t* as an argument.

868 **25.4.10.3 Returns**

869 Upon successful completion, this function shall return a *cap_t* value. Otherwise,
870 a value of (*cap_t*)NULL shall be returned and *errno* shall be set to indicate the
871 error. |

872 **25.4.10.4 Errors**

873 If any of the following conditions occur, the *cap_get_proc()* function shall return |
874 (*cap_t*)**NULL** and set *errno* to the corresponding value:

875 [ENOMEM] The capability state to be returned requires more memory
876 than is allowed by the hardware or system-imposed memory
877 management constraints. —

878 **25.4.10.5 Cross-References**

879 *cap_free()*, 25.4.5; *cap_init()*, 25.4.11; *cap_get_flag()*, 25.4.9; *cap_set_proc()*, |
880 25.4.15.

881 **25.4.11 Allocate and Initialize a Capability State in Working Storage**

882 Function: *cap_init()*

883 **25.4.11.1 Synopsis**

884 #include <sys/capability.h>
885 *cap_t cap_init (void);* |

886 **25.4.11.2 Description**

887 The function *cap_init()* shall create a capability state in working storage and
888 return a pointer to the capability state. The initial value of all flags for all capa-
889 bilities defined by the implementation shall be cleared.

890 This function may cause memory to be allocated. The caller should free any
891 releasable memory, when the capability state in working storage is no longer
892 required, by calling *cap_free()* with the *cap_t* as an argument.

893 **25.4.11.3 Returns**

894 Upon successful completion, this function returns a non-**NULL** *cap_t* value. Oth-
895 erwise, a value of (*cap_t*)**NULL** shall be returned and *errno* shall be set to indicate|
896 the error.

897 **25.4.11.4 Errors**

898 If any of the following conditions occur, the *cap_init()* function shall return |
899 (*cap_t*)**NULL** and set *errno* to the corresponding value:

900 [ENOMEM] The capability state to be returned requires more memory
901 than is allowed by the hardware or system-imposed memory
902 management constraints. —

903 **25.4.11.5 Cross-References**

904 *cap_free()*, 25.4.5.

905 **25.4.12 Set the Capability State of an Open File**

906 Function: *cap_set_fd()*

907 **25.4.12.1 Synopsis**

908 `#include <sys/capability.h>`
909 `int cap_set_fd (int fd, cap_t cap_p);`

910 **25.4.12.2 Description**

911 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 912 capability state identified by *cap_p*. The new capability state of the file identified 913 by *fd* shall be completely determined by the contents of *cap_p*. +

915 For this function to succeed, the process calling it must have the CAP_SETFCAP 916 capability enabled and either the effective user ID of the process must match the 917 file owner or the calling process must have the effective CAP_FOWNER capability 918 flag set. In addition, if `{_POSIX_MAC}` is defined, then the process must have 919 MAC write access to the file. Implementations may place additional restrictions 920 on setting the capability state of a file.

921 If the file open on the descriptor *fd* is not a regular file, then *cap_set_fd()* shall 922 return an error.

923 **25.4.12.3 Returns**

924 Upon successful completion, the function shall return a value of zero. Otherwise, 925 a value of -1 shall be returned and *errno* shall be set to indicate the error. The 926 capability state of the file shall not be affected if the return value is -1.

927 **25.4.12.4 Errors**

928 If any of the following conditions occur, the *cap_set_fd()* function shall return -1 929 and set *errno* to the corresponding value:

930 [EACCES] The requested access to the file specified is denied,
931 or the `{_POSIX_MAC}` system configuration option is
932 enabled and MAC write access to the file opened on descrip-
933 tor *fd* is denied.

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

935 [EINVAL] The value of the *cap_p* argument does not refer to a capabil-
936 ity state in working storage. +

937		The file open on <i>fd</i> is not a regular file.
938	[EPERM]	The process does not have appropriate privilege or does not meet other restrictions imposed by the implementation to perform the operation.
941	[EROFS]	This function requires modification of a file resident on a file system which is currently read-only.

943 **25.4.12.5 Cross-References**

944 *cap_get_fd()*, 25.4.7; *cap_set_file()*, 25.4.13.

945 **25.4.13 Set the Capability State of a File**

946 Function: *cap_set_file()*

947 **25.4.13.1 Synopsis**

```
948 #include <sys/capability.h>
949 int cap_set_file (const char *path_p, cap_t cap_p);
```

950 **25.4.13.2 Description**

951 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 952 the capability state identified by *cap_p*. The new capability state of the file 953 identified by *path_p* shall be completely determined by the contents of *cap_p*.

955 For this function to succeed, the process must have the CAP_SETFCAP capability 956 enabled and either the effective user ID of the process must match the file owner 957 or the calling process must have the effective flag of the CAP_FOWNER capability 958 set. In addition, if {_POSIX_MAC} is defined, then the process must have MAC 959 write access to the file. Implementations may place additional restrictions on set- 960 ting the capability state of a file.

961 If the file pointed to by *path_p* is not a regular file, then *cap_set_file()* shall return 962 an error. The effects of writing capability state to any file type other than a regu- 963 lar file are undefined.

964 **25.4.13.3 Returns**

965 Upon successful completion, the function shall return a value of zero. Otherwise, 966 a value of -1 shall be returned and *errno* shall be set to indicate the error. The 967 capability state of the file shall not be affected if the return value is -1.

968 **25.4.13.4 Errors**

969 If any of the following conditions occur, the *cap_set_file()* function shall return -1
970 and set *errno* to the corresponding value:

971	[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 <i>path_p</i> is denied.	-
975	[EINVAL]	The value of the <i>cap_p</i> 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.	+
978		The file pointed to by <i>path_p</i> is not a regular file.	
979	[ENAMETOOLONG]	The length of the <i>path_p</i> argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {POSIX_NO_TRUNC} is in effect.	
982	[ENOENT]	The named file/directory does not exist or the <i>path_p</i> argument points to an empty string.	-
984	[ENOTDIR]	A component of the path prefix is not a directory.	
985	[EPERM]	The process does not have appropriate privilege or does not meet other restrictions imposed by the implementation to perform the operation.	
988	[EROFS]	This function requires modification of a file resident on a file system which is currently read-only.	
989			

990 **25.4.13.5 Cross-References**

991 *cap_get_file()*, 25.4.8; *cap_set_fd()*, 25.4.12.

992 **25.4.14 Set the Value of a Capability Flag**

993 Function: *cap_set_flag()*

994 **25.4.14.1 Synopsis**

```
995 #include <sys/capability.h>
996 int cap_set_flag (cap_t cap_p,
997                     cap_flag_t flag,
998                     int ncap,
999                     cap_value_t caps[],
1000                    cap_flag_value_t value);
```

1001 **25.4.14.2 Description**

1002 This function shall set the flag *flag* of each capability in the array *caps* in the
1003 capability state in working storage identified by *cap_p* to *value*. The argument
1004 *ncap* is used to specify the number of capabilities in the array *caps*. Implementa-
1005 tions may place restrictions on the setting of the flags in a capability state.

1006 **25.4.14.3 Returns**

1007 Upon successful completion, the function shall return a value of zero. Otherwise,
1008 a value of -1 shall be returned and *errno* shall be set to indicate the error. The
1009 capability state identified by *cap_p* shall not be affected if the return value is -1.

1010 **25.4.14.4 Errors**

1011 If any of the following conditions occur, the *cap_set_flag()* function shall return -1
1012 and set *errno* to the corresponding value:

1013 [EINVAL] At least one of the values of *cap_p*, *ncap*, *flag* and *value*, or
1014 at least one of the first *ncap* elements in *caps*, does not refer
1015 to the corresponding entity. —

1016 The resulting capability state identified by *cap_p* violates %
1017 one or more implementation restrictions. —

1018 **25.4.14.5 Cross-References**

1019 *cap_get_flag()*, 25.4.16.

1020 **25.4.15 Set the Process Capability State**

1021 Function: *cap_set_proc()*

1022 **25.4.15.1 Synopsis**

```
1023 #include <sys/capability.h>
1024 int cap_set_proc (cap_t cap_p);
```

1025 **25.4.15.2 Description**

1026 The function *cap_set_proc()* shall set the values for all capability flags for all capa-
1027 bilities defined in the implementation with the capability state identified by
1028 *cap_p*. The new capability state of the process shall be completely determined by
1029 the contents of *cap_p* upon successful return from this function. If any flag in
1030 *cap_p* is set for any capability not currently permitted for the calling process, the
1031 function shall fail, and the capability state of the process shall remain unchanged.

1032 **25.4.15.3 Returns**

1033 Upon successful completion, the function shall return a value of zero. Otherwise,
1034 a value of -1 shall be returned and *errno* shall be set to indicate the error. Neither
1035 the state represented in the object identified by *cap_p* nor the capability state of
1036 the calling process shall be affected if the return value is -1.

1037 **25.4.15.4 Errors**

1038 If any of the following conditions occur, *cap_set_proc()* shall return -1 and set
1039 *errno* to the corresponding value:

1040 [EINVAL] The value of the *cap_p* argument does not refer to a capability
1041 state in working storage. —

1042 [EPERM] The caller attempted to set a capability flag of a capability
1043 that was not permitted to the invoking process.

1044 [ENOMEM] The function requires more memory than is allowed by the
1045 hardware or system-imposed memory management constraints.
1046

1047 **25.4.15.5 Cross-References**

1048 *cap_get_proc()*, 25.4.10.

|

1049 **25.4.16 Get the Size of a Capability Data Record**

1050 Function: *cap_size()*

1051 **25.4.16.1 Synopsis**

```
1052 #include <sys/capability.h>
1053 ssize_t cap_size (cap_t cap_p)
```

1054 **25.4.16.2 Description**

1055 The *cap_size()* function returns the total length (in bytes) that the capability state
1056 in working storage identified by *cap_p* would require when converted by
1057 *cap_copy_ext()*. This function is used primarily to determine the amount of buffer
1058 space that must be provided to the *cap_copy_ext()* function in order to hold the
1059 capability data record created from *cap_p*.

1060 **25.4.16.3 Returns**

1061 Upon successful completion, the *cap_size()* function returns the length required to
1062 hold a capability data record. Otherwise, a value of (*ssize_t*)-1 shall be returned
1063 and *errno* shall be set to indicate the error.

|

1064 **25.4.16.4 Errors**

1065 If any of the following conditions occur, *cap_size()* shall return **-1** and set *errno* to
1066 one of the following values:

1067 [EINVAL] The value of the *cap_p* argument does not refer to a capability state in working storage.

1069 **25.4.16.5 Cross-References**

1070 *cap_copy_ext()*, 25.4.2.

1071 **25.4.17 Convert a Capability State in Working Storage to Text**

1072 Function: *cap_to_text()*

1073 **25.4.17.1 Synopsis**

1074 `#include <sys/capability.h>`
1075 `char *cap_to_text (cap_t cap_p, size_t *len_p);`

1076 **25.4.17.2 Description**

1077 This function shall convert the capability state in working storage identified by
1078 *cap_p* into a null terminated human-readable string. This function allocates any
1079 memory necessary to contain the string, and returns a pointer to the string. If the
1080 pointer *len_p* is not (*size_t*)**NULL**, the function shall also return the full length of
1081 the string (not including the null terminator) in the location pointed to by *len_p*.
1082 The capability state in working storage identified by *cap_p* shall be completely
1083 represented in the returned character string.

1084 The format of the string pointed to by the returned pointer shall comply with the
1085 specification in 25.3.

1086 This function may cause memory to be allocated. The caller should free any
1087 releasable memory, when the capability state in working storage is no longer
1088 required, by calling *cap_free()* with the *cap_t* as an argument.

1089 **25.4.17.3 Returns**

1090 Upon successful completion, a non-**NULL** value is returned. Otherwise, a value of
1091 (*char* *)**NULL** shall be returned and *errno* shall be set to indicate the error.

1092 **25.4.17.4 Errors**

1093 If any of the following conditions occur, *cap_to_text()* shall return (*char* *)**NULL**
1094 and set *errno* to the corresponding value:

1095	[EINVAL]	Either the <i>cap_p</i> argument does not refer to a capability state in working storage or the <i>len_p</i> argument is invalid, or both.
1098	[ENOMEM]	The string to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.
1101	25.4.17.5 Cross-References	
1102	<i>cap_free()</i> , 25.4.5; <i>cap_get_flag()</i> , 25.4.16; <i>cap_from_text()</i> , 25.4.6.	

1

Section 26: Mandatory Access Control

2 26.1 General Overview

3 This section describes the Mandatory Access Control Option. The section defines
4 and discusses MAC concepts, outlines the MAC policy adopted in this standard
5 and the impact of MAC on existing POSIX.1 functions. Support for the interfaces
6 defined in this section is optional but shall be provided if the symbol
7 `{_POSIX_MAC}` is defined.

8 26.1.1 MAC Concepts

9 MAC Labels

10 MAC labels form the basis for mandatory access control decisions. In order to
11 promote flexibility in which conforming implementations may define a MAC pol-
12 icy, specific components of MAC labels and their textual representations are
13 implementation-defined.

14 Label Relationships

15 Two relationships are defined between MAC labels: *equivalence*, and *dominance*.
16 The details of *dominance* are left to the definition of the conforming implemen-
17 tation, however the dominance relation shall constitute a partial order on MAC
18 labels. *Equivalence* is defined relative to *dominance*. If two MAC labels are
19 equivalent, then each dominates the other.

20 MAC Objects

21 MAC objects are the interface-visible data containers, i.e., entities that receive or
22 contain data, to which MAC is applied. In POSIX.1, these include the following:

23 Files

24 This includes regular files, directories, FIFO-special files, and (unnamed)
25 pipes.

26 Processes

27 In cases where a process is the target of some request by another process,
28 that target process shall be considered an object.

29 MAC Subjects

30 A subject is an active entity that can cause information to flow between controlled
31 objects. Since processes are the only such interface-visible element of POSIX.1

32 they are the only subjects treated in this document.

33 **26.1.2 MAC Policy**

34 The MAC policy presented below is logically structured into the following named
35 policies:

36 **P:** The fundamental statement of mandatory access control policy

37 **FP.*:** The refinements of **P** that apply to file objects (**FP.1**, **FP.2**, etc.)

38 **PP.*:** The refinements of **P** that apply to process objects

39 The following labeling requirement shall be imposed:

40 Each subject and each object shall have a MAC label associated with it at all
41 times.

42 A physically unique MAC label is not required to be associated with each subject
43 and object. The requirement is only that a MAC label shall always be associated
44 with each subject and object. For example, all files in a file system could share a
45 single MAC label.

46 Policies for initial assignment and constraints on the changing of MAC labels are
47 given in the refining policies below.

48 The fundamental MAC restriction **P** is simply stated:

49 **P:** Subjects cannot cause information labeled at some MAC label **L₁** to
50 become accessible to subjects at **L₂** unless **L₂** dominates **L₁**.

51 This covers all data entities visible at the POSIX.1e interface, and includes res-
52 trictions on re-labeling data, i.e., changing the label of an object, as well as move-
53 ment of that data between objects. **P** covers all forms of data transmission visible
54 through the POSIX interface.

55 There are several important exceptions or limitations to the application of **P** and
56 its refinements to POSIX.1 functions:

57 **Covert Channel Exceptions**

58 Policy statement **P** strictly requires that there be no covert channels. Consis-
59 tent with this policy statement the new POSIX.1e functions and the
60 changes to existing POSIX.1 functions have been specified such that covert
61 channels are not inherent in their definition. This standard does not require
62 conforming implementations to be free of covert channels.

63 **Processes Possessing Appropriate Privileges**

64 Implicit in the statement of **P** is the assumption that none of the policies need
65 necessarily apply to processes possessing appropriate privilege unless expli-
66 citly stated. If {_POSIX_CAP} is defined, the list of capabilities that satisfy
67 the appropriate privilege requirements are defined in this standard in section
68 25.2.

69 **Devices**

70 The MAC policy on devices may have additional restrictions or refinements
71 not addressed here. The MAC policy on devices is unspecified.

72 **Additional Implementation Restrictions**

73 It is understood that a conforming implementation may enforce additional
74 security restrictions consistent with these policies.

75 **26.1.2.1 FP: File Function Policies**

76 Mandatory access control for files results from the application of basic policies
77 (**FP.***) to a simple assumption of the file data object. The straightforward applica-
78 tion of these rules to the object model determines the specific MAC restrictions
79 for a large number of file-related interfaces. The object that encompasses a
80 POSIX.1 file shall be defined to consist of a data portion and an attribute portion.
81 For the purposes of mandatory access control, the following assumption is made:

82 Both the data and attribute portion of a file are considered a single MAC-
83 labeled data container. Note that the MAC label shall be considered to be in
84 the attribute portion.

85 Note that, within this standard, and as a basis for defining interface behavior,
86 link names are considered as the contents of directories, and are not a property of
87 the file that they indicate. They are protected by and considered labeled at the
88 MAC label of their containing directory.

89 The following policy rules apply:

90 **FP.1:** The MAC label of a file shall be dominated by the MAC label of a sub-
91 ject for the subject to read the data or attributes of a file.

92 **FP.2:** The MAC label of a file shall dominate the MAC label of a subject for
93 the subject to write the data or attributes of a file.

94 The general POSIX.1e mandatory access control policy shall be that
95 subjects may write objects if the MAC label of the subject is dominated
96 by the object's MAC label. In accordance with the policy in 2.3.2 that
97 further restrictions may be placed on a policy, an implementation could
98 choose to be more restrictive by allowing a subject to write to a file only
99 when the MAC labels are equivalent.

100 **FP.3:** If reading from a FIFO-special file changes either the attributes or the
101 data of the FIFO object, both **FP.1** and **FP.2** shall be satisfied.

102 **FP.4:** A newly created object shall be assigned a MAC label which dominates
103 the MAC label of the creating subject.

104 The general POSIX.1e mandatory access control policy shall be that
105 newly created objects shall be assigned a MAC label which dominates

106 the MAC label of the creating subject. Although this policy statement
107 allows creation of upgraded objects, this standard only provides inter-
108 faces which will create objects with equivalent MAC labels to the MAC
109 label of the creating subject.

110 The MAC label of a file object cannot be modified in violation of **P**, e.g., processes
111 which do not possess appropriate privilege cannot downgrade the label of a file
112 object.

113 (Unnamed) pipes are considered objects, although they are not addressable by
114 pathname.

115 **26.1.2.1.1 Summary of POSIX.1 System Interface Impact**

116 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
121	access	5.6.3
122	chdir	5.2.1
123	chmod	5.6.4
124	chown	5.6.5
125	creat	5.3.2
126	execl	3.1.2
127	execv	3.1.2
128	execle	3.1.2
129	execve	3.1.2
130	execlp	3.1.2
131	execvp	3.1.2
132	fcntl	6.5.2
133	getcwd	5.2.2
134	link	5.3.4
135	mkdir	5.4.1
136	mkfifo	5.4.2
137	open	5.3.1
138	opendir	5.1.2
139	pipe	6.1.1
140	rename	5.5.3
141	rmdir	5.5.2
142	stat	5.6.2
143	unlink	5.5.1
144	utime	5.6.6
145	New Function	POSIX.1e Synopsis
146		
149	acl_delete_def_file	Delete a Default ACL of a File
150	acl_get_fd	Get an ACL of an Open File
152	acl_get_file	Get an ACL of a File
154	acl_set_fd	Set an ACL of an Open File
156	acl_set_file	Set an ACL of a File
159	inf_get_fd	Get the Information Label of an Open File
160	inf_get_file	Get the Information Label of a File
162	inf_set_fd	Set the Information Label of an Open File
164	inf_set_file	Set the Information Label of a File
166	mac_get_fd	Get the MAC Label of an Open File
168	mac_get_file	Get the MAC Label of a File
170	mac_set_fd	Set the MAC Label of an Open File
172	mac_set_file	Set the MAC Label of a File
174	cap_get_fd	Get the Capability State of an Open File
176	cap_get_file	Get the Capability State of a File
178	cap_set_fd	Set the Capability State of an Open File
180	cap_set_file	Set the Capability State of a File

WITHDRAWN DRAFT. All Rights Reserved by IEEE.
Preliminary—Subject to Revision.

182 **26.1.2.2 PP: Process Function Policies**

183 Mandatory access control for processes stems from the application of the basic
184 MAC restriction to the affected POSIX.1 functions. When treated as an object,
185 the process shall consist of its internal data (including the environment data), its
186 executable image, and its status information.

187 The following policy rules apply:

188 **PP.1:** No process at MAC label L_1 may write to a process at label L_2 unless L_2
189 dominates L_1 .

190 **PP.2:** A newly created process shall be assigned a MAC label which dominates
191 the MAC label of the creating process.

192 The general POSIX.1 mandatory access control policy shall be that newly
193 created processes shall be assigned a MAC label which dominates the
194 MAC label of the creating process. Although this policy statement allows
195 creation of upgraded processes, this standard only provides interfaces
196 which create processes with equivalent MAC labels as the creating pro-
197 cess.

198 The MAC label of a process cannot be altered in violation of **P**, e.g., a process
199 which do not possess appropriate privilege cannot downgrade its own MAC label.

200 **26.1.2.2.1 POSIX.1 Functions Covered by MAC Process Policies**

201 This policy shall be applied to the POSIX.1 functions listed in Table 26-2.

202 **Table 26-2 – POSIX.1 Functions Covered by MAC Process Policies**

203 Existing 204 Function	POSIX.1 Section
206 fork	3.1.1
207 kill	3.3.2
208 New 209 Function	POSIX.1e Synopsis
212 mac_set_proc	Set the Process Label

213 **26.2 Header**

214 Some of the data types used by the MAC label functions are not defined as part of
215 this standard, but shall be implementation-defined. If `{_POSIX_MAC}` is defined,
216 these types shall be defined in the header `<sys/mac.h>`, which contains
217 definitions for at least the following type: `mac_t`.

218 **26.2.1 mac_t**

219 This type defines a pointer to an "exportable" object capable of holding a MAC
220 label. The object is opaque, persistent, and self-contained. It shall be possible to
221 create an independent copy of the entire MAC label in a user-defined location
222 using normal byte-copy of *mac_size()* bytes starting at the location pointed to by
223 the *mac_t*. It shall be possible to byte-copy the copy back into system-managed
224 space, and recommence processing of it there, even if the copy has been stored in
225 a file or elsewhere, or moved to a different process. The internal structure of the
226 MAC label is otherwise unspecified.

227 **26.3 Functions**

228 The functions in this section comprise the set of services that permit processes to
229 retrieve, compare, set, and convert MAC labels. Support for the functions and
230 policy described in this section is optional. If the symbol *{_POSIX_MAC}* is
231 defined, the implementation supports the Mandatory Access Control (MAC) labels
232 option and all of the MAC functions shall be implemented as described in this sec-
233 tion. If *{_POSIX_MAC}* is not defined, the result of calling any of these functions
234 is unspecified.

235 The error [ENOTSUP] shall be returned in those cases where the system supports
236 MAC labeling but the particular MAC label operation cannot be applied because
237 of restrictions imposed by the implementation.

238 **26.3.1 Test MAC Labels for Dominance**

239 Function: *mac_dominate()*

240 **26.3.1.1 Synopsis**

```
241 #include <sys/mac.h>
242 int mac_dominate (mac_t labela, mac_t labelb);
```

243 **26.3.1.2 Description**

244 The function *mac_dominate()* determines whether *labela* dominates *labelb*. The
245 precise method for determining domination is implementation-defined.

246 This function is provided to allow conforming applications to test for dominance
247 since a comparison of the labels themselves may yield an indeterminate result.

248 **26.3.1.3 Returns**

249 If an error occurs, the *mac_dominate()* function shall return a value of -1 and
250 *errno* shall be set to indicate the error. Otherwise, a value of 1 shall be returned
251 if label *labela* dominates *labelb*, and a value of 0 shall be returned if *labela* does
252 not dominate *labelb*.

253 **26.3.1.4 Errors**

254 If any of the following conditions occur, the *mac_dominate()* function shall return
255 -1 and set *errno* to the corresponding value:

256 [EINVAL] At least one of the labels is not a valid MAC label as defined by
257 *mac_valid()*. —

258 **26.3.1.5 Cross-References**

259 *mac_equal()*, 26.3.2; *mac_valid()*, 26.3.15.

|

260 **26.3.2 Test MAC Labels for Equivalence**

261 Function: *mac_equal()*

262 **26.3.2.1 Synopsis**

263 #include <sys/mac.h>

264 int mac_equal (mac_t *labela*, mac_t *labelb*);

265 **26.3.2.2 Description**

266 The function *mac_equal()* determines whether *labela* is equivalent to *labelb*. The
267 precise method for determining equivalence is implementation-defined.

268 This function is provided to allow conforming applications to test for equivalence
269 since a comparison of the labels themselves may yield an indeterminate result.

270 **26.3.2.3 Returns**

271 If an error occurs, a value of -1 shall be returned and *errno* shall be set to indicate
272 the error. Otherwise, the *mac_equal()* function returns 1 if *labela* is equivalent to
273 *labelb*, and a value of 0 shall be returned if *labela* is not equivalent to *labelb*.

|

274 **26.3.2.4 Errors**

275 If any of the following conditions occur, the *mac_equal()* function shall return -1
276 and set *errno* to the corresponding value:

277 [EINVAL] At least one of the labels is not a valid MAC label as defined by –
278 *mac_valid()*. –

279 **26.3.2.5 Cross-References**

280 *mac_dominant()*, 26.3.1; *mac_valid()*, 26.3.15.

|

281 **26.3.3 Free MAC Label Storage Space**

282 Function: *mac_free()*

283 **26.3.3.1 Synopsis**

```
284 #include <sys/mac.h>  
285 int mac_free (void *buf_p);
```

286 **26.3.3.2 Description**

287 The function *mac_free()* shall free any releasable memory currently allocated to
288 the buffer identified by *buf_p*. The *buf_p* argument may be either a (*void**)*mac_t*,
289 or a (*void**)char* allocated by the *mac_to_text()* function.

|

290 **26.3.3.3 Returns**

291 Upon successful completion, the function *mac_free()* returns a value of 0. Other-
292 wise, a value of -1 is returned and *errno* is set to indicate the error.

|

293 **26.3.3.4 Errors**

294 This standard does not specify any error conditions that are required to be
295 detected for the *mac_free()* function. Some errors may be detected under condi-
296 tions that are unspecified by this part of the standard.

|

297 **26.3.3.5 Cross-References**

298 *mac_from_text()*, 26.3.4; *mac_get_fd()*, 26.3.5; *mac_get_file()*, 26.3.6;
299 *mac_get_proc()*, 26.3.7; *mac_glb()*, 26.3.8; *mac_lub()*, 26.3.9.

|

300 **26.3.4 Convert Text MAC Label to Internal Representation**

301 Function: *mac_from_text()*

302 **26.3.4.1 Synopsis**

```
303 #include <sys/mac.h>
304 mac_t mac_from_text (const char *text_p);
```

305 **26.3.4.2 Description**

306 The function *mac_from_text()* converts the text representation of a MAC label
307 *text_p* into its internal representation.

308 This function may cause memory to be allocated. The caller should free any
309 releasable memory, when the MAC label is no longer required, by calling
310 *mac_free()* with the *mac_t* as an argument. In event an error occurs, no memory
311 shall be allocated and **NULL** shall be returned.

312 **26.3.4.3 Returns**

313 Upon successful completion, the function *mac_from_text()* shall return a pointer |
314 to the MAC label. Otherwise, no space shall be allocated, a (*mac_t*) **NULL** pointer|
315 shall be returned, and *errno* shall be set to indicate the error.

316 **26.3.4.4 Errors**

317 If any of the following conditions occur, the *mac_from_text()* function shall return
318 a **NULL** pointer and set *errno* to the corresponding value:

319 [EINVAL] The string *text_p* is not a valid textual representation of a MAC
320 label as defined by *mac_valid()*. —

321 [ENOMEM] The MAC label requires more memory than is allowed by the
322 hardware or system-imposed memory management constraints. —

323 **26.3.4.5 Cross-References**

324 *mac_free()*, 26.3.3; *mac_valid()*, 26.3.15. |

325 **26.3.5 Get the Label of a File Designated by a File Descriptor**

326 Function: *mac_get_fd()*

327 **26.3.5.1 Synopsis**

```
328 #include <sys/mac.h>
329 mac_t mac_get_fd (int fildes)
```

330 **26.3.5.2 Description**

331 The *mac_get_fd()* function returns the MAC label associated with an open file.
332 The function accepts a valid file descriptor to the file, allocates memory in which
333 to store the MAC label to be returned and copies the file MAC label into the allo-
334 cated memory.

335 A process can get the MAC label for any file for which the process has a valid file
336 descriptor and MAC read access.

337 This function may cause memory to be allocated. The caller should free any
338 releasable memory, when the MAC label is no longer required, by calling
339 *mac_free()* with the *mac_t* as an argument. In event an error occurs, no memory
340 shall be allocated and **NULL** shall be returned.

341 **26.3.5.3 Returns**

342 Upon successful completion, the function shall return a pointer to the MAC label. |
343 Otherwise, no space shall be allocated, a (*mac_t*)**NULL** pointer shall be returned |
344 and *errno* shall be set to indicate the error.

345 **26.3.5.4 Errors**

346 If any of the following conditions occur, the *mac_get_fd()* function shall return a |
347 (*mac_t*)**NULL** and set *errno* to the corresponding value:

348 [EACCES] MAC read access is denied to the file referred to by *fildes*.

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

350 [ENOMEM] The MAC label requires more memory than is allowed by the
351 hardware or system-imposed memory management constraints. –

352 **26.3.5.5 Cross-References**

353 *mac_free()*, 26.3.3; *mac_get_file()*, 26.3.6; *mac_set_fd()*, 26.3.10; *mac_set_file()*, |
354 26.3.11.

355 **26.3.6 Get the Label of a File Designated by a Pathname**

356 Function: *mac_get_file()*

357 **26.3.6.1 Synopsis**

358 #include <sys/mac.h>

359 *mac_t mac_get_file (const char *path_p);*

360 **26.3.6.2 Description**

361 The *mac_get_file()* function returns the MAC label associated with the pathname
362 pointed to by *path_p*. The function allocates memory in which to store the MAC
363 label to be returned and copies the file MAC label into the allocated memory.

364 A process can get the MAC label for any file for which the process has search
365 access to the path specified and MAC read access to the file.

366 This function may cause memory to be allocated. The caller should free any
367 releasable memory, when the MAC label is no longer required, by calling
368 *mac_free()* with the *mac_t* as an argument. In event an error occurs, no memory
369 shall be allocated and **NULL** shall be returned.

370 **26.3.6.3 Returns**

371 Upon successful completion, the function shall return a pointer to the MAC label.
372 Otherwise, no space shall be allocated, a (*mac_t*)**NULL** pointer shall be returned |
373 and *errno* shall be set to indicate the error.

374 **26.3.6.4 Errors**

375 If any of the following conditions occur, the *mac_get_file()* function shall return a |
376 (*mac_t*)**NULL** and set *errno* to the corresponding value:

377 [**EACCES**] Search permission is denied for a component of the path prefix
378 or MAC read access to the file is denied.

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

383 [**ENOENT**] The named file/directory does not exist, or the *path_p* argument
384 points to an empty string.

385 [**ENOMEM**] The MAC label requires more memory than is allowed by the
386 hardware or system-imposed memory management constraints. –

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

388 **26.3.6.5 Cross-References**

389 *mac_free()*, 26.3.3; *mac_get_fd()*, 26.3.5; *mac_set_fd()*, 26.3.10; *mac_set_file()*, |
390 26.3.11.

391 **26.3.7 Get the Process Label**

392 Function: *mac_get_proc()*

393 **26.3.7.1 Synopsis**

```
394 #include <sys/mac.h>
395 mac_t mac_get_proc (void);
```

396 **26.3.7.2 Description**

397 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.

400 Any process may query its MAC label.

401 This function may cause memory to be allocated. The caller should free any 402 releasable memory, when the MAC label is no longer required, by calling 403 *mac_free()* with the *mac_t* as an argument. In event an error occurs, no memory 404 shall be allocated and **NULL** shall be returned.

405 **26.3.7.3 Returns**

406 Upon successful completion, *mac_get_proc()* returns a pointer to the MAC label of+
407 the process. Otherwise, no space shall be allocated, a (*mac_t*)**NULL** pointer shall |
408 be returned and *errno* shall be set to indicate the error.

409 **26.3.7.4 Errors**

410 If any of the following conditions occur, the *mac_get_proc()* function shall return a|
411 (*mac_t*)**NULL** and set *errno* to the corresponding value:

412 [ENOMEM] The MAC label requires more memory than is allowed by the
413 hardware or system-imposed memory management constraints. –

414 **26.3.7.5 Cross-References**

415 *mac_free()*, 26.3.3; *mac_set_proc()*, 26.3.12.

416 **26.3.8 Compute the Greatest Lower Bound**

417 Function: *mac_glb()*

418 **26.3.8.1 Synopsis**

```
419 #include <sys/mac.h>
420 mac_t mac_glb (mac_t labela, mac_t labelb);
```

421 **26.3.8.2 Description**

422 The function *mac_glb()* returns a pointer to the (valid) MAC label, if it exists, that+
423 is dominated by both the MAC label *labela* and the MAC label *labelb* and dom-
424 inates all other valid MAC labels that are dominated by both the MAC label
425 *labela* and the MAC label *labelb*.

426 This function may cause memory to be allocated. The caller should free any
427 releasable memory, when the MAC label is no longer required, by calling
428 *mac_free()* with the *mac_t* as an argument. In event an error occurs, no memory
429 shall be allocated and **NULL** shall be returned.

430 **26.3.8.3 Returns**

431 Upon successful completion, this returns a pointer to the allocated bounding MAC
432 label. Otherwise, no space shall be allocated, a (*mac_t*)**NULL** pointer shall be |
433 returned, and *errno* shall be set to indicate the error.

434 **26.3.8.4 Errors**

435 If any of the following conditions occur, the *mac_glb()* function shall return a |
436 (*mac_t*)**NULL** and set *errno* to the corresponding value:

437 [EINVAL] At least one of the input labels is not a valid MAC label as -
438 defined by *mac_valid()*.

439 [ENOENT] The bounding MAC label does not exist or is not valid. -

440 [ENOMEM] The MAC label requires more memory than is allowed by the
441 hardware or system-imposed memory management constraints. -

442 **26.3.8.5 Cross-References**

443 *mac_free()*, 26.3.3; *mac_lub()*, 26.3.9; *mac_valid()*, 26.3.15. |

444 **26.3.9 Compute the Least Upper Bound**

445 Function: *mac_lub()*

446 **26.3.9.1 Synopsis**

```
447 #include <sys/mac.h>
448 mac_t mac_lub (mac_t labela, mac_t labelb);
```

449 26.3.9.2 Description

450 The function *mac_lub()* returns a pointer to the (valid) MAC label (if it exists) |
451 that dominates both the MAC label *labela* and the MAC label *labelb* and is dom- |
452 inated by all other valid MAC labels that dominate both the MAC label *labela* and |
453 the MAC label *labelb*.

454 This function may cause memory to be allocated. The caller should free any |
455 releasable memory, when the MAC label is no longer required, by calling |
456 *mac_free()* with the *mac_t* as an argument. In event an error occurs, no memory |
457 shall be allocated and **NULL** shall be returned.

458 26.3.9.3 Returns

459 Upon successful completion, this function shall return a pointer to the bounding |
460 MAC label. Otherwise, a (*mac_t*)**NULL** pointer shall be returned and *errno* shall |
461 be set to indicate the error.

462 26.3.9.4 Errors

463 If any of the following conditions occur, the *mac_lub()* function shall return a |
464 (*mac_t*)**NULL** and set *errno* to the corresponding value:

465 [EINVAL] At least one of the input labels is not a valid MAC label as –
466 defined by *mac_valid()*.

467 [ENOENT] The bounding MAC label does not exist or is not valid. –

468 [ENOMEM] The MAC label requires more memory than is allowed by the
469 hardware or system-imposed memory management constraints. –

470 26.3.9.5 Cross-References

471 *mac_free()*, 26.3.3; *mac_glb()*, 26.3.8; *mac_valid()*, 26.3.15. |

472 26.3.10 Set the Label of a File Identified by File Descriptor

473 Function: *mac_set_fd()*

474 26.3.10.1 Synopsis

```
475 #include <sys/mac.h>
476 int mac_set_fd (int fildes, mac_t label);
```

477 **26.3.10.2 Description**

478 This function sets the MAC label of a file to *label*. The function requires that
479 *fildes* be a valid file descriptor to indicate the file.
480 A process can set the MAC label for a file only if the process has a valid file
481 descriptor for the file and has MAC write access to the file. Additionally, only
482 processes with an effective user ID equal to the owner of the file or with appropri-
483 ate privileges may change the label of the file. If `{_POSIX_CAP}` is defined, then |
484 appropriate privilege shall include `CAP_FOWNER`. |
485 The `mac_set_fd()` function shall fail if the new MAC label is not equivalent to the |
486 file's previous label and the process does not possess appropriate privilege. If |
487 `{_POSIX_CAP}` is defined, and the new MAC label dominates, but is not |
488 equivalent to the file's previous MAC label, then appropriate privilege shall |
489 include `CAP_MAC_UPGRADE`. If `{_POSIX_CAP}` is defined, and the new MAC |
490 label does not dominate the file's previous MAC label then appropriate privilege |
491 shall include `CAP_MAC_DOWNGRADE`. |
492 It is implementation-defined whether an implementation will return `[EBUSY]` or
493 will perform revocation of access if other processes have current access to the file
494 at the time of MAC label modification.

495 **26.3.10.3 Returns**

496 Upon successful completion, the function shall return a value of 0. Otherwise, a
497 value of `-1` shall be returned and *errno* shall be set to indicate the error.

498 **26.3.10.4 Errors**

499 If any of the following conditions occur, the `mac_set_fd()` function shall return `-1`
500 and set *errno* to the corresponding value:

- 501 [`EACCES`] MAC write access is denied to the file specified.
- 502 [`EBADF`] The *fildes* argument is not a valid file descriptor.
- 503 [`EBUSY`] The file named by the *fildes* argument is currently in a state in
504 which the implementation does not allow the label to be
505 changed.
- 506 [`EINVAL`] The MAC label *label* is not a valid MAC label as defined by
507 `mac_valid()`.
- 508 [`ENOTSUP`] `{_POSIX_MAC}` is defined, but this function is not supported on |
509 the file referred to by *fildes*, i.e., `{_POSIX_MAC_PRESENT}` is |
510 not in effect for the file referred to by *fildes*.
- 511 [`EPERM`] An attempt was made to change the MAC label of a file and the
512 process does not possess appropriate privilege.
- 513 [`EROFS`] This function requires modification of a file system which is
514 currently read-only.

515 **26.3.10.5 Cross-References**

516 *mac_get_fd()*, 26.3.5; *mac_set_file()*, 26.3.11; *mac_valid()*, 26.3.15.

517 **26.3.11 Set the Label of a File Designated by Pathname**

518 Function: *mac_set_file()*

519 **26.3.11.1 Synopsis**

520 `#include <sys/mac.h>`
521 `int mac_set_file (const char *path_p, mac_t label);`

522 **26.3.11.2 Description**

523 This function sets the MAC label of the pathname pointed to by *path_p* to *label*.

524 A process can set the MAC label for a file only if the process has search access to
525 the path and has MAC write access to the file. Additionally, only processes with
526 an effective user ID equal to the owner of the file or with appropriate privileges
527 may change the label of the file. If `{_POSIX_CAP}` is defined, then appropriate
528 privilege shall include `CAP_FOWNER`.

529 The *mac_set_file()* function shall fail if the new MAC label is not equivalent to the
530 file's previous MAC label and the process does not possess appropriate privilege.
531 If `{_POSIX_CAP}` is defined, and the new MAC label dominates, but is not
532 equivalent to the file's previous MAC label, then appropriate privilege shall
533 include `CAP_MAC_UPGRADE`. If `{_POSIX_CAP}` is defined, and the new MAC
534 label does not dominate the file's previous MAC label then appropriate privilege
535 shall include `CAP_MAC_DOWNGRADE`.

536 It is implementation-defined whether an implementation will return [EBUSY] or
537 will perform revocation of access if other processes have current access to the file
538 at the time of MAC label modification.

539 **26.3.11.3 Returns**

540 Upon successful completion, the function shall return a value of 0. Otherwise, a
541 value of -1 shall be returned and *errno* shall be set to indicate the error.

542 **26.3.11.4 Errors**

543 If any of the following conditions occur, the *mac_set_file()* function shall return -1
544 and set *errno* to the corresponding value:

545 [EACCES] Search permission is denied for a component of the path prefix
546 or MAC write access to the target file is denied.

547 [EBUSY] The file or directory indicated by *path_p* is currently in a state in
 548 which the implementation does not allow the label to be
 549 changed.
 550 [EINVAL] The MAC label *label* is not a valid MAC label as defined by —
 551 *mac_valid()*.
 552 [ENAMETOOLONG]
 553 The length of the *path_p* argument exceeds {PATH_MAX}, or a
 554 pathname component is longer than {NAME_MAX} while
 555 {POSIX_NO_TRUNC} is in effect.
 556 [ENOENT] The named file/directory does not exist, or the *path_p* argument —
 557 points to an empty string.
 558 [ENOTDIR] A component of the path prefix is not a directory.
 559 [ENOTSUP] {_POSIX_MAC} is defined, but this function is not supported on |
 560 the file specified, i.e., {_POSIX_MAC_PRESENT} is not in effect
 561 for the file specified.
 562 [EPERM] An attempt was made to change the MAC label of a file and the
 563 process does not possess appropriate privilege.
 564 [EROFS] This function requires modification of a file system which is
 565 currently read-only.

566 26.3.11.5 Cross-References

567 *mac_get_file()*, 26.3.6; *mac_set_fd()*, 26.3.10; *mac_valid()*, 26.3.15. |

568 26.3.12 Set the Process Label

569 Function: *mac_set_proc()*

570 26.3.12.1 Synopsis

```
571 #include <sys/mac.h>
572 int mac_set_proc (mac_t label);
```

573 26.3.12.2 Description

574 The *mac_set_proc()* function is used to set (write) the MAC label of the requesting
 575 process. The new label is specified by *label*. A process may only alter its MAC
 576 label if it possesses appropriate privilege. If {_POSIX_CAP} is defined, then |
 577 appropriate privilege shall include CAP_MAC_RELABEL_SUBJ.

578 **26.3.12.3 Returns**

579 Upon successful completion, *mac_set_proc()* shall return a value of 0. Otherwise,
580 a value of -1 shall be returned and *errno* shall be set to indicate the error.

581 **26.3.12.4 Errors**

582 If any of the following conditions occur, the *mac_set_proc()* function shall return
583 -1 and set *errno* to the corresponding value:

584 [EINVAL] The MAC label *label* is not a valid MAC label as defined by –
585 *mac_valid()*.

586 [EPERM] The process does not have appropriate privilege to perform the
587 operation requested.

588 **26.3.12.5 Cross-References**

589 *mac_valid()*, 26.3.15.

|

590 **26.3.13 Get the Size of a MAC Label**

591 Function: *mac_size()*

592 **26.3.13.1 Synopsis**

```
593 #include <sys/mac.h>
594 ssize_t mac_size (mac_t label);
```

|

595 **26.3.13.2 Description**

596 The *mac_size()* function returns the size in bytes of the MAC label specified by
597 *label* if the label is valid. Note: this is the size of the internal MAC label, not the
598 size of the text representation as produced by the *mac_to_text()* function.

599 **26.3.13.3 Returns**

600 Upon successful completion, this function shall return the size of the MAC label.
601 Otherwise, a value of -1 shall be returned and *errno*

602 **26.3.13.4 Errors**

603 If any of the following conditions occur, the *mac_size()* function shall return -1
604 and set *errno* to the corresponding value:

605 [EINVAL] The MAC label *label* is invalid as defined by *mac_valid()*. -

606 **26.3.13.5 Cross-References**

607 *mac_valid()*, 26.3.15.

608 **26.3.14 Convert Internal MAC Label to Textual Representation**

609 Function: *mac_to_text()*

610 **26.3.14.1 Synopsis**

611 `#include <sys/mac.h>`
612 `char *mac_to_text (mac_t label, size_t *len_p);`

613 **26.3.14.2 Description**

614 The function *mac_to_text()* converts the internal representation of the MAC label
615 pointed to by *label* into a human-readable, **NULL** terminated, character string.
616 The output of *mac_to_text()* shall be suitable for re-input as the *text_p* parameter
617 to *mac_from_text()* in 26.3.4, or as the *label* operand to the *setfmac* utility as
618 defined in section 11 of POSIX.2c on the same system or other systems with ident-
619 ical MAC label definitions. The function returns a pointer to the text representa-
620 tion of the MAC label. If the pointer *len_p* is not **NULL**, the function shall return
621 the length of the string (not including the **NULL** terminator) in the location
622 pointed to by *len_p*.

623 This function may cause memory to be allocated. The caller should free any
624 releasable memory, when the text label is no longer required, by calling
625 *mac_free()* with the string address as an argument. In event an error occurs, no
626 memory shall be allocated and **NULL** shall be returned.

627 **26.3.14.3 Returns**

628 Upon successful completion, the function *mac_to_text()* returns a pointer to the
629 text representation of the MAC label, and if the pointer *len_p* is not **NULL**,
630 returns the length of the string (not including the **NULL** terminator) in the loca-
631 tion pointer to by *len_p*. Otherwise, no memory shall be allocated, the memory
632 referred to by *len_p* shall be unchanged, a (*char **) **NULL** pointer shall be
633 returned and *errno* shall be set to indicate the error.

634 **26.3.14.4 Errors**

635 If any of the following conditions occur, the *mac_to_text()* function shall return a
636 **NULL** pointer and set *errno* to the corresponding value:

637 [EINVAL] The MAC label *label* is not a valid MAC label as defined by –
638 *mac_valid()*.
639 [ENOMEM] The text to be returned requires more memory than is allowed
640 by the hardware or system-imposed memory management con-
641 straints. –

642 **26.3.14.5 Cross-References**

643 *mac_from_text()*, 26.3.4; *mac_valid()*, 26.3.15; *setfmac*, POSIX.2c - 11.3. |

644 **26.3.15 Label Validity**

645 Function: *mac_valid()*

646 **26.3.15.1 Synopsis**

647 `#include <sys/mac.h>`
648 `int mac_valid (mac_t label);`

649 **26.3.15.2 Description**

650 The *mac_valid()* function determines if *label* is a valid MAC label. The meaning –
651 of validity is implementation-defined. –

652 **26.3.15.3 Returns**

653 Upon successful completion, the function shall return 1 if *label* is valid, and 0 if it –
654 is invalid. Otherwise a value of -1 shall be returned and *errno* is set to indicate
655 the error. –

656 **26.3.15.4 Errors**

657 This standard does not specify any error conditions that are required to be |
658 detected for the *mac_valid()* function. Some errors may be detected under condi- |
659 tions that are unspecified by this part of the standard. |

660 **26.3.15.5 Cross-References**

661 None.

1

Section 27: Information Labeling

2 27.1 General Overview

3 This section describes the Information Label Option. The section defines and
4 discusses the information label concepts, outlines the information label policy
5 adopted in this standard, and outlines the impact of information labels on exist-
6 ing POSIX.1 functions. Support for the interfaces defined in this section is
7 optional but shall be provided if the symbol {POSIX_INF} is defined.

8 27.1.1 Information Label Concepts

9 Information Labels

10 The Information Label is the item visible at the POSIX.1 interface that is used for
11 associating labeling information with data. This labeling information is not
12 related to Mandatory Access Control, nor does the information labeling policies in
13 any way override the MAC or DAC options, if they are in effect.

14 In order to promote the flexibility with which conforming implementations may
15 define an information labeling policy, specific components of information labels
16 and their textual representation are not defined by this standard.

17 Information Label Relationships

18 Two relationships are defined between information labels: *equivalence* and *domi-*
19 *nance*. A conforming implementation must provide the interfaces for determining
20 whether two information labels have these relationships. Note that it would be
21 acceptable for a conforming implementation to implement information labels in
22 such a manner that no information label is equivalent to, nor dominates, any
23 information label other than itself. Thus, while interfaces for determining domi-
24 nance and equivalence must be provided, the detailed definitions of these rela-
25 tionships are left undefined.

26 Information Label Floating

27 The *inf_float()* operation is used in the statement of the information label policy.
28 The operation *inf_float(inf_p1, inf_p2)* returns an information label whose value is
29 dependent on the values of *inf_p1* and *inf_p2* and the implementation-defined
30 floating policy. The precise definition of *inf_float()* is left to the conforming imple-
31 mentation, however, its intended use is described in 27.1.2. (As a result of this
32 permitted flexibility, a conforming implementation could, for example, choose to
33 always return just *inf_p2*.)

34 **Information Label Subjects**

35 In the broad sense, a subject is an active entity that can cause information of any
36 kind to flow between controlled objects. Since processes are the only such
37 interface-visible element in this standard, they are the only subjects treated in
38 the information label section.

39 **Information Label Objects**

40 Objects are passive entities that contain or receive data. Access to objects potentially
41 implies access to the data they contain. However, objects not only contain
42 data, but also possess attributes. The data portion of an object is that portion
43 that contains the bytes intended to be stored by the object (e.g., the bytes written
44 to a regular file comprise that file's data portion). The attribute portion of an
45 object is that portion that contains descriptive, or control, information pertaining
46 to the object (e.g., a regular file's access and modification times, permission bits,
47 length, and so forth). The granting of access to an object's data and to that object's
48 attributes may be based upon different criteria. Information labeling, as
49 described in greater detail below, relies on this distinction.

50 The objects to which information labeling applies include the data portion of the
51 following objects: regular files, FIFO-special files, and (unnamed) pipes. Note
52 that conforming implementations may choose to apply the information labeling
53 policy more broadly by including, for example, object attributes.

54 **27.1.2 Information Label Policy**

55 The information label policy presented below is logically structured into the following named policies:

57 **I:** The fundamental statement of information labeling

58 **FI.*:** The refinements of **I** that apply to file objects (**FI.1**, **FI.2**, etc.)

59 **PI.*:** The refinements of **I** that apply to process objects

60 The following information labeling requirement is imposed:

61 Each subject and each object that contains data, as opposed to attributes (e.g., mandatory access control label and access time), shall have
62 as an additional attribute an information label at all times.

64 Policies for initial assignment and constraints on the changing of information
65 labels are given in the refining policies below.

66 The fundamental information label policy **I** is:

67 **I:** When subjects cause data (as opposed to attributes) to flow from a
68 source with information label *inf_p1* to a destination with information
69 label *inf_p2*, the destination's information label shall be
70 automatically set to the value returned by *inf_float* (*inf_p1*, *inf_p2*).

71 There are several important exceptions or limitations to the application of **I** and
72 its refinements to POSIX.1 functions:

73 **Processes Possessing Appropriate Privilege**

74 Implicit in the statement of **I** is the assumption that none of the policies
75 need necessarily apply to processes possessing appropriate privilege
76 unless explicitly stated. If {POSIX_CAP} is defined, the list of capabilities
77 that satisfy the appropriate privilege requirements are defined by
78 this standard in section 25.2. Note that conforming implementations
79 can further restrict the policies that can be bypassed using capabilities.
80 For example, if {POSIX_CAP} is defined, the effect of the
81 CAP_INF_RELABEL_OBJ capability may be limited to a range of information
82 labels, where such a range is implementation defined.

83 **Additional Implementation-Defined Floating**

84 It is understood that a conforming implementation may cause the floating
85 described above through the automatic application of the *inf_float()*
86 operation to occur at other times in addition to those covered by the general
87 policy. Additionally it may cause other changes (including “downward”
88 adjustments) of information labels under implementation-defined
89 circumstances.

90 **27.1.2.1 FI: File Function Policies**

91 Information labeling for files results from the application of basic policies (**FI.***) to
92 the file data object. The straightforward application of these rules to the object
93 model determines the specific information label restrictions for a large number of
94 file-related interfaces.

95 The object that encompasses a POSIX.1 file is defined to consist of a data portion,
96 and an attribute portion that contains the POSIX-defined attributes including the
97 information label. For the purposes of information labeling, the information label
98 of a file applies only to the data portion of the file.

99 The following policy rules apply:

100 **FI.1:** When a process with information label *inf_p1* writes data to a file with
101 information label *inf_p2*, the information label of the file shall automatically
102 be set to the value returned by *inf_float(inf_p1, inf_p2)*.

103 **FI.2:** The information label of a newly created file object shall automatically be
104 set to a value that dominates the value returned by *inf_default()*.

105 A conforming implementation may modify these policy rules for certain objects.
106 For example, some objects may be designated “non-floating.” The information
107 label of these objects will not change on process writes. Other objects may support
108 additional or finer-grained labeling which will modify the application of **FI.1** (as
109 well as **PI.1** below.) Precisely which objects are subject to modified rules is
110 implementation-defined.

111 **27.1.2.1.1 POSIX.1 Functions Covered by IL File Policies**

112 This policy is applied to the following POSIX.1 functions:

113 **Table 27-1 – POSIX.1 Functions Covered by Information Label File Policies**

114	Existing 115 Function	POSIX.1 Section
117	creat	5.3.2
118	mkfifo	5.4.2
119	open	5.3.1
120	pipe	6.1.1
121	write	6.4.2
122	New 124 Function	POSIX.1e Synopsis
125	aud_write	Write an Audit Record
126	inf_get_fd	Get the Information Label of a File Identified by File Descriptor
127	inf_get_file	Get the Information Label of a File Identified by Pathname
128	inf_set_fd	Set the Information Label of a File Identified by File Descriptor
129	inf_set_file	Set the Information Label of a File Identified by Pathname

130 **27.1.2.2 PI: Process Function Policies**

131 Information labeling for processes stems from the application of the basic information label policy to the few affected POSIX.1 functions.

133 When treated as an object, the process shall consist of its internal data (including the environment data), its executable image, and its status information.

135 The following policy rules apply:

136 **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)*.

139 **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)*.

142 **PI.3:** A newly created process shall be assigned the information label of the creating subject (process).

144 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.

148 **27.1.2.2.1 POSIX.1 Functions Covered by IL Process Policies**

149 This policy is applied to the following POSIX.1 functions:

150 **Table 27-2 – POSIX.1 Functions Covered by Information Label Process Policies**

151 152	Existing Function	POSIX.1 Section
154	execl	3.1.2
155	execv	3.1.2
156	execle	3.1.2
157	execve	3.1.2
158	execlp	3.1.2
159	execvp	3.1.2
160	fork	3.1.1
161	read	6.4.1
162 163	New Function	POSIX.1e Synopsis
165	aud_read	Read an Audit Record
166	inf_get_proc	Get the Process Information Label
167	inf_set_proc	Set the Process Information Label

168 **27.2 Header**

169 Some of the data types used by the information label functions are not defined as
 170 part of this standard, but shall be implementation-defined. If {POSIX_INF} is
 171 defined, these types shall be defined in the header <sys/inf.h>, which contains
 172 definitions for at least the following type.

173 **27.2.1 inf_t**

174 This type defines a pointer to an “exportable” object containing an information
 175 label. The object is opaque, persistent, and self-contained. Thus, the object can be
 176 copied by duplicating the bytes without knowledge of any underlying structure.

177 **27.3 Functions**

178 The functions in this section comprise the set of services that permit a process to
 179 get, set, and manipulate information labels. Support for the information label
 180 facility functions described in this section is optional. If the symbol
 181 {_POSIX_INF} is defined, the implementation supports the information label
 182 option and all of the information label functions shall be implemented as
 183 described in this section. If {_POSIX_INF} is not defined, the result of calling any
 184 of these functions is unspecified.

185 The error [ENOTSUP] shall be returned in those cases where the system supports
186 the information label facility but the particular information label operation can-
187 not be applied because of restrictions imposed by the implementation. —

188 **27.3.1 Initial Information Label**

189 Function: *inf_default()*

190 **27.3.1.1 Synopsis**

191 `#include <sys/inf.h>`
192 `inf_t inf_default (void)`

193 **27.3.1.2 Description**

194 The *inf_default()* function returns a pointer to an information label with an initial+
195 information label value that a conforming application may associate with newly-
196 created or fully truncated objects.

197 The system may allocate space for the information label to be returned. The
198 caller should free any releasable memory when the new label is no longer
199 required by calling *inf_free()* with the *inf_t* as an argument. In the event an error|
200 occurs, no memory shall be allocated and (*inf_t*)**NULL** shall be returned.

201 The precise method by which this label is determined is implementation-defined
202 and therefore may vary arbitrarily (e.g., based on process ID). As a result, the ini-
203 tial information label may not be the same on all newly created objects. However,
204 this label is guaranteed to be a valid label which, if applied to a newly-created
205 object, will be consistent with the implementation's information label policy.

206 **27.3.1.3 Returns**

207 The function *inf_default()* returns a pointer to the initial information label unless|
208 one of the errors below occurs, in which case no space is allocated, a value of|
209 (*inf_t*)**NULL** is returned, and *errno* is set to indicate the error.

210 **27.3.1.4 Errors**

211 If any of the following conditions occur, the *inf_default()* function shall return a
212 value of (*inf_t*)**NULL** and set *errno* to the corresponding value:

213 **[ENOMEM]** The label to be returned required more memory than was
214 allowed by the hardware or by system-imposed memory manage-
215 ment constraints. —

216 **27.3.1.5 Cross-References**

217 *inf_free()*, 27.3.5; *inf_set_fd()*, 27.3.10; *inf_set_file()*, 27.3.11.

218 **27.3.2 Test Information Labels For Dominance**

219 Function: *inf_dominate()*

220 **27.3.2.1 Synopsis**

221 `#include <sys/inf.h>`
222 `int inf_dominate (inf_t labela, inf_t labelb);`

223 **27.3.2.2 Description**

224 The *inf_dominate()* function determines whether *labela* dominates *labelb*. The
225 precise method for determining dominance is implementation-defined. Domi-
226 nance includes equivalence. Hence, if one label is equivalent to another, then
227 each shall dominate the other. Note that it is possible for neither of two labels to
228 dominate the other.

229 **27.3.2.3 Returns**

230 The function *inf_dominate()* returns 1 if *labela* dominates *labelb*. A value of 0 is
231 returned if *labela* does not dominate *labelb*. Otherwise, a result of -1 is returned,
232 and *errno* is set to indicate the error.

233 **27.3.2.4 Errors**

234 If any of the following conditions occur, the *inf_dominate()* function shall return
235 -1 and set *errno* to the corresponding value:

236 [EINVAL] One or both of the labels is not a valid information label as
237 defined by *inf_valid()*.

238 **27.3.2.5 Cross-References**

239 *inf_equal()*, 27.3.3; *inf_valid()*, 27.3.15.

240 **27.3.3 Test Information Labels For Equivalence**

241 Function: *inf_equal()*

242 **27.3.3.1 Synopsis**

243 `#include <sys/inf.h>`
244 `int inf_equal (inf_t labela, inf_t labelb);`

245 **27.3.3.2 Description**

246 The *inf_equal()* function determines whether *labela* is equivalent to *labelb*. The
247 precise method for determining equivalence is implementation-defined.

248 This function is provided to allow conforming applications to test for equivalence
249 since a comparison of the labels themselves may yield an indeterminate result.

250 **27.3.3.3 Returns**

251 The function *inf_equal()* returns 1 if *labela* is equivalent to *labelb*. A value of 0 is
252 returned if *labela* not equivalent to *labelb*. Otherwise, a value of -1 is returned,
253 and *errno* is set to indicate the error.

254 **27.3.3.4 Errors**

255 If any of the following conditions occur, the *inf_equal()* function shall return -1
256 and set *errno* to the corresponding value:

257 [EINVAL] One or both of the labels is not a valid information label as
258 defined by *inf_valid()*.

259 **27.3.3.5 Cross-References**

260 *inf_dominate()*, 27.3.2; *inf_valid()*, 27.3.15.

261 **27.3.4 Floating Information Labels**

262 Function: *inf_float()*

263 **27.3.4.1 Synopsis**

264 `#include <sys/inf.h>`
265 `inf_t inf_float (inf_t labela, inf_t labelb);`

266 **27.3.4.2 Description**

267 The *inf_float()* function returns a pointer to an information label that represents a+
268 combination of *labela* and *labelb* in a manner dependent on the implementation-
269 defined floating policy.

270 The system may allocate space for the information label to be returned. The
271 caller should free any releasable memory when the new label is no longer
272 required by calling *inf_free()* with the returned *inf_t* as an argument. In the +

273 event an error occurs, no memory shall be allocated and `(inf_t)NULL` shall be |
274 returned.

275 Note, that the notion of floating presupposes the introduction of data with one
276 label into a separately labeled subject or object. The `labela` argument represents
277 the information label of the data being introduced, the argument `labelb`
278 represents the subject's or object's current information label.

279 **27.3.4.3 Returns**

280 Upon successful completion, this function returns a pointer to the new informa- -
281 tion label. Otherwise, no space is allocated, a value of `(inf_t)NULL` is returned, |
282 and `errno` is set to indicate the error.

283 **27.3.4.4 Errors**

284 If any of the following conditions occur, the `inf_float()` function shall return a
285 value of `(inf_t)NULL` and set `errno` to the corresponding value: |

286 [EINVAL] One or both of the labels is not a valid information label as
287 defined by `inv_valid()`

288 [ENOMEM] The label to be returned required more memory than was
289 allowed by the hardware or by system-imposed memory manage-
290 ment constraints. -

291 **27.3.4.5 Cross-References**

292 `inf_free()`, 27.3.5; `inf_valid()`, 27.3.15.

293 **27.3.5 Free Allocated Information Label Memory**

294 Function: `inf_free()`

295 **27.3.5.1 Synopsis**

296 `#include <sys/inf.h>`
297 `int inf_free (void *buf_p);` |

298 **27.3.5.2 Description**

299 The `inf_free()` function frees any releasable memory currently allocated to the
300 buffer identified by `buf_p`. The `buf_p` argument may be either a `(void*)inf_t`, or a |
301 `(void*)char*` allocated by the `inf_to_text()` function.

302 **27.3.5.3 Returns**

303 Upon successful completion, the function *inf_free()* returns a value of 0. Other- |
304 wise, a value of -1 is returned and *errno* is set to indicate the error.

305 **27.3.5.4 Errors**

306 This standard does not specify any error conditions that are required to be |
307 detected for the *inf_free()* function. Some errors may be detected under conditions|
308 that are unspecified by this part of the standard.

309 **27.3.5.5 Cross-References**

310 *inf_default()*, 27.3.1; *inf_float()*, 27.3.4; *inf_get_fd()*, 27.3.7; *inf_get_file()*, 27.3.8;
311 *inf_get_proc()*, 27.3.9; *inf_from_text()*, 27.3.6; *inf_to_text()*, 27.3.14.

312 **27.3.6 Convert Text Label to Internal Representation**

313 Function: *inf_from_text()*

314 **27.3.6.1 Synopsis**

315 #include <sys/inf.h>
316 inf_t inf_from_text (const char *text_p); |

317 **27.3.6.2 Description**

318 The *inf_from_text()* function converts the text representation of an information
319 label, *text_p* into its internal representation, and returns a pointer to a copy of the
320 internal representation.

321 The system may allocate space for the information label to be returned. The
322 caller should free any releasable memory when the new label is no longer
323 required by calling *inf_free()* with the *inf_t* as an argument. In the event an error
324 occurs, no memory shall be allocated and (*inf_t*)NULL shall be returned. |

325 **27.3.6.3 Returns**

326 Upon successful completion, this function returns a pointer to the information |
327 label. Otherwise, no space is allocated, a value of (*inf_t*)NULL is returned, and |
328 *errno* is set to indicate the error.

329 **27.3.6.4 Errors**

330 If any of the following conditions occur, the *inf_from_text()* function shall return a |
331 value of (*inf_t*)NULL and set *errno* to the corresponding value:

332 [EINVAL] *text_p* is not a valid textual representation of an information
333 label as defined by *inf_valid()*.
334 [ENOMEM] The label to be returned required more memory than was
335 allowed by the hardware or by system-imposed memory manage-
336 ment constraints. —

337 **27.3.6.5 Cross-References**

338 *inf_free()*, 27.3.5; *inf_to_text()*, 27.3.14; *inf_valid()*, 27.3.15.

339 **27.3.7 Get the Information Label of a File Identified by File Descriptor**

340 Function: *inf_get_fd()*

341 **27.3.7.1 Synopsis**

342 #include <sys/inf.h>
343 inf_t inf_get_fd (int *fildes*);

344 **27.3.7.2 Description**

345 The *inf_get_fd()* function returns the information label associated with a file. The
346 function accepts a valid file descriptor and returns a pointer to the information
347 label of the file referenced by the descriptor.

348 The system may allocate space for the information label to be returned. The
349 caller should free any releasable memory when the new label is no longer
350 required by calling *inf_free()* with the *inf_t* as an argument. In the event an error
351 occurs, no memory shall be allocated and (*inf_t*)NULL shall be returned.

352 A process can get the information label of any file for which the process has a
353 valid file descriptor. If {_POSIX_MAC} is defined, the process must also have
354 MAC read access to the file.

355 **27.3.7.3 Returns**

356 Upon successful completion, this function returns the information label. Other-
357 wise, no space is allocated, a value of (*inf_t*)NULL is returned, and *errno* is set to |
358 indicate the error.

359 **27.3.7.4 Errors**

360 If any of the following conditions occur, the *inf_get_fd()* function shall return a
361 value of (*inf_t*)NULL and set *errno* to the corresponding value:

362 [EACCES] The required access to the file referred to by *fildes* was denied.

363 [EBADF] The *fildes* argument is not a valid file descriptor.
364 [ENOMEM] The label to be returned required more memory than was
365 allowed by the hardware or by system-imposed memory manage-
366 ment constraints.

367 **27.3.7.5 Cross-References**

368 *inf_free()*, 27.3.5; *inf_get_file()*, 27.3.8; *inf_set_fd()*, 27.3.10.

369 **27.3.8 Get the Information Label of a File Identified by Pathname**

370 Function: *inf_get_file()*

371 **27.3.8.1 Synopsis**

372 `#include <sys/inf.h>`
373 `inf_t inf_get_file (const char *path_p);`

374 **27.3.8.2 Description**

375 The *inf_get_file()* function returns the information label associated with a file.
376 The function accepts a pathname to indicate the file. The function returns a
377 pointer to the information label of the pathname pointed to by *path_p*.
378 The system may allocate space for the information label to be returned. The
379 caller should free any releasable memory when the new label is no longer
380 required by calling *inf_free()* with the *inf_t* as an argument. In the event an error
381 occurs, no memory shall be allocated and (*inf_t*)NULL shall be returned.

382 A process can get the information label of any file for which the process has
383 search access to the path specified. If {_POSIX_MAC} is defined, the process must
384 also have MAC read access to the file.

385 **27.3.8.3 Returns**

386 Upon successful completion, this function returns the information label. Other-
387 wise, no space is allocated, a value of (*inf_t*)NULL is returned, and *errno* is set to |
388 indicate the error.

389 **27.3.8.4 Errors**

390 If any of the following conditions occur, the *inf_get_file()* function shall return a
391 value of (*inf_t*)NULL and set *errno* to the corresponding value:

392 [EACCES] Search permission is denied for a component of the path prefix
393 or the required access to *path_p* is denied.

394 [ENAMETOOLONG]
395 The length of the pathname exceeds {PATH_MAX}, or a

396 pathname component is longer than {NAME_MAX} while
397 {POSIX_NO_TRUNC} is in effect.
398 [ENOENT] The named file does not exist or the *path_p* argument points to
399 an empty string.
400 [ENOMEM] The label to be returned required more memory than was
401 allowed by the hardware or by system-imposed memory manage-
402 ment constraints. —
403 [ENOTDIR] A component of the path prefix is not a directory.

404 **27.3.8.5 Cross-References**

405 *inf_free()*, 27.3.5; *inf_get_fd()*, 27.3.7; *inf_set_file()*, 27.3.11.

406 **27.3.9 Get the Process Information Label**

407 Function: *inf_get_proc()*

408 **27.3.9.1 Synopsis**

409 #include <sys/inf.h>
410 inf_t inf_get_proc (void);

411 **27.3.9.2 Description**

412 The *inf_get_proc()* function returns a pointer to the information label associated
413 with the requesting process.

414 The system may allocate space for the information label to be returned. The
415 caller should free any releasable memory when the new label is no longer
416 required by calling *inf_free()* with the *inf_t* as an argument. In the event an error
417 occurs, no memory shall be allocated and (*inf_t*)NULL shall be returned. —

418 **27.3.9.3 Returns**

419 Upon successful completion, this function returns the information label. Other-
420 wise, no space is allocated, a value of (*inf_t*)NULL is returned, and *errno* is set to
421 indicate the error.

422 **27.3.9.4 Errors**

423 If any of the following conditions occur, the *inf_get_proc()* function shall return a
424 value of (*inf_t*)NULL and set *errno* to the corresponding value:

425 [ENOMEM] The label to be returned required more memory than was
426 allowed by the hardware or by system-imposed memory manage-
427 ment constraints. —

428 **27.3.9.5 Cross-References**

429 *inf_free()*, 27.3.5; *inf_set_proc()*, 27.3.12.

430 **27.3.10 Set the Information Label of a File Identified by File Descriptor**

431 Function: *inf_set_fd()*

432 **27.3.10.1 Synopsis**

433 `#include <sys/inf.h>`
434 `int inf_set_fd (int fildes, inf_t label);`

435 **27.3.10.2 Description**

436 The *inf_set_fd()* function sets (writes) the information label of a file. The new
437 information label is *label*. The function accepts a valid file descriptor to indicate
438 the file.

439 A process can set the information label for a file using this function only if the
440 process has a valid file descriptor for the file. If `{_POSIX_MAC}` is defined, the
441 process must have mandatory write access to the file. Use of this function may
442 also require appropriate privilege. If `{_POSIX_CAP}` is defined, and the effective
443 user ID of the process is not equal to the file owner, appropriate privilege includes
444 the `CAP_FOWNER` capability. In addition, if *label* is not equivalent to the infor-
445 mation label associated with the file referred to by *fildes*, appropriate privilege
446 includes the `CAP_INF_RELABEL_OBJ` capability.

447 **27.3.10.3 Returns**

448 Upon successful completion, this function returns a value of 0. Otherwise, a value
449 of -1 is returned and *errno* is set to indicate the error.

450 **27.3.10.4 Errors**

451 If any of the following conditions occur, the *inf_set_fd()* function shall return -1
452 and set *errno* to the corresponding value:

- 453 [**EACCES**] The required access to the file referred to by *fildes* is denied.
- 454 [**EBADF**] The *fildes* argument is not a valid file descriptor.
- 455 [**EINVAL**] The label in *label* is not a valid information label as defined by
456 *inf_valid()*.
- 457 [**ENOTSUP**] *pathconf()* indicates that `{_POSIX_INF_PRESENT}` is not in
458 effect for the file referenced.
- 459 [**EPERM**] The process does not have appropriate privilege to perform this

460 operation.

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

463 27.3.10.5 Cross-References

⁴⁶⁴ *inf_get_fd()*, 27.3.7; *inf_set_file()*, 27.3.11; *inf_valid()*, 27.3.15.

465 27.3.11 Set the Information Label of a File Identified by Pathname

466 Function: *inf_set_file()*

467 27.3.11.1 Synopsis

```
468 #include <sys/inf.h>
```

```
469 int inf_set_file (const char *path_p, inf_t label);
```

470 27.3.11.2 Description

471 The *inf_set_file()* function sets (writes) the information label of a file. The new
472 information label is *label*. The function accepts a pathname to indicate the file.

473 A process can set the information label for a file only if the process has search
474 access to the path specified. If `{_POSIX_MAC}` is defined, the process must have
475 mandatory write access to the file. Use of this function may also require appropri-
476 ate privilege. If `{_POSIX_CAP}` is defined, and the effective user ID of the process
477 is not equal to the file owner, then appropriate privilege includes the
478 `CAP_FOWNER` capability. In addition, if `label` is not equivalent to the informa-
479 tion label associated with the file referred to by `path_p`, appropriate privilege
480 includes the `CAP_INF_RELABEL_OBJ` capability.

481 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.

484 27.3.11.4 Errors

485 If any of the following conditions occur, the *inf_set_file()* function shall return -1
486 and set *errno* to the corresponding value:

487 [EACCES] Search permission is denied for a component of the path prefix
488 or the required access to *path* *p* is denied.

489 [EINVAL] The label in *label* is not a valid information label as defined by
490 *inf_valid()*.

491 [ENAMETOOLONG]

492 The length of the pathname exceeded {PATH_MAX}, or a

493 pathname component is longer than {NAME_MAX} while
494 {POSIX_NO_TRUNC} is in effect.
495 [ENOENT] The named file does not exist or the *path_p* argument points to
496 an empty string. -
497 [ENOTDIR] A component of the path prefix is not a directory.
498 [ENOTSUP] *pathconf()* indicates that {_POSIX_INF_PRESENT} is not in
499 effect for *path_p*.
500 [EPERM] The process does not have appropriate privilege to perform this
501 operation.
502 [EROFS] This function requires modification of a file system which is
503 currently read only.

504 **27.3.11.5 Cross-References**

505 *inf_get_file()*, 27.3.8; *inf_set_fd()*, 27.3.10; *inf_valid()*, 27.3.15.

506 **27.3.12 Set the Process Information Label**

507 Function: *inf_set_proc()*

508 **27.3.12.1 Synopsis**

509 #include <sys/inf.h>
510 int inf_set_proc (inf_t *label*);

511 **27.3.12.2 Description**

512 The *inf_set_proc()* function sets (writes) the information label of the requesting
513 process. The new information label is *label*. If *label* is not equivalent to the infor-
514 mation label associated with the process, then appropriate privilege is required
515 for this operation. If {_POSIX_CAP} is defined, appropriate privilege includes the
516 CAP_INF_RELABEL_SUBJ capability.

517 **27.3.12.3 Returns**

518 Upon successful completion, *inf_set_proc()* returns a value of 0. Otherwise, a
519 value of -1 is returned and *errno* is set to indicate the error.

520 **27.3.12.4 Errors**

521 If any of the following conditions occur, the *inf_set_proc()* function shall return -1
522 and set *errno* to the corresponding value:

523 [EINVAL] The label in *label* is not a valid information label as defined by
524 *inf_valid()*.
525 [EPERM] The process does not have appropriate privilege to perform this
526 operation.

527 **27.3.12.5 Cross-References**

528 *inf_get_proc()*, 27.3.9; *inf_valid()*, 27.3.15.

529 **27.3.13 Get the Size of an Information Label**

530 Function: *inf_size()*

531 **27.3.13.1 Synopsis**

532 #include <sys/inf.h>
533 ssize_t inf_size (inf_t *label*);

534 **27.3.13.2 Description**

535 The *inf_size()* function returns the size in bytes of the internal representation of
536 the information label in *label*, if it is valid.

537 **27.3.13.3 Returns**

538 Upon successful completion, the function returns the size of the information label.
539 Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

540 **27.3.13.4 Errors**

541 If any of the following conditions occur, the *inf_size()* function shall return -1 and
542 set *errno* to the corresponding value:

543 [EINVAL] The *label* argument is not a valid information label as defined by
544 *inf_valid()*.

545 **27.3.13.5 Cross-References**

546 *inf_free()*, 27.3.5; *inf_valid()*, 27.3.15.

547 **27.3.14 Convert Internal Label Representation to Text**

548 Function: *inf_to_text()*

549 **27.3.14.1 Synopsis**

```
550 #include <sys/inf.h>
551 char *inf_to_text (inf_t label, size_t *len_p);
```

552 **27.3.14.2 Description**

553 The *inf_to_text()* function converts the information label contained in *label* into a
554 human readable, **NULL**-terminated, character string which shall be suitable for
555 the *text_p* parameter to *inf_from_text()* in section 27.3.9 and for re-input as the
556 *inflabel* operand to the **setfinf** utility as defined in section 12 of POSIX.2c. This function
557 returns a pointer to the string. If the pointer *len_p* is not **NULL**, the function shall also return
558 the length of the string (not including the **NULL** terminator) in the location pointed to by
559 *len_p*. The information label in *label* shall be completely represented in the returned character
560 string.

561 The system may allocate space for the string to be returned. The caller should free any releasable
562 memory when the string is no longer required by calling *inf_free()* with the *char ** as an
563 argument. In the event an error occurs, no memory shall be allocated and (*inf_t*)**NULL** shall
564 be returned.

565 **27.3.14.3 Returns**

566 Upon successful completion, *inf_to_text()* returns a pointer to the text representation.
567 Otherwise, in all cases, the memory referred to by *len_p* shall remain
568 unchanged, a value of (*char **)**NULL** is returned, and *errno* is set to indicate the
569 error.

570 **27.3.14.4 Errors**

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

573 [EINVAL] The label in *label* is not a valid information label as defined by
574 *inf_valid()*.

575 [ENOMEM] The text to be returned required more memory than was allowed
576 by the hardware or by system-imposed memory management
577 constraints.

578 **27.3.14.5 Cross-References**

579 *inf_free()*, 27.3.5; *inf_from_text()*, 27.3.6; *inf_valid()*, 27.3.15; **setfinf**, 12.3.

580 **27.3.15 Information Label Validity**

581 Function: *inf_valid()*

582 **27.3.15.1 Synopsis**

583 `#include <sys/inf.h>`
584 `int inf_valid (inf_t label);`

585 **27.3.15.2 Description**

586 The *inf_valid()* function determines whether the label in *label* is a valid information label. The precise meaning of validity is implementation-defined. Examples
587 of some reasons why a label may be considered invalid include: the label is mal-
588 formed, the label contains components that are not currently defined on the sys-
589 tem, or the label is simply forbidden to be dealt with by the system.

591 **27.3.15.3 Returns**

592 Upon successful completion, the function returns 1 if *label* is valid, and 0 if it is
593 invalid. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

594 **27.3.15.4 Errors**

595 This standard does not specify any error conditions that are required to be
596 detected for the *inf_valid()* function. Some errors may be detected under condi-
597 tions that are unspecified by this part of the standard.

598 **27.3.15.5 Cross-References**

599 None.

Annex B

(informative)

Revisions to Rationale and Notes

3 B.1 Revisions to Scope and Normative References

4 ⇒ **B.1.1 Scope** This rationale is to be revised and integrated appropriately into
5 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.

15 This interface is extendible to allow for innovations that provide greater (or
16 different) security functions in various markets. It is expected that conforming
17 implementations may augment the mechanisms defined in this standard and
18 may also provide security functions in areas not included in this standard.

19 It was not a goal of this document to address assurance requirements which
20 constrain the implementation and not the interface. POSIX.1 standards define
21 operating system interfaces only and attempt to allow for the greatest possible
22 latitude in implementation so as to promote greater acceptance of the stan-
23 dards.

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.

WITHDRAWN DRAFT. All Rights Reserved by IEEE.
Preliminary—Subject to Revision.

34 Even though the TCSEC was a source of requirements for the interfaces
35 defined in this standard, this standard is not to be construed as defining a set
36 of interfaces intended to satisfy the requirements of any particular level.

37 ⇒ **B.1.3.6 Supported Security Mechanisms (POSIX.1: line 474)** *Add the fol-*
38 *lowing new section:*

39 **B.1.3.6 Supported Security Mechanisms**

40 The security mechanisms supported by this standard were chosen for their gen-
41 erality. The specific interfaces defined were selected because they were perceived
42 to be generally useful to applications (trusted and untrusted). Two mechanisms,
43 access control lists and privilege, are defined specifically to address areas in the
44 POSIX.1 standard that were deferred to this standard.

45 ⇒ **B.1.3.7 Unsupported Security Mechanisms** *Add the following new sections*
46 *B.1.3.7 - B.1.3.7.11:*

47 **B.1.3.7 Unsupported Security Mechanisms**

48 The purpose of this standard is to provide for application portability between con-
49 forming systems. As a result, this standard does not address several functional
50 security-related issues. Specifically, the POSIX.1e standard does not address:

- 51 (1) Identification and Authentication
- 52 (2) Networking Services and Protocols
- 53 (3) Administrative Services and Management of Security Information
- 54 (4) Covert Channels
- 55 (5) Assurance Issues
- 56 (6) Evaluation Ratings Based on Current Trust Criteria
- 57 (7) The General Terminal Interface as described in the POSIX.1 standard

58 The rationale for excluding these and other potentially relevant topics is provided
59 below.

60 **B.1.3.7.1 Identification and Authentication**

61 I&A mechanisms are being deferred to a future version of this standard. It was
62 felt that the I&A mechanism should take into consideration third-party authenti-
63 cation schemes. It was also felt that deferring this area to a future standard
64 would allow existing practice to become more stabilized prior to standardization.

65 **B.1.3.7.2 Networking Services**

66 Networking services are being deferred to a future version of this standard. This
67 was done to allow the various POSIX Distributed Services working groups to
68 further progress their work prior to standardization. It was also felt that defer-
69 ring this area to a future standard would allow existing practice to become more
70 stabilized prior to standardization.

71 **B.1.3.7.3 Administrative Services and Management of Security Informa-
72 tion**

73 Administrative services and the management of security information are being
74 deferred to a future version of this standard. This was done to allow the POSIX
75 System Administration working group to further progress their work prior to
76 standardization. System administration will ultimately be standardized through
77 a document that is distinct from the POSIX.1 or POSIX.2 standards. The current
78 POSIX.1e work is limited to modifications to the POSIX.1 and POSIX.2 stan-
79 dards.

80 **B.1.3.7.4 Covert Channels**

81 Covert channel analysis is undertaken from the perspective of the interface, and
82 not the underlying implementation. This means that covert channels associated
83 with resource exhaustion, e.g., process IDs, i-nodes, and file descriptors, are not
84 considered. Covert channels visible at the interface are treated. These include the
85 use of exclusive locks and the updating of file access times.

86 **B.1.3.7.5 Assurance Issues**

87 Assurance issues that do not require function or utility interfaces are not expli-
88 citly treated as part of the standard. But assurance requirements that constrain
89 the system interfaces are implicitly part of the standard. The principal issues
90 here are:

91 **B.1.3.7.5.1 Modularity, Security Kernels, Software Engineering**

92 These are mostly kernel internals design and implementation issues, which are
93 beyond the scope of POSIX standards.

94 **B.1.3.7.5.2 Minimality**

95 The TCB minimality assurance requirement is not addressed by this standard.
96 This is an implementation question only.

97 The minimality requirement, introduced at the B3 level of the TCSEC, does not
98 constrain the definition of any POSIX.1e interface, because minimality pertains
99 only to the definition of the partition between the trusted code of the system,i.e.,
100 the TCB, and the untrusted code of the system. This standard does not specify
101 that the interfaces it defines must be TCB interfaces.

102 **B.1.3.7.5.3 System Integrity**

103 System Integrity interfaces are being deferred to a future version of this standard.
104 It was felt that deferring this area to a future standard would allow existing
105 practice to become more stabilized prior to standardization.

106 **B.1.3.7.5.4 Formal Security Policy Model**

107 No security policy models are defined as part of this standard because the stan-
108 dard is not intended to define a complete system. In some areas the implementa-
109 tion may want to extend the standard, and in other areas the implementation will
110 have to extend the standard. Given this incompleteness, a model would be
111 difficult (and perhaps impossible) to define. Also, a full, formal model would con-
112 strain implementations beyond the point necessary for application portability.

113 **B.1.3.7.5.5 Separation of Administrative Roles**

114 Without a complete definition of administrative function, this is clearly beyond
115 the scope of this standard. Also, this is an area where implementations may wish
116 to target particular and isolated installations.

117 **B.1.3.7.5.6 Resource Controls**

118 Resource controls (quotas) are used to support a system availability policy. They
119 are not included in this standard because of a lack of existing practice in UNIX
120 systems and, more importantly, the resources controlled tend to reflect implemen-
121 tation limits (static tables, ...) rather than physical ones.

122 **B.1.3.7.5.7 Trusted Path**

123 A Trusted Path mechanism is not defined because the notion of terminal defined
124 in POSIX.1 is limited to dumb ttys, and is incomplete as well. Existing practice is
125 lacking here as well. The standardization of the key sequence used for invoking
126 the trusted path is possible, but it would also be necessary to define the behavior
127 of the system upon trusted path invocation. It was felt that this would be impossi-
128 ble without a well-defined Trusted Path.

129 **B.1.3.7.5.8 Protected Subsystems**

130 The UNIX-protected subsystem mechanism (programs with the set-user-ID or
131 set-group-ID mode bits set) is subject to abuse by knowledgeable users and
132 misuse by naive users. Its shortcomings are not addressed due to some notable
133 disagreements concerning the desirability of the mechanism. It also doesn't add
134 much to portability.

135 **B.1.3.7.6 Evaluation Ratings Based on Current Trust Criteria**

136 Evaluations of products under current trust criteria involve analysis of all aspects
137 of the product, especially of implementation details. This standard only deals
138 with interfaces. Therefore, it is inherently incomplete and unsuited for evalua-
139 tion under these criteria. In addition, a conforming system could implement the

140 functionality under the interfaces in an insecure manner. Therefore, conformance
141 to this standard does not guarantee that a system should be trusted. |

142 **B.1.3.7.7 General Terminal Interface**

143 This standard does not extend General Terminal Interfaces described in sections
144 7.1 and 7.2. This section explains some of the problems with the GTI from a secu-
145 rity perspective.

146 The existing interfaces do not require that the file descriptor used for changing
147 terminal attributes be opened for writing. Given the MAC policy of read-down, a
148 process could open a terminal which it dominates, and by manipulating terminal
149 attributes perform data downgrade. This violates the basic MAC policies.
150 Requiring that the device is opened for write (or that the process have MAC write
151 access) solves this problem.

152 Manipulation of device attributes can interfere with invocation of trusted path.
153 For example, a process could change the baud rate of its controlling terminal.
154 The trusted path would be unable to determine if the baud rate was changed at
155 the user's request, i.e., because the baud rate was adjusted on the physical termi-
156 nal, or by a malicious or malfunctioning application. Thus, the user might be
157 unable to communicate via the trusted path. Changing the baud rate should be
158 restricted using privilege or trusted path.

159 Applications may cause output to be suspended (using the *tcflow()* function with
160 the action set to TCOOFF). If the trusted path is invoked in such a case, the
161 standard would need to define what happens, i.e., the trusted path can re-enable
162 output, but the status of queued output would need to be determined. An
163 appropriate solution to this problem is not clear.

164 While these problems generally involve trusted path (which is not a part of the
165 standard), it is important not to enact a standard which would preclude building
166 a system that includes a trusted path mechanism.

167 => **B.1.3.8 Portable Trusted Applications** *Add the following new sections*
168 *B.1.3.8:*

169 **B.1.3.8 Portable Trusted Applications**

170 Portable trusted applications are those applications that are: portable because the
171 system call interface they use is that defined by POSIX.1e; and trusted, because
172 they perform some security-related functionality and/or need some privilege from
173 the system in order to function correctly, and which therefore must be trusted to
174 perform the security-related functionality correctly and/or to not abuse the
175 privilege granted to the application.

176 Such portable trusted applications may rely on the TCB of the host system to per-
177 form certain security-critical functions that are necessary to ensure the correct
178 and secure operation of the portable trusted application. For example: a portable

179 trusted application may need to protect some persistent data from tampering by
180 unauthorized processes, and may therefore use DAC features to control access to
181 the persistent data as stored in a file.

182 If the secure operation of the portable trusted application depends on the correct
183 operation of such POSIX.1e functions, then those POSIX.1e functions must be
184 implemented by the TCB of the host system on which the application is running;
185 otherwise, the portable trusted application would be relying on untrusted code to
186 perform functionality upon which the security of the portable trusted application
187 depends.

188 Furthermore, the secure state of the entire system may be at stake if the portable
189 trusted application runs with system privileges, because the portable trusted
190 application may operate incorrectly and abuse its privilege as a result of malfunc-
191 tion of untrusted code performing functionality which is security-related as used
192 by the portable trusted application. However, the interfaces defined in this stan- |
193 dard are not required to be TCB interfaces.

194 As a result, a portable trusted application may be portable to various POSIX.1e- |
195 conformant systems, but only some of those conformant systems may actually
196 implement as TCB interfaces those POSIX.1e interface functions upon which
197 depends the secure operation of the portable trusted application. Therefore, port-
198 able trusted applications under some circumstances may not be trust-worthy even
199 when run on conformant systems. Proper use of portable trusted applications
200 depends on the specification of the system interfaces which are security-critical to
201 the portable trusted application, and the determination of whether all those inter-
202 faces are implemented by the TCB of a system which can run the portable trusted
203 application.

204 **B.2 Revisions to Definitions and General Requirements**

205 ⇒ **B.2.2.2 General Terms** *Insert the following after line 986:*

206 **user:** the term user is used in this document to denote a person who interacts
207 with a computer system. It is not meant to include programs that “look like”
208 users.

209 ⇒ **B.2.10 Security Interface (POSIX.1: line 1741)** *Add the following sections*
210 *B.2.10 and B.2.10.1:*

211 **B.2.10 Security Interface**

212 **B.2.10.1 Opaque Data Objects**

213 Each functional area (MAC, ACL, IL, capabilities, and audit) defines one or more
214 opaque data objects. Certain restrictions are applied to some of those opaque data
215 objects, namely persistence and self-containment. This section describes the
216 rationale for these requirements and their implications.

217 Opaque data objects by definition can contain any type of data, in any form, so
218 long as the functions which manipulate those objects understand that form. For
219 example, Access Control Lists are frequently implemented as linked lists. How-
220 ever, some applications need to pass opaque objects to other processes (e.g., by
221 writing them in FIFOs), or to store them in files. For example, a trusted database
222 system might store a MAC label for each record in the database. Truly opaque
223 data cannot be stored, because an application does not know how much to store,
224 and there is no guarantee that the data will be meaningful when retrieved from
225 the database.

226 In each section, an interface is provided to free memory associated with data
227 structures. (Thus, for example, there are *mac_free()*, *inf_free()*, etc., routines).
228 The description of these routines state that they free any "releasable" memory.
229 Once these routines have been called, the data structure freed can no longer be
230 used by applications: in general, these routines will deallocate all memory associ-
231 ated with the data structure. That is, the **_free()* routines generally work analog-
232 ously to the *malloc()* and *free()* routines of standard C. However, no require-
233 ment imposed by this standard that requires all allocated memory to be freed.
234 Conforming implementations, then, can use their own memory management
235 schemes. Nevertheless, portable applications must assume that the memory
236 freed has been completely deallocated and that any pointers to the freed data
237 structure are no longer valid.

238 **B.3 Revisions to Process Primitives**

239 ⇒ **B.3.1.2 Process Creation (POSIX.1: 1770)** *Rationale for changes to this sec-
240 tion in POSIX.1 is provided below:*

241 When a new process is created via a *fork()* call, the new process is an exact
242 copy of its parent, including the current MAC label, information label, etc.
243 Because this standard does not define the contents of many data structures, it
244 is important to note that both the parent and child may continue using data
245 structures independently.

246 For example, consider an implementation where a MAC label structure (that is
247 an object referenced by *mac_t*) is simply a number. That number could be an
248 index into a kernel table. Functions which use the MAC label could make ker-
249 nel calls, and all manipulation of the MAC label would take place in the

250 kernel. When the *fork()* function is executed, the system must duplicate the
251 kernel table so both the parent and child processes are able to modify the MAC
252 label without interfering with each other.

253 ⇒ **B.3.1.2 Execute a File (POSIX.1: line 1821)** *Rationale for changes to this*
254 *section in POSIX.1 is provided below:*

255 At first glance it might appear that a child's information label should be set
256 either to the information label of the file being executed, or to the lowest label
257 in the system. However, the process performing the *exec*()* operation can pass
258 information to the new process image by way of file descriptors and environ-
259 ment variables. Hence, the old process information label should be incor-
260 porated in the new process information label. Note that the standard recom-
261 mends an information label, but does not require it; other information label
262 policies are possible and allowed by this standard. Additionally, the standard
263 does not require use of the *inf_float()* function to calculate the new information
264 label; this is a suggestion of one way to perform the calculation.

265 ⇒ **B.3.3.2 Send a Signal to a Process (POSIX.1: line 2428)** *Rationale for*
266 *changes to this section in POSIX.1 is provided below:*

267 Using a signal between two processes is effectively sending data. While the
268 amount of data (the signal number) is small, this standard is careful to avoid
269 requiring information flow which contradicts the MAC security policy. Hence,
270 the four cases described in the standard:

271 MAC label of sender equivalent to MAC label of receiver: no MAC restric-
272 tions

273 MAC label of sender dominates MAC label of receiver (i.e., write-down):
274 appropriate privilege is required, and if *{_POSIX_CAP}* is defined,
275 appropriate privilege includes the capability *CAP_MAC_WRITE*.

276 MAC label of receiver dominates MAC label of sender (i.e., write-up):
277 appropriate privilege may or may not be required. A write-up is not
278 an inherent violation of the security policy, except that the sender is
279 able to determine the existence of a higher level process. Systems
280 which address covert channels may wish to close this channel by
281 requiring appropriate privilege. If *{_POSIX_CAP}* is defined,
282 appropriate privilege includes the capability *CAP_MAC_READ*
283 (because the existence of the higher level process is read).

284 MAC label of sender and receiver are incomparable: in this case, appropri-
285 ate privilege is certainly required at least as strong as the case where
286 the label of the sender dominates that of the receiver. If
287 *{_POSIX_CAP}* is defined, appropriate privilege includes the capabil-
288 ity *CAP_MAC_WRITE*. In addition, implementations may require
289 appropriate privilege to perform the read-up, viewing the operation as

290 a write-down followed by a read-up. In this case, if `{_POSIX_CAP}` is
291 defined, appropriate privilege includes the capability
292 `CAP_MAC_READ`. However, the additional capability is not defined
293 by the standard, since implementations are free to add additional res-
294 trictions as desired.

295 The `kill()` function allows notification of a process group. The error code is
296 defined in POSIX.1 as success if any signal was sent, and a failure only if no
297 processes could be signaled. This standard extends that notion: if a process
298 group contains processes with different MAC labels, then a signal is success-
299 fully sent to the process group if even a single process in the group can be sig-
300 naled. This is consistent with the notion in IEEE Std 1003.1-1990 where a sig-
301 nal could be successful even if processes in the process group have different
302 user IDs, and hence only some of them can be signaled.

303 If not even one process can be signaled, then there are two possible errors
304 returned: `[EPERM]` and `[ESRCH]`. `[EPERM]` is used when the sending process
305 dominates at least one of the potential receiving processes, but did not have
306 the required appropriate privilege to send the signal. In this case, the sending
307 process could determine the existence of the potential receiver, so no informa-
308 tion channel exists by returning `[EPERM]`. By contrast, `[ESRCH]` is returned
309 to indicate that either the process group did not exist, or none of the processes
310 in the process group were visible to the sending process.

311 While this standard imposes no information label requirements on signals,
312 implementations may consider the signal as having an information label, and
313 hence float the information label of the receiving process to include the infor-
314 mation label of the sending process.

315 This standard does not extend the notion of access control based on user IDs to
316 include the notion of an access control list on a process.

317 Another architecture not discussed by this standard is to allow overrides of the
318 signaling policy based on the privileges of the receiver. In such an architec-
319 ture, a daemon process could be set up to accept signals from any process,
320 regardless of the MAC label of the sender. However, the POSIX.1 standard
321 does not recognize this notion for user ID based privileges, so this standard
322 does not extend it for MAC.

323 ⇒ **B.4 Revisions to Process Environment (POSIX.1: line 2645)** *Rationale for*
324 *changes to this section in POSIX.1 is provided below:*

325 As previously described, each of the options described in this standard may be
326 selected independently. The `sysconf()` variables listed in this section are to
327 allow programs to determine at runtime whether the option is available.

328 ⇒ **B.5 Files and Directories (POSIX.1: line 2896)** *Rationale for changes to*
329 *this section in POSIX.1 is provided below:*

330 The extensions specified in this standard for file access avoid changing the
331 interfaces specified in POSIX.1 any more than necessary. Specifically, no
332 changes are made to parameter types, and where data structures are involved,
333 no changes are made to add or remove elements from the structure. In some
334 cases the data returned by the interface may be changed. This is most notice-
335 able when examining the file permission bits of a file which has an access con-
336 trol list.

337 ⇒ **B.5.3.1 Open a File (POSIX.1: line 3077)** *Rationale for changes to this sec-*
338 *tion in POSIX.1 is provided below:*

339 While it might appear that a newly created file would always have the infor-
340 mation label *inf_default()* this is not true. For example, implementations
341 might set the information label of a new file to the information label of the con-
342 taining directory or the information label of the creating process.

343 When opening a FIFO, the MAC restriction should be that process and FIFO
344 MAC labels should be equivalent to avoid massive covert channels associated
345 with MAC inequalities. Since the MAC policy defined by this standard allows
346 MAC write-up, it is possible to be POSIX compliant and still include this
347 covert channel. However, since the normal MAC policy is write-equals, this is
348 not a major concern.

349 ⇒ **B.5.6.2 Get File Status (POSIX.1: line 3208)** *Rationale for changes to this*
350 *section in POSIX.1 is provided below:*

351 The *stat()* call in POSIX.1 provides the caller with all file attributes. This
352 standard does not extend *stat()* to return the extended attributes such as MAC
353 label or access control list. There were several reasons:

354 This standard had as a goal to leave the syntax of existing interfaces
355 unchanged.

356 The data structures defined in this standard are potentially variable
357 length, unlike in POSIX.1 where they are all fixed length. Thus, the
358 *stat* structure would have to be adapted to handle pointers to the vari-
359 able length items. This would make the interface more complicated.

360 Each portion of this standard is independent, so not all data types are
361 necessarily defined. Thus, the *stat* structure would have to be set up
362 differently depending which options are provided.

363 Existing programs designed to use a version of *stat()* as defined in POSIX.1
364 might get back additional information. If the program had not been
365 recompiled to allow for a larger structure, this might overwrite other
366 data, and cause the program to fail.

367 Thus, the standard leaves *stat()* unchanged, and adds new functions for get-
368 ting the individual extended file attributes.

369 Note that if `{_POSIX_ACL}` is defined and `{_POSIX_ACL_EXTENDED}` is in
370 effect for the pathname, the semantics of *stat()* and *fstat()* are changed.
371 Specifically, *stat()* and *fstat()* no longer return all the discretionary access
372 information, so applications that depend on it doing so (e.g., when copying dis-
373 cretionary file attributes to another file) may have to be changed.

374 ⇒ **B.5.6.3 File Access (POSIX.1: line 3216)** *Rationale for changes to this sec-
375 tion in POSIX.1 is provided below:*

376 POSIX.1 does not list the specific permissions required for each function (e.g.,
377 *open()*, *mkdir()*). Rather, it relies on the descriptions of pathname resolution
378 and file access in POSIX.1, 2.3, together with additional information (e.g.,
379 error codes) in the individual function descriptions. For example, the descrip-
380 tion of *open()* does not specify that the caller must have search access to each
381 pathname component, and must also have write access to the directory if a
382 new file is being created. The pathname resolution portion is implicit from
383 POSIX.1, 2.3, and write access to the parent directory is provided by the
384 description of the EACCES error number.

385 In a similar fashion, this standard does not describe the MAC requirements for
386 file access, instead referring to POSIX.1, 2.3. Additional information is pro-
387 vided where appropriate, such as linking files (which requires MAC write per-
388 mission to the existing file) and opening a FIFO (which requires MAC write
389 permission to the FIFO file).

390 Unlinking a file might appear to need MAC write access to the containing
391 directory only. However, the unlink operation updates the link count on the
392 file, which is effectively a write operation to the file. Hence, MAC write access
393 is required. Similarly, removing a directory updates the directory link count,
394 and consequently MAC write access is required to the directory being removed.

395 **Clearing setuid/setgid and Privileges**

396 One security-relevant issue not addressed by this standard is resetting of the
397 setuid/setgid bits. For example, most historical implementations clear the
398 setuid and setgid bits when a file is written into. The security risk is that if a
399 setuid utility is improperly installed (e.g., with write permission) and the
400 setuid bit is not cleared, a malicious user could replace the utility with a dif-
401 ferent version. However, neither IEEE Std 1003.1-1990 nor this standard
402 require (nor prohibit) clearing the setuid and setgid bits.

403 There were several reasons for not specifying the behavior. The most impor-
404 tant was determining which interfaces should trigger clearing setuid/setgid
405 bits. Should they be cleared when the file is opened, when it is written to,
406 when it is closed, or some combination? Each leaves certain timing windows,
407 and has potential performance implications.

408 The capability flags provided by this standard provide an extension to the
409 notion of setuid/setgid, with somewhat finer granularity. If setuid/setgid bits
410 are to be cleared, should capability flags also be cleared? Just as this standard
411 makes no statements about setuid/setgid, it does not require (nor prohibit)
412 clearing of capability flags.

413 If capability flags are cleared when a file is written, the implementor should
414 also consider whether they should be cleared when file attributes are changed.
415 For example, consider a program file which has the MAC read-up exemption
416 capability, and the file has a MAC label of secret. When executed, that pro-
417 gram may read top secret data, but at worst it can relabel it as secret (because
418 only a user with at least a secret security level will be able to access the file,
419 and hence execute the program). If the file's MAC label is changed to
420 unclassified, then an uncleared user may be able to execute it, thus allowing
421 top secret data to be written into an unclassified file. Thus, the change in the
422 MAC label of a file impacted the system security, by allowing additional risks.
423 System implementors may wish to consider these types of threats, even though
424 they are not required by this standard.

425 Finally, system implementors should consider whether capability and
426 setuid/setgid bits should be cleared when the file owner is changed.

427 **Object Reuse and File Erasure**

428 Another topic of concern in trusted systems is object reuse, particularly as it
429 applies to files. POSIX.1 requires that newly allocated files be cleared, so the
430 previous contents of the file are inaccessible. While some historical systems
431 overwrite the contents of a file when the file is deleted, this standard imposes
432 no such requirement. Because the contents are cleared when the file is first
433 read, this is not an issue except when the device which stores the file (i.e., the
434 disk) can be accessed outside the file system (e.g., through a raw device). Such
435 concepts are beyond the scope of this standard.

436 **Initial Information Labels**

437 When a file (including a directory or FIFO) is created, the initial information
438 label on the file must be set. This standard does not specify an information
439 label policy. Hence, the standard does not specify what the initial label will
440 be. In most cases the initial label will be the same as the result of a call to
441 *inf_default()*.

442 ⇒ **B.6.1 Pipes (POSIX.1: line 3380)** *Rationale for changes to this section in
443 POSIX.1 is provided below:*

444 Pipes provide communication between related processes (typically a parent
445 and child). Excluding the effects of privileged processes, the related processes
446 by definition have the same MAC label. Hence, specifying the MAC label of
447 the pipe is somewhat irrelevant. However, processes can request the MAC
448 label of the file associated with a file descriptor. This standard defines the
449 MAC label of the pipe as the MAC label of the creating process so such a

450 request can be answered.

451 ⇒ **B.6.5.2 File Locking (POSIX.1: line 3613)**

452 The file locking mechanism defined in IEEE Std 1003.1-1990 allows advisory
453 locks to be placed and detected on a file. The mechanism does not specify the
454 file mode used by processes placing or testing the locks. When a MAC policy is
455 added, the locking mechanism can be used as an information flow channel. At
456 earlier stages of development of this standard strict requirements for MAC
457 access were specified and varying capabilities specified to obtain MAC access.
458 Due to significant ballot objections to the granularity of the capabilities
459 required, it was decided to let this standard be mute on the enforcement of
460 MAC for file locking operations. Implementations concerned with closing the
461 information flow channel have been left free to handle the channel in whatever
462 way they choose. See B.25.4.3 for more discussion of this issue. |

463 ⇒ **B.8 Language-Specific Services for the C Programming Language**
464 *Rationale for changes to this section in POSIX.1 is provided below:*

465 Historical implementations implement the interfaces defined in this section
466 using the base POSIX.1 interfaces. This concept is reflected by the description
467 of the interfaces as having *underlying functions*. However, there is no require-
468 ment that implementations use the underlying functions, as noted in POSIX.1
469 Section 8, lines 341-345. As a result, this standard defines the extensions to
470 the C standard I/O primitives.

471 Some consideration was given to defining security effects of making a
472 *longjmp()* call. For example, to provide time bounding of capabilities the
473 current capability set could be restored to its state as of the *setjmp()* call. This
474 standard makes no such requirements, as applications are not required to time
475 bound capabilities. Rather, applications developers are encouraged to clear
476 appropriate capabilities in the code invoked from the *longjmp()* call.

1 **B.23 Access Control Lists**

2 The overall requirements for an Access Control List (ACL) mechanism in a secure
3 system include the following:

- 4 (1) Allow authorized users to specify and control sharing of objects
- 5 (2) Supply discretionary access controls for objects.
- 6 (3) Specify discretionary access by a list of users and groups with their
7 respective access rights to the protected objects
- 8 (4) Allow discretionary access to an object to be denied for a user or, in cer-
9 tain cases, a group of users.

- 10 (5) Allow changes to the ACL only by the owner of the object or by a process
11 with the required access or appropriate privilege.
12 (6) Not allow more permissive discretionary access than either the initial or
13 final access rights while the ACL is being written by *acl_set_file()* or
14 *acl_set_fd()*.

15 The primary goal in defining access control lists in a POSIX.1e system is to pro-
16 vide a finer granularity of control in specifying user and/or group access to objects.
17 Additional goals for the ACL mechanism are:

- 18 (1) The mechanism should be compatible with the existing POSIX.1 and
19 POSIX.2 standards and, to the extent possible, existing interfaces should
20 continue to work as expected.
21 (2) Reasonable vendor extensions to the ACL mechanism should not be pre-
22 cluded. At a minimum, the specification of read, write and
23 execute/search permissions should be supported. Other permissions
24 should neither be required nor should they be precluded as extensions.
25 (3) New interfaces should be easy to use.
26 (4) Intermixing use between the existing mechanism and newly defined ACL
27 functions/utilities should provide predictable, well understood results.

28 Another goal is to be compatible with existing POSIX.1 standards. Current inter-
29 faces will continue to exist and will affect the overall ACL. Some users will con-
30 tinue to only use the file permission bits. Existing programs may not be modified
31 to use the ACL interface and may continue to manipulate DAC attributes using
32 current POSIX.1 interfaces. These programs should operate on objects with ACLs
33 in a manner similar to their operation on objects without ACLs. However, com-
34 plete compatibility between the existing POSIX.1 DAC interfaces and the
35 POSIX.1e ACL interfaces is simply not achievable. For a discussion of these
36 issues, please refer to B.23.1.

37 The POSIX.1e ACL interfaces should not restrict vendors from providing exten-
38 sions to the basic ACL mechanism; the POSIX.1e ACL interface should not
39 exclude such extensions.

40 For the sake of usability and user acceptance, new interfaces should be as simple
41 as possible while maintaining a reasonable level of compatibility with existing
42 POSIX.1 interfaces.

43 The intermixing of usage between the existing POSIX.1 DAC and the POSIX.1e
44 ACL mechanisms should be well defined and produce reasonable results.

45 The DAC interfaces described in POSIX.1 are adequate for some needs. The file
46 permission bits defined in POSIX.1 are associated with three classes: owner,
47 group, and other; access for each class is represented by a three-bit field allowing
48 for read, write, and execute/search permissions. The POSIX.1e ACL interfaces
49 extend the POSIX.1 interfaces by defining access control lists (ACLs) in order to
50 provide finer granularity in the control of access to objects. ACLs can provide the
51 ability to allow or deny access for individually-specified users and groups of users.
52 However, implementations which allow processes to modify the process' group

53 membership may not be capable of denying access to users based on groups.

54 Several methods exist for allowing discretionary access control on objects. These
55 methods include capability lists, profiles, access control lists (ACLs), permission
56 bits, and password DAC mechanisms. ACLs were selected for the POSIX.1e inter-
57 faces because they meet the goals stated earlier in this section. ACLs are a
58 straightforward extension of the existing POSIX.1 file permission bits which may
59 be viewed as a limited form of ACL containing only three entries.

60 The following features are outside the scope of this document:

61 — Shared ACLs

62 An ACL is shared if it is associated with more than one object; changes to a
63 shared ACL affect the discretionary access for all objects with which the
64 ACL is associated. Shared ACLs are useful as a single point of control for
65 the specification of DAC attributes for large numbers of objects.

66 Although the implementation of shared ACLs is not precluded, shared
67 ACLs are not defined in this standard for the following reasons:

- 68 • It may be difficult to determine the set of objects sharing an ACL. A
69 user could modify the ACL associated with an object and unintention-
70 ally grant access to another object.
- 71 • When changing a shared ACL, it may be necessary to produce an audit
72 record for each file system object that is protected by the ACL.
- 73 • Any changes to a shared ACL which have an unintended security result
74 affect all objects sharing the ACL.

75 — Named ACLs

76 A named ACL is an ACL which exists in the file system space and can be
77 referred to by name. Named ACLs are primarily useful for implementing
78 shared ACLs.

79 Although the implementation of named ACLs is not precluded, named
80 ACLs are not defined in this standard for the following reasons:

- 81 • As file system objects, ACLs themselves may be required to contain dis-
82 cretional access controls which could require recursive ACLs.
- 83 • The owner of a named ACL may not be the owner of the object(s) with
84 which the ACL is associated. The owner of an object could lose control of
85 the DAC attributes associated with that object.

86 **B.23.1 General Overview**

87 POSIX.1 specifies basic DAC interfaces consisting of permissions which specify
88 the access granted to processes in the file owner class, the file group class, and the
89 file other class. These classes correspond to the intuitive notions of the file's
90 owner, members of the file's owning group, and all other users.

91 **B.23.1.1 Extensions to POSIX.1 DAC Interfaces**

92 The specification of the POSIX.1 interfaces provides for two ways to extend discretionary access controls beyond the basic file permission bits:

- 94 • 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.
- 97 • 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.

100 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.

109 **B.23.1.2 Extensions to File Classes**

110 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.

116 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.

122 Refer to B.23.3 for more details on how ACL entries map to the different file classes.

124 **B.23.2 ACL Entry Composition**

125 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.

129 **B.23.2.1 ACL Entry Tag Type Field**

130 Seven distinct ACL entry tag types are defined to be the minimum set of tag types
131 which must be supported by a conforming implementation: ACL_USER_OBJ,
132 ACL_GROUP_OBJ, ACL_OTHER, ACL_USER, ACL_GROUP, ACL_MASK, and
133 ACL_UNDEFINED_TAG.

134 The ACL_USER_OBJ, ACL_GROUP_OBJ, and ACL_OTHER tag type ACL
135 entries are required to exist in all ACLs. If no other entries exist in the ACL, then
136 these entries correspond to the owner, group, and other file permission bits. Since
137 these permission bits can never be removed from a file, the ACL entries
138 corresponding to the permission bits are also required. If an ACL contains any
139 additional ACL entries, then an ACL_MASK entry is also required since it then
140 corresponds to the file group permissions and serves as the maximum permissions
141 that may be granted to the additional ACL entries.

142 While implementations can define additional tag types, the standard does allow
143 an implementation to require the existence of any additional entries in an ACL. If
144 this were allowed, then a file containing only the file permission bits (i.e., an
145 ACL with only three entries) would not be a valid ACL. This would prevent a
146 strictly conforming application from executing correctly on such an implementa-
147 tion which would violate the goal of providing compatibility with the existing
148 POSIX.1 interfaces.

149 An additional ACL entry tag type that could be defined is a “user and group”
150 where such entries specify the access permissions for an individual user within a
151 specific group. While such an ACL entry is useful in some environments, it is not
152 required in the standard since it does not appear to provide widely useful func-
153 tionality. Implementations are not precluded from defining a “user and group” tag
154 type.

155 Implementations which currently allow “user and group” tag type ACL entries
156 can consider the ACL_USER_OBJ and ACL_USER ACL entry tag types to
157 represent access to a user regardless of group membership, e.g., “user.*”. Like-
158 wise, ACL_GROUP_OBJ and ACL_GROUP ACL tag types represent group access
159 regardless of user identity, e.g., “*.group”, and ACL_OTHER represents anybody
160 in any group, e.g., “*.*”.

161 The names of all ACL entry tag types all begin with the prefix “ACL_” in order to
162 provide consistency in naming with other areas of the POSIX standards. While
163 this may make the use of such names slightly more cumbersome for the program-
164 mer, avoiding name conflict through a consistent naming scheme is more impor-
165 tant.

166 POSIX.1e defines two types of ACLs: access and default ACLs. All objects have an
167 access ACL since the POSIX.1 file permission bits are interpreted as a minimal
168 ACL. In addition, a default ACL may be associated with a directory. The rules for
169 ACL entry tag types are the same for both types of ACL. As such, an application
170 can create an ACL and apply it to a file as either an access ACL or a default ACL
171 without changing the ACL structure or any of the ACL data. If POSIX.1e defined
172 ACL entry types which applied to only one type of ACL or if the rules for required
173 ACL entries differed between the types of ACL, then a single ACL could not be

174 applied as both an access and a default ACL.

175 **B.23.2.2 ACL Entry Qualifier Field**

176 The data type of the qualifier field in an ACL entry is specific to the ACL entry
177 tag type. Also, the qualifier field is not extensible for POSIX.1e defined tag types.
178 However, implementations may define the type and structure of the qualifier for
179 entries with implementation-defined tag types. For example, an implementation
180 that wishes to allow the assignment of permissions to an individual user within a
181 specific group could create a tag type, ACL_USER_GROUP, with a qualifier con-
182 taining the identification of both the user and the group. An implementation could
183 also define a user/time entry which could use the qualifier to identify a process
184 within a specified time of day interval.

185 If an implementation could extend the POSIX.1e defined ACL entry qualifier
186 fields, then a strictly conforming application might not function as expected when
187 manipulating an ACL with extended qualifier fields. For example, an implemen-
188 tation extends the qualifier field of the ACL_USER entry type to include a time of
189 day (TOD) interval. A strictly conforming application attempts to manipulate an
190 object's ACL which contains two entries for user fred; one entry contains a TOD
191 qualifier for 0800->1800 and one entry has a TOD qualifier for 1800->0800. If the
192 strictly conforming application intends to change the access allowed for user fred,
193 then the application would call *acl_get_entry()* and *acl_get_qualifier()* until it
194 locates an ACL_USER entry for fred and would then update the entry. The appli-
195 cation would expect only one ACL_USER entry for fred and would only update
196 one entry; since there are two entries for fred, the resulting access for user fred
197 may not be as desired.

198 The special qualifier field value, ACL_UNDEFINED_ID, is defined as a value
199 which cannot be used by the implementation as a valid group or user id. This
200 value is used to initialize the qualifier field within a newly created ACL entry to a
201 value which is not a valid group or user id.

202 **B.23.2.3 ACL Entry Permissions Field**

203 ACL entries are required to support read, write, and execute/search permissions
204 for the following reasons:

- 205 (1) These permissions allow the abstraction of the POSIX.1 file permission
206 bits as ACL entries.
- 207 (2) Existing practice dictates that at least these permissions must be
208 retained.

209 File permissions in addition to read, write, and execute/search are allowed by an
210 implementation because this would allow finer-grained and extended control of
211 access to objects. For example, an implementation could add "append only" or
212 "delete object allowed" permissions. However, such extended permissions are not
213 required by this standard because such permissions are not universally required.

214 **B.23.2.4 Uniqueness of ACL Entries**

215 The combination of ACL entry tag type and qualifier are required to be unique
216 within an ACL. The requirement for unique ACL entries, in combination with the
217 order in which access is checked, provides a simple and unambiguous model for
218 the specification of access information for an object.

219 Note that it is possible for the owner of a file to be explicitly named in an
220 ACL_USER entry within the ACL associated with the file. While this entry may
221 appear to conflict with the entry for the file's owner (i.e., the ACL_USER_OBJ
222 entry), the ACL_USER_OBJ entry will be encountered before any ACL_USER
223 entries during the ACL access check algorithm. Thus, in this case the
224 ACL_USER_OBJ entry would uniquely determine the access permissions for the
225 owner of the file; the individual ACL_USER entry for the file's owner would be
226 ignored. The requirement is that the combination of tag type and qualifier must
227 be unique. Also, the ACL_USER_OBJ entry and the ACL_USER entry are quite
228 different semantically even if the ACL_USER entry contains the identity of the
229 file owner.

230 Likewise, an ACL_GROUP entry with a qualifier id matching the owning group of
231 a file does not conflict with the ACL_GROUP_OBJ entry in the ACL. In such a
232 case, all applicable group entries would be examined to determine if any entry
233 grants the access requested by the process. Both the ACL_GROUP_OBJ entry
234 and the ACL_GROUP entry matching the owning group would be examined and
235 might provide the desired access.

236 **B.23.3 Relationship with File Permission Bits**

237 ACLs expand upon the discretionary access control facility which is already pro-
238 vided by the file permission bits. Although file permission bits do not provide fine
239 granularity DAC, they are sufficient for many uses and are the only mechanism
240 available to existing applications. All existing applications that are security cons-
241 cious use file permission bits to control access. The relationship between the ACL
242 and the file permission bits must be defined in order to determine the level of
243 compatibility provided to existing programs which manipulate the file permission
244 bits.

245 Several approaches are possible for handling the interaction of ACLs with file per-
246 mission bits. Each approach is presented in a separate sub-section with a
247 description of the approach, a list of the advantages, and a list of the disadvan-
248 tages. Final commentary and a conclusion follow the presentation of the
249 approaches.

250 **B.23.3.1 ACL Always Replaces File Permission Bits (Pure ACL)**

251 In this approach, the file permission bits are no longer consulted for ACL deci-
252 sions. Instead, each object has an ACL and the ACL completely determines
253 access. File permission bits would be unused in the standard and the interaction
254 between the file permission bits and ACL entries should be implementa-
255 tion-defined. This method would prevent the use of the old access control mechanism

256 in a strictly conforming application.

257 This approach has the following advantages:

258 — Reduces complexity because there are no compatibility issues between
259 ACLs and permission bits. Permission bits are no longer used for DAC
260 decisions.

261 — A single, well defined discretionary access policy is employed.

262 — Increases security. The old access control mechanism does not provide the
263 proper level of security to meet the requirements of this document.

264 This approach has the following disadvantages:

265 — existing applications that use *chmod()* or *stat()* must be examined to see if
266 they are making DAC decisions. This is because *chmod()* and *stat()* update
267 and return, respectively, more than just DAC information.

268 — existing applications that make DAC decisions must be rewritten to use the
269 new interfaces.

270 — Compatibility between file permission bits and ACLs is left up the vendors
271 who, realistically, must provide some compatibility with their old imple-
272 mentations. Without standardization the compatibility solutions will be
273 vendor specific and not portable.

274 **B.23.3.2 Owner Selects ACL Or File Permission Bits**

275 In this approach, either the file permission bits or the ACL are consulted for the
276 access control decision on a per object basis. The owner of the object determines
277 whether to use the file permission bits or the ACL. If an ACL is set on a file, then
278 the functions that manipulate file permission bits would return an error. If file
279 permission bits are set on a file, then the ACL manipulation functions would
280 return an error for that file.

281 This approach has the following advantages:

282 — If ACLs are never set, then there are no compatibility problems.

283 — If an access ACL is set on an object or a default ACL set on a directory,
284 then the behavior is like the pure ACL system.

285 This approach has the following disadvantages:

286 — Like the previous approach, existing applications that use *chmod()* or *stat()*
287 must be examined to see if they are making DAC decisions.

288 — Existing applications that make DAC decisions must be rewritten to deter-
289 mine which mechanism is in effect for each object it manages and then use
290 the correct interface.

291 **B.23.3.3 Independent ACL And File Permission Bits (AND)**

292 In this approach, both the file permission bits and the ACL are consulted for the
293 discretionary access control decision. Access is granted if and only if it is granted
294 by both the ACL and the file permission bits.

295 This approach has the following advantages:

- 296 — Calls to *chmod()* have the desired effect from a restrictive point of view;
297 ACL entries can further restrict access.
- 298 — The relationship between ACLs and file permission bits is easily defined:
299 to be allowed access both must grant access.

300 This approach has the following disadvantages:

- 301 — To fully utilize the ACL as the effective access control mechanism requires
302 that the file permission bits be set wide-open, i.e. read, write, and execute
303 bits are set for user, group and other.
- 304 — In order to grant access, users must be prepared to change both the ACL
305 and the file permission bits.
- 306 — An application would have to use *chmod()* and *stat()* to manipulate the file
307 permission bits and the ACL functions to manipulate the ACL entries on a
308 file.

309 **B.23.3.4 Independent ACL And File Permission Bits (OR)**

310 In this approach, both the file permission bits and the ACL are consulted for the
311 discretionary access control decision. Access is granted if it is granted by either
312 the ACL or the file permission bits. The ACL is used to grant access beyond what
313 is set in the file permission bits.

314 This approach has the following advantage:

- 315 — Calls to *chmod()* have the desired effect from a permissive point of view.
- 316 — The relationship between ACLs and file permission bits is easily defined:
317 to be allowed access either must grant access.

318 This approach has the following disadvantages:

- 319 — A *chmod(<object>, 0)* call does not deny all access to an object with an ACL.
- 320 — In order to deny access, users must be prepared to change both the ACL
321 and the file permission bits.
- 322 — An application would have to use *chmod()* and *stat()* to manipulate the file
323 permission bits and the ACL functions to manipulate the ACL entries on a
324 file.

325 **B.23.3.5 File Permission Bits Contained Within ACL Without a Mask**

326 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
327 ACL. Calls to *chmod()* modify the ACL_USER_OBJ, ACL_GROUP_OBJ, and
328 ACL_OTHER entries contained in the ACL. Calls to *stat()* return this information
329 from the ACL.

331 This approach has the following advantages:

- 332 — The mapping of ACL entries to permission bits is straight forward. There
333 is no mask entry that may or may not be there.
- 334 — With no additional entries, the semantic meaning of the file permission bits
335 are preserved.
- 336 — There is some compatibility between file permission bits and ACLs. Use of
337 *chmod()* to grant access is compatible. Use of *stat()* to return access for the
338 owning group is compatible.

339 This approach has the following disadvantages:

- 340 — *chmod(<object>, 0)* may or may not prevent access to the object depending
341 on the number of ACL entries. With additional entries, the *chmod()* call
342 does not prevent access to the object and this breaks old style file locking.
- 343 — *chmod go-rwx <object>* may or may not restrict access only to the owner
344 depending on the number of ACL entries. With additional entries, the
345 *chmod()* call does not give owner only access.
- 346 — *creat(<object>, 0600)* may or may not restrict access to the newly created
347 object to the owner. If a non-minimal default ACL exists on the parent
348 directory, then owner only access is not guaranteed.

349 **B.23.3.6 File Permission Bits Contained Within ACL Including a Mask**

350 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
351 *chmod()* modify the ACL entries corresponding to the file permission bits. Calls
352 to *stat()* return this information from the ACL.

355 If there are ACL_USER, ACL_GROUP or implementation-defined ACL entries,
356 then an ACL_MASK entry is required and it restricts the permissions that can be
357 granted by these entries. If there is an ACL_MASK entry, then *chmod()* changes
358 the ACL_MASK entry instead of the ACL_GROUP_OBJ entry and *stat()* returns
359 information from the ACL_MASK entry instead of the ACL_GROUP_OBJ entry.

360 This approach has the following advantages:

- 361 — *chmod(<object>, 0)* prevents access to the object. This provides compatibility
362 with the old locking mechanism.
- 363 — *chmod go-rwx <object>* restricts access only to the owner. This utility call,
364 especially when used with the find utility, is useful for restricting access

365 to objects to the owner.

- 366 — The ACL_MASK entry restricts the permissions that are granted via
367 ACL_USER, ACL_GROUP and implementation-defined ACL entries during
368 object creation. For example, without these restrictions, a *creat(<object>,*
369 *0600)* would not restrict access of a newly created object to the owner.

370 This approach has the following disadvantages:

- 371 — The mapping between the file group class permission bits is not constant.
372 If the ACL_MASK entry exists, then the bits map to it. Otherwise, the bits
373 map to the ACL_GROUP_OBJ entry. This means that *chmod()* and *stat()*
374 update and return, respectively, different information based on the
375 existence of the ACL_MASK entry. This behavior adds complexity to the
376 ACL mechanism.
- 377 — The ACL_MASK entry does not provide complete compatibility with the
378 uses of *chmod()* and *stat()*. *chmod g+rwx <object>* may grant more access
379 than expected due to additional ACL entries.

380 There are several sub-issues with having an ACL mask. The following sub-
381 sections describe those issues.

382 (1) Using ACL_GROUP_OBJ as a Mask

383 The working group considered having the ACL_GROUP_OBJ perform
384 the masking for additional ACL entries.

385 This approach has the following advantages:

- 386 • Removes the five (5) ACL entry to four (4) ACL entry transition prob-
387 lem as described in "Automatic Removal of the ACL_MASK".
- 388 • Removes the special cases in *chmod()* for four (4) ACL entries versus
389 five (5) or more ACL entries as described in "Requiring ACL_MASK to
390 be Present".

391 This approach has the following disadvantages:

- 392 • The permission bits associated with the ACL_MASK limit the access
393 granted by additional ACL entries that are added during object crea-
394 tion. There are two solutions if the ACL_MASK is removed. First,
395 simply do not limit the access granted by the additional ACL entries.
396 See section "File Permission Bits Contained Within ACL Including a
397 Mask" for more details on why this solution is not acceptable. The
398 second solution is to modify the additional ACL entries to grant no
399 more access than was specified by the creating process. See B.23.5.1
400 for more details on why this solution is not acceptable.
- 401 • It is not possible to grant an additional ACL entry more access than
402 the owning group. It is possible to solve this by using a special group
403 with no members as the owning group. However, this solution compli-
404 cates the *setfacl* utility. In the case where an object only grants
405 read access to the owning group and a user wants to add an addi-
406 tional ACL entry that grants read-write access, the *setfacl* utility

407 would have to add an explicit entry for the owning group, change the
408 owning group to the special group, and add the new ACL entry. This
409 solution adds extreme complexity that will be visible to the user.

- 410 • If the file is setgid, then write access is unlikely to be granted by the
411 ACL_GROUP_OBJ entry. This means that additional ACL entries
412 would be unable to be granted write access. However, it is question-
413 able if the owner would want to grant write access to a setgid file.

414 While using the ACL_GROUP_OBJ entry as the mask reduces the com-
415 plexity associated with masking additional ACL entries, its benefits do
416 not outweigh the disadvantages in the areas of object creation and useful-
417 ness of the ACL_GROUP_OBJ entry itself. Therefore, a separate
418 ACL_MASK entry is defined and the ACL_GROUP_OBJ entry is used
419 only to specify the permissions granted to the owning group.

420 (2) Requiring ACL_MASK to be Present

421 The working group considered a strategy to require the ACL_MASK ACL
422 entry to always be present.

423 Either decision adds complexity to the *chmod()* interface. If the
424 ACL_MASK is required, then *chmod()* will behave differently if there are
425 four (4) ACL entries versus five (5) or more ACL entries. If the
426 ACL_MASK is optional, then *chmod()* will behave differently if the
427 ACL_MASK is present versus if the ACL_MASK is absent.

428 This approach has the following advantages:

- 429 • Requiring the presence of an ACL_MASK ACL entry provides con-
430 sistency. Consider the following sequence: A user creates an object in
431 a directory without a default ACL. The user examines the ACL and
432 will only see the ACL_USER_OBJ, ACL_GROUP_OBJ and
433 ACL_OTHER entries. The user adds an additional ACL entry. The
434 user examines the ACL and will see the new ACL entry and the
435 ACL_MASK entry, in addition to the ACL_USER_OBJ,
436 ACL_GROUP_OBJ and ACL_OTHER entries. The ACL_MASK entry
437 has suddenly "sprung" into existence.

438 This approach has the following disadvantages:

- 439 • Requiring the presence of an ACL_MASK entry requires mapping four
440 ACL entries (ACL_USER_OBJ, ACL_GROUP_OBJ, ACL_OTHER and
441 ACL_MASK) onto three groups of permission bits if only the base ACL
442 entries are present.
- 443 • The ACL_MASK serves no purpose if there are no additional ACL
444 entries. Since it serves no purpose in this case, it should not be
445 required.

446 The expected use of a system with ACLs includes the use of default
447 ACLs. Therefore, objects without an ACL_MASK ACL entry are expected
448 to be rare, and most users will not see an ACL_MASK entry "spring" into
449 existence. The standard does not require the ACL_MASK entry to be

450 present if there are no ACL_GROUP, ACL_USER or implementation-
451 defined ACL entries present.

452 (3) Automatic Removal of the ACL_MASK

453 The working group considered requiring that the ACL_MASK entry
454 automatically be removed when all ACL entries other than
455 ACL_USER_OBJ, ACL_GROUP_OBJ, ACL_OTHER and ACL_MASK
456 were removed.

457 This approach has the following advantages:

- 458 • Requiring automatic removal makes the existence of the ACL_MASK
459 less obvious to the user.
- 460 • Requiring automatic removal is simply a clean-up step. The
461 ACL_MASK has performed its function and is no longer needed.

462 This approach has the following disadvantage:

- 463 • Requiring automatic removal of the ACL_MASK and the resultant
464 resetting of the ACL_GROUP_OBJ permission bits leads to execution
465 order specific results (in the absence of automatic recalculation). See
466 below for an example.

467 If ACL_MASK is explicitly removed, then the permissions of
468 ACL_GROUP_OBJ must be set to reasonable values. The working group
469 considered the following cases:

- 470 • Leave ACL_GROUP_OBJ unchanged.

471 If the ACL_GROUP_OBJ has more access than the old ACL_MASK,
472 this case could unintentionally grant increased access rights. Since
473 this is a security violation, this case is rejected.

- 474 • Set ACL_GROUP_OBJ to the value of ACL_MASK.

475 If the ACL_MASK has more access than the old ACL_GROUP_OBJ,
476 this case could unintentionally grant increased access rights. Since
477 this is a security violation, this case is rejected.

- 478 • Return an error to the user if an attempt is made to delete
479 ACL_MASK when ACL_MASK and ACL_GROUP_OBJ differ.

480 This case was viewed as confusing and was rejected, because deleting
481 an ACL entry should be independent of the ACL_MASK and
482 ACL_GROUP_OBJ interactions. It does force the user to understand
483 the problem and take immediate action, rather than waiting until the
484 inadvertent access reductions from the next case are discovered.
485 Finding out about a problem immediately is generally better than dis-
486 covering it inadvertently much later.

- 487 • Logically AND the ACL_MASK and ACL_GROUP_OBJ together and
488 set ACL_GROUP_OBJ to the result.

489 This case can lead to inadvertent access reduction (in the absence of
490 automatic recalculation). For example, an object has an ACL with
491 ACL_GROUP_OBJ ACL entry with read-only access and an
492 ACL_USER(fred) entry with read-write access. Deleting the
493 ACL_USER(fred) entry and then adding an ACL_USER(wilma) entry
494 will produce an ACL that does not allow wilma to have write access to
495 the object. However, adding ACL_USER(wilma) followed by deleting
496 ACL_USER(fred) produces the desired affect.

497 While automatically removing the ACL_MASK when it is no longer
498 needed makes the mask less obvious to the user, its benefits do not
499 outweigh the complexity it adds to the programmatic interface. Therefore,
500 the application must take an explicit action to remove the
501 ACL_MASK entry when it is no longer needed within the ACL.

502 (4) Migration Path Flag

503 It is possible to define a flag to indicate whether masking is enabled or
504 disabled for the implementation.

505 This approach has the following advantages:

- 506 • This flag would give individual system administrators the choice of
507 determining the type of operation required for their specific installation.
508
- 509 • The flag would provide a migration path for some applications which
510 use the *chmod()* function for file locking.

511 This approach has the following disadvantages:

- 512 • The existence of a flag would complicate DAC knowledgeable applications.
513 Software vendors would have to provide different versions of
514 the applications for the different environments or will have to modify
515 their applications to work within the different environments.
- 516 • The existence of a flag will complicate the utility interfaces defined by
517 this standard when used in a networked environment where some
518 systems have the flag enabled and some systems have the flag
519 cleared.
- 520 • The working group is chartered with only producing interfaces. Providing
521 a migration path to a future usage model is beyond the scope of
522 this standard.

523 Given the complexity involved with providing a migration path flag, this
524 standard does not include such a flag.

525 **B.23.3.7 The Conclusion**

526 Compatibility with the existing DAC interfaces in some form or another is the
527 overriding goal of this section. Most of the approaches considered provided some
528 level of compatibility with the existing DAC interfaces. The file permission bits
529 cannot reflect all the information that can be contained in an ACL. However, the

530 *stat()* function should still reflect a reasonable amount of information regarding
531 the access rights of files and the *chmod()* function should still be reasonably com-
532 patible with the previous semantics regarding the update of access information on
533 files. Each approach has compelling advantages and discouraging disadvantages.

- 534 • The "ACL Always Replaces File Permission Bits (Pure ACL)" approach was
535 rejected because it provides no compatibility.
- 536 • The "Owner Selects ACL Or File Permission Bits" approach was rejected
537 because it requires existing applications that manage DAC to be modified
538 to be used on a system with ACLs.
- 539 • The "Independent ACL and File Permission Bits (AND)" approach was
540 rejected because it leads to wide-open file permission bits on systems that
541 make use of ACLs with additional entries.
- 542 • The "Independent ACL and File Permission Bits (OR)" approach was
543 rejected because a user of the existing DAC interfaces can be fooled into
544 thinking that an object with additional ACL entries is secure when, in fact,
545 others have access to the object.
- 546 • The "File Permission Bits Contained Within ACL Without a Mask"
547 approach was rejected because a user of the existing DAC interfaces can be
548 fooled into thinking that an object with additional ACL entries is secure
549 when, in fact, others have access to the object.
- 550 • The "File Permission Bits Contained Within ACL Including a Mask"
551 approach was chosen because it provides the "best" compatibility with the
552 existing DAC interfaces.

553 **B.23.3.8 Altering Permission Bit Mapping**

554 Allowing implementation-defined ACL entries to alter the mapping between file
555 permission bits and ACL entries defined by this standard was considered. If an
556 implementation-defined entry is allowed to modify the permission bits, then it is
557 possible for a strictly conforming POSIX.1e application to fail. Note that a strictly
558 conforming application cannot add the implementation-defined entry to an ACL,
559 but the strictly conforming application may not function properly if it modifies an
560 ACL that contains the implementation-defined ACL entry. Consider the follow-
561 ing: an strictly conforming application modifies the ACL_USER_OBJ entry in an
562 ACL that contains an implementation-defined ACL entry. The implementa-
563 tion-defined ACL entry modifies the permission bits. The strictly conforming applica-
564 tion expects the middle permission bits to be identical to the permission bits in
565 the ACL_GROUP_OBJ entry. However, the permission bits have been modified
566 by the implementation-defined ACL entry. The strictly conforming application is
567 broken.

568 **B.23.4 Default ACLs**

569 A default ACL is a defined set of ACL entries that are automatically assigned to
570 an object at creation time. There were five major decisions with default ACLs.
571 The following subsections explain the rationale for these decisions.

- 572 (1) Why Define Default ACLs?
573 (2) Types of Default ACLs
574 (3) Inheritance of Default ACLs During Object Creation
575 (4) Compulsory versus Non-compulsory ACLs
576 (5) Default ACL Composition

577 **B.23.4.1 Why Define Default ACLs**

578 Should support for default ACLs be defined by the standard? The following rea-
579 sons support inclusion of default ACLs in the standard:

- 580 (1) ACL use is encouraged in secure systems.
581 (2) Default ACLs allow the finer granularity of control provided by ACLs to
582 be automatically applied to newly created objects. This control can be
583 either restrictive or permissive.
584 (3) In a pure ACL environment, it is necessary to provide some initial access
585 rights to a newly created object.

586 The following reasons support exclusion of default ACLs from the standard:

- 587 (1) It is not clear that the benefit of default ACLs outweighs the complexity
588 introduced in object creation and object attribute management. Object
589 creation will have to accommodate the existence of default ACLs in addition
590 to the umask and the object creation mode bits. Either a new set of
591 interfaces has to be created for manipulating default ACLs or the inter-
592 faces for access ACL manipulation will have to be modified to accommo-
593 date default ACLs.
594 (2) The default ACL in any form is a new influence on the ACL of a newly
595 created object and cannot be manipulated or worked around by existing
596 applications. Most existing applications will be able to coexist with
597 default ACLs. However, existing applications that make security
598 relevant decisions may not work on a system with default ACLs. See
599 B.23.5 for specific examples.

600 In general, default ACLs appear to be a useful feature. Several existing ACL
601 implementations have some form of default ACL mechanism. Certainly, default
602 ACLs add complexity to the standard; however, they also add considerable value
603 and should have a well defined standard interface.

604 **B.23.4.2 Types of Default ACLs**

605 Several different types of default ACLs were discussed by the working group. The
606 advantages and disadvantages of each type of default ACL are discussed in the
607 following paragraphs. The final paragraph of this section discusses why a partic-
608 ular type of default ACL was chosen.

609 (1) System Wide Default ACLs

610 One specific default ACL is assigned to any object created on the system
611 by any process, in any directory. System wide default ACLs have the fol-
612 lowing advantages:

- 613 — Can only be set by the system administrator who is likely to be secu-
614 rity conscious

615 — Is not complex or difficult to understand and explain

616 System wide default ACLs have the following disadvantage:

- 617 — Limits the specification of the initial discretionary access control on
618 objects to system administrators rather than the user

619 (2) Per-Process Default ACLs

620 Each user process defines a default ACL which is assigned to any object
621 created by the process. Per-process default ACLs have the following
622 advantages:

- 623 — Models an existing interface, i.e., the umask paradigm
- 624 — Allows the user to retain complete control over the configuration of
625 discretionary access

626 Per-process default ACLs have the following disadvantages:

- 627 — Follows a paradigm that is considered to be inadequate for present
628 needs, i.e., the umask paradigm
- 629 — Requires the user to be security cognizant at all times; however, a
630 knowledgeable user will only make security relevant decisions with a
631 modest degree of frequency
- 632 — Might not be the right default ACL in a shared directory
- 633 — Allows the user to set only a single default ACL for all files created

634 (3) Per-Directory Default ACLs

635 Each directory is allowed to have a default ACL which is assigned to all
636 objects created in the directory. Newly created subdirectories inherit the
637 default ACL of the parent directory. Per-directory default ACLs have the
638 following advantages:

- 639 — Allows the user to set up the hierarchy once
- 640 — Prevents the user from having to set a new default ACL as working
641 directories are changed

642 — Allows system administrators to establish initial default ACLs on
643 users' home directories which will propagate to objects created within
644 the directories

645 — Allows project administrators to establish initial default ACLs on
646 shared directories which will propagate to objects created within the
647 directories

648 Per-directory default ACLs have the following disadvantages:

649 — Propagates the default ACL down through the file system hierarchy in
650 cases where it is not necessary

651 — An implementation written to conserve disk space may have to imple-
652 ment a default ACL sharing mechanism

653 — Gives the choice of the default ACL to the directory owner instead of
654 the file creator

655 The working group recognizes that a per-directory default ACL gives the
656 directory owner control over the default value. However, the directory
657 owner currently has control over at least one attribute of objects created
658 in the directory: specifying the owning group. Also note that the direc-
659 tory owner has control over object creation, deletion, renaming and
660 replacement.

661 The value added by per-directory default ACLs outweighs the complexity intro-
662 duced by the mechanism and was, therefore, selected as the default ACL mechani-
663 smism.

664 **B.23.4.3 Inheritance of Default ACLs During Object Creation**

665 While the working group felt that default ACLs on a per-directory basis provided
666 the best solution, it considered alternatives to simply propagating the default
667 ACL to all newly created objects in a directory. The working group considered
668 two basic schemes for inheritance of ACLs involving the default ACL mechanism:

669 (1) Inheritance of Default ACLs for All Objects

670 The first alternative considered was to have all objects created in a direc-
671 tory inherit the default ACL of the directory. The working group felt that
672 this solution provided an ACL inheritance mechanism that was con-
673 sistent across all objects. This option does not take into account any
674 differing permission requirements for directories as opposed to non-
675 directory objects.

676 (2) Inheritance of Access ACLs for Directory Objects

677 The second alternative specified inheritance of the default ACL as the
678 access ACL for all newly-created objects except directories. A newly-
679 created directory would inherit the access ACL of its parent directory as
680 its access ACL instead of inheriting the parent's default ACL. This
681 approach was attractive because it allowed propagation of common pro-
682 perties through a sub-hierarchy which was thought to be the most

683 common case. It further allowed different permissions to be applied to
684 directories and non-directories which was considered a useful feature.

685 The disadvantages to this approach were the following:

686 — The implementation would not be consistent across all objects. The
687 semantics for applying initial access control information to a single
688 type of file object would differ from the semantics for all other types of
689 file.

690 — In the case where a parent directory has no default ACL, counter-
691 intuitive side effects were unavoidable.

692 • If the access ACL were applied to a newly created directory object
693 only when a default ACL is present, the application of initial
694 access attributes to the directory is determined by an event unre-
695 lated to the action of creating the directory, i.e., the presence of a
696 default ACL. This behavior violates the Principle of Least Aston-
697 ishment.

698 • If the access ACL were always applied to a newly created direc-
699 tory, the semantics of POSIX.1 are violated. The method for apply-
700 ing initial access attributes to directories no longer would allow
701 the capability to create a minimal ACL, i.e., one corresponding to
702 permission bits, in a manner consistent with the POSIX.1 umask
703 capability.

704 The working group selected the first mechanism because the ease in which it
705 could be consistently applied. The working group felt that the advantages of the
706 second approach were not sufficiently beneficial to warrant accepting the disad-
707 vantages. If a more flexible default ACL mechanism providing some of the advan-
708 tages of the second alternative is desired, an implementation may include addi-
709 tional default ACLs for this purpose.

710 **B.23.4.4 Compulsory Versus Non-Compulsory Default ACLs**

711 The standard requires a conforming implementation to support a per-directory
712 default ACL mechanism. The working group discussed whether or not default
713 ACLs should be required on every directory.

714 The following supports requiring default ACLs on every directory:

715 (1) Allows a consistent ACL policy to be maintained for all newly created
716 objects

717 (2) minimizes the need for the umask

718 The following supports the optional use of default ACLs:

719 (1) Allows users who wish to use only the permission bits to use only the
720 existing DAC mechanism

721 (2) Allows existing mechanisms to further restrict access on the newly
722 created object, i.e. creat and umask

723 The working group feels that allowing users to use either default ACLs or the
724 umask interface provides a significant amount of flexibility. Thus, the working
725 group decided to make the use of default ACLs on directories optional.

726 **B.23.4.5 Default ACL Composition**

727 The working group discussed having the same required entries for default and
728 access ACLs or to have no required entries in default ACLs.

729 The following supports having identical required entries for default and access
730 ACLs:

731 (1) Supporting optional default ACL entries leads to a more complex object
732 creation algorithm that is difficult to explain.

733 The following supports having no required entries in default ACLs:

734 (1) The user has the flexibility to configure the default ACL with the
735 minimum amount of access information that is necessary.

736 The working group feels that consistency between default ACLs and access ACLs
737 contributes dramatically to the conceptual simplicity of the default ACL mechan-
738 ism and that the need for simplicity far outweighs the small increase in flexibility
739 provided by optional default ACL entries. Therefore, default ACLs have the same
740 required entries as access ACLs.

741 Note that default ACLs are optional on individual directories. However, if a direc-
742 tory has a default ACL, then that ACL must contain at least the three required
743 entries for owning user, owning group, and all other users. It may contain addi-
744 tional named user and group entries. If a default ACL contains ACL_USER,
745 ACL_GROUP or implementation-defined ACL entries, then an ACL_MASK entry
746 is also required.

747 Also note that a default ACL with no entries is not equivalent to no default ACL
748 existing on a directory. A default ACL with no entries is an error and any attempt
749 to associate such a default (or access) ACL on an object will be rejected with an
750 appropriate error code. The appropriate functions (or options on the setfacl
751 utility) must be used to completely remove a default ACL from a directory.

752 **B.23.5 Associating an ACL with an Object at Object Creation Time**

753 The following goals guided the working group in determining how ACLs should be
754 assigned on object creation:

755 • The object creation calls and the *open()* call with the **O_CREAT** flag specify
756 the mode to use when an object is created. The mode provided is the
757 program's way of indicating the access limitations for the object. It was a
758 goal that no access be permitted to the object if it would not traditionally
759 have been granted.

760 • There are many existing programs that use *creat(filename, 0)* as a locking
761 mechanism. Although this is no longer a recommended way of doing

- 762 locking, preserving this functionality shall be given high priority.
- 763 • The process umask is the user's way of specifying security for newly created
764 objects. It was a goal to preserve this behavior unless it is specifically over-
765 ridden in a default ACL.
- 766 • The access determined by an ACL is *discretionary* access control. But dis-
767 cretion of whom, the creator or the directory owner? Traditionally, discre-
768 tion has been up to the creator. However, ACLs are often used by projects
769 in shared directories. It was a goal to permit the directory owner to have
770 control, but only within the limits specified by the creator.
- 771 • The Principle of Least Astonishment is a guideline that states that changes
772 to existing interfaces should provide a minimal amount of surprise.
- 773 The working group considered whether the creating process should be allowed to
774 control the inheritance of default ACLs. If the process controls inheritance, then
775 the process can keep a default ACL from further restricting the permissions. But
776 the creator can achieve this anyway, by changing the ACL after creation. There-
777 fore no additional control for the creator was provided.
- 778 The algorithm chosen for determining the mode of a newly-created object is in the
779 body of the standard. The reasons why this algorithm was chosen are:
- 780 (1) If there is no default ACL on the parent directory of the created object,
781 the ACL assigned to the object is fully compatible with the access granted
782 to the object in a POSIX.1 system.
- 783 (2) The entries of the default ACL are used in place of the equivalent umask
784 bits. Thus, the creator of the default ACL can control the maximum per-
785 missions for newly created files in the directory.
- 786 If umask were used when a default ACL exists, then the user is likely to
787 set a very permissive umask to permit the full utilization of the default
788 ACL. This permissive umask would be inappropriate in a directory
789 without a default ACL. The chosen solution allows umask and default
790 ACLs to co-exist.
- 791 (3) The newly created object has all the ACL_USER and ACL_GROUP ACL
792 entries specified in the default ACL. The ACL_USER_OBJ,
793 ACL_GROUP_OBJ, and ACL_OTHER entries are as close to the ones
794 specified in the default ACL as possible, within the constraints of the
795 creator's mode parameter. If the default ACL contains an ACL_MASK
796 entry, then it is constrained by the creator's mode parameter instead of
797 the ACL_GROUP_OBJ entry. In this case, the newly created object has
798 the ACL_GROUP_OBJ entry as specified in the default ACL.
- 799 (4) The overall effect is that the access granted to the newly created object
800 has the granularity specified by the default ACL, while preserving the
801 constraints specified by the object creator.
- 802 The only disadvantage recognized by the working group for this algorithm is that
803 the umask is not taken into consideration when creating files in a directory with a
804 default ACL. This solution gives the user little protection against a program that

805 specifies an unwise create mode when creating a file in a directory with an inap-
806 propiate default ACL.

807 Another possible approach is to ignore both the mode parameter of the *creat()*
808 function and the umask value if a default ACL entry exists. This approach was
809 considered because it gives the directory owner complete control over newly
810 created objects in her/his directory. Allowing the directory owner to have control
811 over the permissions of newly created objects is a logical extension. This solution
812 also supports the contention that the directory owner knows how to set up the
813 permissions for newly created objects in a particular hierarchy.

814 This algorithm was not selected because the directory owner can override the
815 program's advice about the use of a newly created object, i.e., override the create
816 mode. Traditionally, the creator of an object has complete control over the mode
817 of a newly created object. This solution would completely usurp that control from
818 the creator.

819 The specification of the semantics for applying ACLs on a newly created object is
820 included as part of this standard so that applications can predict reliably the
821 access that will be granted (or more accurately, the maximum access that will be
822 granted) based on a default ACL set by that application. This is simply an exten-
823 sion of the specification of the setting of the file permission bits for newly created
824 files in the POSIX.1 standard without the ACL option.

825 **B.23.5.1 Modification of ACL Entries on Object Creation**

826 The working group considered changing the default ACL mechanism to modify
827 the permissions granted by additional ACL entries that are added during object
828 creation. The permissions would be modified to grant no more access than was
829 specified by the creating process.

830 This strategy has the following advantage:

- 831 • If the permissions of the additional ACL entries are modified as described
832 above, then the mode parameter specified at object creation could be used
833 to remove undesired permissions from all entries in the new object's access
834 ACL.

835 This strategy was rejected for the following reasons:

- 836 • If the permissions of the additional ACL entries are modified as described
837 above, then information that the creator of the default ACL entered is lost.
838 The most common example is that a *creat(file, 0600)* would lose the infor-
839 mation in the default ACL for all ACL_USER and ACL_GROUP entries.
840 This represents a potential for considerable information loss.

841 **B.23.6 ACL Access Check Algorithm**

842 The ACL access check algorithm has several important characteristics.

843 (1) Support for concurrent membership in multiple groups.

844 If a process belongs to multiple groups, the specific access modes
845 requested are granted if they are granted by the owning group entry or
846 by a matching group entry in the ACL.

847 (2) Consistency with existing POSIX.1 features.

848 The *chmod()* and *stat()* functions will continue to operate on the permis-
849 sions associated with the object's owner, owning group, and other users
850 not matching entries in the ACL.

851 (3) Relative ordering of algorithm steps.

852 The relative ordering of the algorithm steps is essential to be able to
853 exclude specific users even if they belong to a group that otherwise may
854 be granted access to the resource.

855 (4) Support for extensibility.

856 Implementations that include additional ACL entry tag types or exten-
857 sions may insert them as appropriate into the relative order of the
858 defined steps in the algorithm.

859 The rationale for the first of these characteristics is covered in detail below. The
860 issue of interoperability is discussed in detail in B.23.3.

861 **B.23.6.1 Multiple Group Evaluation**

862 The design of supplemental groups in POSIX.1 was intended to provide flexibility
863 in allowing users access to files without requiring separate actions to first change
864 their group identities. The ACL mechanism facilitates that intent by allowing the
865 inclusion of multiple named group entries in the ACL. Since it is possible for a
866 process to match more than one named group entry in the ACL at a time, it is
867 necessary to define the access that is granted by the matched entries.

868 The following paragraphs discuss the approaches that were considered:

869 (1) First group-id match. In this approach, the first entry that matches one
870 of the process's groups is used to determine access. Access is granted if
871 the matched entry grants the requested permissions.

872 This approach does provide a simple solution to the problem, but it does
873 so by putting a burden on the user to order the ACL_GROUP entries
874 correctly to get the desired result. Also, while this is an efficient method
875 to implement, it does dictate implementation details because the ACL
876 entries must be maintained by the system in the order that they were
877 entered by the user.

- 878 (2) Intersection of matching entries. In this approach, the permissions of all
879 the entries which match groups of the process are intersected (ANDed)
880 together. Access is granted if the result of the intersection grants the
881 requested permissions.
- 882 This approach does provide a slightly complex solution (from a user point
883 of view) to the problem, but it is considered very restrictive. It is difficult
884 to justify that a process that is granted read access through one group
885 and write access through another group should actually get no access.
- 886 (3) Union of matching entries. In this approach, the union is taken of the
887 permissions of all the entries which match groups of the process. Access
888 is granted if the result of the union grants the requested permissions.
- 889 This approach does provide a slightly complex solution (from a user point
890 of view) to the problem, but it is considered rather permissive. It is not
891 possible to ensure denial of access to all members of a group via a restric-
892 tive group entry because members of that group may be allowed access
893 via membership in other groups. It is also possible for a process to be
894 granted more access than is granted by a single entry, e.g., one entry
895 grants read access, one entry grants write access and the process is
896 granted read *and* write access.
- 897 (4) Permission match. In this approach, the permissions of all the entries
898 which match groups of the process are compared with the requested
899 access. Access is granted if at least one matched entry grants the
900 requested permissions.
- 901 This approach provides a simple solution to the problem that is very
902 similar to the POSIX.1 semantics. In POSIX.1, if a process is in the file
903 group class and the file group class permissions grant at least the
904 requested access, then the process is granted access. In this approach, if
905 a process is in the file group class and the permissions of one of the ACL
906 entries in the file group class grant at least the requested access, then
907 the process is granted access.
- 908 One of the goals of the ACL mechanism is to be compatible with POSIX.1. Of the
909 different approaches considered, the "Permission match" approach provides the
910 semantics that most closely match POSIX.1 and is the chosen approach.

911 **B.23.6.2 Multiple User Evaluation**

912 If the effective group ID or any of the supplementary group IDs of a process
913 matches the group ID of an object, then the POSIX.1e access check algorithm uses
914 the permissions associated with the ACL_GROUP_OBJ entry and the permis-
915 sions associated with any matching ACL_GROUP entries in determining the
916 access which can be granted to the process. However, if the effective user ID of
917 the process matches the user ID of an object owner, then only permissions associ-
918 ated with the ACL_USER_OBJ entry are used to determine the access allowed for
919 the process. No ACL_USER entries are used even if the process matches the

920 qualifier information for one or more entries.

921 This type of behavior is consistent with the previous POSIX.1 interface since a
922 process could not match multiple user identities yet could match multiple groups.

923 **B.23.7 ACL Functions**

924 **B.23.7.1 ACL Storage Management**

925 These issues apply to both access ACLs and default ACLs. The decision to mani-
926 pulate ACL entries in working storage was made for two reasons: 1) the possibil-
927 ity of unsecure states and 2) the fact that there can be a variable number of ACL
928 entries.

929 If ACL entries could be manipulated directly, or if ACL entries could be manipu-
930 lated while the ACL continued to protect the object, unsecure states could arise.
931 This is because the functions which manipulate ACL entries only manipulate sin-
932 gle entries. The procedural interfaces we have chosen are not capable of changing
933 several entries in a single autonomous operation. Because of this the possibility
934 exists that a less secure state could arise during the modification of an ACL.

935 **B.23.7.1.1 Allocating ACL Storage**

936 Since an ACL can contain a variable number of ACL entries, mechanisms to allo-
937 cate and free dynamic memory are required. The working group considered four
938 approaches. The first approach was to have a single function that allocates a
939 specific amount of memory for the ACL. The disadvantage to this approach is
940 that the user must allocate enough storage or an error will occur and new larger
941 working storage will have to be allocated and the ACL entries recreated.

942 The second approach is to have two functions that allocate space for the ACL.
943 The first function allocates a specific amount of space for the ACL and the second
944 function increases the space allocated by the first function to a specific size.

945 The third approach is to have a single function that allocates an initial amount of
946 memory. Applications would then provide the address of the pre-allocated ACL
947 storage area to the ACL manipulation functions. The *acl_copy_int()*,
948 *acl_create_entry()*, *acl_from_text()*, *acl_get_fd()*, and *acl_get_file()* functions would
949 manipulate the ACL within the ACL storage area provided by the application and
950 would allocate additional memory as needed.

951 The fourth approach is to have the routines which work with working storage
952 areas for opaque data types allocate the working storage as needed and then
953 return pointers to descriptors for those areas. Functions which then manipulate
954 the ACL in the working storage area would allocate additional memory for work-
955 ing storage as needed. In addition, a function to allocate storage for an ACL with
956 no entries would be provided.

957 The final approach has been chosen for inclusion in the standard in order to pro-
958 vide a consistent interface among the various sections of POSIX.1e.

959 **B.23.7.1.2 Copying ACL Storage**

960 The *acl_copy_entry()* function is provided for several reasons: an *acl_entry_t* is a
961 descriptor and cannot be byte copied; an implementation can have extensions and
962 without the function it is not possible for a portable application to copy an entry.

963 The *acl_copy_entry()* function is also provided to allow an application to copy an
964 entry from one ACL to another ACL. This is useful when the source ACL is a list
965 of "defaults" that the application provides for building ACLs to apply to arbitrary
966 objects.

967 The *acl_copy_entry()* function allows an application an easy means of copying an
968 ACL entry from one ACL to another ACL. For example, one implementation of an
969 ACL builder application may maintain an ACL "scratch pad" that is used to build
970 ACLs to be applied to objects. The application may provide a means of highlighting
971 specific ACL entries in the "scratch pad" to be copied to the ACL that is being
972 built.

973 **B.23.7.1.3 Freeing ACL Storage**

974 An explicit interface for freeing ACL storage is provided. The working group con-
975 sidered embedding this functionality into the *acl_set_file()* and *acl_set_fd()* inter-
976 faces. The disadvantage is that a program wanting to apply a single ACL to mul-
977 tiple files would have to create or read the ACL for each application of the ACL.

978 **B.23.7.2 ACL Entry Manipulation**

979 Interfaces are provided to manipulate ACL entries. There were five major deci-
980 sions with ACL entry manipulation. The following subsections explain the
981 rationale for these decisions.

982 **B.23.7.2.1 Procedural Versus Data Oriented Interfaces**

983 This standard uses a procedural interface to manipulate ACL entries instead of
984 the traditional UNIX style data oriented interface.

985 A data oriented interface specification typically defines a small set of primitives to
986 access data objects, e.g. read, write, or commit. The application must be aware of
987 the structure of the data and is responsible for direct manipulation of the data.
988 The advantages of a data oriented interface is that it provides the application a
989 substantial amount of flexibility in accessing and manipulating the data. How-
990 ever, because the application must know the structure of the data, any change in
991 the ordering, size, or type of the data will impact the application.

992 A procedural interface isolates the application from the structure of the data. The
993 interface consists of a larger set of functions where each function performs one
994 operation on one field within the object. The application manipulates the data
995 items within an object by using a series of functions to get/set each data item and
996 a smaller set of functions to read and write the object. The advantage of a pro-
997 cedural interface is that it allows changes and extensions to the structure of the
998 data without any impact to applications using that data. However, isolating the

999 application from the data structure provides the application with less flexibility in
1000 accessing and manipulating the data and exhibit poorer performance.

1001 A data oriented interface has the following advantages:

1002 • consists of a small set of functions.

1003 • can be manipulated by language primitives.

1004 • is consistent with traditional UNIX calls, e.g., *stat()*, *chmod()*, etc.

1005 A procedural interface has the following advantages:

1006 • allows changes/extensions to the data structures without impacting appli-
1007 cations.

1008 • contains fewer visible data structures

1009 • supports a move toward object oriented interfaces which tends to encourage
1010 more portable code

1011 The advantages of isolating applications from the structure of ACLs and ACL
1012 entries are substantial. Thus, a procedural interface was chosen to manipulate
1013 access control list information.

1014 We originally did not choose to define a procedural interface for manipulating the
1015 permission set within an ACL entry. Our reason was that the application must
1016 be aware of the structure of permission sets (bits within a long data type) and
1017 should be responsible for manipulating the bits directly. In our original opinion,
1018 the ease of direct language manipulation of the permission bits far exceeded any
1019 advantage gained in hiding the structure of the information.

1020 During balloting it became clear that procedural interfaces for permission bits
1021 had additional advantages. Functions to manipulate permission sets were added
1022 later to allow an implementation to have more permissions than could fit in a
1023 natural data type (32 bits). While it is somewhat difficult to imagine why more
1024 than 32 permissions are needed, it is not good design to preclude such an imple-
1025 mentation.

1026 **B.23.7.2.2 Automatic Recalculation of the File Group Permission Bits**

1027 The initial proposal was to recalculate the **file group permission bits** whenever
1028 a new ACL entry is added. The following example illustrates a problem with this
1029 approach.

1030 Consider a file created with a file creation mask of 0 in a directory that
1031 contained a fully populated default ACL. This file will have **file group**
1032 **permission bits** of 0, i.e., ---, yet may have named ACL_USER or
1033 ACL_GROUP entries specifically granting permissions. (These entries
1034 will be effectively ignored during access checking because of the masking
1035 effect of the 0 **file group permission bits**.) If the **file group permis-**
1036 **sion bits** are automatically recalculated whenever a new ACL entry is
1037 added, the result of adding a ACL_USER entry specifically denying a
1038 user access will be to effectively grant access to the previously masked
1039 ACL entries.

1040 It seems counter-productive at best to have an entry that denies a user access also
1041 grant access to other users. However, there does not exist a technique to allow for
1042 the application of a single entry in an ACL and the exclusion of others.

1043 Other proposed alternatives include providing a mechanism in the `setfac1` util-
1044 ity to specifically request recalculation. A problem with this alternative is that
1045 typically a user adds an entry to an ACL with the intent of having the new entry
1046 affect the access decision. It isn't possible to have one new named `ACL_USER` or
1047 `ACL_GROUP` entry be guaranteed effective in the access algorithm without recal-
1048 culating the **file group permission bits** based on all entries.

1049 The final alternative considered by the working group is to provide an explicit
1050 interface for recalculating the mask.

1051 **B.23.7.2.3 Convenience Functions**

1052 The `acl_calc_mask()` function is provided for the convenience of applications.
1053 Applications could be required to perform this function, but DAC knowledgeable
1054 applications are likely to need it. Therefore, it is better to provide a standard
1055 interface.

1056 The `acl_valid()` function is provided as a convenience for applications. Applications
1057 could be required to perform this function, however this functionality will
1058 likely be used by ACL cognizant applications. Therefore it is better to provide a
1059 standard interface for this functionality.

1060 It is possible to merge the `acl_valid()` and `acl_set_*`() functions together. How-
1061 ever, it may be useful for ACL cognizant applications to be able to perform the
1062 `acl_valid()` function without having to apply (write out) the ACL to an object.
1063 This was seen as particularly useful for interactive tools in dealing with access
1064 and default ACLs.

1065 The group considered providing program interfaces for the creation of objects with
1066 a specified ACL and other security attributes. The motivation for this is that
1067 security-conscious programs may wish to ensure that objects they create have
1068 correct ACL and other security attributes throughout their life, from the instant
1069 they are created. The group decided not to standardize such interfaces because
1070 programs can achieve the security objective by creating the object using existing
1071 POSIX.1 interfaces specifying very restrictive permissions and then setting the
1072 ACL to the required value.

1073 The `acl_first_entry()` function was added to allow applications to revisit ACL
1074 entries previously referenced with `acl_get_entry()`. This is particularly needed by
1075 applications which are creating an ACL in working storage and need to revisit a
1076 previously created entry.

1077 **B.23.7.2.4 Hooks For Sorting**

1078 The `acl_valid()` function may change the ordering of ACL entries. This behavior
1079 allows an implementation to sort ACL entries before passing them to the
1080 `acl_set_*`() function. This allows a performance improvement to be recognized.
1081 Since the `acl_set_*`() function does not require any specific ordering, the system

1082 will likely sort all entries so that it may check for duplicates. If the sorting is per-
1083 formed by the *acl_valid()* function, the system may only need to make one pass
1084 through the ACL resulting in an order (N) sort when the *acl_set_**() function is
1085 called.

1086 Functions which may add entries to an ACL, or remove them, are also allowed to
1087 reorder the entries of an ACL. This permits, but does not require, an implemen-
1088 tation to keep an ACL in some implementation specific order.

1089 Note that the standard requires that even implementations that reorder the
1090 entries of an ACL do not invalidate any existing ACL entry descriptors that refer
1091 to the ACL: these must continue to refer to the same entries even if the imple-
1092 mentation reorders the entries.

1093 **B.23.7.2.5 Separate Functions for Tag and Permission**

1094 A single function (for example, *acl_get_entryinfo()*) could have been provided for
1095 retrieving ACL entry fields rather than separate functions. However, the stan-
1096 dard provides individual interfaces for retrieving and setting each logical piece of
1097 information within an ACL entry. Implementations can add information to an
1098 entry and add a separate interface for that implementation-specific information
1099 rather than changing the ones specified in this standard.

1100 Implementations are allowed to define additional ACL entry types with arbitrary
1101 size qualifier fields. Because of this, *acl_get_qualifier()* cannot simply copy out a
1102 user ID or group ID size object. The *acl_get_qualifier()* interface returns a pointer
1103 to an independent copy of the qualifier data in the ACL entry. The copy is
1104 independent because the ACL entry may be relocated by an *acl_create_entry()* or
1105 *acl_delete_entry()* call. When the application is done with the ACL entry, the
1106 space needs to be released; hence, the need for for a call to *acl_free()*.

1107 **B.23.7.3 ACL Manipulation on an Object**

1108 Interfaces for manipulating an ACL on an object are provided for reading an ACL
1109 into working storage and for writing an ACL to a file. These functions provide a
1110 type parameter to allow for implementations which include additional types of
1111 default ACLs not defined in the standard. See the rationale for “ACL Storage
1112 Management” for additional information.

1113 An earlier version of the draft contained a requirement that modifying an an ACL
1114 on an object and removing a default ACL from a directory be implemented as
1115 “atomic operations”. The specific requirement was that the operations be atomic
1116 with respect to the invocation and termination of the function calls and any use of
1117 the ACL (access or default ACL). There was also the requirement that changes to
1118 an existing access or default ACL could not result in any intermediate state such
1119 that both the original ACL and the result ACL were both associated with the tar-
1120 get file. While these requirements are certainly necessary, they are requirements
1121 upon the implementation, not the functional interface. As such, it is left to the
1122 implementation to define and enforce its own atomicity requirements. In addition
1123 to not being an interface issue, such atomicity requirements are inherently non-
1124 testable. As such, it is unreasonable to require the construction of tests to

1125 demonstrate conformance these atomicity requirements. For these reasons, all
1126 atomicity requirements were removed from the *acl_delete_def_file()*, *acl_set_fd()*,
1127 and *acl_set_file()* functions.

1128 **B.23.7.4 ACL Format Translation**

1129 There are three formats of an ACL visible to the programmer:

1130 (1) An internal representation that is used by the ACL interfaces.

1131 (2) A self contained data package which can be written to audit logs, stored
1132 in databases, or passed to other processes on the same system.

1133 (3) A **NULL** terminated text package (string) that can be displayed to users.

1134 The ACL copy and conversion functions provide the means to translate an ACL
1135 among the various ACL representations.

1136 The **NULL** terminated text package may contain a representation of an ACL in
1137 either a long text form or a short text form. The following is an example of a valid
1138 ACL in the long text form:

```
1139                     user::rwx
1140                     mask::rwx
1141                     user:jon:rwx
1142                     user:lynne:r-x
1143                     user:dan:---
1144                     group::rwx
1145                     group:posix:r-x
1146                     other:---x
```

1147 The following is a representation of the same ACL in the short text form:

```
1148 u::rwx,m::rwx,u:jon:rwx,u:lynne:r-x,u:dan:---,g::rwx,g:posix:r-x,o:---x
```

1149 The working group considered using the self contained data package as the inter-
1150 nal representation of an ACL. The working group rejected this option for the fol-
1151 lowing reasons:

1152 (1) Implies that some implementations would have to translate an internal
1153 form into a self contained form on every POSIX.1e compliant ACL opera-
1154 tion.

1155 (2) The programmer has to keep track of the size and location of the ACL
1156 with every operation that can modify the ACL. The size must be tracked
1157 because the ACL size may grow or shrink. The location must be tracked
1158 because an ACL may not be able to grow in its present location and
1159 would have to be relocated.

1160 **B.23.7.5 Function Return Values and Parameters**

1161 The *acl_get_**() functions can return pointers, descriptors and discrete values. If
1162 an *acl_get_**() function returns a pointer, then it is returned as the function
1163 return value. This is because a **NULL** pointer is valid indicator for error

1164 conditions. If an `acl_get_*`() function returns a descriptor or a discrete value, then
1165 it is returned as a write-back parameter. This is because there is not a well
1166 defined value that can be returned to indicate that an error has occurred.

1167 **B.23.7.6 File Descriptor Functions**

1168 The working group decided to specify functions that operated via file descriptors
1169 in addition to functions that operated via a file name. These functions allow an
1170 application to open an object and then pass around a file descriptor to that object
1171 instead of both the name and the file descriptor. BSD has found the related
1172 `fchdir()`, `fchmod()`, `fchown()` and `fchroot()` interfaces to be useful.

1173 **B.23.8 Header**

1174 Values for `acl_perm_t` are defined in the header because no definitions in POSIX.1
1175 were suitable. Those definitions considered in POSIX.1 were:

- 1176 (1) Definitions in POSIX.1, 5.6.1.2. These definitions refer to the nine per-
1177 mission bits whereas ACL entry permissions have only three values.
1178 (2) Definitions in POSIX.1, 2.9.1. These names, e.g., `R_OK`, were not
1179 appropriate for ACL entry permissions.

1180 **B.23.9 Misc Rationale**

1181 **B.23.9.1 Objects Without Extended ACLs**

1182 This standard specifies that each file will always have an ACL associated with the
1183 file, but does not require each file to have an extended ACL.

1184 Originally, the provided ACL functions allowed for returning [ENOSYS] if
1185 `{_POSIX_ACL}` was defined and the specified file cannot have an extended ACL.
1186 This was subsequently changed because of objections to the overloading of
1187 [ENOSYS] to return [ENOTSUP] for the cases where a file cannot have an
1188 extended ACL.

1189 A `pathconf()` variable `{_POSIX_ACL_EXTENDED}` is provided to allow applica-
1190 tions to determine if a file can have an extended ACL. This standard does not
1191 specify the specific situations where a file cannot have an extended ACL. Exam-
1192 ples of possible situations are: CD-ROM file systems, and pre-existing file systems
1193 with insufficient space to insert extended ACLs. The `acl_get_fd()` and
1194 `acl_get_file()` functions will always return an ACL because each file will always
1195 have an ACL associated with the file. The `acl_delete_def_file()`, `acl_set_fd()`, and
1196 `acl_set_file()` functions can return [ENOTSUP] if the specified file cannot have an
1197 extended ACL.

1 **B.24 Audit**

2 **B.24.1 Goals**

3 The goals for the POSIX.1e audit option are:

4 (1) Support for Portable Audit-generating Applications

5 (a) Define standard interfaces for applications to generate audit
6 records.

7 (b) Define standard interfaces for applications to request that the sys-
8 tem suspend its generation of audit records for the current process.

9 (c) Define capabilities for these interfaces.

10 (2) Support for Portable Audit Post-processing Applications

11 (a) Define a standard format for system- and application-generated
12 audit records, as viewed through audit post-processing interfaces.

13 (b) Define a minimum set of the POSIX.1e interfaces which shall be
14 reportable in a conforming implementation.

15 (c) Define a standard set of record types, corresponding to the report-
16 able POSIX.1e interfaces, and the required content of those record
17 types as viewed through the audit post-processing interfaces.

18 (d) Define standard interfaces for reading an audit log and processing
19 the audit records that are read.

20 (3) Extensibility for Implementation-specific Requirements

21 (a) Ensure that standard reading and writing interfaces allow
22 specification of arbitrary data in application-defined audit records.

23 (b) Allow for reporting of additional implementation-defined events by
24 conforming implementations.

25 (c) Ensure that standard definitions of the content of required audit-
26 able events allow for extension by conforming implementations.

27 (d) Define standard interfaces for access to implementation-specific
28 audit storage mechanisms (audit logs).

29 The auditing interfaces specified by this standard are intended to be compatible
30 with the auditing requirements of a number of specifications, including but not
31 limited to the U.S. TCSEC levels C2 and above and the European ITSEC func-
32 tionality levels F-C2 and above. It should be noted that this compatibility extends-
33 only to the functional specifications; and also that meeting the requirements of
34 this standard would not necessarily be sufficient to meet all of the audit require-
35 ments of any of the above specifications.

36 There was recognition by the working group that it should be possible for a
37 number of differing implementations to be developed all meeting the POSIX.1e
38 audit requirements. Additionally, consideration was given to the fact that

39 implementations may (will) wish to extend the set of audit functions, audit events
40 and audit records in various ways. For these reasons, flexibility in the POSIX.1e
41 audit requirements was a primary goal.

42 In developing the POSIX.1e audit functions, the working group envisaged two dis-
43 tinct types of auditing applications. First were the class of applications which
44 need to generate their own audit data. These applications, usually trusted, should
45 be able to generate audit data in a standard audit log, rather than simply adding
46 data to an application specific log file. Second were the class of applications that
47 process audit logs. These analysis tools typically read, analyze and produce
48 reports based on the audit data contained in the log. Optimally, these tools
49 should be able to read and analyze audit logs from any POSIX.1e audit conform-
50 ing application. Currently this goal is only partially met. The POSIX.1e audit
51 option provides functions which could be used to develop a audit analysis tool,
52 however, a common (portable) audit log format is not currently defined by this
53 standard. Note that the POSIX.1e audit option specifies only the functions which
54 an analysis tool would use, not the tool itself. The definition of a portable post-
55 processing utility is left to a later stage, when security administration utilities are
56 standardized.

57 **B.24.1.1 Goal: Support for Portable Audit-Generating Applications**

58 Commonly, portable applications, for example a data base, generate and record
59 application specific audit data. Preferably, this data should be recorded in a sys-
60 tem audit log rather than maintaining application-specific log files, or, worse, just
61 ignoring security-relevant events as is common today. It is clearly more desirable
62 for applications to use the standard system auditing mechanism than for each to
63 invent its own.

64 In support of this goal, POSIX.1e audit provides a set of portable interfaces which
65 an application could use to construct audit records and deliver them to an
66 appropriate destination. In some cases it may be desirable to have these records
67 added directly to the system audit log while in other cases a separate log may be
68 required.

69 In order to provide maximum flexibility, the ability to support multiple audit logs
70 has been provided. Applications get access to logs (other than write access to the
71 current system audit log) via the POSIX file abstraction: that is, the POSIX.1
72 *open()* function is used. An additional function, *aud_write()*, is provided to allow
73 records to be added to an audit log by self-auditing applications, since records
74 written will normally have additional data added to them, and may be
75 transmuted into some internal format, by the system in a way which is not con-
76 sistent with the normal semantics of *write()*. A file descriptor parameter is nor-
77 mally used to tell this *aud_write()* interface which log is the destination, but a
78 special value is defined to identify the system audit log (see “Protecting the Audit
79 Log” below for rationale for this).

80 Records of security-relevant events, generated by an application, often relate to
81 actions performed by, or on behalf of, a process (ie, acting as a subject), on one or
82 more objects. The record needs to be structured so that the data that relates to

83 the subject, or a particular object, or other aspects of the event, can be related
84 together: for example, if the record contains a UID, it needs to be clear which sub-
85 ject or object it is related to. The standard therefore provides means for an appli-
86 cation to build structured audit records, with separate sections for each subject or
87 object. Such records can be quite complex, and it would be inefficient if the appli-
88 cation had to build each one from scratch. The standard therefore provides means
89 for the application to alter fields within a record it has constructed, allowing reuse
90 of records.

91 In general, applications that generate audit records will also perform operations
92 that cause the system to record audit records on their behalf. For example, a data
93 base may open several files in normal course of action. For some applications,
94 these system-generated records may be irrelevant and confusing, because the
95 application itself might generate records that are more precise and informative.
96 Therefore a provision is made to allow these, presumably trustworthy, applica-
97 tions to request that recording of system-generated records be suspended because
98 they will provide their own. To ensure the integrity of the audit log, appropriate
99 privilege is required to request suspension of audit records. Also note that this is
100 a “request” to suspend the generation of audit records; an implementation is free
101 to ignore this request.

102 **B.24.1.2 Goal: Support for Portable Audit Post-Processing Applications**

103 The working group recognized that a practical need for audit analysis tools, appli-
104 cations which read, analyze and formulate audit reports, existed. Additionally, to
105 be of maximum value, these tools must be able to access and analyze audit logs
106 from any conforming implementation. Currently, few audit analysis tools exist,
107 and none of the tools examined by the working group were very sophisticated. It
108 is therefore difficult to determine what functions are required for these analysis
109 tools to function adequately. The working group determined that, at minimum,
110 an analysis tool would need to access (open), read and terminate access (close) to
111 the audit log.

112 In Draft 14 the working group recognized the need to make audit records avail-
113 able as they are committed to the audit log. The group felt that tools such as
114 intrusion detection programs would require such a feature. The function
115 *aud_tail()* was added to allow an application to request that records be made
116 available to it as they are being written. However, it was later pointed out that
117 the required effects could be obtained without use of a specialized interface: for
118 example, an intrusion detection application could read from the end of the file
119 currently used for the system audit log, using mechanisms similar to tail(1); and
120 it could be told by the administrator (or other software) when the file correspond-
121 ing to the system audit log gets altered. Accordingly, the interface was removed
122 again. (There was some concern that this might result in records not being
123 delivered for analysis until after a delay due to system buffering, but this was felt
124 to be an implementation matter.)

125 The working group considered the addition of functions to query (selectively read)
126 the audit log but rejected the idea for several reasons:

- 127 1. Understanding of need. The group could not determine what type of query
128 functionality would be required by a portable analysis tool. Lack of market
129 models made the task more difficult.
- 130 2. Defined query language. The group was unable to locate an agreed upon
131 standard language for formulating a query. The working group was reluc-
132 tant to invent a query language for POSIX.1e audit.
- 133 3. Extraneous functionality. The working group felt that as long as an analysis
134 tool could access the next sequential record, that an analysis tool could pro-
135 vide its own query capability.
- 136 In addition to a set of common functions, a portable analysis tool may need to
137 read and analyze audit logs from various sources. Thus, a portable tool may be
138 dependent upon the definition of a standard audit record format. This standard
139 does define a set of standard audit events, and the required record content for
140 those events; it also defines means by which additional information in those
141 records, and information in other records, can be obtained in a syntactically
142 meaningful way.
- 143 Early versions of this standard contained requirements for storage of data in a
144 standard form. This form proved to be unacceptable for most implementations,
145 which have varying requirements for efficient storage of audit data. The working
146 group decided to allow for storage of data in “native format” by default with an
147 option to record data in a “portable format”, to be defined. Without this inter-
148 change format, analysis of audit data across multiple storage implementations
149 requires the application to do several conversions; from native format to human
150 readable text (e.g., internal to external), gather the data on a single machine and
151 then convert the human readable text to internal format (e.g., external to inter-
152 nal).
- 153 Since the portable audit log definition has yet to be developed, a possible goal of
154 support for portable audit post-processing applications is currently satisfied only
155 in part, primarily by defining functional interfaces to audit data.
- 156 In addition to the definition of standard functions, POSIX.1e audit also defines a
157 set of standard audit events. These events, based on standard POSIX.1 and
158 POSIX.1e interfaces, define the minimum data elements to be supplied by a con-
159 forming implementation when the event occurs (assuming auditing is enabled).
- 160 Events generated by standard POSIX.1 operations are defined to ensure that a
161 portable analysis tool has some common ground in any system, although in prac-
162 tice, application-specific analysis tools (using standard interfaces to read
163 application-specific data) will probably be fairly common. By defining the event
164 types in this standard, a consistent mapping across all conforming systems is
165 achieved.
- 166 There was some debate on whether to include events related to the relatively
167 small set of POSIX.2 interfaces that are (arguably) security-relevant. However, a
168 POSIX.2 interface is not necessarily built over POSIX.1; conversely, a POSIX.1
169 system does not necessarily provide POSIX.2 commands and utilities. There is
170 thus no basis for defining POSIX.1e audit events for the POSIX.2 interfaces. The

171 following were also seen to be reasons for excluding these events:

- 172 1. If a POSIX.2 implementation is built over POSIX.1, many of the POSIX.2
173 interfaces are adequately audited by the underlying audit events: eg,
174 chmod(1) is adequately audited by the events for exec(2) of the command and
175 chmod(2).
- 176 2. The most important security relevant commands, such as login, are not
177 included in POSIX.2; those that are administrative are generally deferred to
178 the POSIX 1387 working group.
- 179 3. In many cases, the commands that are included in POSIX.2 are not the ones
180 that need to be audited. For example, it is not particularly relevant that a
181 user has requested that a file be printed, or a batch job be started; what is
182 relevant is the actual printing or starting of the job, which may or may not
183 occur. POSIX.2 does not define the means by which these latter actions
184 actually occur, any more than it specifies login or administrative interfaces,
185 so it is not possible to standardize audit records for these occurrences.

186 The working group had debated including commonly known functions such as
187 login, cron, etc to the set of standard events. However, the majority of the working
188 group felt that adding non-POSIX events was not acceptable because (a) while
189 these events were “common” they were not “standard”, hence the “common”
190 events were not deemed acceptable for inclusion and (b) systems which did not
191 support these “common” functions would still have to support all the POSIX.1e
192 audit event types. Additionally, there was some variance between the implemen-
193 tations of the “common” events. For these reasons the working group decided to
194 limit the scope of POSIX.1e audit events to the domain of POSIX standards.

195 The working group debated the set of included events at great length; the goal
196 was to include only those events which were security related and/or critical to the
197 audit log. For example, consideration was given to including AUD_READ in the
198 set of auditable events, however, it was felt that the information deemed desirable
199 would be obtained by auditing the opening of the audit log. The AUD_WRITE
200 event was also debated, with similar results (except that it was decided to audit
201 AUD_WRITE failures). The working group felt that the amount of information
202 derived from events such as these did not justify the potential performance
203 penalty (e.g., auditing each read/write). Consideration was given to making these
204 events optional. The group felt that the concept of “optional” events had little
205 value because portable applications could not depend on the events being sup-
206 ported (because the events were optional) and hence the “optional” events would
207 be of little use.

208 **B.24.1.3 Goal: Extensibility for Implementation-Specific Requirements**

209 It is important to allow applications to generate arbitrary records. Rather than
210 having a single generic record, however, applications are permitted to place infor-
211 mation in audit records that, while application-specified, has existing syntax asso-
212 ciated with it that allows an analysis program to process the information. For
213 instance, an application refers to a file by pathname, and because there is a

214 standard way to describe a file in an audit record, an analysis program can select
215 records concerning a particular file without knowing anything about the applica-
216 tion generating the record that mentions the file.

217 Similarly, it is important to allow applications to specify arbitrary information in
218 audit records, because not all the items an application needs to specify will be of
219 the sort that can be interpreted in a portable way. The set of audit attributes is
220 extensible to allow this, and additionally includes an explicitly defined opaque
221 data object for application use.

222 Not all applications will want to use the system audit log; indeed, a particular
223 implementation may not permit such use. So, it is important to allow implemen-
224 tations to provide other audit logs. Because the POSIX file abstraction provides
225 defined interfaces without mandating any particular implementation mechanism,
226 it is appropriate to use this for access to audit logs. Some proposals for this stan-
227 dard specified that audit logs were independent of the normal file systems, having
228 their own set of interfaces (e.g., *aud_open()*, *aud_close()*) however these were not
229 seen to provide any particular advantages.

230 Apart from the above application-oriented considerations, it is important that
231 implementations be able to extend the set of auditable system interfaces, and to
232 extend the set of data that is reported in audit records for the standard auditable
233 interfaces. They will thus be able to report the occurrence of security relevant
234 events that are beyond the current scope of ratified POSIX.1 standards, and to
235 record additional security information for the standard events.

236 **B.24.2 Scope**

237 The scope of security auditing specifications in POSIX.1e is defined by the above
238 goals. In addition, the following items are specifically excluded:

239 (1) Administration
240 Functions and utilities to support security audit administration are
241 excluded. These exclusions include the assignment of audit control
242 parameters to specific users, and pre-selection of which auditable events
243 are to be recorded.

244 (2) Audit data storage
245 The definition of formats and organization for permanent audit data
246 storage is not addressed, nor is there any required storage organization
247 for a system's audit log.

248 (3) Portability/Data interchange
249 The definition of formats and organization required for a portable audit
250 log and for interchange of audit data are not addressed.

251 (4) Audit delivery mechanism
252 The definition of a mechanism for delivering records is not addressed,
253 although the interface to this mechanism, 24.4.40, is included.

254 Administrative functions are excluded from the POSIX.1e auditing scope, these
255 are the province of POSIX.7.

256 The specification of criteria for the pre-selection of which audit records should be
257 recorded is deemed to be an administrative issue. It was felt that portable
258 trusted applications could not reasonably make use of interfaces to control pre-
259 selection.

260 A grouping of event types into *classes* of events for post-processing were excluded
261 from the scope because it was felt that not enough is currently known about post-
262 processing to allow a solid set of post-processing classes to be included in the
263 POSIX.1 standard. The group felt there were two compelling reasons why it was
264 inappropriate to standardize event classes: (1) the grouping of events into classes
265 is inherently arbitrary; while the group could easily agree on a standard set of
266 common events (based on POSIX.1) the grouping of these events into classes dif-
267 fered widely, (2) the definition of classes does not add greatly to application porta-
268 bility because the event type rather than class is what is stored in the audit
269 record.

270 This standard does not address audit data storage. It is expected that each con-
271 forming implementation may have a different form of permanent storage for audit
272 data. Similarly, the issues of interchange of audit data are not addressed. A key
273 problem in the definition of data interchange is that current standards do not
274 address data size issues at all.

275 This standard does not address the actual mechanism for delivering audit records
276 from a trusted application (or from the operating system itself) to a system's audit
277 log. However, the interfaces that an application (or the operating system) would
278 use to perform the delivery are specified. An actual delivery mechanism might
279 involve spooling daemons, special network protocols, etc.

280 This standard also does not address the issue of protection of the audit data, that
281 being an implementation's responsibility (see below for further rationale for this).

282 **B.24.3 General Overview**

283 In this standard, the general architecture for audit record processing is that the
284 internal format of audit records is opaque, and functional interfaces are provided
285 both for audit-generating applications to construct audit records (adding, chang-
286 ing and deleting fields) and for audit post-processing applications to analyze
287 records (reading fields). The system manages the working storage used to hold
288 the record; interfaces are provided to create new (empty) records in the working
289 store, to read records from an audit log into the working store, and to write
290 records from working store into an audit log.

291 An earlier version of this document used explicitly different storage representa-
292 tions for data structures used in reading records and in writing records. Writing
293 records used opaque storage (called an Audit Record Descriptor), whereas reading
294 used a caller-supplied buffer that was implied (but not required) to be a directly
295 accessible storage representation of the portable audit record format. In princi-
296 ple, this would have allowed a processing program to have performed manipula-
297 tions directly on the record contents, without using the reading interfaces.

298 A major criticism of this proposal was that it required that all data should be
299 written in a portable format that was biased toward machines that support
300 expanded data types. In abandoning the requirement that all audit data should
301 be stored directly in the portable format, it became impossible to provide this abil-
302 ity. It also became apparent that the defined set of interfaces had become
303 sufficiently complete and efficient that the ability was no longer important.

304 The original proposal defined audit records as consisting of individual “tokens”
305 where a “token” represented an independent element of a record, for example a
306 *pathname*. To make the token opaque all manipulation of the token (read/write)
307 was done using per-token interfaces. For example *get.pathname_token* and
308 *put.pathname_token* would be required to get (read) and put (write) a specific
309 token. It is easy to see how this style of interface could lead to an excessive
310 number of token types and in turn, an excessive number of interfaces required to
311 manipulate each token type. There was also the possibility of inconsistent use of
312 the tokens by applications performing their own auditing. The concerns regarding
313 efficiency of storage and number of interfaces led to the replacement of the “token
314 based” proposal.

315 In draft 13, self-auditing applications were required to construct audit records in
316 user-managed storage, because the user (application) knows the size and contents
317 of the record, and there is no point in making the data opaque. Also, the record
318 may be used as a “template”, that is the record may be modified and written mul-
319 tiple times without requiring multiple allocate/free operations of system managed
320 storage. However, this proposal was criticized in ballot for not providing either
321 sufficient record structuring capabilities or sufficient support for portable applica-
322 tions; extending the proposal to provide additional structuring would add consid-
323 erably to the complexity of the data structures applications would have to mani-
324 pulate (giving problems in some language bindings), and would exacerbate the
325 second criticism. In contrast, system-managed storage was used for reading
326 records, because in many cases the application will rapidly eliminate most records
327 from the analysis, and keeping them in system-managed space saves the cost of
328 converting the whole of each record from an internal to a standard format. Also,
329 programs reading records are likely to be processing many records sequentially,
330 and correspondingly benefit from eliminating application-level storage manage-
331 ment overhead.

332 The current set of interfaces and corresponding data structures have been
333 designed to provide reasonable application support with reasonable efficiency,
334 without an excessive number of interfaces. Data storage representations are not
335 defined. The interfaces deal with opaque structures at the top level, and indivi-
336 dual components at a lower level; the latter use ‘get item’ interfaces, and a ‘type
337 length pointer’ data structure, thus providing flexible functionality through a
338 small number of interfaces. The interface for application generation of audit
339 records similarly uses ‘put_item’ interfaces and the ‘type length pointer’ structure
340 to specify the data to be recorded. Several tradeoffs exist, as described below, and
341 these are not the most efficient interfaces imaginable; merely the most efficient
342 portable interface proposed so far.

343 One tradeoff exists in the granularity of information access to the audit record. An
344 audit record consisting of individual attributes is the more general interface but
345 also is more inefficient. Structure-based interfaces that put and get information in
346 large chunks are more familiar to programmers but it may be more difficult to
347 validate the attributes; and structures are inconvenient if there are a large
348 number of variable size components (or components with opaque structure that
349 may be variable size).
350 Another tradeoff is caused by offering only indirect access to the audit record,
351 because the information must be retrieved procedurally. The cost could be minim-
352 ized by implementing these interfaces as macros and a procedural interface
353 allows an implementation greater flexibility in defining audit log storage and
354 access methods.

355 **B.24.4 Audit Logs and Records**

356 **B.24.4.1 Protecting the Audit Log**

357 Of all the data in a secure computing system, the audit log is perhaps the one
358 item which is most important to protect against invalid manipulations EVEN by
359 apparently authorized users. For instance, if an intruder can defeat a system's
360 access control mechanisms, and assume all the rights and powers of an author-
361 ized system administrator, it would still be extremely useful to be able to audit
362 the intruder's activities. To any extent possible, the auditing mechanism and the
363 audit log should be protected against external attacks.

364 The group considered specifying a few possible mechanisms that provide elements
365 of protection against this threat, but decided not to do so. The group took this
366 position because any mechanism that is sufficiently general (not implementational-
367 dependent) to specify in a standard would not, itself, provide significant protec-
368 tion. Only a combination of mechanisms, most of them implementational-
369 dependent and outside the scope of POSIX, can protect a system's audit log to a
370 meaningful degree beyond basic file protection.

371 If the audit file is protected using the normal filesystem protection mechanisms,
372 the degree of protection increases with the security of the system. Thus in an
373 ACL based system with a single super-user, it could be read/write to superuser
374 only. On a system with the administrative roles divided according to the principle
375 of least privilege, it could be owned by the audit administrator, with read access
376 available also to the security administrator. On a system with MAC controls of
377 disclosure and integrity, it could be owned by audit administrator with a disclo-
378 sure label making it readable only to security and audit administrators, and an
379 integrity label making it writable only by the system. Of course, these access con-
380 trols do not prevent the audit subsystem itself from writing to the audit log to
381 record actions of users, even though the users don't have write access to the audit
382 log file.

383 Thus when audit logs are accessed via the POSIX file abstraction, this standard
384 does not mandate any protection mechanism other than the normal file system
385 access control mechanisms. The exception to this occurs in the case where an

386 application needs to write to the current system audit log. There are two reasons
387 why it would not be appropriate to rely on the usual file protection mechanisms,
388 exercised through *open()*, in this case. Firstly, a self-auditing application should
389 generally not have the ability to open the system audit log for write, since this
390 would confer the ability to corrupt data that was already in the log, for example
391 by writing random data at random positions in the log. Thus in this case an alter-
392 native means of accessing the log is needed to ensure its integrity. Secondly, an
393 implementation may not have a fixed mapping between the current system log
394 and a POSIX file: either the log data may be sent to different files at different
395 times (e.g. when the current file reaches a certain size), or the data may not be
396 sent to a medium that is accessible through a POSIX file name. Therefore this
397 standard specifies that the current system log is written without use of *open()*,
398 and uses appropriate privileges as the means to control access to that log.

399 The working group debated whether self-auditing applications should be permit-
400 ted to provide all the data of an audit record, some people holding the view that
401 the system should be required to provide some of the data (especially in the record
402 header) in order to protect the integrity of the audit log and provide accountability
403 for application-generated records. However, others held that it is only necessary
404 to protect the integrity of the audit log, and that the application is trusted to
405 create the entire contents of the audit record itself - some even suggested that the
406 application should not even have to be privileged to do this. The final consensus
407 took the ‘middle way’: that the integrity of the audit log should be protected (by
408 allowing applications to write records without giving them general write access;
409 and by allowing the system to check the format of audit records); and that only
410 ‘trusted’ applications should be able to write records, the control being provided
411 by use of appropriate privilege. The latter control allows implementations, or
412 even installations, to set their own policy about the degree of trust needed in self-
413 auditing applications, since they can control how widely the privilege to write
414 audit records is distributed.

415 **B.24.4.2 Audit Log and Record Format**

416 The logical audit log is a stream of audit records. That is, an audit log appears to
417 the application program as a sequence of discrete, variable length records. Each
418 record contains a complete description of an audit event: records are intended to
419 be largely independent entities. An important distinction must be made between
420 the “logical” and “physical” descriptions of the audit record. The “logical” appear-
421 ance of the audit log refers to the appearance of the audit records returned by the
422 functions defined by this standard. The “physical” description of the audit record
423 refers to the audit record as it exists in the audit log, that is how the record would
424 appear if the audit log were read in its raw state. This standard does not define
425 the “physical” view of the audit log. Additionally, this standard does not define
426 the “logical” view of audit records when viewed by interfaces not defined by this
427 standard.

428 B.24.4.3 Audit Record Contents

429 The statement above that audit records should be largely independent is an ack-
430 nowledgement that no audit data can be completely context-independent, and an
431 encouragement that audit records contain enough context to be meaningful for
432 analysis in most circumstances.

433 Each audit record contains at least a header and a set of subject attributes (the
434 term ‘subject attributes’ is used in preference to ‘process attributes’ because a pro-
435 cess can also be an object (e.g. when receiving a signal), and also the particular
436 set of attributes reported is that appropriate to the process’s role as a subject, as
437 opposed to all the attributes of the process). Most records also contain one or
438 more sets of event specific data, and zero or more sets of object specific informa-
439 tion. The header defines a version number (see below), the data format the record
440 is written in, and includes fields for event type, event time, and event status. The
441 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.

452 The format field specifies the format of the data contained in the audit log.
453 Currently, only the format AUD_NATIVE is supported. The AUD_NATIVE for-
454 mat indicates that the audit data contained in the log is written in native
455 machine format. This field is primarily a place holder for future revisions of this
456 standard which are expected to add other formats such as a portable audit format.
457 It is important for the portable application to know what type of data is written in
458 the field, thus the application knows what kind of data to expect (i.e., byte order-
459 ing, data type sizes, etc.)

460 The status value was added to indicate the status of the audit event with some
461 indication greater than success or failure. The following event statuses are
462 currently defined:

463 AUD_SUCCESS The event completed successfully.

464 AUD_PRIV_USED The event completed successfully and privilege was exer-
465 cised. Conforming implementations are not required to
466 report this value (reporting AUD_SUCCESS instead), since
467 not all audit policies require that use of privilege be audited.
468 If the value is reported, however, this does imply that
469 privilege was required, not just that privilege was available
470 and was used. The working group felt this distinction was
471 important because although some implementations may not
472 need to distinguish between a privilege which was used and

473 a privilege which was required, existing practice has shown
474 that for security auditing it is important to report the use of
475 privilege to achieve an operation that would have failed
476 without it.

477 AUD_FAIL_DAC The event failed because of discretionary access control
478 checks.

479 AUD_FAIL_MAC The event failed because of mandatory access control checks.

480 AUD_FAIL_PRIV The event failed because of lack of appropriate privilege. –
481 The audit record does not contain an indication of what the
482 appropriate privileges were, though if the POSIX capability
483 option is in use it does indicate the capabilities available to
484 the subject, and other security attributes of the subject and
485 object; thus it would be possible to deduce which capabilities
486 would have been needed to complete the operation.

487 AUD_FAIL_OTHER The event failed for some reason, none of the above. This
488 includes implementation-defined policy extensions.

489 Note that implementations are free to extend this list with additional status
490 values. Note also that the standard does not define which of the various
491 AUD_FAIL statuses is to be returned if the event could have failed for more than
492 one reason: if this were specified it would imply that implementations had to per-
493 form tests in a certain order, or carry out all tests even if one had already failed,
494 and the working group did not think this a reasonable requirement.

495 The audit record header includes an identifier, the *audit ID*, for the individual
496 human user accountable for the event: it is a fundamental principle of accounta-
497 bility that each event should identify the human user accountable for it (see below
498 for further rationale related to audit IDs). For system-generated events, if the
499 process initiating an action does so on behalf of a user who is not directly associ-
500 ated with the process (e.g., a server process acting on behalf of a client) the
501 directly accountable user should probably be the one that initiated the server.
502 However, if there is no accountable user (e.g., the server was started automati-
503 cally at system initiation) then the standard does allow the system to provide a
504 null audit ID. For application-generated records, the standard specifies interfaces
505 that allow a server process to record the audit ID of the client process for which it
506 is acting.

507 The subject attributes are required to include the process ID and the basic secu-
508 rity attributes of the subject: the effective UID and GID; means for reporting
509 other security attributes (e.g., supplementary groups, labels, capabilities) is also
510 provided. The working group considered requiring that all these attributes be
511 present (at least if the relevant POSIX options are implemented) but rejected this
512 because it was not clear that all systems implementing audit would need to pro-
513 vide this information. It is a matter for the policy of the system. Accordingly the
514 standard defines how the information can be provided, and what happens if it is
515 not, and allows implementations to decide on policy.

516 The object specific information includes fields for the type and name of the object,
517 and object security attributes. Again, some of the security attributes are optional,

518 it being up to the implementation security policy to define whether they are pro-
519 vided. In general, the standard requires that object details be supplied whenever
520 the attributes or data of an object may be accessed or altered; it does not require
521 it otherwise (for example, on a *chdir()*).

522 The audit events for interfaces that operate on files via file descriptors include the
523 fd among the data reported. There was some feeling that this was in itself not
524 very useful, since the file descriptor is not directly meaningful to an audit
525 administrator, but the audit record for the *open()* call that created the file descrip-
526 tor is also reportable, and does enable an audit post-processing tool, or audit
527 administrator, to make the link back to a human-readable name.

528 For records that report changes to subject or object attributes, the standard
529 includes the new attributes, through inclusion of the function arguments. It also
530 requires that details of the relevant subject/object are included; it specifies that if
531 the relevant attribute is included in the details, then the old value shall be given.
532 However, it does not generally require that the relevant attribute must be
533 included in the details. There are several reasons for this: not all security policies
534 require that the old attributes be audited; in some implementations there is no
535 reason for the old attribute to be available to the audit subsystem; for some attri-
536 butes there could be a significant performance/space impact (e.g. recording 1000-
537 entry ACLs!). Thus the standard always requires the new attribute to be
538 recorded, and permits (but does not require) the old attribute.

539 **B.24.4.3.1 Semantics of Audit Event Types**

540 The standard includes a set of pre-defined system event types with fixed interpre-
541 tations (corresponding to interfaces defined in POSIX.1). These system event
542 types are defined primarily for use by audit analysis tools such that they can have
543 a base set of defined, standard event types for analysis. It was felt by the working
544 group that a standard means of uniquely identifying these system event types
545 was required to avoid collisions (e.g., various definitions of the same event type);
546 therefore the standard includes a means of identifying the event types them-
547 selves, that is, a standard naming of system event types is provided. The event
548 type defines the minimum logical content of the record as it is returned by the
549 POSIX.1e audit functions.

550 The working group felt that some applications may need to query the list of sys-
551 tem event types supported by a system. For example, a interactive audit analysis
552 tool may want to get all the system event types supported on a system, then
553 prompt the user to determine what event types to analyze the audit log for. This
554 type of capability also requires a interface to convert the audit event type from its
555 internal representation (numeric) to text for display purposes, then from text (or
556 numeric-text) to internal format (numeric). To provide this functionality to an
557 analysis tool the following interfaces were defined: *aud_get_all_evid()*,
558 *aud_evid_to_text()*, *aud_evid_from_text()*.

559 Applications also need some defined semantics for audit events. A portable appli-
560 cation wishing to generate its own audit records must be able to specify the form
561 and content of the record so that it can convey this information to an audit
562 analysis application. Like system events, application events also require some

563 means of identifying the event type.

564 The working group debated how best to define the event types. Some iterations of
565 this standard specified the event types as numeric constants (e.g., 1,2, ... nnnn).
566 The working group felt that a portable analysis tool would be most efficient
567 searching for and comparing numeric event type identifiers. For example, an
568 analysis tool searching for records of type AUD_AET_KILL could simply search
569 for records of event type 1. However, the working group felt that the expression of
570 event types as character strings, e.g., "AUD_AET_AUD_OPEN" allowed for easier
571 future expansion. The standard could thus reserve the AUD_AET_ prefix for
572 future use (as opposed to reserving 1-xxx). The former option was proposed in the
573 first ballot of the standard (attracting ballot objections related to extensibility,
574 and the likelihood of applications choosing the same event types); the latter was
575 proposed in the second ballot (attracting ballot objections related to efficiency of
576 processing and storage). Finally, it was decided to adopt a combination, using
577 numeric identifiers for system events and string identifiers for application events.
578 This accomplished several goals:

- 579 A. System events can be recorded and processed with maximum efficiency.
580 B. Applications wishing to do self-auditing were less likely to have audit event
581 type collisions. For example a database could generate records of
582 AET_<MYNAME>_DB as opposed to records of event type 150. The group
583 felt it was far less likely that two applications would choose the same char-
584 acter string.
585 C. Application event types cannot clash with system event types.

586 **B.24.4.4 Audit Record Data Format**

587 The physical format of an audit record is unspecified - that is, a post-processing
588 application may make no assumptions about the format and location of the
589 header, subject, object and event specific data as it actually exists in the record.
590 Logically, an audit record is a collection of opaque segments (headers, sets of sub-
591 ject attributes, etc) each of which is referred to by a descriptor and accessed only
592 by functions referencing that descriptor.

593 The segments of a particular type in a record are ordered, so that semantics may
594 be attached to their relative positions; this is likely to be particularly important if
595 a record contains details of more than one object, since these may represent the
596 source and sink of data. Descriptors for the various structures can be obtained
597 either serially or by random access (e.g. to the second set of object attributes). |

598 The segments which comprise a system-generated audit record contain at least
599 the data items defined by this standard and may include additional,
600 implementation-defined data items. The data type of each of the required data
601 items is defined by this standard, as is the ordering of the items. Note that the
602 size and byte-ordering of the data items may vary from system to system. That is,
603 there is no intention that the binary data in the opaque structures is directly port-
604 able from system to system.

605 A header segment must be (logically) included in every record. For system-
606 generated records, the fields of it are set by the system; for application-generated
607 records, the application is required to specify values for certain fields, and may
608 supply more (the system will supply certain fields if the application does not).
609 Similarly, the subject attributes of system-generated records are provided entirely
610 by the system, but for application generated records the application is trusted to
611 provide subject attributes, for example of a client process (again, the system will
612 supply 'default' values, describing the current process, if the application provides
613 none). There was considerable debate in the working group about whether the
614 application could be trusted to supply header and subject attributes; some
615 members felt that the system should always provide the header (except for the
616 event type and status) and subject attributes for the current process. However,
617 the alternative view prevailed, that an application that is trusted to generate
618 audit records (in the system audit log) can also be trusted to do it right.

619 Although the number and ordering of segments in the record is important, it
620 should be noted that for failed events, some objects and data in the record for the
621 standard event types may be omitted, because this information may be missing
622 from the function or utility invocation.

623 **B.24.4.4.1 Portable Audit Record Format**

624 The current version of the standard does not contain a definition for a portable
625 audit log format. This is currently being investigated for a future iteration of the
626 standard. Earlier drafts of this standard did contain a portable audit log format.
627 However, the standard required that all records be written in this format which
628 proved to be controversial. The rationale contained here defines the reasons why
629 the working group felt a portable format was necessary.

630 A portable audit log format allows the audit data to be analyzed on systems other
631 than the systems which generated it. Several methods were proposed to place
632 audit records in portable format. One method proposed was to write all audit
633 records in the portable format. This method was rejected because it had the
634 potential to impose performance penalties on those implementations which did
635 not support the data sizes required by the portable format as their "native" data
636 types, in other words, some systems may be required to do size and type conver-
637 sions on each record written.

638 To avoid this unnecessary penalty, the data that is returned in the structures is
639 always in the local format. An alternate method proposed was to allow the audit
640 records to be written in native machine format with the conversion to the portable
641 format to be done by some form of audit record filter. These records can then be
642 transferred to other systems.

643 There are two costs to this approach, however. The first is that each system must
644 be prepared to read the portable format(s) defined. The second is that these
645 records are always translated twice - once on the generating system and once on
646 the system used for analysis.

647 The portable data formats are not defined in this document. That is, the size of
648 uids, gids, MAC labels, etc. is unspecified at this point. While the elements of a

649 portable audit log can be outlined, the definition of the portable audit log format
650 is not defined.

651 Auditing by nature is the gathering of data, not the definition of it. Almost all the
652 data types contained in a typical audit record are external to the audit group.
653 True data portability is a problem much larger than the need for a portable audit
654 log. Currently, neither the POSIX.1 standard nor the POSIX.2 standard address
655 data interchange sufficiently to define a portable audit format.

656 After extensive research and discussion, the audit subgroup has concluded that
657 the portable audit format is a subject that cannot be resolved with the present
658 amount of information obtainable from other internationally recognized standards
659 bodies.

660 Analysis of the problem revealed the following issues, all of which need to be
661 resolved before a portable audit log format can be developed. The issues are:

- 662 (1) Data format (byte ordering)
- 663 (2) Data field sizes (very specific! number of bytes or equivalent)
- 664 (3) Field mappings (user ID <-> user name, etc.)
- 665 (4) Time coherence (time zone, etc.)
- 666 (5) Internationalization issues (at least for text strings contained in the file)
- 667 (6) Byte size
- 668 (7) Field identification and boundaries (how to tell where a record begins
669 and ends)
- 670 (8) Naming convention (uniqueness of user, for example user ID plus process
671 ID)

672 It was decided that the audit log header file needs to contain: an indicator that
673 marks the log as being in POSIX.1e portable format, the version of the standard
674 of the portable format, the data format indicator of the log (XDR, NDR, or ASN1
675 format), the time zone in which the log was created and any applicable maps
676 required by that machine. There may be several machine identifiers and associ-
677 ated maps, keyed by machine_id. Not much more information can be generated
678 without input from the interchange format group.

679 The audit subgroup has also yielded the format of the MAC label, ACL, and capa-
680 bilities associated with the portable audit format to those associated groups.
681 However, they too will be unable to determine the data sizes to be used in a port-
682 able interchange format without input from the interchange format subgroup.

683 **B.24.5 Audit Event Types and Event Classes**

684 The distinction between event types and event classes has generated considerable
685 controversy. Two differing proposals were considered. One suggested that group-
686 ing types into event classes is arbitrary and may differ from one system to
687 another. Another proposal suggested that event types should belong to a small,
688 fixed set of standard event classes. This proposal also suggested that the event
689 class be recorded in a header, with the event type, thus making it the responsibil-
690 ity of the auditing program to fix the relationship between the two.

691 Initially the latter proposal was accepted. However, after further reflection, it
692 was decided that recording the event class in a header was not tenable. If an
693 event type belongs to many classes, but only one can be recorded in a header, then
694 the inclusion of such a value might serve to confuse rather than clarify the reason
695 for the audit record. Eventually it proved impossible to reach consensus on how
696 event classes should be standardized; there was also a body of opinion that said it
697 was unnecessary to standardize them, because post-processing applications could
698 group event types into classes at that level. Accordingly, the concept of event
699 class was removed from the standard.

700 It also turns out to be very hard to define precisely when an event deserves an
701 event type of its own. For instance, are successful and failed **open** calls the same
702 event type? Probably so, because they can be differentiated by the result field in
703 the record header (though looked at another way, that really means that the
704 result field is part of the event type, and so they are two different types).

705 Are open of a file for reading, and open of a file for read/write, different event
706 types? Though they differ only in one bit of a system call argument, maybe they
707 ought to be different types, because they represent very different abilities being
708 exercised. This example leads to a circular definition of event types: two types
709 should be separate when it would make sense to assign them to separate classes.

710 It was finally decided to define no more than one event type for each of the POSIX
711 interfaces being audited; in a few cases a single event type was used for several
712 closely related interfaces (e.g. the *exec()* family). The separation of, eg open-read |
713 and open-write can then be done by post-processing tools on the basis of informa-
714 tion in the record; implementation-specific means could be used to separate these
715 for event pre-selection purposes too (see below).

716 **B.24.6 Selection Criteria**

717 At various times, drafts of this standard have included facilities for both pre-
718 selection and post-selection of audit records: that is, selection of the records that
719 are recorded in the log, and those that are reported from the log to an audit post-
720 processing application. However, the standard does not finally contain any selec-
721 tion facilities. The pre-selection interfaces have been removed because they are
722 seen to be an administrative facility, and therefore out of scope. The post-
723 selection interfaces have been removed on more pragmatic grounds: there was no
724 agreement on what facilities are needed, or how post-selection criteria should be
725 specified. Additionally, the group felt that so long as the next sequential record

726 could always be made available, applications could build selection criteria them-
727 selves.

728 **B.24.7 Audit Interfaces**

729 **B.24.7.1 Gaining access to the Audit Log**

730 In earlier drafts of this standard, to provide some separation of audit log from file
731 the concept of an *audit_log_descriptor* was conceived. The audit log descriptor pro-
732 vides a level of abstraction above the file descriptor interface. An attempt was
733 made to define a set of interfaces for use in analyzing abstract audit logs, conceal-
734 ing the storage method, location and format of the actual data. In draft 13 (and
735 previous drafts) there were two functions provided to initiate and terminate
736 access to the audit log; *aud_open()* and *aud_close()*. However, this resulted in a
737 need to reinvent a complete I/O package for such objects. Also, it did not succeed
738 in defining any particularly useful interfaces, other than a record-oriented read
739 function.

740 In draft 13 several balloters objected to the concept of an "audit descriptor". There
741 were two flavors of objection. One type of objection cited existing practice claiming
742 that existing practice (or all that was known to the objector) used files so the
743 abstraction of an audit descriptor was not reflective of current practice. Another
744 type of objection stated that since the descriptor was largely implementation
745 defined that it was of little use to the portable application. In response to these
746 ballot objections, the *aud_open()* and *aud_close()* as well as all concept of "audit
747 descriptor" was deleted. The *aud_open()*, *aud_close()* and "audit descriptor" were
748 replaced by the P1003.1 *open()* and *close()* calls while the audit log descriptor was
749 replaced with a file descriptor. The result of this change was to make the POSIX
750 audit functions more reflective of existing practice.

751 **B.24.7.2 Distinction Between System Audit Log and Audit Log Files**

752 With the removal of the audit descriptor abstraction some semantic differences
753 between the "system audit log" and file-based audit logs (i.e. non-system logs) sur-
754 faced. The primary difference being the fact that the system has some a priori
755 knowledge of the system audit log while the file-based audit log may only be
756 known by the application. An example of the difference between the "system log"
757 and file based logs lies in the amount of support which may be provided by the
758 system in ensuring the integrity of the audit records and the audit files. In the
759 case of the system log, the system is responsible for ensuring the integrity of the
760 audit log. For example if an application issues an *aud_write()* call on the system
761 audit log, the system is responsible for ensuring that the audit data is eventually
762 written to a properly formatted audit log. The system is also responsible for
763 proper sequencing of the records and supplying any accessory information neces-
764 sary to post-process the record (e.g. UID to text representation). When dealing
765 with a file based audit log the system cannot guarantee that the file specified as
766 an "audit log" is in fact properly formatted (i.e. meets the system's requirements
767 for a proper audit log), that the file offset is correct or that any accessory

768 information required for later translation (by *aud_rec_to_text()*) is properly
769 represented in the file. Additionally, if multiple *aud_write()* calls are made to the
770 file based audit log the system has little control over the sequencing of the
771 records. The only method provided by the standard for providing the concept of
772 "next" record is via the POSIX concept of file append. That is if the file based
773 audit logs are opened with the O_APPEND option the system can provide the
774 assurance that the "next" record written is properly placed in the audit log.

775 **B.24.7.3 Read/Write access to the Audit Log**

776 Appropriate privilege is required to write to the system log, but is not normally
777 required to read it. The rationale for this is that the write interface does not
778 require that the log has previously been opened (because the application should
779 not have unrestricted write access to the audit log, but only the ability to request
780 that records be added to the log (subject to an implementation specific pre-
781 selection policy); indeed, it may not even know the name of the file in which the
782 log is stored). However, for the read interface the log must first be opened, and
783 normal system access controls can be applied.

784 No privilege requirements are placed on implementation-defined audit logs
785 though implementation-defined forms of access control (including privilege) may
786 be applied.

787 **B.24.7.4 Space Allocation**

788 Space allocation for auditing functions is handled by the system throughout, with
789 the user only being required to notify the system when an item is no longer
790 required (by calling *aud_free()*). Functions that create or read in data on behalf of
791 the user automatically allocate space for the data: for example, for records read
792 from audit logs and for text strings created by *aud_id_to_text()*. The only excep-
793 tion is *aud_copy_ext()* which specifically copies into user-created space.

794 **B.24.7.5 Audit Identifiers**

795 The audit ID, an identifier conceptually different from a UID, was introduced as a
796 means of satisfying the requirement for individual accountability. While this
797 requirement can be met in other ways (e.g., unique UIDs) it was felt that the
798 introduction of the audit ID was the best means of meeting the requirement.

799 In many existing systems, the user has a username and a user ID. Neither of
800 these is appropriate for use as the audit ID, because POSIX.1 does not require
801 that either of these be mapped to an individual human user. Further, the user ID
802 is the basis of the (DAC) authorization policy of the system, which is logically dis-
803 tinct from the accountability policy. In particular, some systems allow aliasing of
804 one user ID to several usernames that all have the same DAC authorizations, or
805 permit several users to share a username; this is incompatible with use of the
806 user ID as an audit ID. An audit ID has its own unique type *aud_id_t*, because
807 only by doing this could an audit file be analyzed on systems of a different type to
808 the one on which it was generated. Some implementations might wish to define

809 mappings between *aud_id_t* values and implementation-defined identifiers, such
810 as personnel numbers; this is not subject to standardization.

811 Note that there is nothing to stop a particular implementation from implementing
812 user ID and audit ID for each user as the same value, as long as it maintains individual
813 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
814 is assigned, thus it can be administrator or system assigned (in the latter case,
815 perhaps equal to the UID).

817 Currently, two functions are provided for processing audit IDs. The function,
818 *aud_id_to_text()* is provided to allow an application to convert an audit ID to a
819 string identifying the corresponding individual user. The function,
820 *aud_id_from_text()* is provided to allow an application to convert a string identifying
821 an individual user to an audit ID. The audit option does not define any relationship
822 between the strings handled by these functions and the *pw_name* field
823 obtainable from the function *getpwuid()*. Using these interfaces, an audit post-
824 processing application could provide record-selection facilities that permit an
825 auditor to select records based on the identity of the individual accountable for
826 actions; or could present the identity of the individual responsible for a particular
827 record to the auditor.

828 A further function, *aud_get_id()* is provided to allow a process that generates
829 records of its own activities to obtain the audit ID of the user accountable for the
830 actions of a client process and include it in such records.

831 Note that the functions to set, store and allocate audit IDs are not defined by this
832 standard, since these are considered to be administrative and therefore out of
833 scope.

834 **B.24.7.6 Audit Post-Processing Interfaces**

835 **B.24.7.6.1 Reading the Audit Log**

836 This standard provides a single read function *aud_read()* which operates with a
837 file descriptor returned via *open()*.

838 The *aud_read()* function returns a pointer to the next sequential record in the
839 audit log. Note that it is up to the underlying implementation to ensure that the
840 next sequential record is returned. Certain events occur on a system for which
841 sequence is important. For example, a parent process forks a child. It is possible
842 that audit records from the child may appear in the *physical* log prior to the
843 record indicating the fork event had occurred. In any case, it is important that the
844 record for the parent's fork is returned prior to any subsequent records for the
845 child (provided, of course, that the implementation-specific pre-selection policy
846 causes the fork event to be recorded). While the records in the internal audit log
847 may not be in the proper logical sequence, the sequence returned by *aud_read()*
848 must reflect the proper sequence.

849 Note that if an application chooses to write its own audit records to a file-based
850 audit log (e.g. not the system log) it is left largely up to the application to ensure |
851 that the records are properly sequenced. The only mechanism provided by POSIX

852 for maintaining the sequencing of records written to a file-based audit log is via
853 the O_APPEND flag supplied on *open()*.

854 Since the system is not controlling the file-based audit log there may be no addi-
855 tional (system supplied) sequencing information provided.

856 **B.24.7.6.2 Parsing Audit Records**

857 An audit log may contain records in multiple data formats. All data in any given
858 record will be of the same format. The only format currently defined is
859 AUD_NATIVE; previously an AUD_PORTABLE format specifier was also
860 included, but this has been removed in the current draft because of the decision to
861 delay addressing the issue of portable data interchange formats.

862 A previous draft stated that access to the sets of data within the various sections
863 (headers, subjects, event-specific data and objects) of an audit record and to the
864 individual fields within these sets was sequential, i.e., to get to the nth field
865 required reading all the fields up to that one also. Several objections were made
866 to this claiming that it was both restrictive and inefficient: it prevented the read-
867 ing of the fields or sets in an arbitrary order and it required the processing of
868 fields or sets that were not needed. To respond to these objections, a third param-
869 eter has been added to the aud_get_*(*)* and the aud_get_*_info(*)* functions, where
870 * is one of hdr, subj, event or obj.

871 For the aud_get_*(*)* functions, this parameter represents the ordinal number of
872 the set being requested in the appropriate section. This allows random access to
873 the sets, while at the same time allowing all of the sets in a section to be pro-
874 cessed sequentially.

875 For the aud_get_*_info(*)* functions, the third parameter represents a field_id,
876 identifying the field being requested; for system-generated records there are
877 defined values of field_id for each item; and the interfaces for construction of
878 application-generated records allow the application to specify the field_id for each
879 item (see below). Thus the field_id allows access to specific fields within the set.
880 Note that field_ids are not necessarily sequential. In addition, two special
881 field_ids, AUD_FIRST_ITEM and AUD_NEXT_ITEM are provided to allow
882 sequential access to the fields within a set. This can be used for rewinding a set.
883 Thus, both random and sequential access to the fields in a set are provided.

884 Note that the *aud_get_**(*)* interfaces operate on audit record descriptors as
885 returned by any of *aud_read()*, *aud_init_record()* and *aud_dup_record()*. The
886 decision to use symmetric interfaces allows applications greater latitude in pro-
887 cessing a record and allows the implementation to be considerably simplified
888 because separate writing functions are not needed for records that are read from
889 the log as opposed to those that are created from scratch.

890 As mentioned above, the *aud_read()* function returns a pointer to an opaque
891 structure defining the next sequential record in the audit log. This record is then
892 read in logical pieces: the record header, subject attributes, event-specific informa-
893 tion and object attributes. The record segments are read by calls to the following
894 functions:

895 1. *aud_get_hdr()*
896 2. *aud_get_hdr_info()*
897 3. *aud_get_subj()*
898 4. *aud_get_subj_info()*
899 5. *aud_get_event()*
900 6. *aud_get_event_info()*
901 7. *aud_get_obj()*
902 8. *aud_get_obj_info()*

903 *aud_get_hdr()* returns a descriptor for the header information.
904 *aud_get_hdr_info()* takes the descriptor returned by *aud_get_hdr()*, and returns
905 the data item from within the header of the audit record identified by the field_id.
906 If sequential access is being used, then repeated calls using AUD_NEXT_ITEM as
907 the field_id return the data items from the header in a predefined order.

908 *aud_get_subj()* returns a descriptor for a set of subject attributes.
909 *aud_get_subj_info()* takes the descriptor returned by *aud_get_subj()*, and returns
910 the data item from within the subject information of the audit record identified by
911 the field_id. If sequential access is being used, then repeated calls return the
912 data items from the subject attributes in a predefined order.

913 *aud_get_event()* returns a descriptor for an opaque data item defining a set of
914 event-specific data from the record. *aud_get_event_info()* takes the descriptor
915 returned by *aud_get_event()* and returns the data item from within the event-
916 specific information identified by the field_id. There are defined items of informa-
917 tion to be returned in a defined order for the standard audit event types when
918 sequential access is being used. Repeated calls to *aud_get_info()*, are required to
919 read all items of event specific information.

920 *aud_get_obj()* returns a descriptor for an opaque data item defining a set of object
921 attributes. *aud_get_obj_info()* takes the descriptor returned by *aud_get_obj()* and
922 returns the data item from within the object specific information of the audit
923 record identified by the field_id. If sequential access is being used, then repeated
924 calls return data items from the object information segment in a predefined order.

925 Implementations are free to add additional fields to system audit records. As
926 such, any of the audit record segments defined above may be extended. If the
927 implementation extends an audit record segment, the implementation-defined
928 data items are appended. That is, the implementation-defined data items will be
929 read using AUD_NEXT_ITEM after all the items defined by this standard. Note
930 that this means that an application must issue successive calls to the above inter-
931 faces to make sure all data items in a record are read.

932 **B.24.7.6.3 Example of Use**

933 The following describes a brief example of the POSIX.1e audit functions used to
934 read records from an audit log:

```

935     int          sys_ad1; /* file descriptor to the audit log */
936     aud_rec_t    aud_rec1; /* record descriptor */
937     aud_hdr_t    aud_hdr; /* audit record header */
938     aud_subj_t   aud_subj; /* audit subject info */
939     aud_event_t  aud_event_info; /* audit event information */
940     aud_obj_t    aud_obj; /* audit object information */
941     aud_info_t   aud_info_descr; /* audit info descriptor */

942     sys_ad1 = open (log, O_RDONLY) /* Open an audit log */

943     while ((aud_rec1 = aud_read (sys_ad1)) != (aud_rec_t) NULL)
944     {

945     /* Get audit header & header information */

946         aud_get_hdr (sys_rd1, 1, &aud_hdr);
947         aud_get_hdr_info (aud_hdr, AUD_EVENT_TYPE_ID, &aud_info_descr);

948         [ repeated calls to aud_get_hdr_info to get all hdr info ]

949     /* Get audit subject & related information */

950         aud_get_subj (sys_rd1, 1, &aud_subj);

951     /* Get the UID from the subject portion of the record */

952         aud_get_subj_info (aud_subj, AUD_EUID_ID, &aud_info_descr);

953     [additional calls to aud_get_subj_info for example ...]

954         aud_get_subj_info (aud_subj, AUD_MODE_ID, &aud_info_descr);

955     /* Get audit object & related information */

956         aud_get_obj (sys_rd1, 1, &aud_obj);

957     [additional calls to aud_get_obj_info for example ...]

958         aud_get_obj_info (aud_obj, AUD_ACL_ID, &aud_info_descr);

959     /* You could now use the POSIX.1e ACL i/fs to analyze the ACL */

960     /* Get audit event & related information */

961         aud_get_event (sys_rd1, 1, &aud_event_info);

```

WITHDRAWN DRAFT. All Rights Reserved by IEEE.
Preliminary—Subject to Revision.

```

962     [additional calls to aud_get_event_info for example ... ]

963             aud_get_event_info (aud_event_info, AUD_PATHNAME,
964                                     &aud_info_descr);
965     }

966     close (sys_ad1);

967 In the above example, the while loop reads records sequentially from the audit
968 log, referenced by sys_rd1. The record is then parsed by a series of calls to;
969 aud_get_hdr(), aud_get_hdr_info(), aud_get_subj(), aud_get_subj_info(),
970 aud_get_obj(), aud_get_obj_info(), aud_get_event(), aud_get_event_info().

971 Note that the addition of the field_id allows for (somewhat) random access to the
972 record. In previous versions of this standard, access of this nature was not pro-
973 vided and access to a particular part of the record was sequential. Additionally
974 there had to be some a priori knowledge of the format of the record (i.e. that UID
975 was the 4th field in the record). This problem has been eliminated with the addi-
976 tion of the field_id.

977 The audit records are processed in logical blocks, the header, subject, object and
978 event information. The aud_get_*() interfaces are used to (logically) extract the
979 corresponding logical block of the audit record so it may be processed by the appli-
980 cation. In an implementation, the aud_get_*() may simply position a index to a
981 portion of the audit record. For example a call to aud_get_hdr() may simply posi-
982 tion a index to the beginning of the audit record header. After the call to
983 aud_get_*(), subsequent calls to aud_get_*_info() are used to extract data fields
984 from the record. For example, repeated calls to aud_get_hdr_info() are made to
985 extract the header data items from the audit record.

```

986 B.24.7.6.4 Audit Record Conversion

987 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.

988 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 allo-
989 cated by the underlying implementation. Aside from the ordering of information
990 in the converted record, the standard does not specify any details of the text; thus
991 the output of the function can be displayed to a user, but cannot be further pro-
992 cessed by an application (e.g. adding special formatting). Portable post-
993 processing applications that want to provide formatted text for audit records
994 themselves can do so by using the *aud_get_**() and *aud_get_**_info() functions to
995 obtain the content of the record, and other POSIX.1 functions to convert each item
996 to text. In draft 13 and 14 there was an attempt to define more details of the out-
997 put of *aud_rec_to_text()*, but this was widely criticized (for example, it used new-
998 line characters as delimiters, but these were taken to be formatting which was
999 stated to be inappropriate to POSIX.1); therefore these details were withdrawn.

1004 The converse function is not provided by the standard: there is no requirement to
1005 be able to take a human readable record and convert it to internal form in order to
1006 support either post-processing or self-auditing applications.

1007 **B.24.7.6.5 Copying Audit Records**

1008 The working group determined that some applications would find it desirable to
1009 save audit records. They may be saved for functions such as backup/restore or for
1010 applications which are building a database of audit records for later processing.
1011 One way to achieve this is just to *aud_read()* records from one log and
1012 *aud_write()* them to another. However, this is not very flexible, since the destination
1013 has to be an audit log. It is desirable that it be possible to store a record in
1014 any user-defined destination. Since the POSIX.1e audit functions use system
1015 allocated space to store audit records, a provision needed to be made to copy the
1016 audit record from system managed space into user-managed space. Conversely,
1017 the ability to move the record back into system managed space and allow it to be
1018 processed by the POSIX.1e audit functions was also needed.

1019 The function *aud_copy_ext()* copies an audit record from system managed space to
1020 user-managed space. It is the responsibility of the application to ensure that adequate
1021 space is reserved for the copied record. To allow the application to determine the
1022 space required to hold the copied record, the function *aud_size()* is provided.
1023 The *aud_size()* function, accepts a pointer to an audit record in internal format and
1024 returns the size required to hold the audit record in user-managed space.
1025 Note that the size returned by *aud_size()* may not be reflective of the space allo-
1026 cated for the internal record because pointers or various compression techniques
1027 may be used by the underlying implementation to reduce the amount of space
1028 required to store audit records.

1029 The function *aud_copy_int()* copies an audit record from user-managed space back
1030 to system managed space. This function was provided to allow applications to re-
1031 process audit records that have previously been copied to user space and, maybe,
1032 saved. It was suggested that if the POSIX.1e audit functions could be made to
1033 operate on the user-managed copy of the record this capability would not be
1034 needed. However, because the underlying implementation may use various tech-
1035 niques to compress the size of internally stored records (e.g., pointers) the
1036 assumption that the POSIX.1e audit functions could be used on copied records
1037 was not valid. The working group did not want to constrain implementations by
1038 requiring that the internal and user-managed copies of audit records be identical.

1039 **B.24.7.7 Application Auditing Interfaces**

1040 **B.24.7.7.1 Constructing Audit Records**

1041 In draft 12, interfaces were defined that allowed an application to construct an
1042 audit record before writing it to an audit log. However, although it was clearly the
1043 intent that the application should be able to alter fields in the record, and thus
1044 reuse the record, this was not in fact possible. In ballot, this deficiency was
1045 widely criticized, as was the efficiency of such interfaces without an ability to
1046 reuse records.

1047 In draft 13, a major simplification of the interfaces was proposed. All interfaces
1048 for constructing audit records were removed, and instead a data structure
1049 approach was proposed; the application constructed the record using rows of
1050 type/length/pointer structures to define the data, then passed these structures to
1051 *aud_write()*. This too was widely criticized, as being insufficient application sup-
1052 port, incompatible with the style of the rest of the standard, and providing
1053 insufficient structuring capabilities: there was, for example, no means to indicate
1054 that a particular group of data items in the record all related to one object or sub-
1055 ject.

1056 The standard has now reverted to interfaces based on those of draft 12, but
1057 extended and completed to allow records to be built with all the structure of a
1058 system-generated record, and full facilities for altering and reusing records.
1059 Thus the objections to both the draft 12 and the draft 13 proposals should be
1060 satisfied. Indeed, much of the flexibility of the earlier, token based, proposal has
1061 been achieved, without however proposing as many interfaces as that did.

1062 The intent of the supplied interfaces is that an application should be able to
1063 implement any reasonable strategy for constructing audit records. For instance,
1064 an application is able to include much or little structure information in records: it
1065 can specify that most of the data in the record has no defined structure; or it can
1066 structure the data according to the subject(s) and object(s) to which it relates, and
1067 give meaningful data types for much of the data. Also, an application can chose to

- 1068 • Create a new *aud_rec_t* for each record it constructs, deleting the *aud_rec_t*
1069 when the record has been written to the audit log, or
- 1070 • Reuse a single *aud_rec_t* for various records, using the various *aud_put_**()
1071 and *aud_put_*_info()* interfaces to add information, and using the various
1072 *aud_delete_**() and *aud_delete_*_info()* interfaces to remove information,
1073 between invocations of *aud_write()*.

1074 The *aud_put_**() interfaces allow an application to ask the implementation to
1075 create new sections (e.g. header, sets of object attributes) in an audit record; the |
1076 interface returns to the application an identifier (an opaque data item) for the
1077 newly created section. The application used this identifier when adding content to
1078 the section, and also when it wants to add another new section before the existing
1079 one.

1080 The *aud_put_*_info()* interfaces allow an application to add content to a section
1081 created as above. The application tells the i/f an identifier (an integer) for the
1082 item that it is adding to the section; it can also give the i/f an identifier for a previ-
1083 ously added item before which the new item is to be placed.

1084 **B.24.7.7.2 Writing the Audit Log**

1085 The ability to write to the system audit log cannot be generally available, because
1086 it could provide a malicious user with a means of denying service to other users
1087 (by filling up the audit file) or misleading an audit administrator (by seeding the
1088 audit log with disinformation). Accordingly, utilities that use the *aud_write()*
1089 interface to write to the system audit log must have appropriate privilege and be
1090 trusted to use it properly.

1091 The *aud_write()* function accepts an *aud_rec_t* pointing to an audit record which
1092 may have been constructed using the interfaces described above, or may have
1093 been read from another audit log. Some earlier drafts of the standard did not per-
1094 mit an application calling *aud_write()* to specify all the sections of a record; some
1095 did not permit the internals of a record to be structured into sets of related details
1096 (e.g. object attributes); some did not permit a record to be read from one log and
1097 written to another. All of these earlier restrictions were the subject of ballot
1098 objections, leading to the current interfaces.

1099 **B.24.7.7.3 Auditing Suspension and Resumption**

1100 Any process doing its own auditing may wish to suspend standard auditing of its
1101 operations. This is likely to be used mainly by processes auditing themselves at a
1102 much finer or coarser granularity than the kernel. For example, a program that
1103 scans the filestore periodically and moves to tape any files that have been unused
1104 for a long time could audit the movement of the files itself (in a more meaningful
1105 way than the kernel); it seems unnecessary to record that it checked the access
1106 dates of all files in the system, which would merely clutter the audit log with
1107 data. Even standard utilities (with appropriate privilege) might make use of this
1108 facility, to provide a higher level view of events than would be given by the kernel.
1109 The interface used to request that the system suspend and resume system audit-
1110 ing of the current process is *aud_switch()*.

1111 **B.24.7.7.4 Error Return Values**

1112 If the symbol *{_POSIX_AUD}* is defined, then the implementation supports the +
1113 audit option and is required to support the audit functions as described in this +
1114 standard. If the symbol *{_POSIX_AUD}* is not defined, then the implementation +
1115 does not claim conformance to the audit option and the results of an application +
1116 calling any of the audit functions are not specified within this standard. An alter- +
1117 native is for the audit functions to specify that the error return code [ENOSYS] be+
1118 returned by the functions if the audit option is not supported. However, in order +
1119 to remain compliant with the policies of POSIX.1, this standard cannot specify +
1 any requirements for implementations that do not support the option.

2 **B.25 Capability**

3 **B.25.1 General Overview**

4 **Goals**

5 The primary purpose of defining interfaces for a capability mechanism within this
6 standard is to provide for a finer granularity of controlling and granting system
7 capabilities than the traditional super-user model.

8 The major goals of this standard are to:

- 9 (1) Provide a portable means of supporting the assignment of an ability for a
10 process to invoke or perform restricted system services.
11 (2) Support the implementation of *least privilege* security policies by provid-
12 ing the means to constrain a process by enabling it to invoke only those
13 system capabilities necessary to perform its specific tasks.

14 The additional goals of this standard are to:

- 15 (1) Define a common terminology for addressing the topic of capability.
16 (2) Define the semantics of how process capabilities are acquired and
17 altered.
18 (3) Define the system functions and utilities necessary to utilize capabilities.
19 (4) Provide compatibility with programs that depend upon the set user id on
20 execution and set group id on execution behavior to gain access to system
21 resources.
22 (5) Provide the means by which an implementation may grant capabilities in
23 order to emulate the traditional super-user.
24 (6) Allow for extensibility by future implementations.
25 (7) Define a minimum set of capabilities necessary to support the develop-
26 ment and execution of security-relevant programs.
27 (8) Ensure that there is a mechanism by which capabilities may be trans-
28 ported with their associated files.

29 It has been pointed out that the term *privilege* has been commonly used for a
30 mechanism that achieves the above stated goals. However, the term *privilege* is
31 also commonly used in the international community to mean something else
32 entirely. It is felt that the confusion that would result from using the term
33 *privilege* would not serve this standard well. —

34 A capability mechanism is a common requirement for most operating systems.
35 Capability controls the availability of particularly important system services to
36 processes that are known to maintain system integrity.

37 The principle of *least privilege* is a common requirement of security policies, that
38 is, granting to a process only the minimum rights and capabilities necessary to
39 perform a task. The purpose of this principle is to constrain the damage that may
40 arise from a violation of the security policy, e.g., disclosing confidential informa-
41 tion or corrupting the integrity of the system. We must emphasize here that the
42 standard does not (nor can it) specify a *least privilege* mechanism—only interfaces
43 that, when used with a correctly defined set of capabilities, could successfully be
44 used to implement a *least privilege* security policy.

45 An example of the application of the principle of *least privilege* in the commercial
46 environment is the separation of roles in an accounting department. In most firms
47 of any size, the person who records and manages the Accounts Receivable is **NOT**
48 the person who records and manages the Accounts Payable. This is so one person
49 cannot create false bills and then write checks to pay them. A current example in

50 the computer world is the use of the restricted shell (rsh) for computer operator
51 consoles — the operator, who has a great deal of potential access to the entire
52 computer system by virtue of his or her physical access to the machine, can have
53 that access limited to those functions actually required to perform his or her job
54 by the system administrator.

55 Additional goals 1 and 2 are natural intermediate goals for meeting our major
56 goals. Before a capability mechanism can be defined, a terminology and the basic
57 concepts of capability must be laid out. Once that has been achieved, then the
58 semantics of how capabilities are acquired, manipulated, and controlled need to
59 be defined. Only after this step has been accomplished — deciding what opera-
60 tions are required to provide a capability mechanism — can the next step be
61 taken.

62 Additional goal 3 is the end result of this effort — the definition of the interfaces
63 that can be used to provide the semantics developed above. The specification of
64 these interfaces is the entire purpose of this effort — to provide a set of tools that
65 can be used by conforming applications to perform those tasks necessary for its
66 functions.

67 Additional goals 4 and 5 are compatibility goals. The set user id and set group id
68 mechanisms of POSIX.1 continue to function as they have in the past, providing
69 DAC access to objects based upon the owning ids of the executed file. Set uid root
70 functionality may be provided by appropriate use of the file permitted capability
71 set. While our goal is to provide a mechanism that will support implementations
72 intended for high levels of trust, there will be implementations that will still need
73 to support existing setuid root programs, and implementations that will still pro-
74 vide the 'superuser' identity to administrators. While we would like to discourage
75 both of these practices, we understand that current practice is often slow to
76 change and that some existing applications will have to run unmodified on secure
77 machines for at least a transition period.

78 Goal 6 is a basic goal of all systems — motherhood and apple pie to engineers. All
79 systems need to permit extensibility and flexibility so that unforeseen situations
80 and future improvements do not require an architectural change in order to
81 accommodate them. At some point, every system will need to be completely
82 replaced, but one would like to push that off as long as possible. Implementations
83 will need to provide capabilities not specified here to accommodate various secu-
84 rity policies and system functions not part of this standard. Extensibility is there-
85 fore an absolute requirement.

86 Goal 7 is the specification of a standard set of capabilities — is a necessary part of
87 this effort. Trusted applications will need to be able to acquire a certain capabili-
88 ty to perform a specific function across all compliant implementations in order to
89 be portable, and that capability will need to have the same meaning across imple-
90 ments.

91 Goal 8 was agreed upon primarily to support system backup and restoration
92 operations. This goal does not include the transfer of capabilities from system to
93 system necessarily. Indeed, there is a good argument that requiring that degree
94 of portability adds risk to a system, and that a system administrator should be

95 required to approve every new trusted program before it is assigned capability
96 attributes. As a result we define file capability attributes, but not their actual
97 representation or how they are stored with the file on a tape.

98 **Scope**

99 The scope is the natural result of our goals. In order to support the principle of
100 least privilege, interfaces that provide the means for programs to enable and dis-
able capabilities while running are necessary. In order to support the compati-
101 bility goals, there must be a means for programs to pass capabilities to other pro-
102 grams that they execute, and the semantics of that inheritance must therefore be
103 specified to some degree. Because it is programs that are the “trusted” agents on
104 implementations, there must be some method to identify them as trusted —
105 therefore attributes associated with program files must be specified. Finally, a
106 small set of capabilities to be used with the interfaces and utilities in the existing
107 POSIX.1 specifications must be defined so that writers of conforming applications
108 know which capabilities will be available to perform various functions and their
109 appropriate use.

111 **Purpose of a Capability Mechanism**

112 The purpose of a capability mechanism is to provide a finer granularity of control
113 over the access to restricted system services to specific users or processes than
114 that provided by the traditional POSIX.1 "UID 0" access mechanism. A general
115 purpose capability mechanism supports not only the ability to implement the
116 principle of least privilege, but also provides the foundation for building an
117 authorization mechanism to support security administration. The interfaces and
118 concepts presented in this document have been designed to meet these require-
119 ments.

120 **Authorization vs Capability**

121 The power to perform an action in a trusted system based on user identity is
122 called an “authorization.” Authorizations are generally designed around opera-
123 tional requirements and tasks rather than system services. For example, an
124 authorization to perform backups would be granted to a user. The backup pro-
125 gram however, would enable and disable specific capabilities to perform the
126 backup function. A system that supports authorizations simplifies the adminis-
127 trative task of the security officer by eliminating the need to comprehend exactly
128 which capabilities each program requires and how to allocate those capabilities to
129 users.

130 The establishment of a user identity and a user’s authorizations based on that
131 user’s identity is presently outside the scope of the POSIX standards. Because of
132 this, the assignment of authorizations to users through a program such as login
133 and the use of an authorization mechanism for determining utility capability
134 bracketing is presently undefined, as is the relationship between the authoriza-
135 tion mechanism and the capability mechanism used by a program. It is not, how-
136 ever, our intention to preclude any implementation of a user authorization
137 mechanism with this standard.

138 General Discussion of Capability

139 Currently, most POSIX.1 and POSIX-like implementations grant all capabilities
140 to a particular user ID – 0 (root). Most of the time, the ability to log in as or to
141 assume this identity is restricted to a small set of users on a system, one of whom
142 is the system administrator. The “root” account has the ability to execute any
143 utility or use any system function regardless of what security restrictions may be
144 involved. Such special rights are necessary to do many administrative tasks, such
145 as system backups and restores, writes into special files, and to operate processes
146 such as line printer daemons and mail handling servers.

147 In the vast majority of cases, however, a process needs to invoke only a few
148 specific restricted system services or override a single type of access permission in
149 order to accomplish its task. A line printer daemon, for instance, needs to be able
150 to read any file in the system, but does not need the ability to write into them,
151 and the same with a backup program. A login program needs to be able to change
152 its user identity, but it does not need to modify disk quotas, and so forth.

153 As has been demonstrated numerous times, the requirement that a process be
154 granted the ability to bypass all the security restrictions in a system just to
155 bypass some of them leads to accidents and purposeful misuse. Many times,
156 users do not realize that they are in privileged mode and perform a destructive
157 action (`rm *`) without realizing that the system will not stop them in their current
158 state. Other times, a user acquires the ability to become “root” for a perfectly legiti-
159 mate reason, and then passes it on to other users or applies the special abilities
160 “root” provides in ways not intended by the system administrator.

161 A capability mechanism provides the means for a system administrator to grant a
162 program the ability to use a restricted system service or bypass specific security
163 checks. For instance, user Joe can run the backup program for an entire network
164 (that can read every file on the network) from the “admin” host. Properly imple-
165 mented and administered, the capability mechanism could permit Joe to perform
166 his assigned task, but could prevent abuse of the world read access capability
167 such as browsing files normally not accessible to Joe.

168 Principle of Least Privilege

169 A process’s need for capability access to system resources and functions does not
170 justify giving the process uncontrolled use of capabilities. It is also not appropri-
171 ate to establish for a process chain (a sequence of programs within a single pro-
172 cess) a set of capabilities that remains fixed and active throughout the life of that
173 chain. Rather, the set of active capabilities of a process can be expected to change
174 as the functions of the process change, so that the process has active at any time
175 just those capabilities needed to perform its current function. This is an applica-
176 tion of the principle of least privilege, and it applies equally to users and to
177 processes.

178 Implications of the Principle of Least Privilege

179 Any capability mechanism will associate with each process a set of capabilities
180 that the process can potentially use, but capabilities should be controlled at the

181 level of granularity of individual programs. The most straightforward way to do
182 that is to associate capability controls with the individual program files. The first
183 requirement implied by the principle of least privilege, *to control capability at the*
184 *granularity of individual programs*, leads to the assignment of capability attri-
185 butes to program files; this is the file capability state.

186 If a program is always executed in a single context, e.g., by a single user to per-
187 form a single function, then the specific set of capabilities for that context-
188 program combination applies to all invocations of the program. However, in gen-
189 eral a program is executed in varying contexts, e.g., by different users, or on dif-
190 ferent files, or for different purposes—such as a printer spooling program. Thus
191 we need to be able to change the capabilities of a process as its circumstances
192 change.

193 This is the second requirement implied by the principle of least privilege: to con-
194 trol use of capability within the context of intended use. It further suggests that
195 process capabilities be divided into two classes: capabilities that are currently
196 active, and capabilities that could be activated. We do this by creating two
197 corresponding kinds of process capability flags: *effective* (indicating that the capa-
198 bility is active) and *permitted* (indicating that the capability could be activated).
199 Thus a process can increase its current set of active capabilities by making *effec-*
200 *tive* any capability that it is currently *permitted*, and can reduce its active capa-
201 bilities at any time while retaining the ability to restore them. This ability of a
202 process to adapt its active capabilities to the needs of the moment is referred to in
203 the standard as the “time bounding of capability” and is sometimes also referred
204 to as capability bracketing.

205 If a process image is instantiated from a program file, its capabilities will be
206 affected by the capability state associated with the file. A program will *exec* a pro-
207 gram file to instantiate its successor program in a process chain. Here too the
208 principle of least privilege implies that we adapt the use of capability to the con-
209 text of use. There are two general ways to do this.

210 In the first, *exec()* constrains the maximal extent of capabilities for the
211 process image it instantiates from a program file. In this way the
212 invoking process image can limit the capabilities of the successor pro-
213 cess image.

214 In the second, *exec()* plays no role in limiting the set of capabilities that
215 the instantiated process image may have; rather, the successor process
216 image sets the capabilities itself, choosing them from the set of capabili-
217 ties associated with the program file from which it was instantiated,
218 and possibly from a set of capabilities that a predecessor process image
219 had passed on.

220 In either case, the advantage of passing capabilities along a process chain is that
221 it allows the process to dynamically build up a capability context, rather than lim-
222 iting its capability context to a single, per-process image state.

223 Besides providing a capability for a process image to pass capability information
224 to subsequent process images, it may be desirable that a specific process image
225 have capabilities that are not *permitted* to any of its predecessors. We therefore

226 need a way to increase the capabilities of a process based on the program file
227 being *exec*'ed.

228 Finally, we observe that a process image may wish to pass capabilities to some
229 successor process image through an intermediate third process image that is not
230 itself trusted to properly use the passed capabilities. For example, a process may
231 initiate an untrusted shell that in turn will *exec* a third program file.

232 **B.25.2 Major Features**

233 There must be some method for a process to acquire capability(s) it needs if it is to
234 be able to use it(them) at some point. Because capabilities are security relevant,
235 this method must be restricted to a trusted part of the system, which must grant
236 the ability to use capability based on one or more characteristics of the process.
237 We assert that the characteristic most relevant is the identity of the program or
238 programs that are run within that process.

239 A result of this assertion, that the identity of programs is the primary characteris-
240 tic used to assign trust, is the requirement that there be some means to identify a
241 program file as trusted. There are several means available to do this. The first is
242 to embed some form of identification in the program file itself in such a way that
243 the loader can interpret it. This leads to problems, however, in that different ins-
244 tallations may have different security policies, and that system administrators
245 may not trust the program developer enough to set the proper capability attri-
246 butes. The second alternative is to attach capability attributes to the program
247 file. This alternative provides a much larger degree of flexibility, in that system
248 administrators can differ in their trust of a particular program without modifying
249 or altering the actual program itself, and is much more consistent with current
250 practice and methods. As a result, file capability attributes were proposed.

251 **B.25.2.1 Task Bounding of Capability**

252 This standard has the advantage of being flexible enough that a given capability
253 may be bound either for the duration of an executable program or the duration of
254 a single system call. This allows flexibility in the granularity of capability, pro-
255 vides support for backwards compatibility, and allows trusted programs to sup-
256 port capability bracketing. The main advantage in task bounding of capability is
257 that it reduces the chance that program errors will have security-relevant side
258 effects.

259 **B.25.2.2 Capability Inheritance**

260 Trusted programs can perform complicated functions and, as a result, can be very
261 large. The larger and more complicated a program is, however, the harder it is to
262 evaluate for trust and the more difficult it becomes to maintain. In addition, one
263 of the basic tenants of the POSIX.1 operating system is to provide a set of simple
264 utilities that can be executed together or in series to perform more complicated
265 functions. As a result, it is desirable for a trusted program to be able to pass on
266 its capability characteristics to other programs to perform functions it would

267 otherwise have to implement itself.

268 While one trusted program may want to pass **all** of its capabilities to another,
269 more often the child program only needs a subset of the parent program's capabili-
270 ties to perform its functions. Also, should the child program be trusted, the
271 parent trusted program may not be aware of how much trust that child program
272 actually has at any given time. Finally, a conforming program CANNOT be
273 trusted to handle implementation-defined capabilities. Therefore, the developer
274 needs to have the ability to restrict what capabilities he or she desires to pass on
275 to the child program, and the system developer and administrator need to have a
276 means of controlling what capabilities they are willing to permit the child pro-
277 gram to have.

278 Since the *exec()* function is the means by which one program invokes another, it
279 must be modified:

- 280 • To grant capabilities to programs when they are executed.
281 • To permit programs to pass capabilities to other programs.
282 • To restrict which capabilities may be passed from one program to another.

283 So far, we have provided the basis for program level capabilities. In other words,
284 programs that are granted capabilities using the attributes specified so far have
285 those capabilities during their entire scope of execution. For many systems,
286 program-level capabilities may not provide the level of granularity desired by the
287 security policy. For instance, a program may need to have the capability to write
288 to a system administrative file only during a single call to the *open()* system func-
289 tion. For the remainder of the time the program executes, the capability is avail-
290 able but not required. In order to support implementations that support the con-
291 cept of least privilege to a finer level of granularity, we need to provide the means
292 by which a program can enable a capability only during the scope of execution for
293 which it is actually required.

294 In summary, then, the view of the principle of least privilege presented here and
295 the desired functionality described above implies the following:

- 296 (1) There is capability state associated with program files as well as with
297 processes.
- 298 (2) There are two kinds of process capability attributes: one which defines
299 what capabilities **may** be invoked by the process image, and another that
300 defines what capabilities **are currently** invoked by the process.
- 301 (3) There is a way to increase the capabilities of a process that depends on
302 the process image file that it *exec*'s.
- 303 (4) There is a way to conditionally transmit capabilities from a process
304 image to its successor image(s).
- 305 (5) There is a way to restrict which capabilities may be passed to any partic-
306 ular process image that depends on the process image file.
- 307 (6) The *exec()* system function determines the capability attributes of the
308 process it instantiates.

309 **B.25.2.3 Process Capability Flags**

310 A process image acquires capabilities from the set of capabilities attached to the
311 program file from which it is initiated. The *effective* flag determines whether the
312 capability is active for the process. The *permitted* flag determines whether the
313 process may choose to make the capability *effective*. The *inheritable* flag deter-
314 mines whether the process may pass on to its successor process image a condi-
315 tional right to use a capability. The right must be conditional because the capa-
316 bility may be inappropriate for intermediate image(s). Indication of the
317 successor's appropriate capabilities is reasonably associated with the successor's
318 process image file. In fact, this indication can be made precisely by the *permitted* |
319 file capability flag. The determination of the right to use a capability depends on |
320 the current process's value of the *inheritable* flag and on the values of the *permit-|
321 ted* and *inheritable* flags of the corresponding file capability. This determina- |
322 tion is made by *exec()*. In implementations that depend more heavily on use of the |
323 *effective* flag, the *inheritable* flag can be used by a process image to determine the |
324 trust associated with its predecessor process image and therefore provide a basis |
325 for enforcing its own security policy.

326 **B.25.2.4 File Capability Flags**

327 As we have seen, the principle of least privilege requires that with each program
328 file there is associated the set of capabilities that a process image, instantiated
329 from that file, requires to do any of its functions.

330 The *inheritable* flag determines which capabilities the resulting process image
331 may pass to subsequent process images and which ones the program may chose to
332 use if the previous program image possessed the capability.

333 The *permitted* flag determines which capabilities the resulting process image
334 needs to have available in order for the program to function properly, regardless
335 of the capabilities of the previous process image.

336 The *effective* flag determines which capabilities the resulting process image will |
337 possess in its *effective* process set.
338 The ability to support capability unaware applications on a per executable basis |
339 ensures that these programs will continue to function with a limited set of capa- |
340 bilities, thus reducing the risk of unauthorized access to restricted functions.
341 Additionally, the risk of a trojan horse gaining unauthorized access to capabilities |
342 is reduced if the inclusion of capabilities into the effective set is automatically lim- |
343 ited to a per file basis.
344 Earlier versions of this standard provided a single *set_effective* flag instead of the |
345 *effective* set. The new process *permitted* set was promoted to the *effective* set on |
346 *exec()* when this flag was set.

347 **B.25.2.5 The Determination of Process Capability by fork()**

348 This is a simple case. The *fork()* system function is meant to create a new process |
349 that is, as much as possible, identical to its parent. Because capability is not an |
350 attribute that uniquely identifies a process, such as process ID, the capability |
351 state of a child process should be identical to that of its parent immediately after |
352 the execution of the *fork()* system function.

353 **B.25.2.6 The Determination of Process Capability by exec()**

354 The *inheritable* and *permitted* capability flags of the program file and the *inherit-|*
355 *able* capability flags of the current process together determine the context- |
356 dependent set of capabilities *permitted* to the instantiated process. The |
357 context-independent set of capabilities that is included in the *permitted* capability |
358 set of the program when it is executed is derived from the *permitted* file capability |
359 flags associated with the program file. The union of these two sets comprise the |
360 set of capabilities that the *exec()* function permits the new process image to use.

361 The initial state of the *effective* flags of the new process image depends on the |
362 *inheritable* flags in the old image and the values of *inheritable*, *permitted*, and |
363 *effective* flags of the program file. The justification for selecting the transforma- |
364 tion function for process capability state is incorporated throughout the text of |
365 this section.

366 **B.25.2.7 Support of the Capability State Attribute on Files**

367 The intent of these interfaces is not to limit the manner in which processes can |
368 gain appropriate privilege. Thus, if the value of the pathname variable |
369 {_POSIX_CAP_PRESENT} is zero (meaning that the file does not support the |
370 POSIX capability state attributes), then it is possible for an implementation to |
371 specify other mechanisms. For example, the USL implementation provides both a |
372 privilege mechanism and a superuser mechanism.

373 Certainly, there are implementations that allow files to be exec'ed from file sys- |
374 tems that do not support capability attributes (for example, an NFS file system |
375 mounted from a system not supporting the capability option). In this case, it is |

376 suggested that an implementation treat this file exactly as it would a file without
377 a capability state attribute from a file system that does support capability attri-
378 butes.

379 **B.25.2.8 Extensions to This Standard**

380 This specification does not preclude providing additional implementation-defined
381 constraints, such as a system-wide configuration variable to further constrain the
382 capability inheritance rules. The value of this variable could be used to act as an
383 additional gating function to permit a single global value to be manipulated by a
384 system security officer to help stop or slow a security breach in progress by
385 preventing any permitted capabilities from being automatically included in every
386 process effective capability set. Additional file capability attributes and file capa-
387 bility flags can also be defined by an implementation. It must be emphasized that
388 such extensions are compliant only if they further constrain (prevent from becoming
389 effective) capability.

390 **B.25.2.9 Process Capability Manipulation**

391 When a process image is instantiated from a program file, its capability flags
392 describe its capability state. As noted earlier, the *effective*, *permitted*, and *inherit-
393 able* flags respectively denote which capabilities are active, which may be
394 activated, and which the process image will (conditionally) pass on to its suc-
395 cessors. A process should not be permitted to arbitrarily modify these flags, but is
396 restricted according to the following set of rules.

397 A process can promote to *effective* only those capabilities whose *permitted* flag is
398 set. This lets the process adapt its degree of active capabilities to its current con-
399 text, and so supports the principle of least privilege. On the other hand, the pro-
400 cess can never promote a capability to *effective* if the *permitted* flag is turned off,
401 and can never enable a *permitted* flag that is turned off. Thus the process cannot
402 assume for itself capabilities to which it is not entitled.

403 To prevent it from accumulating capabilities through inheritance, a process can
404 enable an *inheritable* flag only if the corresponding *permitted* flag is set.

405 If a process disables a *permitted* flag, the corresponding *effective* flag is automati-
406 cally disabled. The corresponding *inheritable* flag is not affected, so capabilities
407 can be conditionally transmitted along a process chain whose intermediate
408 processes may themselves have no capabilities. In no other case does changing
409 the value of any flag affect the value of any other flag.

410 **B.25.3 Function Calls Modified for Capability**

411 The standard defines the capabilities required by each of the POSIX.1 functions.
412 However, many implementations included additional functions that should be
413 modified to support the capabilities defined in this standard. While the list
414 presented here is by no means exhaustive, it is included as helpful information for
415 the reader.

416 **Table B-3 – Other System Functions Potentially Affected by Capability Policies**

417

Function

419	adjtime
420	bind
421	chroot
422	killpg
423	limit
424	mincore
425	mknod
426	mount
427	ptrace
428	readv
429	reboot
430	sethostname
431	settimeofday
432	shutdown
433	socket
434	socketpair
435	swapon
436	symlink
437	syscall
438	umount
439	vadvise
440	vfork
441	vhangup
442	writev
443	sysattr

444 **B.25.4 Capability Header**

445 These types were defined to provide opaqueness and avoid specifying detail that
 446 should be left to the implementation. The capabilities defined in this section are
 447 limited to those specifically called for in the POSIX.1 standard. Included also are
 448 those capabilities defined in POSIX.1e.

449 **B.25.4.1 Rationale for the Selection of Capabilities Defined in the Stan-
450 dard**

451 This section will describe the process that the capability group used to develop the
 452 set of capabilities specified in this standard. Enough detail is provided about the
 453 process so that an implementor can duplicate it when analyzing an implementa-
 454 tion to determine what additional capabilities, if any, are required.

455 We began the process of defining a capability set for the standard by first develop-
 456 ing a set of guidelines to be used. These guidelines are contradictory to a degree,
 457 and the group made trade offs between them when discussing each individual
 458 capability in order to come up with a minimum set of capabilities that were
 459 deemed necessary for the support of conforming applications.

460 **Principles for Determining a Capability Set**

461 Principle #1: A capability should permit the system to exempt a process from a
462 specific security requirement.

463 In most cases, security requirements found in the function descriptions take the
464 form: “In order for this function to succeed, <requirement>, or the process must
465 possess *appropriate privilege*.” A specific example can be found in the POSIX.1
466 description of the *chown()* function, which states “In order for this function to
467 succeed, the UID associated with the process must match the owner ID of the file,
468 or the process must possess *appropriate privilege*.” |

469 This principle is meant to support the principle of least privilege, in that a capa-
470 bility should provide only the minimum rights or authority to perform a specific
471 task.

472 Principle #2: There should be a minimal overlap between the effects of capabili-
473 ties.

474 Capabilities should be defined such that they apply to logically distinct opera-
475 tions, and the granting of a set of capabilities should not, as a side effect, grant an
476 additional capability that is not in that set.

477 This principle was developed to address the concerns that capabilities should be
478 distinct and unique—no capability or combinations of capabilities should provide
479 the capabilities afforded by another capability. When a system administrator
480 grants one or more capabilities to a specific user or program, they should have
481 some assurance that the recipient is not gaining any additional capabilities.

482 Principle #3: Insofar as principles #1 and #2 are supported, fewer capabilities are
483 better than more.

484 When it makes sense to do so, and identical or nearly identical security require-
485 ments exist, a single capability should be defined for all those security require-
486 ments instead of a separate capability for each individual security requirement.

487 This principle was defined primarily to support ease of use and ease of adminis-
488 tration. If each individual security requirement in an implementation had a
489 unique capability, several hundred capabilities would be required, a management
490 nightmare that would be prone to misunderstanding, confusion and error. If a
491 specific security requirement is especially critical or sensitive, however, it was
492 generally agreed that it should be assigned a unique capability in order to assure
493 positive control over which processes/programs are exempted from the require-
494 ment.

495 **Determining the Capability Set**

496 Once the above general principles were agreed to, the group turned to the existing
497 and draft POSIX documents to begin the process of actually developing the set of
498 capabilities included in this standard.

499 The set of capabilities defined in this document is not intended to be all-inclusive.
500 Implementations may (and probably should) define additional capabilities to sup-
501 port the operation and maintenance of their systems. Finally, it should be

502 emphasized that the development of a capability set is not a cookbook process—
503 implementors must consider their own system security requirements and the
504 design of their own systems when determining what capabilities they will sup-
505 port. Our requirement was to develop a minimum set of capabilities we deter-
506 mined necessary to support conforming POSIX applications.

507 Step one in the process was to develop a list of security requirements from the
508 POSIX.1, and POSIX.2 documents. This involved searching through the descrip-
509 tions of the functions and utilities looking for the phrase “appropriate privilege” |
510 and also looking for text that implied a security requirement that was not directly
511 stated.

512 Once we had developed the list of security requirements, or “checks”, we grouped
513 sets of identical or nearly identical requirements together, and developed a
514 descriptive name for each individual or group of requirements that remained.
515 When grouping requirements, each case was discussed to ensure that it really did
516 belong to the group, and it was not uncommon for a decision to be re-made as the
517 list developed and additional considerations were brought up.

518 The last step in the process was to review the entire list. Capabilities were
519 deleted or combined with another capability when it was deemed appropriate to
520 do so with respect to the third principle in B.25.4.1

521 **B.25.4.2 Rationale for DAC Capability Specification**

522 The DAC group defines the extensions to POSIX.6 for a finer granularity of discre-
523 tionary access control beyond POSIX.1. For systems with `{_POSIX_CAP}` -
524 configured, it is necessary to define the policy override capabilities.

525 The DAC group initially considered separating DAC overrides into 4 distinct
526 capabilities. These were:

- 527 • CAP_DAC_READ
- 528 • CAP_DAC_WRITE
- 529 • CAP_DAC_SEARCH
- 530 • CAP_DAC_EXECUTE.

531 The CAP_DAC_READ and CAP_DAC_WRITE separation was considered neces-
532 sary for providing read-only access for a wide range of applications that have no
533 need to write to the objects they are examining. The CAP_DAC_SEARCH and
534 CAP_DAC_EXECUTE capabilities were suggested because it was not necessarily
535 appropriate to group these abilities with the CAP_DAC_READ and
536 CAP_DAC_WRITE capabilities. Also, specification of four separate capabilities
537 maps one-to-one with the existing POSIX.1 features.

538 The group also considered a single CAP_DAC_OVERRIDE capability, but this
539 granularity was considered insufficient for the following reasons:

- 540 • Demonstrated commercial need on other operating systems to support
541 separate CAP_DAC_READ and CAP_DAC_WRITE overrides based on func-
542 tional requirements. For example, a backup program requires the ability to
543 read all file objects on the system but only requires the ability to write to
544 the backup device. Additionally, this separation provided programmatic
545 support for administrative roles which allow for protection from inadver-
546 tent modification of system critical objects.
- 547 • Worked examples of trusted systems evaluated at class B2 or higher
548 against the TCSEC on which similar mechanisms were required to meet
549 the System Architecture requirement.

550 Because the specification of four separate capabilities seemed to be unnecessary,
551 and the specification of a single capability is not sufficient to support commercial
552 requirements, we decided to specify three capabilities and permit implemen-
553 tations to add additional capabilities if appropriate.

554 In fact, an analysis of the requirements determined that these three capabilities
555 are sufficient to support the principle of least privilege as well as the anticipated
556 commercial demand. Note, however, the specification provides support for imple-
557 mentation defined capabilities where deemed necessary.

558 The consensus was that applications that required CAP_DAC_READ override
559 would also require CAP_DAC_SEARCH override. Therefore these two capabili-
560 ties were combined.

561 **B.25.4.3 Rationale for MAC Capability Specification**

562 A MAC policy differs from a DAC policy in that an untrusted process or user does
563 not participate in establishing the access criteria. Rather, the system is responsi-
564 ble for enforcing the policy established by the security officer. As such, the MAC
565 policy can be considered to impose a higher degree of assurance on the protection
566 of an object compared to DAC. Therefore, MAC policy override capabilities must
567 be carefully considered.

568 The MAC group has established a set of policy overrides that are designed to sup-
569 port sufficient granularity of control to meet the needs of current security stan-
570 dards as well as to meet the needs of future trusted applications, such as data-
571 bases, multi-level mailers, etc.

572 **CAP_MAC_UPGRADE and CAP_MAC_DOWNGRADE**

573 The MAC group originally considered a single MAC override capability to cover
574 both the upgrade and downgrade cases for manipulating object labels. Although
575 this level of granularity meets the needs of the current TCSEC, more recent secu-
576 rity criteria, such as the '91 Compartmented Mode Workstation Evaluation Cri-
577 teria do require separation of the MAC override capabilities. In addition, the
578 separation of the upgrade and downgrade functions is a common operational
579 requirement. Supporting distinct capabilities is a logical extension of this opera-
580 tional requirement.

581 **CAP_MAC_LOCK**

582 At one time during the writing of this standard, the standard required that a process
583 have MAC write access to a file at the time of a lock operation, or have
584 CAP_MAC_LOCK enabled. These protections were necessary because the set of
585 locks associated with a file are considered to be an object. More specifically,
586 because the data structure which defines the lock on a file can be directly written
587 by processes (by setting locks) and can be directly read by processes (by querying
588 locks), this data structure was deemed a communication channel that must be
589 subject to MAC constraints.

590 The straightforward application of MAC policy to locks requires that a process
591 have MAC write access to the file prior to setting locks. In a system with only
592 CAP_MAC_WRITE, a process must be trusted to use the override capability
593 appropriately. It can be argued that processes that need to use locks should be
594 trusted enough to use the MAC write override capabilities for this purpose. This
595 approach also has the added feature of minimizing the number capabilities neces-
596 sary for the MAC policy.

597 However, the use of CAP_MAC_WRITE to bypass this policy constraint was con-
598 sidered non intuitive and a violation of the principle of least privilege. For exam-
599 ple, a process merely wishing to set a read lock on a lower level file simply to read
600 the file, e.g., a password file, would then need to be granted the MAC write capa-
601 bility, despite having no need to write data to the lower level file. Thus in cases
602 such as these, which in actual implementations are likely to be frequent, not only
603 is a powerful capability being used to cover a relatively innocuous activity, but
604 also the use of a write capability to effectively perform a read is confusing. For
605 this reason CAP_MAC_LOCK was originally adopted.

606 Based on significant ballot objections, this capability was removed and the stan-
607 dard was made mute on the subject of how an implementation handles the chan-
608 nel created by *fcntl* and reading locks.

609 **CAP_MAC_READ and CAP_MAC_WRITE**

610 While the TCSEC does not require separation of the MAC override capability into
611 distinct READ and WRITE capabilities, other security specifications do. In addition
612 MAC is a system enforced policy rather than a discretionary policy, requiring
613 that applications which need only to read an object also have the power to write
614 the object was considered an unwarranted risk. Separation of MAC_READ and
615 MAC_WRITE overrides will encourage application developers to be cautious with
616 their use.

617 **CAP_MAC_RELABEL_SUBJ**

618 The ability of a subject to change its own MAC label is controlled by the
619 CAP_MAC_RELABEL_SUBJ capability. This capability is intended for use by
620 trusted subjects which have the need to modify their label based on some (possibly
621 external) criteria. For example, a trusted server which may need to reset its
622 MAC level prior to executing functions on behalf of a client request. Unlike
623 objects, which tend to have a static label, subjects would need a dynamic label

624 therefore a single capability is more appropriate for subjects.

625 **B.25.4.4 Rationale for Information Labeling Capability Specification**

626 **CAP_INF_NOFLOAT_SUBJ and CAP_INF_NOFLOAT_OBJ**

627 These two capabilities are the override capabilities for the Information Label Policy.
628 The INF_NOFLOAT_OBJ capability is necessary to support programs which
629 need to write a shared single file at many information levels. An example of this
630 is the /etc/utmp file which the login program writes. Similarly, there are
631 processes which may not wish to allow their information label to float. An example
632 of this would be a server process which must fork off children to perform work
633 in response to a specific request. The INF_NOFLOAT_SUBJ supports these types
634 of processes.

635 **CAP_INF_RELABEL_OBJ and CAP_INF_RELABEL_SUBJ**

636 These capabilities allow processes to explicitly set labels on subjects and objects.
637 As information labels are not an access control policy separate overrides for reading
638 and writing object labels are unnecessary. Rather a single capability is
639 sufficient for applications which need to manipulate information labels on objects.

640 **B.25.5 New Capability Functions**

641 **B.25.5.1 Function Naming Scheme**

642 In order to provide for consistency across the sections of this document, a naming
643 scheme for all named entities was adopted. Functions are named with a subsystem
644 identifier—cap_, first, followed by a short name that identifies the type of
645 operation the function performs, then a short name that identifies the data the
646 function operates on. While this scheme generates names that are somewhat
647 longer than are generally customary, it is generally evident from the name of the
648 function what its purpose is and we found it easier to remember them. |

649 **B.25.5.2 Allocate, Duplicate, and Release Storage for Capability State** |

650 The *cap_init()* function is necessary to create a new object to hold capability attributes.
651 We did not desire to specify the contents and storage requirements of this
652 object in order to permit as many differing implementations as possible. Having
653 provided an allocation function, we need also to provide a free function, *cap_free()*,
654 so that an implementor can release memory and structures associated with a
655 process capability data object. In order to permit the representation to be copied, we
656 defined a duplication function, *cap_dup()*. |

657 **B.25.5.3 Initialize a Process Capability Data Object**

658 The *cap_clear()* function permits a program to set the representation of the capa-
659 bility state to a known secure state. This has the advantage that a conforming
660 program need not know all the capabilities defined in the implementation to set
661 this “secure” state.

662 **B.25.5.4 Read and Write the Capability Flags of a Process**

663 The *cap_set_proc()* and *cap_get_proc()* functions permit a program to obtain and
664 set the capability state of a process atomically. The atomicity of these functions is
665 significant—the state of a process could possibly change between multiple invoca-
666 tions of a function that deals with only one capability flag at a time.

667 The *cap_set_proc()* function is an especially security-critical function in any sys-
668 tem that implements a capability mechanism, as it is here that the standard
669 requires that the security policy regarding the manipulation of process capability
670 state be applied. The requirement that the capability be permitted to the running
671 program provides the primary means to limit what capabilities any one program
672 can propagate through the system.

673 **B.25.5.5 Get and Set Values of Capability Flags**

674 The *cap_get_flag()*, and *cap_set_flag()* functions provide the standard interface for
675 getting and setting the values of the capability flags. Portable trusted applica-
676 tions will need to manipulate the process capability state on different implemen-
677 tations so that they can perform “time bounding of capabilities” and set what
678 capabilities they want to pass on to programs that they *exec*. The *cap_get_flag()*
679 function permits a conforming application to determine the state of a capability
680 without actually attempting to use it. Without a *get* function, conforming applica-
681 tions could generate numerous unnecessary audit messages attempting to use
682 capabilities not available to the current invocation of the program. The
683 *cap_set_flag()* is the only means by which a conforming application can alter the
684 state of a specific capability.

685 **B.25.5.6 Exporting Capability Data**

686 The *cap_to_text()* and *cap_from_text()* functions translate process capability |
687 states between human-readable text and capability data object representations. |
688 These functions are necessary to provide a portable means of transferring capabil- |
689 ity information between systems. Implementations may also use these functions |
690 to translate between text and data objects in order to support capability manipu- |
691 lation and display. One possible use is the display of available capabilities using |
692 a trusted shell utility, another is the transport of capability information across a |
693 network in a form recognizable to all machines.

694 There are other valid reasons to want to store process capability data objects—for |
695 instance, the process capability state could be an important field in certain audit |
696 records. Textual data, while easily readable, is not compact. The internal |

697 representation of capability state is not guaranteed by this standard to be valid
698 outside of the context in which it exists. For instance, it may contain pointers to
699 strings spread throughout the system-managed space. This was intentional to
700 permit implementors the maximum possible freedom. Because of this, the
701 *cap_copy_ext()* and *cap_copy_int()* functions are provided to convert the internal
702 representation to and from a self-contained binary format that should be more
703 compact than the textual version.

704 **B.25.5.7 Manipulating File Capability Flags**

705 When we developed the set of functions to manipulate file capability flags, we had
706 several goals in mind. First, we wanted the assignment of capability attributes to
707 files to be atomic—there is a reasonable probability that a program file could be
708 executed by another process in the middle of a sequence of non-atomic file attri-
709 but operations. Second, we wanted to continue to hide the actual representation
710 of capability attributes in the standard and permit a wide variety of implementa-
711 tions. We feel that the interfaces defined support an implementation where the
712 file capability attributes are stored in the files' inode AND an implementation
713 where the files' capability attributes are stored in a central database maintained
714 by a capability server. Finally, the group as a whole decided to specify procedural
715 interfaces wherever possible instead of data-oriented interfaces in order to better
716 support extensibility and flexibility in the future.

717 We did not resolve the atomicity problem to the extent we desired, but felt that
718 the correct solution was really outside of our scope. POSIX has no mandatory file
719 locking mechanism, hence, there exists the possibility that file attributes have
720 been altered by a second process between the time the first process has read them
721 and the time it attempts to set them. This is a general problem not limited to file
722 capability state, but includes all file attributes and data. Instead of solving the
723 general problem, we have specified functions that read and write the entire capa-
724 bility state, rather than permit programs direct access to individual capability
725 flags and attributes. This should minimize, but not eliminate, this problem. —

726 **B.25.5.8 Read and Write the Capability State of a File**

727 The *cap_get_file()* and *cap_set_file()* functions permit a program to obtain and set
728 the capability state of a file atomically. The atomicity of these functions is
729 significant—the state of a file could change between multiple invocations of a
730 function that deals with only one capability flag at a time. In addition, it keeps
731 device I/O required by the capability function set to these two functions—all the
732 rest can (but are not required to) be memory only operations.

733 The *cap_set_file()* function is a security-critical function in any system that imple-
734 ments a capability mechanism. We therefore imposed a number of restrictions on
735 the ability of programs to use this function. The requirement that the capability
736 be permitted to the running program provides the means to limit what capabili-
737 ties any one program can propagate through the system. The requirement to
738 have the CAP_SETFCAP capability effective provides the means to restrict pro-
739 grams that are permitted a capability for other purposes from granting it to

740 programs that the system administrator has not specifically approved. The
741 remaining restriction is that the UID associated with the process be equal to the
742 owner of the file or that the process have the CAP_FOWNER capability
743 effective—this is a standard restriction for all operations dealing with file attri-
744 butes. The combination of restrictions above are the minimum necessary to
745 prevent the unauthorized propagation of capabilities. —

746 Many times a file is already opened when it is being assigned attributes. Many
747 programs use file-descriptor based functions in order to avoid the performance
748 penalty incurred to perform repeated pathname resolutions. To accommodate
749 this class of applications, we have provided the *cap_set_fd()* and *cap_get_fd()*
750 functions to set and get the capability state of an opened file. +

751 **B.25.5.9 Error Return Values** +

752 If the symbol *{_POSIX_CAP}* is defined, then the implementation supports the +
753 capability option and is required to support the capability functions as described +
754 in this standard. If the symbol *{_POSIX_CAP}* is not defined, then the implemen- +
755 tation does not claim conformance to the capability option and the results of an +
756 application calling any of the capability functions are not specified within this +
757 standard. An alternative is for the capability functions to specify that the error +
758 return code [ENOSYS] be returned by the functions if the capability option is not +
759 supported. However, in order to remain compliant with the policies of POSIX.1, +
760 this standard cannot specify any requirements for implementations that do not +
761 support the option. —

762 **B.25.6 Examples of Capability Inheritance and Assignment**

763 **B.25.6.1 A User-based Capability Model**

764 The inheritance mechanism provides a method of controlling a process' capabili-
765 ties based upon the context in which the process is executed. An important part
766 of the context is the identity of the user invoking the process. It is possible to
767 associate capabilities with a user profile which defines a subset of the capabilities
768 available to the trusted programs that a user may execute. Trusted programs
769 may therefore have greater or lesser abilities depending on which user executes
770 them. These user capabilities constitute the *inheritable* capability set on session
771 initialization. A subset of the user capabilities could be selected by utility options
772 to support user roles. The login shell will probably be an untrusted shell, and in
773 itself be incapable of using capability. —

774 It is not possible for a user to alter the set of *inheritable* capabilities within an
775 untrusted shell or program. A user can only modify the set of *inheritable* capabili-
776 ties by executing a program that gains capabilities either by having *effective* capa-
777 bilities or by having *permitted* capabilities that have already been set *inheritable*. |
778 Programs that have *effective* capabilities may validate a user's authorization to
779 use those capabilities, depending on whether or not the execution of the program
780 could have an adverse impact on the security of the system. This mechanism per-
781 mits the emulation of a fully privileged user by executing a program that has all |

782 capabilities *effective*.

783 **B.25.6.2 A Program-based Capability Model**

784 Instead of forcing every trusted application to perform user authorization checks,
785 it is possible to create a single program that does so, and sets the *inheritable* flag
786 of all capabilities authorized to a user. Program files in this style of implementa-
787 tion would have the *permitted* flags of all the capabilities they require for all their
788 possible functions set. When executed, the program would receive only those
789 capabilities actually authorized to the user, not necessarily the full set that they
790 are capable of using. It is thus possible to provide a trusted shell or user interface
791 program that will assign additional capabilities or disable existing capabilities
792 associated with a user based upon the specific functions to be performed and then
793 invoke one or more programs that are relieved of having to perform a user auth-
794 ization check.

795 It is not possible for an executing program to acquire additional capability for
796 itself through the execution of a more trusted program, i.e., through *exec*'ing a
797 more trusted executable file, but only to create a new process image that is more
798 trusted than it is. Since the new process image has, by definition, replaced the
799 old process image, attempts to garner additional capability in this manner will
800 fail.

801 **B.25.7 Capability Worked Examples**

802 This section illustrates the POSIX.1e Capability mechanism by providing both
803 utility and function examples. Included are examples using the POSIX.2 chown
804 utility and POSIX.1 *chown()* function, examples of capability unaware programs,
805 and an illustration of how the capability mechanism defined in this standard can
806 be used to execute shell scripts.

807 **B.25.7.1 CHOWN()**

808 To change the user ID of a file, the *chown()* function imposes the following restric-
809 tions:

- 810 • A process shall possess an effective user ID equal to the user ID of the file, or its
811 effective capability set shall include the CAP_FOWNER capability.
- 812 • If the {_POSIX_CHOWN_RESTRICTED} option is in effect for the file, the pro-
813 cess' effective capability set shall include the CAP_CHOWN capability. Thus, to
814 change the user ID of the file, both the CAP_CHOWN and CAP_FOWNER capa-
815 bilities may be required in the process' effective capability set. If the system
816 implements the MAC option of this standard, the process may also require the
817 CAP_MAC_WRITE capability in the process' effective capability set.
- 818 • If the file is a regular file, the set-user-ID (S_ISUID) and set-group-ID (S_ISGID)
819 bits of the file mode shall be cleared upon successful return from *chown()*, unless
820 the call is made by a process whose effective capability set includes the
821 CAP_FSETID capability, in which case, it is implementation defined whether

822 those bits are altered.

823 In examples 1 through 3 below, the *chown()* executable file is assigned, via the
824 *cap_set_file()* function, an empty effective set, an inheritable capability set that
825 includes:

826 • CAP_FOWNER

827 • CAP_CHOWN

828 • CAP_FSETID

829 and a permitted capability set with flags set to potentially allow bypassing of DAC
830 and MAC restrictions imposed by the *chown()* function (See 25.2 for capability
831 descriptions.)

832 • CAP_DAC_READ_SEARCH

833 • CAP_MAC_READ

834 • CAP_MAC_WRITE

835 EXAMPLE 1

836 If the *chown()* utility is executed by a process that possesses all the above capabilities
837 in its inheritable capability set, then all of these capabilities are included in
838 the resulting process's permitted capability set. When these capabilities are made
839 effective, via the *cap_set_proc()* function, the process may change the user ID of
840 the specified file without regard for mandatory and discretionary access restrictions,
841 file ownership restrictions, or {*POSIX_CHOWN_RESTRICTED*} restrictions.
842 Alteration of the set-user-ID and set-group-ID bits of the file mode is implementation
843 defined upon successful return from *chown()*.

844 EXAMPLE 2

845 If the *chown()* utility is executed by a process that possesses no capabilities in its
846 inheritable capability set, then the resulting process's permitted capability set
847 will not contain the three required capabilities. Therefore, the resulting process
848 shall not possess appropriate capabilities to override any of the *chown()* restrictions
849 described above.

850 EXAMPLE 3

851 If the *chown()* utility is executed by a process that possesses only the
852 CAP_CHOWN and CAP_FOWNER capabilities in its inheritable capability set,
853 then the resulting process will possess the CAP_CHOWN and CAP_FOWNER
854 capabilities in its permitted capability set. When these capabilities are made
855 effective, via the *cap_set_proc()* function, the process may change the user ID of
856 the file, regardless of the file's initial user ID, or value of
857 {*POSIX_CHOWN_RESTRICTED*}. However, this process must satisfy all mandatory
858 and discretionary access requirements, and the set-user-ID and set-group-
859 ID bits of the file mode shall be cleared upon successful return from *chown()*.

860 **EXAMPLE 4**

861 In this example, the file capabilities are initialized as described for examples 1
862 through 3, above, except that the CAP_FSETID capability is removed from the
863 chown executable file's inheritable capability set, and is assigned to the file's per-
864 mitted capability set. The process resulting from execution of the chown utility
865 will possess the CAP_FSETID capability as part of its permitted capability set,
866 regardless of the contents of the exec'ing process's inheritable capability set.
867 When the CAP_FSETID capability is made effective, via the *cap_set_proc()* func-
868 tion, alteration of the set-user-ID and set-group-ID bits of the file mode is imple-
869 mentation defined upon successful return from *chown()*.

870 **B.25.7.2 Capability Unaware Programs**

871 In this section, we examine the behavior of capability unaware programs. This
872 specification provides support for backwards compatibility of binary executables
873 that depend on traditional UNIX set-user-ID behavior for proper operation. This
874 specification also provides a mechanism for overriding capability on a per execut-
875 able basis. Additionally, the *permitted* flag provides for a finer granularity of con-
876 trol to enable capabilities based on the *inheritable* flag of the exec'ing process. For
877 all capability unaware programs that require capability, the program file's *effec-*
878 *tive* flag must be set. This is the only mechanism for enabling capabilities in the
879 *effective* capability set upon execution.

880 **EXAMPLE 1**

881 Suppose an old version of the mailx program requires discretionary and manda-
882 tory override capabilities to operate correctly on a particular implementation.
883 These capabilities can be enabled via the *effective* capability set regardless of the
884 exec'ing process' *inheritable* capability set. This allows mailx to operate on a sys-
885 tem supporting {_POSIX_CAP} without modifying the mailx source code.

886 If an administrator desires to control which capabilities become *effective* based on
887 the exec'ing program's *inheritable* capabilities, then the *permitted* flag is used.
888 The *inheritable* flag is ANDed with the *permitted* flag and this result is included
889 in the new process' *effective* flag.

890 **EXAMPLE 2**

891 The grep program may have the CAP_DAC_READ_SEARCH capability enabled
892 in the *permitted* capability set, which would then permit the invoker to access all
893 files if and only if the CAP_DAC_READ_SEARCH *inheritable* capability was
894 enabled in the exec'ing process. This would permit a trusted process to *exec* the
895 grep program to locate a phrase in a file tree it normally would not have read
896 access to.

897 **B.25.7.3 Shell Script Execution**

898 A shell script can be executed with capability using the capability mechanism
899 defined in this standard. For example, a program stub can be created that can be
900 invoked from the login shell that sets the inheritable capability attributes for
901 those capabilities needed for shell script execution. The *system()* function can
902 then be invoked to execute the shell script file. The capabilities set in the inherit-
903 able capability set are then passed through the shell executed by the *system()*
904 function to the individual utilities constituting the shell script. The capabilities
905 available to each utility are then determined by the *exec()* function as described in
906 the capability mechanism.

907 **B.25.7.4 Textual Representation of Capability States**

908 The purpose of this clause is to specify a single, portable format for representing a
909 capability state. This textual representation is intended for use by the
910 *cap_to_text()* function and the *getcap* command to represent the state of an
911 existing capability state object, and by the *cap_from_text()* function and the
912 *setcap* command to translate a textual representation of a capability state into
913 its internal form.

914 Examples of valid textual capability state specifications include:

915 No flags for any capabilities defined in the implementation are set:

```
916     "all="
917     "="
918     "CAP_CHOWN=
919     CAP_DAC_OVERRIDE=
920     ...
921     <all remaining POSIX-defined capabilities>
922     <implementation-defined capability>=
923     <implementation-defined capability>=
924     ...
925     <all remaining implementation-defined capabilities>
926     "
```

927 Only the permitted flags for CAP_KILL, CAP_CHOWN, and CAP_DAC_OVERRIDE
928 are set. The remaining flags for the remaining capabilities are all
929 cleared:

```
930     "CAP_KILL,CAP_CHOWN,CAP_DAC_OVERRIDE=p"
931
932     "all=
933     CAP_KILL=p CAP_CHOWN=+p-ei
934     CAP_DAC_OVERRIDE=p"
```

935 The inheritable flag for every capability defined by the implementation
936 is set except for the CAP_MAC_* capabilities. The effective flag is set
937 for the CAP_DAC_OVERRIDE capability:

```
938     "all=i
939     CAP_MAC_READ,CAP_MAC_WRITE,CAP_MAC_DOWNGRADE,CAP_MAC_LOCK
940     CAP_DAC_OVERRIDE+e"
941
```

942 In order to promote the portability of capability state information between imple-
943 ments, one representation must be specified in this standard. We chose to
944 standardize the textual representation as this promotes not only application por-
945 tability but user portability as well.

946 We considered an alternative representation that was flag `set` oriented, i.e.,
947 something that would look like:

```
948     i=CAP_KILL,CAP_MAC_WRITE
949     p=all
950     ...
```

951 however, this was rejected as implying a specific implementation (e.g., implemen-
952 tation of capability data objects as multiple set structures) and potentially being
953 less compact (a privilege having all flags set must be named separately for each
954 flag.) In addition, the requirement in such a representation to name a capability
955 multiple times greatly increases the chances for human error when attempting to
956 specify or interpret the representation.

957 In general, it is felt that this specification provides implementations with a wide
958 degree of flexibility in how they can represent capability states, while ensuring
959 that they can correctly interpret such states created on other interpretations with
960 a minimum of difficulty and implementation complexity. The same state can be
961 represented in a compact manner or a lengthy manner, depending on the purpose
962 for which it is intended.

1 **B.26 Mandatory Access Control**

2 **B.26.1 Goals**

3 The primary goal of adding support for a Mandatory Access Control (MAC)
4 mechanism in the POSIX.1 specification is to provide interfaces to mandatory
5 security policies. A mandatory security policy is a system-enforced access control
6 policy that is outside the control of unprivileged users. Additional goals included
7 are to:

- 8 (1) address mandatory access controls that support appropriate, widely
9 recognized criteria, while providing as much flexibility for
10 implementation-specific MAC policies as is practical;
- 11 (2) define MAC interfaces for portable, trusted applications and specify MAC
12 restrictions on all other POSIX.1 functions;
- 13 (3) preserve the provision for POSIX.1 conforming applications to impose
- 14 (4) preserve 100% compatibility to the base POSIX.1 functionality among
15 subjects and objects operating under “single label conditions”, i.e., all
16 subjects and objects have an equivalent MAC label;
- 17 (5) add no new MAC-specific error messages to existing POSIX.1 and other
18 interface standards, as doing so could interfere with the desire to avoid
19 new covert channels.

20 The mandatory access control (MAC) interfaces are intended to be compatible
21 with the mandatory access requirements of a number of criteria, particularly com-
22 patibility with the U.S. TCSEC levels B1-B3, the European ITSEC functionality
23 levels FB1/FB3, and the U.S. CMW requirements for MAC. It should be noted
24 that compatibility with these criteria extends only to the functionality defined in
25 them, and not to the assurances they may require. Additionally, the interfaces
26 were designed to conform with the requirements for adding “extended security
27 controls” to POSIX-conforming systems, as stated in POSIX.1, section 2.3.1.

28 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.

35 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.

40 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.

51 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.

56 **B.26.2 Scope**

57 Section 26 defines and discusses the overall MAC policy and refinements of this 58 overall restriction for the two major current policy areas: files and processes.

59 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.

63 **B.26.2.1 Downgrade and Upgrade**

64 The definitions of downgrade and upgrade are the technically precise ones. They 65 may not be intuitive because downgrade includes incomparable labels. For example, changing *Secret:A* to *Top_Secret:B* is a downgrade.

67 B.26.2.2 Concepts Not Included

68 Several concepts that will commonly be implemented by conforming systems have
69 not been treated by this document, many because they have no basis in the
70 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.

85 The label testing function, *mac_valid()* is intended to help
86 provide an interface to at least some of these restrictions in
87 a more portable manner. For example, the restrictions may
88 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.

94 Devices: Access to devices through device special files is not treated
95 in this document. Often implementations may have special
96 device access rules based on device-specific considerations.
97 Two common examples of such special device access rules
98 are device “ranges” (sets of allowed MAC labels for accessing
99 certain devices), and “public,” generally-accessible devices,
100 such as /dev/null and /dev/tty. Since such device-
101 specific considerations have no basis in POSIX.1, devices as
102 a whole are not addressed in this document.

103 File Systems: Mounted file systems are not included.

104 Trusted User Commands:
105 Commands for both administrators and trusted or partially
106 trusted users have not been included.

107 Label Translation: POSIX.1 does not address networked systems. Thus, the
108 issue of translating MAC labels into a portable form is not
109 addressed in this standard.

110 Process Label Functions:
111 The functions provided as part of this standard to retrieve or
112 set the MAC label associated with a process are limited to
113 the requesting process. That is, no interface is provided
114 whereby a process may specify another process (for example,
115 using a process id) to be the target of the *mac_set_proc()* or
116 *mac_get_proc()* functions. Such mechanisms have been
117 omitted in order to be consistent with the POSIX.1 standard
118 which provides no facilities for processes to manipulate, or
119 be cognizant of, other processes' state information. Note,
120 however, that conforming implementations may choose to
121 provide such functions.

122 **B.26.3 File Object Model**

123 An important part of mandatory access control for files is the seemingly simple
124 assumption that the file attributes and data comprise a logically single data
125 container to which the file MAC label is applied—the “file object.” Virtually all MAC
126 function restrictions arise from applying the following two basic policy rules
127 under this assumption:

- 128 **(FP.1)** The MAC label of a file must be *dominated* by the MAC label of a pro-
129 cess for the process to read the data or attributes of a file, and
130 **(FP.2)** The MAC label of a file must *dominate* the MAC label of a process for
131 the process to write the data or attributes of a file. (Allowed restric-
132 tions on this rule are discussed following in *Direct Write-up*).

133 For example, linking to a file involves altering the link count of that file, and
134 hence MAC write access to the file is required (as well as appropriate restrictions
135 on the directory in which the link is created). This is discussed below in the *link()*
136 example.

137 MAC restrictions for virtually all file-related functions can be straightforwardly
138 derived from these basic policy assumptions. (See the Policy section for a com-
139 plete list.)

140 Two examples:

141 *mkdir()*

142 The *mkdir()* function is used to create a directory, **D**. Apart from actually
143 creating the directory itself, a link name must be placed in the specified
144 parent directory, **PD**. Application of the **FP.1** and **FP.2** yields the MAC res-
145 trictions:

- 146 (1) The process shall have search access to **PD**. (Search access is an out-
147 growth of **FP.1**.)
148 (2) In order to add the link name, the process shall have MAC write access
149 to **PD**, i.e., the MAC label of the process shall be dominated by that of the
150 directory (from **FP.2**).

151 **D** is created with the MAC label of the process (**FP.4**), and hence it is correct
152 to leave the file open to the process.

153 Note that the calls *creat()*, *mkfifo()*, and *open-for-create* are other functions
154 that create files and will have these same MAC restrictions.

155 *link()*

156 The *link()* function is a little more complicated. A new link is to be created in
157 a directory **D** to an existing file **F**. This involves writing the new link name to
158 **F** into **D**. Hence the following MAC rules are applied:

159 (1) The process shall have search access to the directory **D**.

160 (2) The process shall also have search access to the file **F**, because the func-
161 tion is implicitly testing for the existence of **F**.

162 (3) The process shall have MAC write access to **D**, i.e., under **FP.2** the MAC
163 label of the process shall be dominated by that of **D**.

164 In making a new link to **F**, the link count of **F** must be increased. Hence,
165 the process is implicitly writing into **F**, and:

166 (4) The process shall have MAC write access to **F**, under **FP.2**.

167 **B.26.4 Direct Write-up**

168 Originally, FP.2 dictated that a process can only open files for writing whose label
169 equals that of the process (“write-at-label”), but that, a POSIX.1e conforming
170 implementation could allow write access under relaxed conditions, in particular,
171 when the MAC label of the file properly dominates that of the process. Because
172 POSIX.1 mandates that additional conditions can only be more restrictive, this
173 was changed to write-up, with write-equal allowed as part of a fully conforming
174 implementation.

175 The usefulness of allowing *open-for-write* of higher-label files (“direct write-up”)
176 seemed too small given potential implementation difficulties. For this reason,
177 direct write-up was not required by the standard. However, direct write-up may
178 be a useful feature for the vendor willing to address its implementation problems,
179 and for this reason, along with the reason cited above, the change was made.

180 Implementations which implement direct write-up will need to consider the
181 impact on return codes and potential covert channels.

182 Note that the creator of a portable application cannot assume such relaxations are
183 present because they are not required by the standard. Write-at-label must
184 instead be assumed as the rule for MAC write.

185 In the following discussions, it is generally assumed that write-at-label is the
186 case.

187 The option of creating objects with MAC labels dominating that of the creating
188 process is allowed, but interfaces to do so are not provided. This facility would be
189 effected by the same set of concerns expressed with regard to direct write-up,

190 hence the more conservative approach. Furthermore, providing an interface for
191 creating an object with a MAC label begs the question of why we don't provide a
192 mechanism for an ACL and a capability set.

193 **B.26.5 Protection of Link Names**

194 As discussed above, in POSIX.1 there really is no such thing as a “filename.” This
195 is true both logically and physically, i.e., no name is stored in the file itself.
196 Instead there are only link names to files that are both logically and physically
197 data items within the parent directory.

198 This proposal takes the most direct interpretation of the protection of link names
199 within a directory: the link names are simply considered data in the directory.
200 This means that the names are protected by the MAC label of the directory that
201 contains them, even when they indicate files or directories at other MAC labels.

202 A process could determine the link name and hence existence of objects at labels
203 not dominated by the process. However, this cannot be used as a covert channel
204 because the process that defined those names must have had write access to the
205 containing directory, which means that its label equals (or, in some implementa-
206 tions, was dominated by) the label of the directory. More precisely, the covert
207 channel “sender” that creates the link name must be equal to (or, in some imple-
208 mentations, dominated by) the MAC label of the directory, and the “reader” must
209 dominate that label. Hence, because information is at most going to a higher
210 MAC label there is no covert channel.

211 Since link names may be protected at a lower MAC label than the file to which
212 they point, the user must be careful to choose a name that is adequately protected
213 at the MAC label of the parent directory.

214 This interpretation is both natural and common for UNIX file systems and under-
215 scores that link (“file”) names are not a property of the file, but rather of a parent
216 directory.

217 One of the suggested alternatives is the so-called “name-hiding” model where
218 each link in a directory is considered an object labeled at the label of the file to
219 which it links. This alternative was rejected because it is more complex, and
220 doesn’t offer any real improvement over the alternative that was accepted. Access
221 to the link names in a directory must therefore be controlled on a per-link basis.

222 **B.26.6 Pathname Search Access**

223 Files are commonly referenced by a pathname, for example **A/B/F**. If the path-
224 name starts with the “/” character, then the pathname starts at the absolute root
225 of the file system. Otherwise it starts at the current working directory of the pro-
226 cess. Even pathnames that contain only one name, e.g., **F**, are still pathnames.
227 Each such reference requires an implicit reading of a sequence of directories, and
228 **FP.1** must be applied to this process. This is called *search access* in this docu-
229 ment.

230 A pathname consists of a sequence of *link* names (**A**, **B**, and **F** in the previous
231 example) i.e., each name in the pathname is a link name contained in some direc-
232 tory. In other words, the name that most users commonly assume is “attached to”
233 a file is actually the name of a link in a directory, where the link only points to
234 the actual file. There may be many such links with different names to a single
235 file.

236 In locating a file on behalf of the process, the system is in effect opening the
237 sequence of directories that contain the link names in the pathname, finding
238 (reading) the next link name, and proceeding to the next file or directory named
239 by that link. The following basic constraint is required under **FP.1**:

240 **MAC Search Access**

241 In order for the system to perform this implicit reading of the directories in
242 the pathname, the process is required to have MAC read access to each direc-
243 tory that contains a link name of the pathname. Specifically, the MAC label
244 of the process must dominate that of the directory. (Note that ACL execute
245 “x” permission is also required, as in standard POSIX.)

246 For relative pathnames, the current working directory (“.”) is considered the first,
247 implicit directory in the pathname and is checked first. For absolute pathnames,
248 the absolute root directory is checked first, and, because it is customarily at the
249 lowest MAC label on the system, search access will always proceed from absolute
250 root.

251 Note that the last element in the pathname is the final link name. Once this final
252 link name is read from the directory in which it resides, search access is con-
253 sidered complete. Hence, by definition the final target element (**F** in the current
254 example) is not itself checked for any MAC access during search access, although
255 it will certainly be checked in the context of specific operations.

256 Basically, MAC search access determines whether a process can detect the
257 existence of a file, specifically, whether the process can read a directory containing
258 a link to the file.

259 As a general rule, MAC search access is applied to all pathnames presented in a
260 function. If this succeeds, then other MAC checks follow.

261 **B.26.7 Check-Access-on-Open Only**

262 The MAC policies follow the standing POSIX.1 metaphor that access to the data
263 portion of a file object is checked only when access is requested and not for each
264 data read and write. Subsequently, access to the file is not revoked or changed in
265 mode until the process willingly closes the file.

266 With this form of access, it is important that the MAC label of a file object not be
267 altered if the alteration would allow information flow to a subject which would
268 have not occurred at the new label. This requirement was originally stated in a
269 **FP.5**, but this was removed when it was pointed out that **FP.5** is really just say-
270 ing you can not violate **FP.1** or **FP.2**.

271 There are some conditions (which are rejected in this document) where the label
272 could technically be allowed to change:
273 — When the file references were write-only and the label was being raised. How-
274 ever, this seems a relatively rare case.
275 — When the system supported some type of access revocation or recalculation.
276 — Allow changing the label only when the requesting process is currently refer-
277 ring to the file.
278 — If all processes currently referencing the file were appropriately privileged,
279 then the label might be allowed to change. The danger here is that the privileged
280 processes may not be aware of the label change.
281 The application of **FP.1** and **FP.2** to the *mac_set_file()* and *mac_set_fd()* functions
282 takes the simple approach and make the handling implementation-defined as to
283 whether changing of the file label when there are open connections to the file,
284 (other than the calling process in the case of *mac_set_fd()*), are disallowed, even
285 when the processes are privileged, or whether revocation is performed.

286 **B.26.8 Creating Upgraded Directories**

287 An upgraded directory is one whose MAC label properly dominates that of its
288 parent directory.
289 While in general the operation of **FP.2** and **FP.4** do not allow unprivileged
290 processes to create files or directories at other than the process MAC level, some
291 means of creating multi-label file trees is necessary.
292 In particular, the ability to create upgraded directories gives a convenient means
293 for organizing a multi-label file tree appropriately, and need not violate any fun-
294 damental security constraints. Hence it is appropriate to provide unprivileged
295 processes with some means of doing so; though it has been chosen not to do so as
296 part of this standard.

297 **B.26.9 Objects without MAC labels**

298 This standard specifies that each file will always have a MAC label associated
299 with the file, but does not require each file to have its own unique MAC label.
300 Originally, the provided MAC functions allowed for returning [ENOSYS] if
301 {_POSIX_MAC} was defined and the specified file did not have its own MAC label.
302 This was subsequently changed because of objections to the overloading of
303 [ENOSYS] to return [ENOTSUP] for the cases where a file does not have its own
304 MAC label.
305 A *pathconf()* variable {_POSIX_MAC_PRESENT} is provided to allow applications
306 to determine if a file has its own MAC label. This standard does not specify the
307 specific situations where a file does not have its own MAC label. Examples of pos-
308 sible situations are: read only file systems; pre-existing file systems with
309 insufficient space to insert MAC labels; and certain devices such as /dev/null. The

310 *mac_get_file()* and *mac_get_fd()* functions will always return a MAC label because
311 each file will always have a MAC label associated with the file. The *mac_set_file()*
312 and *mac_set_fd()* functions can return [ENOTSUP] if the specified file does not
313 have its own unique MAC label but shares the MAC label of a file system.

314 **B.26.10 Error Return Values**

315 The MAC functions specified in this standard may return one of several errors
316 depending on how the implementation has addressed MAC labeling.

317 If the symbol `{_POSIX_MAC}` is defined, then the implementation supports the +
318 MAC option and is required to support the MAC functions as described in this +
319 standard. If the symbol `{_POSIX_MAC}` is not defined, then the implementation +
320 does not claim conformance to the MAC option and the results of an application +
321 calling any of the MAC functions are not specified within this standard. An alter- +
322 native is for the MAC functions to specify that the error return code [ENOSYS] be+
323 returned by the functions if the MAC option is not supported. However, in order +
324 to remain compliant with the policies of POSIX.1, this standard cannot specify +
325 any requirements for implementations that do not support the option.

326 The error [ENOTSUP] shall be returned in those cases where the system supports
327 MAC but the particular operation cannot be applied because restrictions imposed
328 by the implementation. For example, if an application attempts to set the MAC
329 label on a file on a system where *sysconf()* indicates that an MAC is supported by
330 the system, but the value that *pathconf()* returns for `{_POSIX_MAC_PRESENT}`
331 for that file indicates that individual MAC labels are not supported on that file,
332 the application shall receive the [ENOTSUP] error. Therefore, if an application
333 attempts to set the MAC label on a file, it is the application's responsibility to first
334 use *pathconf()* to determine whether the implementation supports MAC labels on
335 that file.

336 It should be noted that, in general, this standard attempts to avoid adding and
337 defining new errors. However, in the case of [ENOTSUP], the following points
338 were noted: First, the need exists to provide feedback to applications concerning
339 a new error condition. Second, while it is possible to use an existing error code in
340 such cases (for example, ENOSYS), the group felt that this would overload those
341 errors. P1003.1, when consulted, concurred with this view and agreed that the
342 creation of a new error code, in this case, was appropriate. Third, the error
343 [ENOTSUP] is also being used by P1003.4 for roughly the same reasons. There-
344 fore, the consensus of several POSIX working groups is that while adding new
345 errors is generally not recommended, that this case warrants the creation of a
346 new error and that the new error should be [ENOTSUP].

347 The [EINVAL] error is returned by functions when the MAC label specified in the
348 function call is syntactically incorrect or the MAC label is not permitted on the
349 system because implementation-defined restrictions, (e.g., range restrictions).
350 That is, this error is used to indicate the invalidity of the MAC label specified,
351 independent of whether the operation would have succeeded had it been a valid
352 label.

353 Although POSIX.1 does not specify precedence for error return values, careful
354 consideration should be given to this matter in the security standard to ensure
355 that covert channel considerations are adequately addressed. Specifically, if an
356 unprivileged application attempts a function for which privileges are required and
357 the implementation returns the EINVAL error in favor of the EPERM error, it
358 may be possible for the application to determine the system's MAC label range
359 restrictions based on whether EINVAL is returned (indicating the label is outside
360 the system's range), or EPERM is returned (indicating the label is valid for the
361 system, but that the application failed the privilege check). Therefore, despite
362 this standard's silence on the issue, it is recommended that when a function could
363 return multiple errors in a particular instance, that the errors be given the follow-
364 ing precedence (from most favored to least favored): EPERM, EINVAL,
365 ENOTSUP.

366 **B.26.11 Valid MAC Labels**

367 MAC labels have two forms: internal and external.

368 The basic MAC label structure defined in this standard (*mac_t*) is a pointer to an
369 opaque data structure. The binary format of that opaque data structure may
370 include such data as a hierarchical classification and non-hierarchical categories.
371 The standard makes no assumptions regarding the underlying representation
372 other than imposing the following constraint: the structure must be an export-
373 able object. That is, the structure is opaque, persistent, and self-contained. The
374 structure can therefore be copied by duplicating the bytes without knowledge of
375 its syntax. Such a copy can be changed without any effect on the original, and the
376 original can be changed without any effect on the copy.

377 The external format of a label is a text string of undetermined format. Any
378 separator character between fields in the textual representation is
379 implementation-defined. As noted in POSIX.1 section B.2.3.5, the character set
380 used for textual representation of MAC labels is not defined by this standard.

381 The meaning of a valid MAC label is implementation-defined, as described in
382 *mac_valid()*. A MAC label could be invalid for many reasons, such as:

- 383 A. It is malformed, e.g., the label contains a checksum in the opaque type
384 which does not agree with the checksum calculated from the data.
- 385 B. It is out of the security level range of the system, e.g., the label refers to a
386 classification or category or combination which is outside the set of valid
387 MAC labels for the system.
- 388 C. It is out of the security level range of a process, e.g., the label refers to a
389 classification or category or combination which is outside the set of valid
390 MAC labels for a process.
- 391 D. It is outside the representation range, e.g., a system could allow no more
392 than n categories from a universe of m, even though each of the m categories
393 is valid.

394 Invalid MAC labels may appear for a number of reasons. Examples include: con-
395 structing a MAC label in process memory without regard to semantics of the bits,
396 importing a MAC label from a dissimilar system, reading a MAC label previously
397 stored in a file, etc. Note, however, that none of the MAC interfaces defined in
398 this standard will ever return an invalid MAC label.

399 The *mac_valid()* function is the means for an implementation to communicate to
400 a portable application that the application should not “deal with” certain MAC
401 labels—that they are undefined, disallowed, or some implementation-restricted
402 state. Note however that an implementation may impose additional restrictions
403 on the MAC labels for a particular object or process beyond the system-wide con-
404 straints that are addressed by *mac_valid()*.

405 **B.26.12 Modification of MAC labels**

406 Unlike some of the other features in this standard, the basic unit of data for man-
407 datory access control (the MAC label) is not usually manipulated. Interfaces and
408 a memory management model to support manipulation of MAC labels were
409 deemed inappropriate, except for the least upper and greatest lower bounds func-
410 tions discussed below.

411 **B.26.13 Least upper bounds and greatest lower bounds**

412 The function *mac_glb()* is useful for applications that wish to limit their activities
413 to those permitted by both labels. For example, if a user wants to know the max-
414 imum classification of data that the user can transmit via a network cleared for
415 MAC label *labelA* to a machine cleared for MAC label *labelB*. Likewise, the
416 *mac_lub()* function allows applications to determine a MAC label which dom-
417 inates two specified labels.

418 It is the intent that conforming applications only use these functions, rather than
419 more primitive manipulation of the label structures themselves.

420 **B.26.14 Functions returning MAC labels**

421 Functions which return MAC labels should use a common implementation specific
422 allocation mechanism. For example, *mac_get_file()* allocates space for a MAC
423 label, fills in the MAC label from the requested file system object, and returns a
424 pointer to this space to the caller. The system allocates space because a MAC
425 label could be of variable length in some implementations. Such systems include
426 those which use a sparse matrix representation. If the system did not allocate the
427 space a portable application would have to query the system about the size of a
428 (subject’s or object’s) MAC label, reserve space for the label, and then call another
429 function to obtain the MAC label. The overhead for systems with a fixed length
430 MAC label is excessive. The use of additional level of indirection in the present
431 interfaces accommodates systems with both fixed and variable sized labels with
432 reasonable efficiency.

433 The use of an allocator implies the use of a deallocator. The function *mac_free()*
434 frees the storage space allocated by any MAC function which allocated a MAC
435 label.

436 A function to allow for the translation of an internal label to an alternative exter-
437 nal label format was considered and rejected. For example, it is anticipated that
438 some trusted applications will wish to display a short form of the MAC label on a
439 display terminal, perhaps as part of an icon, rather than the entire (possibly very
440 lengthy) external text form. An option considered was to alter the *mac_to_text()*
441 function to include a form argument. Trusted applications could specify the exter-
442 nal form of the label desired, e.g., icon, abbreviated, long. The proposal was
443 rejected because the TCSEC, ITSEC, and CMW requirements criteria do not
444 specify alternative external formats. Thus, most implementations do not provide
445 for alternative text labels.

446 **B.26.15 Multi-level directories**

447 Interfaces to create, remove, and scan multi-level directories were considered and
448 actually appeared in earlier drafts, but were removed because a lack of consensus
449 and ballot objections. The basic reason for a multi-level directory mechanism is
450 that certain portions of the filesystem namespace are “well known” and need to be
451 publicly available. The most obvious example is /tmp; many applications expect
452 to be able to create files within this directory. However, in a system with MAC,
453 allowing applications at any level to freely create visible files in /tmp would be an
454 unacceptable security hole; it allows a trivial means for a Trojan horse program to
455 make great quantities of data visible at lower levels (by encoding the data in file
456 names).

457 Data at a MAC label higher than that of the multi-level directory may be stored in
458 the multi-level directory by an unprivileged user. However, access to this data
459 will still be governed by the MAC policy.

460 **B.26.15.1 Underlying Mechanism**

461 To overcome this problem, while still allowing applications free access to well
462 known directories, some means of hiding parts of the file system name space is
463 needed. The most direct method, what has been called a “true” multi-level direc-
464 tory, is to implement a new directory structure which allows entries to be truly
465 hidden. Here, for example, *readdir()* would only return entries at the requester’s
466 MAC level or lower. While conceptually nice, this is hard to implement properly.
467 For example, compatibility and prevention of a covert channel require lower level
468 processes (at least) to be able to create entries with the same names as pre-
469 existing ones created by higher-level processes. To avoid confusion, the appear-
470 ance of these names then needs to be altered somehow (for example, by appending
471 a representation of the label) for reference by higher-level processes. To avoid
472 other channels, the apparent size of the directory may need to be altered to
473 prevent visibility of creating and deleting files which might cause the size of the
474 directory to change.

475 A simpler implementation uses the separation already provided by subdirectories
476 to achieve the goal. References to pathnames such as /tmp/foo are “redirected”
477 during pathname resolution to “hidden” subdirectories of /tmp, usually to some-
478 thing like

479 /tmp/*LabelRepresentation*/foo

480 Here, *LabelRepresentation* tends to be a base 64 or hex representation of the
481 binary form of the label. These hidden subdirectories must of course be created
482 somehow, presumably either beforehand by a trusted program or administrator,
483 or as needed by the system.

484 **B.26.15.2 Getting Around The Hiding**

485 Both mechanisms hide part of the file system namespace from applications.
486 There are times when this is not desirable, e.g. when backing up filesystems, or
487 when a user simply wants to get at a lower level file. This is especially pressing
488 with the subdirectory approach, which conceals lower level files just as well as
489 higher level ones. Hence some means of generating a reference to an otherwise
490 invisible object is needed.

491 Again, two basic approaches have been taken. Either the reference is generated
492 directly by some special pathname:

493 /tmp/DON'T*DO*REDIRECTION!!/*LabelRepresentation*/foo

494 or it is generated indirectly by setting some process mode which allows using the
495 “real” filename

496 /tmp/*LabelRepresentation*/foo

497 The “modal” methods are less flexible in allowing redirected and real representa-
498 tions to be mixed, although some of this can be ameliorated by having multiple
499 modes such as

500 redirect none

501 redirect “system” directories (/tmp, /usr/tmp) only

502 redirect both system and application (/usr/spool/mail, etc.)
503 directories)

504 Their interaction with things like symbolic links involves difficulties as well.
505 (Allowing a symbolic link to a file in a hidden directory requires some means of
506 specifying the mode in the symbolic link.)

507 The “non-modal” special pathname method has the disadvantage of reserving part
508 of the file name space, something which unfortunately there is no precedent for in
509 historical implementations. If the portion reserved, e.g., the pathname com-
510 ponent

511 DON'T*DO*REDIRECTION!!

512 in the (fictitious) implementation above, were not standardized, a portable appli-
513 cation would have to abide by every namespace restriction imposed by every
514 implementation.

515 Finally, there are ways to address these issues without changing the way direc-
516 tories are processed at all. One such mechanism is the "variable symlink", in
517 which a component of the user's environment is used to replace a specified path-
518 name component in the symlink. Thus, if the symlink /tmp contained
519 "/orary/MACLABEL", a process with the environment variable MACLABEL set to
520 "secret" would be directed to "/orary/secret". Other mechanisms, such as an exotic
521 file system type, are also possible.

522 **B.26.16 The Directory Model**

523 The relationships between the MAC label of a directory and its subdirectories and
524 files is often referred to as the "directory model." One of the more common models
525 for POSIX-like systems is for files to equal and for directories to dominate the
526 label of their parent directories. This is sometimes called the "non-decreasing
527 directory" model because MAC labels at most increase as one traverses from the
528 root of a directory tree to its leaves. Multics, for example, used this model.

529 The following discussion applies only when untrusted processes are allowed to
530 create upgraded directories under one of the schemes above.

531 This proposal does not absolutely impose the non-decreasing directory model.
532 Neither does it prevent conforming implementations from imposing a non-
533 decreasing restriction. However, the application of the basic MAC restrictions on
534 the processes for accessing and creating the files as simple, labeled data con-
535 tainers leads to the restriction that unprivileged processes (users) can only create
536 non-decreasing directory trees. Privileged processes are not bound by these res-
537 trictions and can create files and directories at arbitrary MAC labels.

538 Implicit in the preceding discussion on upgraded directories is the assumption
539 that trees created by unprivileged processes will be non-decreasing.

540 The non-decreasing nature of file trees combined with the minor user difficulties
541 of creating upgraded directories (changing login sessions) will tend to group direc-
542 tories according to MAC label. That is, instead of highly intermixed files and
543 directories at various MAC labels, they will tend to be segregated according to
544 MAC label. This is generally a good practice anyway, because the close intermin-
545 gling of file system elements at different labels tends to be a breeding ground for
546 covert channels and confusion.

547 Basically, this proposal takes the position that non-decreasing hierarchies are
548 appropriate for unprivileged processes, but that POSIX.1e should not so restrict
549 appropriately privileged processes.

550 **B.26.17 File Tranquillity**

551 The original **FP.5** dealt with file object tranquillity. (Note, this rule was removed
552 as an explicit rule when it was pointed out that it is just a restatement of **FP.1**
553 and **FP.2**.)

554 **FP.5:** The MAC label of an object cannot be changed to a new MAC label if the
555 change would allow information flow between a process and an open file
556 object which could not have occurred at the new MAC label.

557 There are two general ways that a conforming implementation could enforce the
558 file change-level constraint:

559 **Tranquillity**

560 The change request could be denied if there were any open connections to the
561 file (other than the requesting process in the case of the *mac_set_fd()* func-
562 tion).

563 **Readjustment**

564 The change request could be fulfilled if it could be determined that all open
565 connections could have been made in the mode requested after the label was
566 changed. The implementation could either preemptively close the newly-
567 disallowed connections, or attempt to readjust the current access modes of the
568 open connections.

569 Readjustment can be difficult to implement and is not required by the standard,
570 but is also not precluded by the standard. Since readjustment is not required,
571 this leaves strict tranquillity as the lowest common denominator of conforming
572 implementations. For this reason, portable applications must assume no more
573 than strict tranquillity for maximum portability under the standard.

574 **B.26.18 Process Tranquillity**

575 Requirements for “process tranquillity” do not exist because any process
576 privileged to change its own label is presumed to ensure it does not subsequently
577 cause undesired information flows.

578 **B.26.19 Unnamed Pipes**

579 Unnamed pipes are considered labeled objects. However, because they are not
580 addressable, i.e., cannot be opened, and because MAC is enforced only when
581 objects are opened for access, there are never any actual MAC checks against the
582 label of the pipe. The label will however need to be retrieved in the *mac_get_fd()*
583 function.

584 The primary rationale for labeling unnamed pipes is so that processes using
585 *mac_get_fd()* (who may not know whether the file descriptor is a pipe) will not see
586 anomalous behavior for pipes.

587 **B.26.20 FIFOs**

588 First-in-first-out (FIFO) data objects have an inherent covert channel in that
589 higher-label readers can affect the state of the object in a manner that can be
590 detected by other (lower-label) readers/writers. For example, a reader/writer at
591 L_1 can write sequences to the FIFO and then determine how much data has been
592 read by a reader at L_2 by reading the FIFO (where L_2 is not dominated by L_1).
593 This constitutes information flow in that is contrary to the basic MAC policy
594 **FP.3.**

595 FIFOs in POSIX.1 include only FIFO-special files. In order to control the covert
596 channels for these FIFO-special files, the following rule is imposed:

597 Unprivileged processes may open FIFO-special files for reading only if the
598 process also has MAC write access to the FIFO, i.e., the process is at the
599 same MAC label as the FIFO-special file.

600 Hence, unprivileged processes at different MAC labels may not obtain a FIFO
601 between them even if opened such that information may only flow in accordance
602 with **P**.

603 **B.26.21 Inclusion of *mac_set_fd()***

604 Originally, this function was not included. It was felt that there was too little
605 demonstrated need for the function against potential implementation difficulties.
606 The only mentioned use was by *login*.

607 One notable implementation difficulty is that it is difficult to find the parent
608 directory (or directories) of a file given only a file descriptor. This makes it
609 difficult for implementations that wish to absolutely enforce the relationship
610 between a file and its parent directory. (Note that the issue of unique parent
611 directory is side-stepped when a pathname is given in that the directory given in
612 the pathname is the one to which various mandatory access controls are applied.)

613 However, in the interest of consistency with the other POSIX.1e options, it was
614 decided to include the *mac_set_fd()* function.

615 **B.26.22 Inclusion of *mac_size()***

616 The *mac_size()* function has been provided to allow applications to obtain the size
617 of a MAC label. Applications need to know the size of MAC labels only if they are
618 going to store the MAC label. There is no reason to know the size to use the pro-
619 vided MAC functions. An example of using the *mac_size()* function is a data base
620 system which needs to store a MAC label for each record. It would use the
621 *mac_size()* function to find out the size of the space to allocate and then could byte
622 copy the MAC label to the data base record.

623 **B.26.23 Restrictions on Signals**

624 The following, minimal MAC restriction governs the sending of signals:

625 An unprivileged process cannot send signals to another unprivileged process
626 when the signals would result in actions other than an upgrading of information,
627 i.e., the signal is only allowed when the label of the receiver dominates
628 that of the sender.

629 The general philosophy is to prohibit only those signals that can be repeatedly
630 sent thus causing high-bandwidth covert channels. This affects mainly the *kill()*
631 function.

632 No additional restrictions are imposed between two processes at the same label or
633 when at least one of the processes is privileged.

634 **B.26.24 Alteration of atime**

635 Many functions require that the file *atime* be marked for update. However, the
636 case where the actions of a process could affect the *atime* of a file whose label does
637 not dominate that of the process presents a potential covert channel. Some imple-
638 mentations can adjust when the *atime* is actually set and thus adequately confine
639 such covert channels, but this is not required by the standard. Instead, the effect
640 on *atime* in such cases is implementation-defined.

641 **B.26.25 Multi-Label Untrusted Process Hierarchies**

642 There are situations where untrusted processes at different MAC labels can have
643 an ancestral relationship. Processes with an ancestral relationship have special
644 opportunities for communicating information, e.g., *wait*, *waitpid* of POSIX.1 sec-
645 tion 3.2.1, and when both processes are untrusted and at different MAC labels
646 these opportunities present potential covert channels. There are no MAC restric-
647 tions for at least some of the following reasons:

648 — These situations can only be set up by trusted processes who change their MAC
649 label. It is assumed that a trusted process who changes its label and creates (by
650 *fork()* or *exec()*) untrusted processes will take actions to confine potential covert
651 channels.

652 — The channels are typically low-bandwidth.

653 — Restricting all such operations seems like too much imposition for too little
654 gain.

655 **B.26.26 File Status Queries**

656 Following the precedence of IEEE Std 1003.1-1990, no DAC access is required to
657 determine the various status attributes of a file (DAC information, labels, owner,
658 etc.) including all new attributes, such as the MAC label. However, MAC read
659 access is required to prevent potential covert channels.

1 **B.27 Information Labeling**

2 **B.27.1 Goals**

3 The primary goal of adding support for an information labeling mechanism in the
4 POSIX.1 specification is to provide interfaces to non-access control related data
5 labeling policies. An information labeling policy, unlike access control related pol-
6 icies (such as mandatory or discretionary access control), provides a means for
7 associating security-relevant information with the data maintained by the sys-
8 tem. More specifically, the information labeling mechanism's goals are to:

- 9 (1) Address the need for non-access control related mechanisms to imple-
10 ment data labeling policies as specified in existing standards and criteria
11 while providing as much flexibility for implementation-specific informa-
12 tion labeling policies as is practical. Specifically, to allow for the varia-
13 nces between existing standards, the interfaces are intended to provide
14 the latitude for implementations to support multiple information label
15 uses. For example: to allow information labels to be applied to subjects
16 and objects by the system, and altered by the system, to record the flow of
17 data between subjects and objects, or to allow information labels to be
18 applied to objects by users, and altered by them on a discretionary basis,
19 to record handling restrictions on the object contents.
- 20 (2) The information label interfaces are intended to be compatible with the
21 information label requirements of a number of standards and criteria. In
22 particular, goals include compatibility with the U.S. Compartmented
23 Mode Workstation Information Label requirements, and the European
24 vendor and customer demands, along with DIA document DDS-2600-
25 5502-87 and DIA document DDS-2600-6243-91. Finally, the interfaces
26 were designed to conform with the requirements for adding "extended
27 security controls" to POSIX-conforming systems, as stated in section
28 2.3.1 of POSIX.1.

29 There is a recognition that the underlying mechanisms involved can be
30 implemented in a number of different ways that still fulfill the
31 POSIX_INF requirements. Another consideration is the expectation that
32 POSIX.1 conforming systems will wish to extend the functionality
33 defined in this standard to meet particular, specialized needs. For these
34 reasons, flexibility in the POSIX_INF requirements while still conform-
35 ing to the criteria mentioned above, is an important objective.

- 36 (3) Define information labeling interfaces for conforming applications. By so
37 doing, it becomes possible to develop trusted applications which are port-
38 able across POSIX_INF-compliant implementations.
- 39 (4) Specify information labeling enhancements on other POSIX.1 functions
40 as necessary. Identifying information labeling modifications to other
41 POSIX.1 functions ensures that application developers are made aware
42 of possible changes required for their applications to function in a
43 POSIX_INF-compliant environment.
- 44 (5) Address information labeling-related aspects of all forms of data access
45 and transmission visible through the POSIX.1 interfaces. (Please note
46 the distinction made between data and control information, clarified later
47 in this section.) The interface, however, is designed for flexibility: the
48 standard defines the *minimum* functionality that must be provided.
49 Naturally, conforming implementations may choose to perform informa-
50 tion labeling on objects, or at times, not required by this standard.
- 51 (6) Preserve 100% compatibility with the base POSIX.1 functionality. That
52 is, it is undesirable to require new restrictions on the operation of exist-
53 ing POSIX.1 interfaces, or to require changes to the syntax of existing
54 POSIX interfaces.
- 55 (7) Add no new information labeling-specific error messages to existing
56 POSIX.1 interfaces and thus minimize the potential for confusing exist-
57 ing applications. While this potential for confusion cannot be entirely
58 eliminated (in particular because existing error codes can now be
59 returned in situations which would not arise without information label-
60 ing present), avoiding new error values at least ensures existing applica-
61 tions will be able to report errors.

62 **B.27.2 Scope**

63 This section examines the information labeling interfaces provided by this stan-
64 dard and explains the overall motivation for including the information labeling
65 interfaces. Rationale and design tradeoffs are presented for the key information
66 label interfaces.

67 This standard supports a security policy of nondisclosure, primarily through the
68 interfaces defined for discretionary and mandatory access control. In particular,
69 mandatory access control mechanisms implemented using the defined interfaces
70 are expected to conform with the overall intent established in the security stan-
71 dards to which they are targeted. These security standards, (e.g., the TCSEC),
72 normally require policies and mechanisms that protect objects at the level of the
73 most sensitive data that they can contain. Often, however, the *data* contained in
74 objects is actually much less sensitive than indicated by the mandatory access
75 control label associated with that object. In addition, many security policies
76 require that certain non-mandatory access control related information be associ-
77 ated with subjects and objects. Thus, in addition to mandatory access control
78 labeling, this standard provides optional interfaces for data labeling. Use of these

79 interfaces by conforming implementations permit support for a variety of data
80 labeling policies.

81 **B.27.3 Concepts Not Included**

82 Several concepts that will commonly be implemented by conforming systems have
83 not been treated by this document, many because they have no basis in the
84 POSIX standards upon which this document is currently based. These include:

85 Label Translation: POSIX.1 does not address networked systems. Thus, the
86 translation of information labels into an exportable form is
87 not addressed in this standard.

88 Process Label Functions: The functions provided as part of this standard to
89 retrieve or set the information label associated with a pro-
90 cess are limited to the requesting process. That is, no inter-
91 face is provided whereby a process may specify another pro-
92 cess (for example, using a process id) to be the target of the
93 *inf_get_proc()* or *inf_set_proc()* functions. Such mechanisms
94 have been omitted in order to be consistent with the
95 POSIX.1 standard which provides no facilities for processes
96 to manipulate, or be cognizant of, other processes' state
97 information. Note, however, that conforming implemen-
98 tations may choose to provide such functions.

99 **B.27.4 Data Labeling Policies**

100 There are many instances when security-related information should be associated
101 with subjects and objects even though that information may not, in general, be
102 used for mandatory access control. Such information may include markings that
103 indicate the source of some data, what the data is about, the "trustworthiness" of
104 the data, or anything else about the data other than how it should be protected.
105 This non-mandatory access control related information is represented in an infor-
106 mation label that should be associated with data when it is printed or otherwise
107 exported. This specification provides functions to assign initial information
108 labels, combine two information labels, and manipulate information labels.

109 A sample non-mandatory access control data labeling policy might be one targeted
110 at virus detection. For example, under this policy, programs downloaded from a
111 public bulletin board might be labeled with the marking "suspect-file." If the pro-
112 gram contained a virus, and if the *inf_float()* function (discussed below) imple-
113 mented the Compartmented Mode Workstation (also discussed below) style of
114 floating labels, then it would be easy to track the spread of any infection
115 throughout the system because every file infected by the virus would automati-
116 cally be stamped with the "suspect-file" marking.

117 Other examples of non-mandatory access control information that should be asso-
118 ciated with data include handling caveats, warning notices, discretionary access
119 control advisories, and release markings. The ability to implement standards-

120 based systems that support these and other non-mandatory access control mark-
121 ings is of great interest to many vendors and users.

122 One example of existing non-mandatory access control policies this interface is
123 intended to support are those proposed by the European trusted system vendor
124 community. The functionality necessary is that users must be allowed to apply
125 data labels to subjects and objects, and alter them on a discretionary basis, in
126 order to record handling restrictions on the objects' contents.

127 To provide a data labeling interface that can easily support the existing multiple
128 data labeling policies, the information label interfaces have been carefully gen-
129 eralized to provide a mechanism to support these policies, without attempting to
130 enforce the specifics of any particular policy. The burden of implementing specific
131 policies is left to conforming implementations.

132 **B.27.4.1 General Information Label Policy**

133 Section 27.1.2 of this standard defines a general information labeling policy capa-
134 ble of supporting multiple particular data labeling policies. The information label
135 policy statement consists of:

- 136 (1) A broad policy statement
- 137 (2) Refinements of this policy for the two major current policy areas: files
138 and processes.

139 It should be noted that the policies in this section do not constitute a *formal secu-*
140 *rity policy model* with proven assertions. It is, however, the most fundamental set
141 of information label policies that should be defined. The general information label
142 policy is as follows.

143 *Information Label Policy*: Each subject (process) and each object that con-
144 tains data (as opposed to control information) shall have as an attribute an
145 information label at all times.

146 Information labels are said to “float” as data from one object is introduced to
147 another object. The general information label floating policy is intentionally flexi-
148 ble and can be stated as follows:

149 *Information Label Floating Policy*: The implementation-defined policy that
150 determines to what degree information labels associated with data are
151 automatically adjusted as data flows through the system.

152 The information label float policy is embodied by the *inf_float()* function. This
153 function computes a new information label that is the combination of two informa-
154 tion labels passed as arguments. As noted above, the new information label is
155 calculated according to implementation-defined policies.

156 Note that the information label policy as applied to process functions specifies (in
157 **PI.2**) that when a process with an information label *inf_p1* executes a file with
158 information label *inf_p2*, the information label of the process shall be set to the
159 value returned by *inf_float(inf_p1, inf_p2)*. However, in implementations where

160 the new file executed completely overlays the process' address space, i.e., there is
161 no data transfer from the originally executing process to the newly executing pro-
162 cess, the information label of the process after executing the file may be set to
163 *inf_p2*. The central factor in determining whether such an implementation con-
164 forms to the information label policy is whether data is transferred: the transfer
165 of control information (such as process id, and various user ids) is inevitable and
166 permissible; the transfer of data is unacceptable.

167 **B.27.4.2 Error Return Values**

168 The information labeling functions specified in this standard may return one of
169 several errors depending on how the implementation has addressed information
170 labeling.

171 If the symbol `{_POSIX_INF}` is defined, then the implementation supports the +
172 information label option and is required to support the information label functions+
173 as described in this standard. If the symbol `{_POSIX_INF}` is not defined, then the+
174 implementation does not claim conformance to the information label option and +
175 the results of an application calling any of the information label functions are not +
176 specified within this standard. An alternative is for the information label func- +
177 tions to specify that the error return code [ENOSYS] be returned by the functions +
178 if the information label option is not supported. However, in order to remain com- +
179 pliant with the policies of POSIX.1, this standard cannot specify any require- +
180 ments for implementations that do not support the option.

181 The error [ENOTSUP] shall be returned in those cases where the system supports
182 the information label facility but the particular information label operation can-
183 not be applied because restrictions imposed by the implementation. For example,
184 if an application attempts to set the information label on a file on a system where
185 `sysconf()` indicates that an information label facility is supported by the system,
186 but the value that `pathconf()` returns for `{_POSIX_INF_PRESENT}` for that file
187 indicates that information labels are not supported on that file, the application
188 shall receive the [ENOTSUP] error. Therefore, if an application attempts to set
189 the information label on a file, it is the application's responsibility to first use
190 `pathconf()` to determine whether the implementation supports information labels
191 on that file.

192 It should be noted that, in general, this standard attempts to avoid adding and
193 defining new errors. However, in the case of [ENOTSUP], the following points
194 were noted: First, the need exists to provide feedback to applications concerning
195 a new error condition. Second, while it is possible to use an existing error code in
196 such cases (for example, ENOSYS), the group felt that this would overload those
197 errors. P1003.1, when consulted, concurred with this view and agreed that the
198 creation of a new error code, in this case, was appropriate. Third, the error
199 [ENOTSUP] is also being used by P1003.4 for roughly the same reasons. There-
200 fore, the consensus of several POSIX working groups is that while adding new
201 errors is generally not recommended, that this case warrants the creation of a
202 new error and that the new error should be [ENOTSUP].

203 The [EINVAL] error is returned by functions when the information label specified
204 in the function call is syntactically incorrect or the information label is not per-
205 mitted on the system because implementation-defined restrictions, (e.g., range
206 restrictions). That is, this error is used to indicate the invalidity of the informa-
207 tion label specified, independent of whether the operation would have succeeded
208 had it been a valid label.

209 Although POSIX.1 does not specify precedence for error return values, careful
210 consideration should be given to this matter in the security standard to ensure
211 that covert channel considerations are adequately addressed. While information
212 labeling is not usually subject to covert channels, in certain cases they may arise.
213 Specifically, if an application that does not possess appropriate privilege attempts
214 a function for which appropriate privilege is required and the implementation
215 returns the EINVAL error in favor of the EPERM error, it may be possible for the
216 application to determine the system's information label range restrictions based
217 on whether EINVAL is returned (indicating the label is outside the system's
218 range), or EPERM is returned (indicating the label is valid for the system, but
219 that the application did not possess appropriate privilege). Therefore, despite this
220 standard's silence on the issue, it is recommended that when a function could
221 return multiple errors in a particular instance, that the errors be given the follow-
222 ing precedence (from most favored to least favored): ENOSYS, EPERM, EINVAL,
223 ENOTSUP.

224 **B.27.4.3 Rationale for Pointer Arguments**

225 The functions provided to support information labeling use an opaque data type.
226 Nevertheless, in order to accommodate systems in which the size of an informa-
227 tion label may vary (e.g., depending on the actual label encoded or depending on
228 the total set of labels supported), the information label functions operate on
229 pointers. For this reason, the basic information label structure defined in this
230 standard (*inf_t*) is defined to be a pointer to an opaque data structure. In this
231 way, conforming applications need not determine the size of a label prior to
232 requesting an operation that will produce or modify that label. (In some cases,
233 such as *inf_float()*, this would be particularly difficult inasmuch as the resultant
234 information label is not known prior to making the request.) Instead, the system
235 functions themselves are responsible for allocating the space necessary to contain
236 a new label, and a function is provided to applications to free that space when the
237 label is no longer needed.

238 The tradeoffs between the approach adopted by the information label functions
239 specified in this standard and alternative approaches are many and varied. The
240 structure of the information label function interfaces have been designed to be
241 consistent with those provided by the interfaces supplied in support of the other
242 features included in this standard, and the mandatory access control interfaces in
243 particular. Thus, a more detailed and complete rationale for the adoption of these
244 types of interfaces can be found in the mandatory access control rationale.

245 **B.27.4.4 Rationale for POSIX.1e Functions**

246 The *inf_float()* function is not specified in detail to allow for a range of
247 implementation-defined floating policies. The range of policies would determine
248 the degree to which information labels associated with data are automatically
249 adjusted as data flows through the system. Two explicit floating policies that
250 have been articulated are intended to be supportable in POSIX through the
251 definition of *inf_float()*.

252 The first policy is that articulated as part of the Compartmented Mode Worksta-
253 tion project (see IEEE Transactions on Software Engineering, Vol. 16, No. 6, June
254 1990, pp 608-618). Under this policy, every data read or write is intended to
255 (potentially) modify the information label of the object being modified through the
256 read or write. In the case of a subject reading an object, the subject's information
257 label would be modified ("floated") to a combination of the information label of the
258 subject before the read, and the information label associated with the object.
259 When a subject writes to an object, the object's information label would be floated
260 to represent the combination of the information label of the object before the
261 write, and the information label associated with the subject. This policy makes a
262 great deal of sense in the case where there are a large number of different infor-
263 mation label values, and it is desired to track the flow of data through the system
264 by having the data's information label follow the data. To accommodate this pol-
265 icy, *inf_float()* would always combine its two arguments and return the result.
266 The details of the combination would depend on the semantics of the particular
267 information labels involved.

268 The second policy makes more sense when there are a relatively small, more
269 static number of information label values. In this policy, the intention is that
270 objects, when created, inherit their creator's information label, but that the infor-
271 mation label does not automatically change thereafter. To accommodate this pol-
272 icy, *inf_float()* would be defined such that it floated an information label only one
273 time. In other words, *inf_float()* would return a result other than its second argu-
274 ment only when its second argument is equal to *inf_default()* and its first argu-
275 ment is not *inf_default()*.

276 **B.27.4.5 System Floating**

277 Because the *inf_float()* routine takes two labels and returns the result of a float
278 operation, it is not an entirely general function. That is, it cannot base the result
279 of the float operation on any factor other than the two input labels. However, it is
280 possible to imagine other data labeling policies that require different floating
281 rules based on any number of factors (e.g., files involved, or time of day). Support
282 for these peculiar types of policies is not explicitly required in this standard. The
283 main reason for this exclusion is that, of the multiple data labeling polices
284 intended to be supported by this standard, none require such extensions to the
285 *inf_float()* function. Indeed, to the group's knowledge, no known data labeling
286 policy currently used in commercially available systems that would require such
287 extensions presently exists.

288 The second major reason for the lack of true generality in the floating function
289 was due to technical obstacles. To make the *inf_float()* function more general,
290 additional arguments would be required. The addition of more information used
291 to characterize the two labels involved in floating was discussed. Particular con-
292 sideration was given to adding type information so that the type of the object with
293 which the information label is associated could be determined. This was to allow
294 the implementation-defined algorithm to act differently based on the types of the
295 objects involved. This addition was rejected because the working group could see
296 no use for it in an external (application level) interface for conforming applica-
297 tions. The group also considered including arguments to identify the specific
298 object being floated. Again, due to lack of motivation, and an inability to devise a
299 useful interface that could be used to identify all POSIX objects that could sup-
300 port ILs, and still be extensible to non-POSIX objects (in a curt acknowledgement
301 of the needs of the real world), this option, too, was dropped.

302 Note that the *inf_float()* function nevertheless remains a valuable and necessary
303 interface: it allows conforming applications to call a routine which the system
304 provides that is guaranteed to provide a label float operation consistent with the
305 system's data labeling policies. Using the function, trusted applications can per-
306 form fine-grained labeling of their own resources.

307 **B.27.4.6 Object Labeling**

308 The objects to which this standard requires information labels be applied include
309 the expected POSIX.1 objects: files. Not included among the objects are
310 processes. As observed in the mandatory access control section, processes may act
311 as objects under certain conditions. For example, when one process sends a signal
312 to another, the former is effectively writing to the latter, and therefore the latter
313 could be considered an object, from the perspective of this function. However,
314 because many data labeling policies consider signals of this type to be a transmis-
315 sion of control information, and therefore not necessarily subject to the informa-
316 tion label policies, many data labeling policies do not consider the process to be an
317 object (from the information label perspective) with respect to these functions.
318 Because POSIX.1 does not provide any other functions in which processes act as
319 objects, the information labeling standard does not include processes as objects.

320 Note that information labels are not required to be applied to directories. Argu-
321 ments for why they should be are as follows. Directories, like any other type of
322 file, contain arbitrary length strings of process-specified data. This data is, by
323 intent, designed to be communicative to users; that is, it is meaningful informa-
324 tion (from the human perspective). Since this is the type of information data
325 labeling policies are intended to label, it would make sense to require that direc-
326 tories be subject to the information label policies.

327 Alternatively, opposing opinions have been expressed that information labels
328 should not be required to be applied to directories. These arguments are as fol-
329 lows. Directories are not containers of data, but rather are organizers of data con-
330 tainers (such as regular files). As such, the notion that information labels are
331 applied to "data" as opposed to "control information" suggests that information
332 labels may not necessarily be needed on directories. In addition, as with

333 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.

340 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.

343 **B.27.5 Initial Information Labels**

344 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".)

360 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.

373 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.

377 The addition of more information used to characterize the object to receive an ini-
378 tial information label was discussed. Particular consideration was given to
379 adding type information so that the type of the object with which the initial infor-
380 mation label is to be associated could be determined. This was to allow the
381 implementation-defined algorithm to act differently based on the type of object to
382 be labeled. This addition was rejected because the working group could see no
383 use for it in an external (user level) interface for conforming applications. Inter-
384 nal (system-specific) initial information labels are not required to use
385 *inf_default()* and therefore can be different based on the object being labeled.

386 **B.27.6 Information Label Validity**

387 Information labels have two forms: internal and external.

388 The basic information label structure defined in this standard (*inf_t*) is a pointer
389 to an opaque data structure. The binary format of that opaque data structure
390 may include such data as a hierarchical classification, non-hierarchical categories,
391 or non-access control related markings. The standard makes no assumptions
392 regarding the underlying representation or contents of the structure other than
393 imposing the following constraint: the structure must be an exportable object.
394 That is, the structure is opaque, persistent, and self-contained. The structure can
395 therefore be copied by duplicating the bytes without knowledge of its syntax.
396 Such a copy can be changed without any effect on the original, and the original
397 can be changed without any effect on the copy.

398 The external format of a label is a text string of unspecified format. Any separa-
399 tor characters appearing between the components of an information label are
400 implementation-defined. Note that this standard does not specify the set of legal
401 characters that may be used in the text representation of an information label.
402 Further rationale for this decision can be found in POSIX.1, section B.2.3.5.

403 The meaning of a valid information label is implementation-defined, as described
404 in *inf_valid()*. An information label could be invalid for a variety of reasons.
405 Some reasons why a label may be invalid on some systems include:

406 It is malformed (e.g., the label contains a checksum in the opaque type
407 that does not agree with the checksum calculated from the data).

408 It is out of the cleared range of the system (e.g., the label refers to a
409 classification that is outside the set of valid classifications for the system).

410 It is outside the representation range (e.g., a system could allow no more
411 than n categories from a universe of m, even though each of the m
412 categories is valid).

413 If *{_POSIX_MAC}* is defined, and the mandatory access control label of a
414 process does not dominate the mandatory access control label associated
415 with all components of an information label, then that information label
416 may be invalid for the process, even though it is valid for other processes

417 executing on the same system.

418 Invalid information labels may appear for a great number of reasons. Examples
419 include: constructing an information label in process memory without regard to
420 semantics of the bits, importing an information label from a dissimilar system,
421 etc. Note, however, that combining two information labels (e.g., using *inf_float()*),
422 will calculate an information label that is valid. This is because information
423 labeling, as noted elsewhere in this section, is used for data labeling, not access
424 control. Therefore, if the other security policies implemented in a conforming sys-
425 tem permit data to be combined, the information labeling mechanism is obligated
426 to calculate an accurate and valid information label for the combined data.

427 **B.27.7 Control Information**

428 The policy discussion contained in section 27.1.2 specifically notes that the infor-
429 mation label of a file applies only to the data portion of the file. That is, manipu-
430 lation of control information need not result in an information label float opera-
431 tion. This “special” treatment for control information results from a tradeoff
432 between functionality and security. If information labels floated when control
433 information was manipulated (e.g., at file open time, instead of at data transfer
434 time), the information labels associated with subjects and objects would have a
435 tendency to float too often and would lose some of their utility as a mechanism to
436 track the flow of data throughout a system. It can be argued that floating when
437 control information is manipulated would result in more “trustworthy” informa-
438 tion labels, however, several groups have expressed interest in favoring func-
439 tionality over security in this case. It is understood that a conforming implemen-
440 tation may cause the float operation to occur at times in addition to those covered
441 by the specified information labeling policy; such implementations may choose
442 enhanced trustworthiness over security.

443 **B.27.8 Relationship between ILs and Mandatory Access Control Labels**

444 In some systems, such as compartmented mode workstations, there exist certain
445 invariants that hold between ILs and mandatory access control labels. In the
446 case of CMWs, this invariant states that for any specific subject’s or object’s
447 labels, the access related portion of the information label (e.g., the classification
448 and categories) must be dominated by the mandatory access control label. While
449 this notion is useful for CMWs, it is not generally applicable to all systems that
450 might support the information label interfaces specified in this document. Most
451 notably, some companies that support the fundamental concept of information
452 labels, employ them in a manner such that mandating a relationship between
453 mandatory access control labels and ILs has no meaning. Indeed, there is no
454 requirement in this standard that the mandatory access control option be sup-
455 ported in order to support the IL section.

456 Note that conforming implementations are always at liberty to enforce additional
457 constraints. Thus a conforming implementation may certainly enforce a relation-
458 ship between mandatory access control labels and ILs (such as dominance). The

459 silence of this standard on the topic of specific relationships between mandatory
460 access control labels and ILs should not dramatically impact portable applica-
461 tions.

462 **B.27.9 Additional Uses of Information Labeling**

463 The Compartmented Mode Workstation (CMW) security requirements are well
464 known in many parts of the computer security community and have attracted con-
465 siderable vendor interest. The CMW requirements are documented formally in
466 “Security Requirements for System High and Compartmented Mode Worksta-
467 tions”, Defense Intelligence Agency document DDS-2600-5502-87 and are dis-
468 cussed less formally in the June 1990 issue of IEEE Transactions on Software
469 Engineering. Information labeling is a key component of the CMW requirements
470 both for meeting certain data labeling policies that concern non-mandatory access
471 control related information, and to avoid a potential data overclassification prob-
472 lem that may result from use of mandatory access control label-only systems.
473 This section of the rationale will further examine the data overclassification prob-
474 lem as an additional example of the utility of information labels.

475 According to mandatory access control policy **FP.4**, a newly created file object
476 shall be assigned the mandatory access control label of the creating subject (pro-
477 cess). Such a policy is necessary to prevent any subjects with mandatory access
478 control labels dominated by the creator’s label from discovering the “fact of
479 existence” of the object, thereby closing a covert channel.

480 Although the mandatory access control label of a newly created object correctly
481 represents the sensitivity of the object from the standpoint of mandatory access
482 control, it most likely incorrectly represents the actual sensitivity of the data con-
483 tained in the object. Since the newly-created object contains no data, the sensi-
484 tivity of the (null) data itself should be considered some system low value.

485 Another example of the overclassification problem is as follows. Consider a shell
486 process (subject) executing with a mandatory access control label of *mac_p2*. Dur-
487 ing the lifetime of this shell the user decides to make a copy of another user’s file
488 containing data with a sensitivity of *mac_p1* and therefore a mandatory access
489 control label of *mac_p1*. *mac_p2* dominates *mac_p1*, so the copy operation would
490 be permitted by mandatory access control policy **FP.1**. The copy process will be
491 created with a mandatory access control label of *mac_p2* (in accordance with man-
492 datory access control policy **PP.2**), will read the data from the original file and
493 store a copy of the data in a newly created file. In accordance with **FP.4**, the
494 newly created file will have a mandatory access control label of *mac_p2*, even
495 though the original data was only sensitive enough to require protection at the
496 *mac_p1* level.

497 These overclassification problems can be mitigated with the use of information
498 labels. In particular, an implementation could define *inf_default()* to return an
499 information label of “system low” and *inf_float()* to combine information labels as
500 per the CMW requirements. In such a system the information label of a newly
501 created (empty) object would be system low—an accurate representation of the
502 actual sensitivity of the (null) data contained within the object. Note that this

503 newly created object (and the fact that this object existed) would still be correctly
504 protected by the object's mandatory access control label. When a process reads
505 from a file, the process information label floats with the file information label.
506 When a process writes to a file, the file information label floats with the process
507 information label.

508 Returning to the copy example, say the information label of the source file is
509 *inf_p1*. The copy process will start with an information label of *inf_p2*, which we
510 assume is system low as defined by *inf_default()* (as will generally be the case).
511 In the model of information label floating described in the paragraph above, when
512 the copy process reads the data from the file to be copied, the copy process' infor-
513 mation label will float to the value returned by *inf_float(inf_p1, inf_p2)*, which,
514 because *inf_p2* is system low, will equal *inf_p1*. When the copy process creates
515 and writes the target file, that file will float to *inf_p1* (the copy process' label).
516 Thus the information label of the data in the source file will follow the data as it
517 moves through the system. So, even though the target file has a mandatory
518 access control label that is higher than the mandatory access control label of the
519 source file, the target file's information label is the same as the source file's infor-
520 mation label and remains an accurate representation of the actual sensitivity of
521 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 emailed 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:

WITHDRAWN DRAFT. All Rights Reserved by IEEE.
Preliminary—Subject to Revision.

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

WITHDRAWN DRAFT. All Rights Reserved by IEEE.
Preliminary—Subject to Revision.

**WITHDRAWN DRAFT. All Rights Reserved by IEEE.
Preliminary—Subject to Revision.**

**WITHDRAWN DRAFT. All Rights Reserved by IEEE.
Preliminary—Subject to Revision.**

**WITHDRAWN DRAFT. All Rights Reserved by IEEE.
Preliminary—Subject to Revision.**

Topical Index

WITHDRAWN DRAFT. All Rights Reserved by IEEE.
Preliminary—Subject to Revision.

Contents

SECTION	PAGE
Section 1: Revisions to the General Section	1
Section 2: Revisions to Terminology and General Requirements	3
Section 3: Revisions to Process Primitives	17
Section 4: Revisions to Process Environment	21
Section 5: Revisions to Files and Directories	23
Section 6: Revisions to Input and Output Primitives	35
Section 8: Revisions to C Programming Language Specific Services	37
Section 23: Access Control Lists	39
23.1 General Overview	39
23.1.1 ACL Entry Composition	40
23.1.2 Relationship with File Permission Bits	41
23.1.3 Default ACLs	42
23.1.4 Associating an ACL with an Object at Object Creation Time	42
23.1.5 ACL Access Check Algorithm	43
23.1.6 ACL Functions	44
23.1.7 POSIX.1 Functions Covered by ACLs	46
23.2 Header	47
23.2.1 acl_entry_t	47
23.2.2 acl_perm_t	48
23.2.3 acl_permset_t	48
23.2.4 acl_t	48
23.2.5 acl_tag_t	48
23.2.6 acl_type_t	49
23.2.7 ACL Qualifier	49
23.2.8 ACL Entry	50
23.3 Text Form Representation	50
23.3.1 Long Text Form for ACLs	50
23.3.2 Short Text Form for ACLs	52
23.4 Functions	52
23.4.1 Add a Permission to an ACL Permission Set	53
23.4.2 Calculate the File Group Class Mask	53
23.4.3 Clear All Permissions from an ACL Permission Set	55
23.4.4 Copy an ACL Entry	55
23.4.5 Copy an ACL From System to User Space	56

SECTION	PAGE
23.4.6 Copy an ACL From User to System Space	57
23.4.7 Create a New ACL Entry	58
23.4.8 Delete a Default ACL by Filename	59
23.4.9 Delete an ACL Entry	61
23.4.10 Delete Permissions from an ACL Permission Set	61
23.4.11 Duplicate an ACL	62
23.4.12 Release Memory Allocated to an ACL Data Object	63
23.4.13 Create an ACL from Text	64
23.4.14 Get an ACL Entry	65
23.4.15 Get an ACL by File Descriptor	66
23.4.16 Get an ACL by Filename	67
23.4.17 Retrieve the Permission Set from an ACL Entry	69
23.4.18 Get ACL Entry Qualifier	70
23.4.19 Get ACL Entry Tag Type	71
23.4.20 Initialize ACL Working Storage	72
23.4.21 Set an ACL by File Descriptor	73
23.4.22 Set an ACL by Filename	74
23.4.23 Set the Permissions in an ACL Entry	76
23.4.24 Set ACL Entry Tag Qualifier	77
23.4.25 Set ACL Entry Tag Type	78
23.4.26 Get the Size of an ACL	79
23.4.27 Convert an ACL to Text	80
23.4.28 Validate an ACL	81
Section 24: Audit	83
24.1 General Overview	83
24.1.1 Audit Logs	83
24.1.2 Audit Records	84
24.1.3 Audit Interfaces	85
24.1.4 Summary of POSIX.1 System Interface Impact	89
24.2 Audit Record Content	89
24.2.1 Auditable Interfaces and Event Types	90
24.2.2 Audit Event Types and Record Content	92
24.3 Header	106
24.3.1 aud_evinfo_t	108
24.3.2 aud_hdr_t	108
24.3.3 aud_id_t	108
24.3.4 aud_info_t	108
24.3.5 aud_obj_t	109
24.3.6 aud_obj_type_t	110
24.3.7 aud_rec_t	110
24.3.8 aud_state_t	110
24.3.9 aud_status_t	110
24.3.10 aud_subj_t	111

SECTION	PAGE
24.3.11 aud_time_t	111
24.4 Functions	112
24.4.1 Copy an Audit Record From System to User Space	112
24.4.2 Copy an Audit Record From User to System Space	113
24.4.3 Delete Set of Event-specific Data from a Record	114
24.4.4 Delete Item from Set of Event-specific Data	115
24.4.5 Delete Header from an Audit Record	116
24.4.6 Delete Item from Audit Record Header	117
24.4.7 Delete Set of Object Attributes from a Record	118
24.4.8 Delete Item from Set of Object Attributes	118
24.4.9 Delete Set of Subject Attributes from a Record	119
24.4.10 Delete Item from Set of Subject Attributes	120
24.4.11 Duplicate an Audit Record	121
24.4.12 Map Text to Event Type	122
24.4.13 Map Event Type to Text	123
24.4.14 Release Memory Allocated to an Audit Data Object	124
24.4.15 Get All Audit Event Types	125
24.4.16 Get Audit Record Event-specific Data Descriptor	126
24.4.17 Examine Audit Record Event-specific Data	127
24.4.18 Get an Audit Record Header Descriptor	129
24.4.19 Examine an Audit Record Header	130
24.4.20 Get a Process Audit ID	132
24.4.21 Get an Audit Record Object Descriptor	133
24.4.22 Examine Audit Record Object Data	134
24.4.23 Get an Audit Record Subject Descriptor	137
24.4.24 Examine Audit Record Subject Data	138
24.4.25 Map Text to Audit ID	141
24.4.26 Map Audit ID to Text	141
24.4.27 Create a New Audit Record	142
24.4.28 Add Set of Event-specific Data to Audit Record	143
24.4.29 Add Item to Set of Event-specific Data	144
24.4.30 Add Header to Audit Record	146
24.4.31 Add Item to Audit Record Header	147
24.4.32 Add Set of Object Attributes to Audit Record	149
24.4.33 Add Item to Set of Object Attributes	150
24.4.34 Add Set of Subject Attributes to Audit Record	151
24.4.35 Add Item to Set of Subject Attributes	153
24.4.36 Read an Audit Record	154
24.4.37 Convert an Audit Record to Text	156
24.4.38 Get the Size of an Audit Record	157

SECTION	PAGE
24.4.39 Control the Generation of Audit Records	158
24.4.40 Validate an Audit Record	159
24.4.41 Write an Audit Record	160
Section 25: Capabilities	163
25.1 General Overview	163
25.1.1 Major Features	164
25.1.2 Capability Functions	167
25.2 Header	169
25.3 Text Form Representation	175
25.3.1 Grammar	176
25.4 Functions	177
25.4.1 Initialize a Capability State in Working Storage	178
25.4.2 Copy a Capability State From System to User Space	178
25.4.3 Copy a Capability State From User to System Space	179
25.4.4 Duplicate a Capability State in Working Storage	180
25.4.5 Release Memory Allocated to a Capability State in Working Storage	181
25.4.6 Convert Text to a Capability State in Working Storage	182
25.4.7 Get the Capability State of an Open File	183
25.4.8 Get the Capability State of a File	184
25.4.9 Get the Value of a Capability Flag	185
25.4.10 Obtain the Current Process Capability State	186
25.4.11 Allocate and Initialize a Capability State in Working Storage	187
25.4.12 Set the Capability State of an Open File	188
25.4.13 Set the Capability State of a File	189
25.4.14 Set the Value of a Capability Flag	190
25.4.15 Set the Process Capability State	191
25.4.16 Get the Size of a Capability Data Record	192
25.4.17 Convert a Capability State in Working Storage to Text	193
Section 26: Mandatory Access Control	195
26.1 General Overview	195
26.1.1 MAC Concepts	195
26.1.2 MAC Policy	196
26.2 Header	200
26.2.1 mac_t	201
26.3 Functions	201
26.3.1 Test MAC Labels for Dominance	201
26.3.2 Test MAC Labels for Equivalence	202
26.3.3 Free MAC Label Storage Space	203

SECTION		PAGE
26.3.4	Convert Text MAC Label to Internal Representation	203
26.3.5	Get the Label of a File Designated by a File Descriptor	204
26.3.6	Get the Label of a File Designated by a Pathname	205
26.3.7	Get the Process Label	207
26.3.8	Compute the Greatest Lower Bound	207
26.3.9	Compute the Least Upper Bound	208
26.3.10	Set the Label of a File Identified by File Descriptor	209
26.3.11	Set the Label of a File Designated by Pathname	211
26.3.12	Set the Process Label	212
26.3.13	Get the Size of a MAC Label	213
26.3.14	Convert Internal MAC Label to Textual Representation	214
26.3.15	Label Validity	215
Section 27: Information Labeling		217
27.1	General Overview	217
27.1.1	Information Label Concepts	217
27.1.2	Information Label Policy	218
27.2	Header	221
27.2.1	inf_t	221
27.3	Functions	221
27.3.1	Initial Information Label	222
27.3.2	Test Information Labels For Dominance	223
27.3.3	Test Information Labels For Equivalence	223
27.3.4	Floating Information Labels	224
27.3.5	Free Allocated Information Label Memory	225
27.3.6	Convert Text Label to Internal Representation	226
27.3.7	Get the Information Label of a File Identified by File Descriptor	227
27.3.8	Get the Information Label of a File Identified by Pathname	228
27.3.9	Get the Process Information Label	229
27.3.10	Set the Information Label of a File Identified by File Descriptor	230
27.3.11	Set the Information Label of a File Identified by Pathname	231
27.3.12	Set the Process Information Label	232
27.3.13	Get the Size of an Information Label	233
27.3.14	Convert Internal Label Representation to Text	234
27.3.15	Information Label Validity	235

SECTION	PAGE
Annex B (informative) Revisions to Rationale and Notes	237
B.1 Revisions to Scope and Normative References	237
B.2 Revisions to Definitions and General Requirements	242
B.2.10 Security Interface	243
B.3 Revisions to Process Primitives	243
B.23 Access Control Lists	249
B.23.1 General Overview	251
B.23.2 ACL Entry Composition	252
B.23.3 Relationship with File Permission Bits	255
B.23.4 Default ACLs	264
B.23.5 Associating an ACL with an Object at Object Creation Time	268
B.23.6 ACL Access Check Algorithm	271
B.23.7 ACL Functions	273
B.23.8 Header	279
B.23.9 Misc Rationale	279
B.24 Audit	280
B.24.1 Goals	280
B.24.2 Scope	285
B.24.3 General Overview	286
B.24.4 Audit Logs and Records	288
B.24.5 Audit Event Types and Event Classes	296
B.24.6 Selection Criteria	296
B.24.7 Audit Interfaces	297
B.25 Capability	306
B.25.1 General Overview	306
B.25.2 Major Features	312
B.25.3 Function Calls Modified for Capability	316
B.25.4 Capability Header	317
B.25.5 New Capability Functions	322
B.25.6 Examples of Capability Inheritance and Assignment	325
B.25.7 Capability Worked Examples	326
B.26 Mandatory Access Control	331
B.26.1 Goals	331
B.26.2 Scope	332
B.26.3 File Object Model	334
B.26.4 Direct Write-up	335
B.26.5 Protection of Link Names	336
B.26.6 Pathname Search Access	336
B.26.7 Check-Access-on-Open Only	337
B.26.8 Creating Upgraded Directories	338
B.26.9 Objects without MAC labels	338
B.26.10 Error Return Values	339
B.26.11 Valid MAC Labels	340
B.26.12 Modification of MAC labels	341

SECTION	PAGE
B.26.13 Least upper bounds and greatest lower bounds	341
B.26.14 Functions returning MAC labels	341
B.26.15 Multi-level directories	342
B.26.16 The Directory Model	344
B.26.17 File Tranquillity	344
B.26.18 Process Tranquillity	345
B.26.19 Unnamed Pipes	345
B.26.20 FIFOs	346
B.26.21 Inclusion of <i>mac_set_fd()</i>	346
B.26.22 Inclusion of <i>mac_size()</i>	346
B.26.23 Restrictions on Signals	347
B.26.24 Alteration of atime	347
B.26.25 Multi-Label Untrusted Process Hierarchies	347
B.26.26 File Status Queries	348
B.27 Information Labeling	348
B.27.1 Goals	348
B.27.2 Scope	349
B.27.3 Concepts Not Included	350
B.27.4 Data Labeling Policies	350
B.27.5 Initial Information Labels	356
B.27.6 Information Label Validity	357
B.27.7 Control Information	358
B.27.8 Relationship between ILs and Mandatory Access Control Labels	358
B.27.9 Additional Uses of Information Labeling	359
Annex F (informative) Ballot Instructions	361
Identifier Index	373
Topical Index	378

TABLES

Table 23-1 – ACL Data Types	47
Table 23-2 – acl_perm_t Values	48
Table 23-3 – acl_tag_t Values	49
Table 23-4 – acl_type_t Values	49
Table 23-5 – ACL Qualifier Constants	49
Table 23-6 – ACL Entry Constants	50
Table 24-1 – Interfaces and Corresponding Audit Events	91
Table 24-2 – Audit Data Types	107

Table 24-3 – Other Constants	107
Table 24-4 – <i>aud_info_t</i> members	108
Table 24-5 – Values for <i>aud_info_type</i> Member	109
Table 24-6 – <i>aud_obj_type_t</i> Values	110
Table 24-7 – <i>aud_state_t</i> Values	110
Table 24-8 – <i>aud_status_t</i> Values	111
Table 24-9 – <i>aud_time_t</i> Members	111
Table 24-10 – <i>aud_hdr_info_p</i> Values	131
Table 24-11 – <i>aud_obj_info_p</i> Values	135
Table 24-12 – <i>aud_subj_info_p</i> Values	139
Table 25-1 – POSIX.1 Functions Covered by Capability Policies	166
Table 25-2 – Capability Data Types	169
Table 25-3 – <i>cap_flag_t</i> Values	169
Table 25-4 – <i>cap_flag_value_t</i> Values	169
Table 25-5 – <i>cap_value_t</i> Values	170
Table 25-6 – <i>cap_value_t</i> Values for Mandatory Access Controls	173
Table 25-7 – <i>cap_value_t</i> Values for Information Labels	174
Table 25-8 – <i>cap_value_t</i> Values for Audit	174
Table 26-1 – POSIX.1 Functions Covered by MAC File Policies	199
Table 26-2 – POSIX.1 Functions Covered by MAC Process Policies	200
Table 27-1 – POSIX.1 Functions Covered by Information Label File Policies	220
Table 27-2 – POSIX.1 Functions Covered by Information Label Process Policies	221
Table B-3 – Other System Functions Potentially Affected by Capability Policies	317