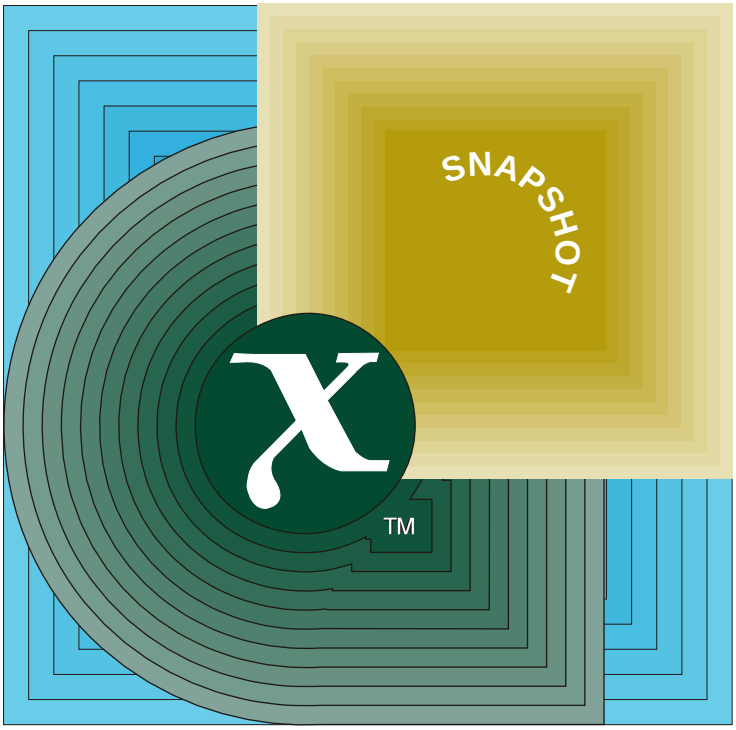
Snapshot

# Security Interface Specifications: Auditing and Authentication




THE Open GROUP

[This page intentionally left blank]

*X/Open Snapshot*

# Security Interface Specifications: Auditing and Authentication

*X/Open Company, Ltd.*



X/Open Snapshot

Security Interface Specifications: Auditing and Authentication

X/Open Document Number: XO/SNAP/90/020

Set in Palatino by X/Open Company Ltd., U.K.
Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to the X/Open Company at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.co.uk

# Contents

## SECURITY INTERFACE SPECIFICATIONS: AUDITING AND AUTHENTICATION

# Preface

## X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring greater value to users through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable system environment, called the *Common Applications Environment (CAE)*. This environment covers all the standards, above the hardware level, that are needed to support open systems. It ensures portability and connectivity of applications, and allows users to move between systems without retraining.

The interfaces identified as components of the Common Applications Environment are defined in the *X/Open Portability Guide*. This guide contains an evolving portfolio of practical applications programming interface standards (APIs), which significantly enhance portability of application programs at the source code level. The interfaces defined in the X/Open Portability Guide are supported by an extensive set of conformance tests and a distinct trademark - the X/Open brand - that is carried only on products that comply with the X/Open definitions.

X/Open is thus primarily concerned with standards selection and adoption. The policy is to use formal approved *de jure* standards, where they exist, and to adopt widely supported *de facto* standards in other cases.

Where formal standards do not exist, it is X/Open policy to work closely with standards development organizations to encourage the creation of formal standards covering the needed functionalities, and to make its own work freely available to such organizations. Additionally, X/Open has a commitment to align its definitions with formal approved standards.

## The X/Open Product Family - XPG

There is a single family of X/Open products, which has the generic name ''XPG''.

### *XPG Versions*

There are different numbered versions of XPG within the XPG family (XPG1, XPG2, XPG3). Each XPG version is an integrated set of elements supporting the development, procurement and implementation of open systems products, and each comprises its own:

- XPG Specifications
- XPG Verification Suite
- XPG descriptive guides

- XPG trademark licensing materials

The XPG trademark (or ‘‘brand’’) licensed by X/Open always contains a particular XPG version number (e.g., ‘‘XPG3’’) and, when associated with a vendor’s system, communicates clearly and unambiguously to a procurer that the software bearing the trademark correctly implements the corresponding XPG specifications. Users specifying particular XPG versions in their procurements are therefore certain as to the XPG specifications to which vendors’ systems conform.

### *XPG Specifications*

There are four types of XPG specification:

- **XPG*n* Formal Specifications**

  These are the long-life XPG specifications that form the basis for conformant/branded X/Open systems, and are the only type of XPG specification released with an XPG version number (e.g., ‘‘XPG3’’). They are intended to be used widely within the industry for product development and procurement purposes. Currently, all XPG Formal Specifications are included in Issue 3 of the X/Open Portability Guide.

  Individual XPG specifications are released as Formal Specifications only as part of the formal release of the complete XPG version to which they belong. However, prior to the launch of that XPG version, they may be made available as:

- **XPG Developers’ Specifications**

  These are specifically designed to allow developers to create X/Open-compliant products and applications in advance of the formal launch of a future version of the XPG.

  Developers’ Specifications may be relied on by product developers as the final, base specification that will appear in a future XPG. They are made available beforehand in order to meet the need of product developers for advance notification of the contents of XPG Formal Specifications, to assist in their product planning and development activities.

  By providing such advance notification, X/Open makes it possible for products conforming to future XPG Formal Specifications to be developed as soon as practicable, enhancing the value of XPG itself as a procurement aid to users.

- **XPG Preliminary Specifications**

  These are XPG specifications, usually addressing an emerging area of technology, and consequently not yet supported by a base of conformant product implementations, that are released in a controlled manner for validation purposes. A Preliminary Specification is not a ‘‘draft’’ specification. Indeed, it is as stable as X/Open can make it, and on publication will have gone through the same rigorous X/Open development and review procedures as XPG Formal and Developers’ Specifications.

  Preliminary Specifications are analogous with the ‘‘trial-use’’ standards issued by formal standards organizations, and product development teams are intended to develop product on the basis of them. Because of the nature of the technology they are addressing, they are untried in practice, and they may therefore change before being published as an XPG Formal or Developers’ Specification.

- **Snapshot Specifications**

  These are ‘‘draft’’ documents, that provide a mechanism for X/Open to disseminate information on its current direction and thinking to a limited audience, in advance of formal publication, with a view to soliciting feedback and comment.

  A snapshot represents the interim results of an X/Open technical activity. While X/Open currently intends to progress this activity towards publication of an X/Open Guide, X/Open is a consensus organisation, and makes no commitment regarding publication.

  Similarly, a snapshot does not represent any commitment on behalf of any X/Open member to make any specific products available now or in the future.

## This Document

This document is a Snapshot specification (see above). It describes extensions to the base X/Open operating system definition that satisfy the requirement for additional accountability and authentication features. The extensions have been designed to be compatible with existing guidelines for secure systems set out by external bodies, such as the DoD Trusted Computer System Evaluation Criteria (TCSEC). X/Open work on security has also been coordinated with that of the IEEE POSIX 1003.6 working group, which is considering auditing on secure IEEE standard 1003.1-1988 conforming systems. The interface definition will be maintained to remain compatible with auditing interfaces defined by that group.

**Disclaimer:**

This document represents the interim results of an X/Open technical activity. While X/Open currently intends to progress this activity towards publication of an X/Open Guide, X/Open is a consensus organisation, and makes no commitment regarding publication.

Similarly, this document does not represent any commitment on behalf of any X/Open member to make any specific products available now or in the future.

**Do not specify or claim conformance to this document.**

# *Trademarks*

X/Open and the 'X' device are trademarks of X/Open Company Limited in the U.K. and other countries.

UNIX is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and other countries.

# Referenced Documents

The following documents are referenced in this guide:

- DoD Document 5200.28-STD, Trusted Computer System Evaluation Criteria (TCSEC).

*Chapter 1*

# *Introduction*

In addition to general purpose systems already defined in the Common Applications Environment (CAE), X/Open members also need to supply systems for use in commercial and civil government roles that require additional accountability and authentication. Accountability applies to users and user processes and implies the need for additional methods of selectively auditing various aspects of process behaviour. Authentication extends the security features already present on most X/Open systems.

The fundamental aim of the X/Open Security Interface is to identify extensions to the base operating system definition that satisfy the above requirements. Concomitant with this objective is the need to provide these extensions in a way that is compatible with existing guidelines for secure systems set out by external bodies such as those defined in the DoD Trusted Computer System Evaluation Criteria (TCSEC) (see **Referenced Documents**).

The IEEE P1003.6 working group is also considering auditing on secure IEEE Standard 1003.1-1988 conforming systems. The work of X/Open on security has been coordinated with the P1003.6 working group and the interface definition will be maintained to remain compatible with auditing interfaces defined by that group.

Further requirements include the definition of a common audit record format, such that tools to process audit trails can be written to be portable between X/Open systems. This only applies to the external audit record format as presented to applications by a procedural interface to the underlying audit data. The internal format of audit data is implementation-defined.

A number of extensions are also required to existing system interfaces and headers. These have been minimised to reduce the risk of destabilising existing base products. Thus, changes to standard interfaces are marked as optional extensions to the base definition.

This issue of the XSI Security interfaces concentrates on the following major areas:

- Security Auditing
- Audit Trails
- Audit Event Classes and Audit Event Types
- Authentication Extensions
- Minimal XSI Changes

**Chapter 2**, **Overview** describes the major requirements in each of these key areas. **Chapter 3**, **Function and Interface** provides a narrative description of the functions provided to meet these requirements. The remaining chapters contain the formal interface definition itself.

Chapter 2

# Overview

## 2.1 GENERAL

The features required of XSI Security include:

- Systems must provide identification and authentication, giving a unique identifier to each user, so that the person who causes any security related event can be identified.
- Systems must be able to selectively audit the actions of any one or more users based on individual audit identity. The resulting audit must be available to an audit administrator.
- The audit trail, in which is recorded the audit of user actions, must be protected from modification, unauthorised access and destruction.

The types of event that must be audited are also identified. Some minimal requirements for the content of audit trail records are given.

This level of security features includes the requirements defined by the C2 level of the DoD Trusted Computer System Evaluation Criteria (TCSEC) (see **Referenced Documents**).

In addition to the TCSEC criteria, X/Open has added a requirement for extra selectivity of audit data collection. In particular, selectivity by event type is required. Events should be selectable on a per-user basis and on an individual per-process basis. For usability, it should also be possible to set a default system-wide level of auditing for new users of the system.

Further requirements beyond TCSEC level C2 may be included in future issues of the interface definition.

At a gross level the requirements can be grouped into two categories; Accountability and Authentication. These terms are formally defined in **Section 2.7**, **Definitions**.

## 2.2 SECURITY AUDITING

The purpose of defining standard interface functions to the audit facilities of a secure X/Open system is to permit the development of portable applications of the following type:

- Audit reduction packages, which require access to audit data in a standard format. This leads to a requirement to provide a standard interface to audit trails, as seen by audit reduction packages.
- Applications which are trusted to do their own auditing. There is thus a requirement for a trusted application to be able to append records to the audit trail. Applications that are trusted to audit themselves at a much coarser level of granularity than the Trusted Computing Base (TCB) may also be trusted to disable TCB auditing of their activities. Another interface is required to achieve this functionality.
- Packages providing improved usability for audit administrators. Standard audit control interfaces are therefore required.

The main technical requirements in defining standard interface functions to the audit facilities are:

- To meet or exceed the auditing criteria defined by external requirements bodies, for example, TCSEC C2.
- To ensure portability of applications and compatibility with possible future extensions to the XSI Security.

## 2.3 AUDIT TRAILS

The main requirement is that applications must be able to access audit trails in a portable way. The goals of this specification are to provide a definition that:

- Allows audit reduction tools to be developed independently of any particular system implementation.
- Allows new types of audit records to be generated either by the TCB or trusted applications without requiring modifications to existing audit tools.
- Can be used by an X/Open-compliant system to describe the operations performed by the system's Trusted Computing Base (TCB).
- Allows implementations considerable flexibility in the choice of what information is recorded in audit records.

How and where audit data is physically recorded is implementation-defined. Additionally the internal format of the audit trail is implementation-defined and is of no concern to this specification.

## 2.4 AUTHENTICATION EXTENSIONS

There is a need to define new system interfaces to:

- Permit the reading and writing of encrypted passwords in implementation-defined authentication databases.
- Examine and modify the rules for password aging.

Although not defined as part of the XSI, it is assumed that all systems supporting the Security Option will implement some form of user authentication mechanism. This may take the form of a password, which the system will validate against the authentication database entry for the specified user name. Access to the system should only be granted if the password is specified correctly.

The above interfaces are necessary to provide a portable interface to user authentication mechanisms, which may vary considerably from one system to another. Password aging is a mechanism by which system administrators can impose rules on the user community about when passwords must be changed. There is a requirement to provide such a mechanism. How it is implemented is system defined.

## 2.5 XSI CHANGES AND EXTENSIONS

As stated in **Chapter 1**, **Introduction**, a number of extensions are necessary to existing system interfaces and headers to support the XSI Security. These have been minimised as much as possible to prevent unnecessary disturbance of the base definition. Extensions are required at both the command and programming levels, primarily to cater for the maintenance of user audit identifiers (see definition of Audit ID). The interfaces affected by this requirement are described in **Chapter 3**, **Function and Interface**.

## 2.6 CONFIGURATION

XSI Security is not applicable to all systems, nor indeed to all deliveries of a system that supports the security interfaces. The XSI Security facility has therefore been divided into two separately supportable security options corresponding to the accountability and authentication categories introduced in **Section 2.1**, **General**. The options are:

i. ACCOUNTABILITY. This option contains the functions and interfaces related to process accounting, including the audit interface functions, audit trail contents and event identification.

ii. AUTHENTICATION. This option contains the functions and interfaces related to user authentication and the processing of entries in the authentication database.

Individual interface specifications identify to which option each function belongs.

An implementation may support one, both or neither of the above security options. If any part of an option is provided by an implementation, all interfaces and functions defined as being part of that option must be fully supported. An application can determine what options are supported by calling the *sysconf*() function.

On systems that offer neither security option, it is desirable that portable tools fail sensibly. To ensure this, the following must be provided even on systems that support neither of the security options:

i. The security option flags defined for the *sysconf*() function.

ii. An <**audit.h**> header.

iii. The symbolic constants defined in <**unistd.h**>.

iv. An embryonic implementation of each interface function, which need do nothing more than set *errno* to ENOSYS and indicate that the call was unsuccessful.

### 2.6.1 The ACCOUNTABILITY Option

The following new interface functions are defined for the ACCOUNTABILITY option:

| Interface |
|---|
| *aud_commit*() |
| *aud_config*() |
| *aud_discard*() |
| *aud_get_header*() |
| *aud_get_object*() |
| *aud_get_event_info*() |
| *aud_length*() |
| *aud_next*() |
| *aud_print*() |
| *aud_put_object*() |
| *aud_put_event_info*() |
| *aud_start*() |
| *aud_switch*() |
| *get_process_audit_ID*() |
| *get_process_audit_events*() |
| *get_user_audit_events*() |
| *map_audit_ID_to_user*() |
| *map_user_to_audit_ID*() |
| *set_process_audit_ID*() |
| *set_process_audit_events*() |
| *set_user_audit_ID*() |
| *set_user_audit_events*() |

**TABLE 1.** New Interfaces for the ACCOUNTABILITY Option

### 2.6.2 The AUTHENTICATION Option

The following new interface functions are defined for the AUTHENTICATION option:

| Interface |
|---|
| *get_password_aging()* |
| *secure_get_passwd_user()* |
| *secure_put_passwd_user()* |
| *set_password_aging()* |

**TABLE 2.** New Interfaces for the AUTHENTICATION Option

## 2.7 DEFINITIONS

### 2.7.1 Accountability

The property that enables activities on a system to be traced to individuals who may then be held responsible for their actions.

### 2.7.2 Appropriate Privileges

Zero or more implementation-defined privileges associated with a process with regard to the function calls and function call options defined in this interface specification that need special privileges.

### 2.7.3 Audit ID

Each authorised user of a secure system is identified by a unique arithmetic value of type **audit_ID_t**. This value is set at login time and for all subsequent child processes, including those initiated via the *at*, *batch* and *crontab* commands.

### 2.7.4 Audit Record

The smallest discrete unit of data that is recorded in the audit trail on the occurrence of an auditable event. An audit record is composed of attributes which define the characteristics of the auditable event.

### 2.7.5 Audit State

The audit state of a process includes the list of event classes and event types currently being audited and additionally whether auditing for that process is switched on or off.

### 2.7.6 Audit Trail

A file-like storage entity used to hold audit records produced by the Trusted Computing Base and Trusted applications.

### 2.7.7 Auditing Style

The set of parameters designed into the system by the Auditor that govern the contents of audit records generated by the system.

### 2.7.8 Auditor

The name assigned to the authorised individual(s) who is (are) responsible for auditing.

### 2.7.9 Authentication

The process that verifies the identity of a user, device, or other entity in a computer system, often as a prerequisite to allowing access to resources in a system. It is also the process that verifies the integrity of data that have been stored, transmitted or otherwise exposed to possible unauthorised modification.

### 2.7.10 Authentication Database

An implementation-specific file or files used to define the mapping between user names and encrypted passwords. The authentication database also contains rules for password aging.

### 2.7.11 Discretionary Access Control

An access control mechanism which allows a subject to grant its acquired access rights to other subjects without restriction.

### 2.7.12 Identification Database

An implementation-specific file or files used to define the one-to-one mapping between user names and audit IDs.

### 2.7.13 Mandatory Access Control

An access control mechanism which does not permit a subject to transfer information freely from one access domain to another.

### 2.7.14 Object

A passive entity in the system used for storage or transferral of information.

### 2.7.15 Subject

An active entity in the system which transfers information from one object to another.

### 2.7.16 TCB

See **Trusted Computing Base**.

### 2.7.17 Trusted Computing Base

The Trusted Computing Base (or TCB) consists of hardware, firmware and software that together enforce a security policy over a product or system. The TCB creates a basic protection environment and provides additional user services. The ability of a trusted computing base to correctly enforce a security policy depends solely on the mechanisms within the TCB and the correct input by system administrative personnel of parameters (e.g., a user's clearance) related to the security policy.

Chapter 3

# Function and Interface

## 3.1 SECURITY AUDITING

### 3.1.1 Audit Identifier Interfaces

TCSEC C2 requires that users be individually accountable for their security-relevant actions. This implies that a user should be accountable for all events in a session, from the moment of logging-in to the point of exiting the session. This is achieved by giving each individual a unique ‘‘audit identifier’’ (*audit_ID*), which is associated with the user’s initial process and inherited by all child processes. The *audit_ID* of a process cannot be changed by any subsequent action of the user. It is also inherited by any *at, batch* or *crontab* jobs (see the **X/Open Portability Guide**, **Issue 3**, **Volume 1**, **XSI Commands and Utilities**) initiated by the user.

It is further required that each user of the system has a distinct, individual user name. Systems supporting the ACCOUNTABILITY option shall maintain an identification database, in which a unique *audit_ID* is associated with each registered user of the system. In programming terms, an *audit_ID* is distinct from a UID and is defined as an arithmetic value of type **audit_ID_t** (see <**audit.h**>). The interface definition identifies various functions for setting and getting the *audit_ID* of processes and users. It does not define how the identification database should be implemented.

Most X/Open systems allow some users to log in to user names that are not accountable to an individual user (e.g., root, uucp, etc.). In a secure system, individual accountability must be maintained even when a user logs in to one of these ‘‘role’’ user names. The ACCOUNTABILITY option defines that the value of *audit_ID* shall accurately identify the user of a ‘‘role’’ user name. Because it is not relevant to application portability, the interface definition does not state how accountability should be achieved when a user logs in to a role user name.

The following interfaces are currently defined for accessing process *audit_IDs* and the identification database. Note that these interfaces can only be called successfully by processes having appropriate privileges.

| Interface |
|---|
| *get_process_audit_ID*() |
| *map_audit_ID_to user*() |
| *map_user_to_audit_ID*() |
| *set_process_audit_ID*() |
| *set_user_audit_ID*() |

### 3.1.2 Audit Reduction Interfaces

Audit reduction utilities need a standard means to access the audit trail and need the format of information read from the trail to be standardised. This section defines the interface functions provided to access audit trails.

Although there is no requirement that audit data be stored in a file when it is collected, nor any requirement placed on the internal form of the data, it is assumed that the data can be made available via file-like interfaces, and in ''records'' of standard format, for analysis by audit reduction tools. The term ''record'' is used to mean the audit data collected on the occurrence of a single auditable event.

Interfaces are provided for reading individual records from an audit trail, either sequentially or by seeking for the next record that matches a given search criterion. Other interfaces provide for the manipulation of data within an audit record.

The standard format in which audit trail records are made available via these interfaces is defined in **Section 3.2**, **Audit Record Format**. For the purpose of this part of the guide, it is sufficient to know that:

- each record contains basic attributes which define the nature of the event recorded, and
- a record may contain additional attributes, depending on the nature of the event and the configuration of the system.

The following interfaces are currently defined for use by audit reduction tools.

| Interface |
|---|
| *aud_discard*() |
| *aud_next*() |
| *aud_get_header*() |
| *aud_get_object*() |
| *aud_get_event_info*() |
| *aud_print*() |
| *aud_length*() |

#### Reading the Audit Trail

Records can be read from the audit trail with the *aud_next*() function. This function returns an audit record in a buffer. The structure of the information in this buffer is not defined, but operations are defined to access the information.

The function for reading audit records from the audit trail is:

size_t aud_next (fd, ard, predicate)

The *aud_next*() function allows an application to search for the next record in the trail matching specified criteria. This call has the side effect of establishing the criteria to be used on subsequent searches. This function will return the next record from the audit trail matching the current search criteria. If none are established, then the next record in the trail is returned.

This function returns the data via *ard*, which is the means of returning an identifier to a buffer allocated by the function.

The *fd* argument must point to an open audit trail. If a call to one of the above functions is successful, the cursor associated with *fd* will be left pointing at the next record in the audit trail. If a call is unsuccessful, the cursor will either be left unchanged or, where no further records are available, it will point to the end of the audit trail.

**Manipulating Audit Records**

Functions are provided to read information from the audit record, to determine its length and to print it. The functions to read information from the audit record are:

```
int aud_get_header(ard, header, version)
int aud_get_object(ard, object, version)
int aud_get_event_info(ard, event_info)
```

It is assumed that *ard* identifies a valid audit record buffer.

The *aud_get_header*() function returns the header information from the audit record in the buffer. The information is returned as a pointer via *header*.

The *aud_get_object*() function returns an object descriptor from the audit record. This function can be invoked repeatedly to read all object descriptors from the record.

The *aud_get_event_info*() function returns event-specific information from the audit record. This information is syntactically described by the record, but has no ascertainable semantics.

The *version* argument on each of the functions *aud_get_header* and *aud_get_object* ensures forward compatibility. The argument specifies the minimum acceptable value for the type of item required. If the audit record contains only items with a lower value of the *version* field, the function call will fail.

The other functions provided to manipulate an audit record are:

```
int aud_discard (ard)
int aud_print (fd, ard, mode)
size_t aud_length (ard)
```

The *aud_discard*() function will free any space allocated by the system to store audit records and attributes that have been read.

The *aud_print*() function will translate an audit record into an external format according to the *mode* parameter. The output is written into the file specified by *fd*. The *aud_length*() function will return the length of the audit record in the buffer.

### 3.1.3 Trusted Application Interfaces

Some processes may be given privilege to append records to the audit trail. The following interface functions are defined to satisfy this requirement:

| Interface |
|---|
| *aud_commit*() |
| *aud_discard*() |
| *aud_put_object*() |
| *aud_put_event_info*() |
| *aud_start*() |
| *aud_switch*() |

#### Writing the Audit Trail

Functions are provided to create and alter audit records. These include:

```
int aud_start(ard, event)
int aud_put_object(ard, object)
int aud_put_event_info(ard, event_info)
```

The *aud_start*() function returns the identifier of an internal audit record buffer in which a trusted application can prepare audit data for appending to the audit trail. The *event* parameter will define the event type.

The *aud_put_object*() and *aud_put_event_info*() functions may be used to alter the audit record defined by the *ard* parameter. *aud_put_object* adds an object descriptor to the record and *aud_put_event_info* adds event-specific information to the record. Both functions may be called repeatedly.

Functions are provided to write the completed audit records. These are:

```
int aud_commit(ard, client, result)
int aud_discard(ard)
```

The *aud_commit*() function may be called to commit the audit record in the *ard* buffer with the specified *result*. The *client* parameter may optionally be used to define the client on whose behalf a server is acting. The system will fill in the additional fields in the header and append the record to the system audit trail.

The *aud_discard*() function will free any space allocated by the system to store the audit record.

Note that standardisation of these interfaces does not restrict implementations to use of a standard internal audit record format. Implementations are free to map the structured data from the audit write calls into any appropriate internal format. However, the fact that the audit data is structured means that the mapping to an internal format should be done in a meaningful way, such that the process can be reversed when converting an audit trail back into the standard format.

**Auditing Suspension and Resumption**

A process with appropriate privileges to insert records into the audit trail may also be given the (possibly different) privilege to advise the TCB that standard auditing of its operations should be suspended or resumed. This may be useful to avoid recording unnecessary detail in the audit trail. The privilege to advise the TCB in this way should be available only to fully trusted software. The TCB may or may not actually suspend its auditing of the process, depending on the audit policy currently in use.

This capability is provided by the *aud_switch*() function, which takes a single argument indicating whether process auditing is to be switched **on** or **off**. Note that the audit state of a process, including the current **on** or **off** state of auditing, is inherited by a child if the process calls *fork*().

### 3.1.4 Audit Control Interfaces

These interfaces are only available to processes with appropriate privilege and their use may itself be audited. The following audit control functions are defined in this issue of the interface definition:

| Interface |
|---|
| *get_process_audit_events*() |
| *get_user_audit_events*() |
| *set_process_audit_events*() |
| *set_user_audit_events*() |
| *update_audit_events*() |

**Specifying Event Lists to be Audited**

Events to be audited are specified to the audit system in event lists. An event list is simply a row of event identifiers. Each identifier is simply a value of type **aud_event_t**. The identifiers may include both individual events and groups of events (see **Chapter 6**, **Audit Event Classes and Event Types**).

The *set_user_audit_events*() function is provided to turn on and off auditing of specified events for one or more users. Setting on or off specified events for all users of the system is permitted, as is setting on or off all events for specified users. For administrative convenience, it is possible to define a system default setting for events. This setting is used as the default set for a user name when it is added to the system. The interface also permits specified events for specified users to be set to the system default.

When a user's auditable events are changed, the new events are used for all subsequent login sessions and jobs initiated by the *at*, *batch* or *crontab* commands. The interface does not define the effect on sessions that already exist. However, an interface (*update_audit_events*()) is defined which ensures that all sessions are updated to reflect the current event lists.

The *get_user_audit_events*() function gets the current selected events for a specified user or the system default event list.

How and where user event lists are stored by a system is implementation-defined.

Two further functions (*set_process_audit_events*() and *get_process_audit_events*()) are provided to set and get the event list for the current process. The initial events for a session are set to the default event list defined for that user. These interfaces allow a process with appropriate privileges to modify its auditable events at a later stage.

### 3.1.5 Protecting the Audit Trail

TCSEC requires that the audit trail is protected from unauthorised access of all types. It is believed that the standard system file protection mechanisms defined in the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **System Interfaces and Headers** are sufficient to meet this requirement. Thus no changes or additions to the system protection mechanisms are defined for XSI Security.

## 3.2 AUDIT RECORD FORMAT

This section describes the common characteristics of audit records.

### 3.2.1 Purpose of Records

Audit records describe events; that is, there is a correspondence between some actual event that occurred and was noticed by the TCB and the audit record it generates. An audit record provides a largely context-independent description of an event. With an audit record, you know what happened, who caused it to happen, what it happened to, and when.

### 3.2.2 Audit Record Contents

Audit records contain a header, object descriptors and event-specific information. The header describes the event and subject information, including the event time and result. The object descriptors contain information about the objects affected by the event. The event-specific information is described syntactically only.

Although there is no requirement on how the system stores the audit record, logically it appears to the application as several structures which are ''read'' from the audit record by audit trail analysers. The structures for object descriptors and the event-specific information can also be put into an audit record by a trusted process; trusted processes will also set certain fields in the audit record header.

The header structure contains the following fields:

| | |
|---|---|
| *version* | the version number of the header |
| *subject* | the audit ID of the subject |
| *client* | the audit ID of the client |
| *event* | the event type |
| *time* | the time of the event |
| *time_off* | the time offset (in nanoseconds) |
| *status* | the audit status of the event |
| *pid* | the process ID |
| *dac* | a pointer to the subject DAC information |
| *mac* | a pointer to the subject MAC information |
| *net* | a pointer to the origin information |
| *priv* | a pointer to subject privilege information |

The definition of the *dac* structure will be inherited from other standards bodies working on discretionary access control. It is likely to contain the following members:

| | |
|---|---|
| *version* | the structure version number |
| *ruid* | the real user ID |
| *euid* | the effective user ID |

| | |
|---|---|
| *suid* | the saved user ID |
| *rgid* | the real group ID |
| *egid* | the effective group ID |
| *sgid* | the saved group ID |
| *ngroups* | the number of concurrent groups |
| *groups* | the array of concurrent groups |

The definition of the *mac* structure will be inherited from other standards bodies working on mandatory access control. It is likely to contain the following members:

| | |
|---|---|
| *version* | the structure version number |
| *type* | the type of MAC label |
| *fmt* | the format of MAC label |
| *size* | the size of the MAC label |
| *label* | a pointer to the MAC security label of the subject |

The *net* and *priv* structures are yet to be defined.

The object descriptor structure contains the following fields:

| | |
|---|---|
| *version* | the version number |
| *type* | the type of object |
| *mode* | the mode of access |
| *namefmt* | the format of the name |
| *namelen* | the length of the name |
| *name* | a pointer to the name of the object |
| *mac* | a pointer to the object MAC information |
| *dac* | a pointer to the object DAC information |

Each item of event-specific information contains the following members:

| | |
|---|---|
| *format* | a format specifier |
| *len* | the length of the information |
| *info* | a pointer to the data |

The *mac*, *dac*, *priv* and *net* fields from these headers may be null, indicating that either this information is not recorded in each record on the system or the attribute is not supported.

There is no limit on the length of any individual attribute except for the maximum size of {AUDIT_REC_MAX} bytes for the entire audit record.

The structures and the audit record may contain other information.

## 3.3 AUDIT EVENT CLASSES AND EVENT TYPES

Each audit record has exactly one audit event type. In addition, each audit event type belongs to one or more audit event class.

The audit event type is intended as a way of identifying all audit records that describe the same type of event; that is, events that differ only in the parameters supplied to an operation.

The audit event class is intended as a way of grouping audit event types in a way that is useful to an auditor.

The audit event type is an inherent property of an event and is recorded in each audit record. The audit event class, on the other hand, is a structure imposed from outside and is not recorded in an audit record.

Because nearly all operations have the common characteristic that they may succeed or fail, the true type of an event should be considered to incorporate both the audit event type and the status (success / failure) from the record header.

## 3.4 AUDITING STYLE

The auditing style of a system determines whether certain types of information are to be included in the audit record. This section describes each style option and the inquiry interface. In all cases, style options are binary (i.e., a system either implements the option or it doesn't). It is required that a system implement the same style in all audit records it generates.

For the most part, style options are a tradeoff of size and cost-to-generate against the ease of analysing audit data and the degree to which certain security requirements are met. Because certain types of information are difficult to record and store, it is important that a conforming implementation have the option not to generate them.

The facility to inquire into the current auditing style of a system is for the benefit of tools which perform their own auditing. Such tools are then able to generate audit records consistent with the style of the system.

The inquiry facility is not for use by portable audit analysis tools. A portable audit analysis tool must be prepared to establish the style for the audit trail under analysis by the contents of the records it encounters - the trail may have been generated by a different system. Such a tool must be prepared to operate on records generated by a system that implements any combination of these style options. Fortunately, since the options are orthogonal, this is comparatively easy for tools to cope with.

### 3.4.1 Auditing Style Interfaces

The inquiry interface for auditing style is:

int aud_config (option)

If the style option specified by the *option* argument is implemented, the *aud_config*() function will return the value 1. If the style option is not implemented, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate the error.

The following values of *option* are defined as symbolic constants in **<audit.h>**:

| Name | Description |
|---|---|
| AUS_ABSPATH | Include Absolute Pathnames |
| AUS_OBJMAC | Include Object MAC Information |
| AUS_OBJDAC | Include Object DAC information |

There are no interfaces for setting auditing style, as this operation is inherently implementation-dependent and probably cannot be adjusted dynamically.

### 3.4.2 Include Absolute Pathnames

Inquire by *aud_config* (**AUS_ABSPATH**);

If a system implements this option, all pathnames in objects are absolute. File relative pathnames are not permitted. This allows inquiries about files by name.

If a system does not implement this option, portable analysis tools must be prepared to satisfy by-name file inquiries by tracking each subject's current working directory and

root directory throughout their life in the audit trail.

### 3.4.3 Include Object MAC Information

Inquire by *aud_config* (**AUS_OBJMAC**);

If a system implements this option, all object descriptors include the MAC information appropriate to the system on which the record was generated.

### 3.4.4 Include Object DAC Information

Inquire by *aud_config* (**AUS_OBJDAC**);

If a system implements this option, all object descriptors include the DAC information appropriate to the system on which the record was generated.

## 3.5 XSI CHANGES AND EXTENSIONS

### 3.5.1 Commands and Utilities

The provision of security auditing requires that all processes set off during a user session are auditable. This means that the *audit_ID* of the session leader must be inherited by all child processes, including those initiated indirectly via autonomous process schedulers (e.g., for batch jobs). This affects the operation of a number of commands currently defined in the **X/Open Portability Guide**, **Issue 3**, **Volume 1**, **XSI Commands and Utilities**:

| Commands |
|---|
| *at* |
| *batch* |
| *crontab* |

The interface definition does not state what *audit_ID* should be used by schedulers themselves.

**Chapter 4**, **Commands and Utilities** provides a modified definition of the above commands for use in a secure environment. Specifically, the *audit_ID* of the initiating process must be exported and inherited by processes set up to run background commands.

### 3.5.2 System Interfaces and Headers

A number of system interfaces defined in the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **System Interfaces and Headers** are affected in different ways by the requirements for secure systems described in **Chapter 2**, **Overview**. Some, like the *fork ()* and *exec ()* functions, must ensure that *audit_ID*s and audit states are inherited correctly. Others, like the *sysconf ()* function, have been extended to enable applications to determine what security options are supported by an implementation.

The following system interfaces and headers are affected by XSI Security:

| Interface |
|---|
| *exec*() |
| *fork*() |
| *sysconf*() |
| <**limits.h**> |
| <**unistd.h**> |

Modified entries for these interfaces are presented in **Chapter 5**, **System Interfaces and Headers**. Note that XSI Security requires that additional symbolic constants are defined in **<limits.h>** and **<unistd.h>**. It does not alter the definition of any existing constants.

### 3.5.3 Passwords and Password Aging

Four new interfaces are defined to provide additional, secure interfaces to the authentication database:

| Interface |
|---|
| *get_password_aging*() |
| *secure_get_passwd_user*() |
| *secure_put_passwd_user*() |
| *set_password_aging*() |

The *secure_get_passwd_user*() function allows a process with appropriate privileges to read the encrypted password of a named user from the authentication database. Conversely, the *secure_put_passwd_user*() function allows a privileged process to write the encrypted password of a named user into the authentication database.

How and where passwords are stored in a system is implementation-defined.

Password aging is a mechanism by which system administrators can force users to change their passwords at regular intervals. This interval is defined by two values which give:

- the maximum number of days for which a password will remain valid, and
- a warning time, before the password expires, during which the users will be warned to change their passwords.

By manipulation of these values, it is possible for administrators to force a user to supply a new password when he or she next attempts to log in to the system. After a password has expired, it is undefined whether the associated user ID is still usable.

The *set_password_aging*() function allows a process with appropriate privileges to set the password aging rules for the system as a whole or for a named user. The *get_password_aging*() function allows a privileged process to determine (for a named user) the number of days for which the current password will remain valid, and the number of days since the password was last changed.

*Chapter 4*

# *Commands and Utilities*

This chapter contains modified definitions of standard commands and utilities for systems supporting XSI Security. Only commands whose behaviour is different in a secure environment are included. For a complete definition of all commands and utilities see the **X/Open Portability Guide**, **Issue 3**, **Volume 1**, **XSI Commands and Utilities**.

**NAME**

at, batch - execute commands at a later time

**SYNOPSIS**

**at time [date] [+increment]**
**at -r job**
**at -l job**

**batch**

**DESCRIPTION**

See *at* in the **X/Open Portability Guide**, **Issue 3**, **Volume 1**, **XSI Commands and Utilities**.

**SECURITY**

On systems supporting the ACCOUNTABILITY option, jobs executed by the *at* and *batch* commands will inherit the audit ID of the initiating process. The default audit state of the associated user will be used as the initial audit state of any resulting processes. How these settings are exported is implementation-defined.

**CHANGE HISTORY**

First released in the **X/Open Portability Guide**, **Issue 2**.

**This Document**

A SECURITY section has been added indicating additional requirements for systems supporting the Security Interfaces.

**NAME**

crontab - user crontab file

**SYNOPSIS**

**crontab [file]**
**crontab -r**
**crontab -l**

**DESCRIPTION**

See *crontab* in the **X/Open Portability Guide**, **Issue 3**, **Volume 1**, **XSI Commands and Utilities**.

**SECURITY**

On systems supporting the ACCOUNTABILITY option, jobs initiated via the *crontab* command will inherit the audit ID of the user who owns the associated *crontab* file. The default audit state of that user will also be used as the initial audit state of any resulting processes.

**CHANGE HISTORY**

First released in the **X/Open Portability Guide**, **Issue 2**.

**This Document**

A SECURITY section has been added indicating additional requirements for systems supporting the Security Interfaces.

*Chapter 5*

# *System Interfaces and Headers*

This chapter contains definitions of system interfaces and headers for XSI Security. In addition, it contains modified definitions of standard system interfaces defined in the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interfaces and Headers**, whose behaviour is different in a secure environment.

**NAME**

aud_commit - append record to audit trail

**SYNOPSIS**

**#include <audit.h>**

**int aud_commit (ard, client, status)**
**aud_rec_t ard;**
**audit_ID_t client;**
**aud_stat_t status;**

**DESCRIPTION**

The *aud_commit*() function will complete the audit record identified by *ard* and write it to the audit trail. The audit record buffer identified by *ard* will have been returned to the process by a previous, successful call to the *aud_start*() function.

Audit records are completed by filling in the defined fields of the header. The *event* field will have been initialized when the audit buffer was created. The *status* and *client* fields are filled in at an implementation-defined time. If the process is not acting on behalf of a client, the *client* parameter should be specified as AUDIT_NOBODY.

Storage space used for the audit record is freed with this call if successful. If unsuccessful, the record should be discarded with the *aud_discard*() function.

A process must have appropriate privileges to call this function successfully.

**RETURN VALUE**

If successful, the *aud_commit*() function returns a value of 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *aud_commit*() function may return any of the errors identified for the *write*() function. It will also fail if:

[EINVAL] The *ard* argument does not identify a valid audit record buffer.

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

**SEE ALSO**

*aud_discard*(), *aud_put_event_info*(), *aud_put_object*(), *aud_start*(), <**audit.h**>, *write*() in the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interfaces and Headers**.

**CHANGE HISTORY**

First released in this document.

**NAME**

aud_config - get auditing style option setting

**SYNOPSIS**

**#include <audit.h>**

**int aud_config (option)**
**int option;**

**DESCRIPTION**

The *aud_config*() function allows an application to inquire about the current setting of an auditing style option. The auditing style of a system determines whether certain types of information are to be included in the audit record. The facility to inquire into the current auditing style of a system is for the benefit of tools which perform their own auditing. Such tools are then able to generate audit records consistent with the style of the system.

The style option is identified by the setting of the *option* argument, which can be set to one of the following symbolic constants defined in <**audit.h**>:

| Name | Description |
|---|---|
| AUS_ABSPATH | Include Absolute Pathnames |
| AUS_OBJMAC | Include Object MAC Information |
| AUS_OBJDAC | Include Object DAC Information |

A process must have appropriate privileges to call this function successfully.

**RETURN VALUE**

If the specified option is implemented, the *aud_config*() function returns a value of 1. If the option is not implemented, a value of 0 will be returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *aud_config*() function will fail if:

[EINVAL] The *option* argument does not contain a valid style option constant.

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

**SEE ALSO**

<**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

aud_discard - discard audit record buffer

**SYNOPSIS**

**#include <audit.h>**

**int aud_discard (ard)**
**aud_rec_t ard;**

**DESCRIPTION**

The *aud_discard*() function will discard the audit record buffer identified by *ard*, including the freeing of any storage space that may be allocated to the buffer. The audit record buffer identified by *ard* may have been returned to the process by a previous, successful, call to the *aud_start*() function or to the *aud_next*() function.

A process must have appropriate privileges to call this function successfully.

**RETURN VALUE**

If successful, the *aud_discard*() function returns a value of 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *aud_discard*() function will fail if:

[EINVAL] The *ard* argument does not identify a valid audit record buffer.

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

**SEE ALSO**

*aud_commit*(), *aud_next*(), *aud_put_object*(), *aud_put_event_info*(), *aud_start*(), <**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

aud_get_header - read the audit record header

**SYNOPSIS**

**#include <audit.h>**

**int aud_get_header (ard, header, version)**
**aud_rec_t ard;**
**struct aud_hdr_t **header;**
**unsigned char version;**

**DESCRIPTION**

The *aud_get_header*() function will return a pointer to the header structure via the argument *header*.

Currently only one header version is defined: AUD_XSTD_HDR, which contains the following members:

```
unsigned char       version;
audit_ID_t          subject;
audit_ID_t          client;
aud_event_t         event;
time_t              time;
unsigned            time_off;
aud_stat_t          status;
pid_t               pid;
struct aud_dac_t    *dac;
struct aud_mac_t    *mac;
struct aud_net_t    *net;
struct aud_priv_t   *priv;
```

The *version* argument specifies the minimum acceptable value for the *version* field within the header. Headers with a lower *version* value will not be returned.

The structures for the *mac* and *dac* header components will be defined by other standards groups.

Storage allocated by this function must be explicitly freed by a call to *aud_discard*().

**RETURN VALUE**

If successful, the *aud_get_header*() function returns a value of 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *aud_get_header*() function will fail if:

[EINVAL] The *ard* argument does not contain a valid audit record.

[EINVAL] The *version* argument does not specify a supported header type.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

The *aud_get_header*() function may fail if:

[ENOMEM] Insufficient storage space is available to hold the header structure.

**SEE ALSO**

*aud_discard*(), *aud_get_event_info*(), *aud_get_object*(), *aud_next*(), <**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

aud_get_object - read an object descriptor from the audit record

**SYNOPSIS**

**#include <audit.h>**

**int aud_get_object (ard, object, version)**
**aud_rec_t ard;**
**struct aud_obj_t **object;**
**unsigned char version;**

**DESCRIPTION**

The *aud_get_object*() function will return a pointer to an object descriptor via the argument *object*.

Currently only one version of object descriptor is defined: AUD_XSTD_OBJ, which contains the following members:

```
unsigned char       version;
unsigned short      type;
unsigned short      mode;
unsigned char       namefmt;
unsigned char       namelen;
char                *name;
struct aud_mac_t    *mac;
struct aud_dac_t    *dac;
```

The *version* argument specifies the minimum acceptable value for the *version* field within the object. Objects with a lower *version* value will not be returned.

The *type* parameter defines the type of object. This should be one of the following values:

AUD_OBJ_FILE
AUD_OBJ_DIR
AUD_OBJ_DEV
AUD_OBJ_FIFO
AUD_OBJ_MSG
AUD_OBJ_SHM
AUD_OBJ_SEM
AUD_OBJ_STOR
AUD_OBJ_IPC

The *AUD_OBJ_STOR* and *AUD_OBJ_IPC* values should be used to denote storage and IPC objects which are implementation-defined. The other values correspond to defined XPG objects.

The *mode* parameter defines how the object was accessed. This is a bitmask and contains one of:

AUD_OBJ_STAT
AUD_OBJ_CONTENTS

and one of:

AUD_OBJ_READ
AUD_OBJ_WRITE
AUD_OBJ_EXEC
AUD_OBJ_SEARCH

The *namefmt* parameter defines the format of the name and may be any of the valid values defined for the format of the event-specific information.

The structures for the *mac* and *dac* object components will be defined by other standards groups.

The objects are returned in the order in which they were added to the record by the implementation or application. The number of remaining objects is returned as the status code. When this value reaches zero, there are no unreturned object descriptors in the record. The number of unreturned descriptors may be queried at any point if this function is invoked with a NULL pointer for the *object* parameter.

Storage allocated by this function must be explicitly freed by a call to *aud_discard*().

**RETURN VALUE**

If successful, the *aud_get_object*() function returns a value greater than or equal to zero which corresponds to the number of object descriptors remaining in the audit record. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *aud_get_object*() function will fail if:

[EINVAL] The *ard* argument does not contain a valid audit record.

[EINVAL] The *version* argument does not specify a supported object type.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

The *aud_get_object*() function may fail if:

[ENOMEM] Insufficient storage space is available to hold the object descriptor.

**SEE ALSO**

*aud_discard*(), *aud_get_event_info*(), *aud_get_header*(), *aud_next*(), <**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

aud_get_event_info - read event-specific information from the audit record

**SYNOPSIS**

**#include <audit.h>**

**int aud_get_event_info (ard, event_info)**
**aud_rec_t ard;**
**struct aud_event_info_t **event_info;**

**DESCRIPTION**

The *aud_get_event_info*() function will return a pointer to an item of event-specific information via the argument *event_info*. This item contains a structure with the following members:

```
unsigned short    format;
unsigned short    len;
char              *info;
```

The *format* field defines formatting information that may be used to display the data referenced by *info*. Possible values of this field are:

AUD_FORMAT_CHAR
AUD_FORMAT_SHORT
AUD_FORMAT_INT
AUD_FORMAT_LONG
AUD_FORMAT_STRING
AUD_FORMAT_OPAQUE

This function may be invoked multiple times to read all event-specific information from the audit record.

The event-specific items are returned in the order in which they were added to the record by the implementation or application. The number of remaining event-specific items is returned as the status code. When this value reaches zero, there are no unreturned event-specific items in the audit record. The number of unreturned event-specific items may be queried at any point if this function is invoked with a NULL pointer for the *event_info* parameter.

Storage allocated by this function must be explicitly freed by a call to *aud_discard*().

**RETURN VALUE**

If successful, the *aud_get_event_info*() function returns the number of *event_info* descriptors remaining in the audit record. If unsuccessful, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *aud_get_event_info*() function will fail if:

[EINVAL] The *ard* argument does not contain a valid audit record.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

The *aud_get_event_info*() function may fail if:

[ENOMEM] Insufficient storage space is available to hold the event-specific information.

**SEE ALSO**

*aud_discard*(), *aud_get_header*(), *aud_get_object*(), *aud_next*(), <**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

aud_length - get audit record length

**SYNOPSIS**

**#include <audit.h>**

**size_t aud_length (ard)**
**aud_rec_t ard;**

**DESCRIPTION**

The *aud_length*() function will return the total length (in bytes) of the audit record identified by *ard*. The audit record will have been either read into *ard* by a previous, successful, call to the *aud_next*() function or allocated by the *aud_start*() function.

**RETURN VALUE**

If successful, the *aud_length*() function returns the length of the audit record. Otherwise, a value of (*size_t*)-1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *aud_length*() function will fail if:

[EINVAL] The *ard* argument does not identify a valid audit record buffer.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

**SEE ALSO**

*aud_next*(), *aud_start*(), <**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

aud_next - read audit trail record

**SYNOPSIS**

```
#include <sys/types.h>
#include <audit.h>
size_t aud_next (fd, ard, predicate)
int fd;
aud_rec_t *ard;
char *predicate;
```

**DESCRIPTION**

This function attempts to read the ‘‘next’’ record from the audit trail specified with the file descriptor *fd*.

This function also defines the predicate to be used to search for the next record and will return a matching record if one exists. The *aud_next* function can then be used to search for successive records in the trail that match the defined predicate. By default, if no predicate is explicitly defined the function will return the next record read from the audit trail.

The *predicate* may include comparison, IN or LIKE predicates of the form defined in the **X/Open Portability Guide**, **Issue 3**, **Volume 5**, **Data Management**, for SQL search conditions with WHERE clauses. The SQL predicates may be connected with AND, OR or NOT operators and parentheses may be used to change the order of evaluation.

The left hand side of each predicate must be the name of one of the audit record attributes defined below:

| | |
|---|---|
| EVENT | the name of the audit event type. |
| STATUS | the audit status of the record. |
| TIME | the time when the record was generated. Values must be specified in the format defined by the LC_TIME environmental variable. |
| PROCESS | the process ID of the subject. |
| AUDIT_ID | the audit ID of the subject or client (if specified). |
| REAL_UID | the real user ID of the subject or client (if specified). |

If the *predicate* parameter is a NULL pointer, the search criteria remains unchanged from the last call of *aud_next*() which specified a *predicate*.

If the *predicate* parameter points to a NULL or empty string, the search condition will be set to the default (the next sequential record in the trail).

If the file descriptor is valid and points to a valid audit trail and a matching record is found in the trail, the contents of the first such record are returned to the caller in the manner described below.

The function will allocate a buffer, place the contents of the record into it and return an identifier to it via the argument *ard*.

The information returned via *ard* is suitable to supply to calls of the functions *aud_get_header*, *aud_get_object* and *aud_get_event_info*.

Storage allocated by this function must be explicitly freed by a call to *aud_discard*().

If the function successfully reads an audit trail record, the cursor associated with *fd* will be advanced to point at the next record in the audit trail.

If no appropriate record can be found in the audit trail, a value of zero is returned and the cursor is advanced to the end of the audit trail.

If a call is unsuccessful, a value of -1 is returned and the cursor remains unchanged.

**RETURN VALUE**

If successful, the function returns the length of the audit record. If there are no more records in the audit trail, the function returns a value of 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

These functions may return any of the errors identified for the *read*() function. They will also fail if:

[EINVAL] The cursor associated with *fd* is not positioned at a valid audit record.

[EINVAL] The predicate defined by *predicate* is not syntactically valid.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

The functions may fail if:

[ENOMEM] Insufficient storage space is available to hold the audit record.

**SEE ALSO**

*aud_discard*(), *aud_get_event_info*(), *aud_get_header*(), *aud_get_object*(), *aud_length*(), *aud_print*(), *read*(), <**audit.h**>, in the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interfaces and Headers**.

**CHANGE HISTORY**

First released in this document.

**NAME**

aud_print - translates a binary audit record

**SYNOPSIS**

**#include <audit.h>**

**int aud_print (fd, mode, ard)**
**int fd;**
**int mode;**
**aud_rec_t ard;**

**DESCRIPTION**

The *aud_print*() function translates an audit record identified by the *ard* parameter from the system-specific format to an external representation specified by the *mode* parameter. The output is written to the file descriptor specified by the *fd* parameter.

The audit record will have been assembled in *ard* by a previous, successful, call to the *aud_next*() function.

The *mode* parameter can specify one of:

AUD_STD_ASCII
AUD_STD_XDR
AUD_STD_NDR

The AUD_STD_ASCII format provides a textual representation of the audit record.

The AUD_STD_XDR and AUD_STD_NDR formats provide system-independent binary representations of the audit record.

**RETURN VALUE**

If successful, the *aud_print*() function returns a value of 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *aud_print*() function will fail if:

[EINVAL] The *ard* argument does not identify a valid audit record.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

[EINVAL] The *mode* argument is not one of AUD_STD_ASCII, AUD_STD_XDR or AUD_STD_NDR.

[ENOSYS] The requested translation mode is not supported on this implementation.

**SEE ALSO**

*aud_length*(), *aud_next*(), <**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

aud_put_object - put object descriptor into record buffer

**SYNOPSIS**

**#include <audit.h>**

**int aud_put_object (ard, object)**
**aud_rec_t ard;**
**aud_obj_t *object;**

**DESCRIPTION**

The *aud_put_object*() function will insert the object description in the structure pointed to by the *object* parameter into the audit record buffer identified by *ard*. The audit record buffer, *ard*, will have been initialized by a previous, successful, call to the *aud_start*() function.

The *aud_obj_t* structure contains the following members:

```
unsigned char       version;
unsigned short      type;
unsigned short      mode;
unsigned char       namefmt;
unsigned short      namelen;
char                *name;
struct aud_mac_t    *mac;
struct aud_dac_t    *dac;
```

The *type* parameter defines the type of object. This should be one of the following values:

AUD_OBJ_FILE
AUD_OBJ_DIR
AUD_OBJ_DEV
AUD_OBJ_FIFO
AUD_OBJ_MSG
AUD_OBJ_SHM
AUD_OBJ_SEM
AUD_OBJ_STOR
AUD_OBJ_IPC

The *AUD_OBJ_STOR* and *AUD_OBJ_IPC* values are used to denote storage and IPC objects which are implementation-defined. The other values correspond to defined XPG objects.

The *mode* parameter defines how the object was accessed. This is a bitmask and should contain one of:

AUD_OBJ_STAT
AUD_OBJ_CONTENTS

and one of:

AUD_OBJ_READ
AUD_OBJ_WRITE
AUD_OBJ_EXEC
AUD_OBJ_SEARCH

The *namefmt* parameter defines the format of the name and may be any of the valid values defined for the format of the event-specific information (see *aud_get_event_info*()).

The structures for the *mac* and *dac* object components will be defined by other standards groups.

The *aud_put_object*() function may be called repeatedly in order to add several descriptors to the record. The order of these descriptors will be preserved by the system, so that they may be read in the same order by the *aud_get_object*() function. This order will also be preserved by the *aud_print*() function.

**RETURN VALUE**

If successful, the *aud_put_object*() function returns a value of 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *aud_put_object*() function will fail if:

[EINVAL] The *ard* argument does not identify a valid audit record.

[EINVAL] The *version* field of the *object* structure is not valid.

[EINVAL] The *type*, *mode* or *namefmt* fields of the *object* structure contained illegal values.

[EPERM] The caller does not have the appropriate privilege to invoke this function or to set the value of the specified attribute.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

The *aud_put_object*() function may fail if:

[ENOMEM] The function cannot allocate the additional memory in the audit record buffer to contain the new object.

**SEE ALSO**

*aud_commit*(), *aud_discard*(), *aud_get_event_info*(), *aud_get_object*(), *aud_print*(), *aud_put_event_info*(), *aud_start*(), <**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

aud_put_event_info - put event-specific information into record buffer

**SYNOPSIS**

**#include <audit.h>**

**int aud_put_event_info (ard, event_info)**
**aud_rec_t ard;**
**aud_event_info_t *event_info;**

**DESCRIPTION**

The *aud_put_event_info*() function will insert the event-specific information in the structure pointed to by the *event_info* parameter into the audit record buffer identified by *ard*. The audit record buffer, *ard*, will have been initialized by a previous, successful, call to the *aud_start*() function.

The *event_info* structure contains the following members:

```
unsigned short    format;
unsigned short    len;
char              *info;
```

The *format* member defines the format of the information in the *info* buffer, while the *len* member gives the length of the buffer, in bytes (see *aud_get_event_info*() for possible values of *format*).

The *aud_put_event_info*() function may be called repeatedly in order to add several descriptors to the record. The order of these descriptors will be preserved by the system, so that they may be read in the same order by the *aud_get_event_info*() function. This order will also be preserved by the *aud_print*() function.

**RETURN VALUE**

If successful, the *aud_put_event_info*() function returns a value of 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *aud_put_event_info*() function will fail if:

[EINVAL] The *ard* argument does not identify a valid audit record.

[EINVAL] The *format* field of the *event_info* structure contained an illegal value.

[EPERM] The caller does not have the appropriate privilege to invoke this function or to set the value of the specified attribute.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

The *aud_put_event_info*() function may fail if:

[ENOMEM] The function cannot allocate memory in the audit record buffer to contain the new event_info.

**SEE ALSO**

*aud_commit*(), *aud_discard*(), *aud_get_event_info*(), *aud_print*(), *aud_put_object*(), *aud_start*(), <**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

aud_start - allocate audit record buffer

**SYNOPSIS**

**#include <audit.h>**

**int aud_start (ard, event)**
**aud_rec_t *ard;**
**aud_event_t event;**

**DESCRIPTION**

The *aud_start*() function allocates an internal audit record buffer and returns its identifier in *ard*. Information can be added to the record in the buffer with the *aud_put_object*() and *aud_put_event_info*() functions. The audit record can be completed and appended to the audit trail by calling the *aud_commit*() function.

The *event* and *subject* fields in the header will be set according to the *event* parameter and audit ID of the current process, respectively. The *client* and *status* fields in the header are set with the *aud_commit* function. All other fields in the header are completed by the system. It is implementation-defined when the header values are actually established.

If the process terminates before the audit record is finished and committed, the audit record buffer is discarded and no data will be written to the audit trail.

Whether or not partially completed audit record buffers are made available to subsequent child processes is undefined. On some systems, if a child attempts to access such buffers it may invalidate an audit record being built in the parent, or result in two similar records being written to the audit trail.

A process must have appropriate privileges to call this function successfully.

**RETURN VALUE**

If successful, the *aud_start*() function returns a value of 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *aud_start*() function will fail if:

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

The *aud_start*() function may fail if:

[ENOMEM] Insufficient storage space is available to hold the audit record buffer.

**SEE ALSO**

*aud_commit*(), *aud_discard*(), *aud_put_event_info*(), *aud_put_object*(), <**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

aud_switch - suspend or resume process auditing

**SYNOPSIS**

**#include <audit.h>**

**int aud_switch (audit_state)**
**enum { off, on } audit_state;**

**DESCRIPTION**

The *aud_switch*() function sets process auditing *off* or *on*. The request is advisory and may be ignored either wholly or partially if the auditing policy of the system (or the TCB) prohibits the suspension of process auditing. A request to suspend auditing from a process that is privileged to do its own auditing only affects audit data generated by the TCB (i.e., it does not affect auditing performed by the *aud_commit*() function).

The current state of this switch is inherited by a child if the process calls the *fork*() function.

A process must have appropriate privileges to call this function successfully.

**RETURN VALUE**

If successful, the *aud_switch*() function returns the previous value of the audit switch for the calling process. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *aud_switch*() function will fail if:

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

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

**SEE ALSO**

*aud_commit*(), *fork*(), <**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

environ, execl, execv, execle, execve, execlp, execvp - execute a file

**SYNOPSIS**

**extern char **environ;**

**int execl (path, arg0, arg1, ..., argn, (char*)0)**
**char *path, *arg0, *arg1, ..., *argn;**

**int execv (path, argv)**
**char *path, *argv[];**

**int execle (path, arg0, arg1, ..., argn, (char*)0, envp)**
**char *path, *arg0, *arg1, ..., *argn, *envp[];**

**int execve (path, argv, envp)**
**char *path, *argv[], *envp[];**

**int execlp (file, arg0, arg1, ..., argn, (char*)0)**
**char *file, *arg0, *arg1, ..., *argn;**

**int execvp (file, argv)**
**char *file, *argv[];**

**DESCRIPTION**

See *exec* in the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interface and Headers**.

**SECURITY**

On systems supporting the ACCOUNTABILITY option, the following attributes will also be inherited by the new process image:

- audit_ID,
- audit switch state,
- process audit events.

**CHANGE HISTORY**

First released in the **X/Open Portability Guide**, **Issue 1**.

**This Document**

A SECURITY section has been added indicating additional requirements for systems supporting the Security Interfaces.

**NAME**

fork - create a new process

**SYNOPSIS**

**#include <sys/types.h>**

**pid_t fork ()**

**DESCRIPTION**

See *fork*() in .

**SECURITY**

On systems supporting the ACCOUNTABILITY option, the following attributes are also inherited by a child process:

- audit_ID,
- audit switch state,
- process audit events.

**CHANGE HISTORY**

First released in the **X/Open Portability Guide**, **Issue 1**.

**This Document**

A SECURITY section has been added indicating additional requirements for systems supporting the Security Interfaces.

**NAME**

get_password_aging - get information on password aging

**SYNOPSIS**

**#include <audit.h>**

**int get_password_aging (user_name, time_left, when_changed)**
**char *user_name;**
**int *time_left;**
**int *when_changed;**

**DESCRIPTION**

The *get_password_aging*() function returns information about the current state of password aging for user *user_name*. This information is returned in the integers pointed to by the *time_left* and *when_changed* arguments, as follows:

*time_left* The number of days for which the current password will remain valid. A negative value indicates that this time has already expired.

*when_changed*
The number of days since the password was last changed.

Users can and should change their passwords before *time_left* days have elapsed. After a password has expired, it is undefined whether or not the associated user ID is still usable.

A process must have appropriate privileges to call this function.

**RETURN VALUE**

If successful, the *get_password_aging*() function returns a value of 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *get_password_aging*() function will fail if:

[EINVAL] The *user_name* argument does not identify a valid user.

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The AUTHENTICATION option is not supported on this implementation.

**SEE ALSO**

*set_password_aging*(), <**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

get_process_audit_ID - get the audit ID of the calling process

**SYNOPSIS**

**#include <audit.h>**

**audit_ID_t get_process_audit_ID ()**

**DESCRIPTION**

The *get_process_audit_ID*() function returns the audit ID of the calling process.

A process must have appropriate privileges to call this function successfully.

**RETURN VALUE**

If successful, the *get_process_audit_ID*() function returns the audit ID of the process. Otherwise, a value of *audit_ID_t*-1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *get_process_audit_ID*() function will fail if:

[EINVAL] The *audit_ID* of the process is not set.

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

**SEE ALSO**

<**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

get_process_audit_events - get process event list

**SYNOPSIS**

```
#include <audit.h>

aud_event_t *get_process_audit_events (nmemb)
size_t *nmemb;
```

**DESCRIPTION**

The *get_process_audit_events*() function returns a pointer to an array of up to {AUDIT_MAX_SIZE} elements containing a list of the event classes and event types currently being audited in the calling process. This list is initialised at session start and is inherited by all child processes. The initial setting of the list is determined from the default audit event list defined for the associated user (see *set_user_audit_events*()). Subsequent changes to the process event list can only be made by processes with appropriate privileges calling *set_process_audit_events*() or *update_audit_events*().

A value indicating the number of valid elements in the event list is stored in the location pointed to by the *nmemb* argument. If this value is not required, *nmemb* can be set to the NULL pointer and no value will be returned.

A process must have appropriate privileges to call this function successfully.

**RETURN VALUE**

If successful, the *get_process_audit_events*() function returns a pointer to the current event list. Otherwise, a NULL pointer is returned and *errno* is set to indicate the error.

**ERRORS**

The *get_process_audit_events*() function will fail if:

[EINVAL] No events are currently being audited in the process.

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

The *get_process_audit_events*() function may fail if:

[ENOMEM] The implementation requires to allocate space to hold the return value of the function and there is insufficient free memory space currently available.

**APPLICATION USAGE**

The array used to hold the process event list may be held in a static data area whose contents are overwritten by this and other audit system function calls. Applications are recommended to copy this array to a local data area if it is to be saved or reused.

**SEE ALSO**

*set_process_audit_events*(), *set_user_audit_events*(), *update_audit_events*(), <**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

get_user_audit_events - get user event list

**SYNOPSIS**

**#include <audit.h>**

**aud_event_t *get_user_audit_events (user_name, nmemb)**
**char *user_name;**
**size_t *nmemb;**

**DESCRIPTION**

The *get_user_audit_events*() function returns a pointer to an array of up to {AUDIT_MAX_SIZE} elements containing a list of the audit event classes and event types currently turned on for user *user_name*. If *user_name* is the NULL pointer, the function returns a pointer to an array containing the list of audit event classes and event types currently defined as the system default event list (see *set_user_audit_events*()).

A value indicating the number of valid elements in the event list is stored in the location pointed to by the *nmemb* argument. If this value is not required, *nmemb* can be set to the NULL pointer and no value will be returned.

A process must have appropriate privileges to call this function successfully.

**RETURN VALUE**

If successful, the *get_user_audit_events*() function returns a pointer to the audit event list for user *user_name*, or for the system if *user_name* is the NULL pointer. Otherwise, a NULL pointer is returned and *errno* is set to indicate the error.

**ERRORS**

The *get_user_audit_events*() function will fail if:

[EINVAL] The *user_name* argument is not the NULL pointer and does not identify a valid user.

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

The *get_user_audit_events*() function may fail if:

[ENOMEM] The implementation requires to allocate space to hold the return value of the function and there is insufficient free memory space currently available.

**APPLICATION USAGE**

The array used to hold the user event list may be held in a static data area whose contents are overwritten by this and other audit system function calls. Applications are recommended to copy this array to a local data area if it is to be saved or reused.

**SEE ALSO**

*set_user_audit_events*(), <**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

map_audit_ID_to_user - get user name for audit ID

**SYNOPSIS**

```
#include <audit.h>

char *map_audit_ID_to_user (audit_ID)
audit_ID_t audit_ID;
```

**DESCRIPTION**

The *map_audit_ID_to_user*() function returns a pointer to a string containing the user name associated with *audit_ID* in the identification database. How and where this information is stored is implementation-defined.

A process needs appropriate privileges to call this function successfully.

**RETURN VALUE**

If successful, the *map_audit_ID_to_user*() function returns a pointer to a string containing the user name associated with *audit_ID*. Otherwise, a NULL pointer is returned and *errno* is set to indicate the error.

**ERRORS**

The *map_audit_ID_to_user*() function will fail if:

[EINVAL] The *audit_ID* argument does not contain a valid audit identifier.

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

The *map_audit_ID_to_user*() function may fail if:

[ENOMEM] The implementation requires to allocate space to hold the return value of the function and there is insufficient free memory space currently available.

**APPLICATION USAGE**

The string used to hold the user name may be held in a static data area whose contents are overwritten by other audit system function calls. Applications are recommended to copy this string to a local data area if it is to be saved or reused.

**SEE ALSO**

<**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

map_user_to_audit_ID - get audit ID for user name

**SYNOPSIS**

```
#include <audit.h>

audit_ID_t map_user_to_audit_ID (user_name)
char *user_name;
```

**DESCRIPTION**

The *map_user_to_audit_ID*() function returns the audit ID of the user identified by the string pointed to by *user_name*. This information is taken from the system's identification database, the format and location of which are implementation-defined.

A process must have appropriate privileges to call this function successfully.

**RETURN VALUE**

If successful, the *map_user_to_audit_ID*() function returns the audit ID associated with *user_name*. Otherwise, a value of **audit_ID_t**-1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *map_user_to_audit_ID*() function will fail if:

[EINVAL] The *user_name* argument does not identify a valid user.

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

**SEE ALSO**

<**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

secure_get_passwd_user - get password for user

**SYNOPSIS**

**#include <audit.h>**

**int secure_get_passwd_user (name, password, nbyte)**
**char *name, *password;**
**unsigned nbyte;**

**DESCRIPTION**

The *secure_get_passwd_user*() function searches the authentication database for the user entry identified in the string pointed to by *name*. If found, the encrypted form of the password associated with user *name* is returned in the buffer pointed to by *password*.

No more than *nbyte* bytes will be placed in the buffer. If the size of the encrypted password is less than or equal to *nbyte*, the *secure_get_passwd_user*() function will place the password in the buffer and return its size in bytes. If its size is greater than *nbyte*, the first *nbyte* bytes will be placed in the buffer and the function will return a value greater than *nbyte*.

If *nbyte* is 0, *password* is permitted to be a null pointer. This form of a call is useful for determining the size of an encrypted password.

The algorithm for encrypting passwords is implementation-defined.

A process must have appropriate privileges to call this function successfully.

**RETURN VALUE**

If successful, the *secure_get_passwd_user*() function returns the length of the encrypted password in bytes. If the value returned is greater than *nbyte*, only *nbyte* bytes will have been placed in the buffer pointed to by *password*. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *secure_get_passwd_user*() function will fail if:

[EINVAL] The *name* argument does not identify a valid user name.

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The AUTHENTICATION option is not supported on this implementation.

**SEE ALSO**

<**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

secure_put_passwd_user - set password for user

**SYNOPSIS**

**#include <audit.h>**

**int secure_put_passwd_user (name, password, nbyte)**
**char *name, *password;**
**unsigned nbyte;**

**DESCRIPTION**

The *secure_put_passwd_user*() function sets the password associated with user *name* in the authentication database. The *password* argument is a pointer to a character array of *nbyte* bytes, containing an encrypted form of the new password.

The algorithm for encrypting passwords is implementation-defined.

A process must have appropriate privileges to call this function successfully.

**RETURN VALUE**

If successful, the *secure_put_passwd_user*() function returns a value of 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *secure_put_passwd_user*() function will fail if:

[EINVAL] The *name* argument does not identify a valid user name.

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The AUTHENTICATION option is not supported on this implementation.

**SEE ALSO**

<**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

set_password_aging - set information on password aging

**SYNOPSIS**

**#include <audit.h>**

**int set_password_aging (user_name, time_to_expire, warning_time)**
**char *user_name;**
**int time_to_expire, warning_time;**

**DESCRIPTION**

The *set_password_aging*() function sets information about password aging for user *user_name*. This information is defined by the values of *time_to_expire* and *warning_time* as follows:

*time_to_expire*

The maximum number of days for which a password will remain valid. After a password has expired, it is undefined whether or not the associated user ID is still useable.

*warning_time*

The period in days, before the password expires, during which the user is warned to change the password.

If *warning_time* ≥ *time_to_expire*, the password must be changed on the next login. The maximum life of a password may be no greater than one year (365 days).

If *user_name* is the NULL pointer, the function sets information about password aging for all current users of the system. This supersedes existing password aging information, if present.

A process must have appropriate privileges to call this function successfully.

**RETURN VALUE**

If successful, the *set_password_aging*() function returns a value of 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *set_password_aging*() function will fail if:

[EINVAL] The *user_name* argument is not the NULL pointer and does not identify a valid user, or *time_to_expire* or *warning_time* contain an invalid setting.

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The AUTHENTICATION option is not supported on this implementation.

**SEE ALSO**

<**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

set_process_audit_ID - set the audit_ID of the calling process

**SYNOPSIS**

**#include <audit.h>**

**int set_process_audit_ID (audit_ID)**
**audit_ID_t audit_ID;**

**DESCRIPTION**

The *set_process_audit_ID*() function will set the audit identifier of the calling process to *audit_ID*. A call to this function will fail if the audit identifier of the process is already set, or if the value of *audit_ID* is not a valid audit identifier.

A process must have appropriate privileges to call this function successfully.

Once set, the audit ID of a process is inherited by all child processes (see *fork*()), by new process images (see *exec*()), and by all background jobs initiated from the process (see *at* and *crontab*).

**RETURN VALUE**

If successful, the *set_process_audit_ID*() function will return a value of 0. Otherwise, the value -1 will be returned and *errno* will be set to indicate the error.

**ERRORS**

The *set_process_audit_ID*() function will fail if:

[EINVAL] The value of *audit_ID* is not a valid audit identifier.

[EPERM] The process does not have appropriate privileges to call this function, or the audit identifier of the process is already set.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

**SEE ALSO**

*at*, *crontab*, *exec*(), *fork*(), <**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

set_process_audit_events - set process event list

**SYNOPSIS**

**#include <audit.h>**

**int set_process_audit_events (audit_state, event_ptr, nmemb)**
**enum { off, on } audit_state;**
**aud_event_t *event_ptr;**
**size_t nmemb;**

**DESCRIPTION**

The *set_process_audit_events*() function will set the event classes or event types identified in the array pointed to by *event_ptr* to *audit_state* in the calling process. The array pointed to by *event_ptr* is defined as an event list of *nmemb* elements, where *nmemb* is a value in the range (1-{AUDIT_MAX_SIZE}).

A process must have appropriate privileges to call this function successfully.

**RETURN VALUE**

If successful, the *set_process_audit_events*() function will return a value of 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *set_process_audit_events*() function will fail if:

[EINVAL] The value of the *audit_state* or *nmemb* argument is invalid, or an element of the array pointed to by *event_ptr* identifies an invalid event class or event type.

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

**SEE ALSO**

**<**audit.h**>.**

**CHANGE HISTORY**

First released in this document.

**NAME**

set_user_audit_ID - set audit ID for user

**SYNOPSIS**

**#include <audit.h>**

**int set_user_audit_ID (user_name, audit_ID)**
**char *user_name;**
**audit_ID_t audit_ID;**

**DESCRIPTION**

The *set_user_audit_ID*() function sets the audit ID for the user identified in the string pointed to by *user_name*. This function updates the identification database. Existing processes belonging to user *user_name* will not be affected.

A call to this function will fail if the process does not have appropriate privileges to access the identification database.

**RETURN VALUE**

If successful, the *set_user_audit_ID*() function returns a value of 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *set_user_audit_ID*() function will fail if:

[EINVAL] The *user_name* argument does not identify a valid user, or *audit_ID* does not contain a unique audit identifier.

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

**SEE ALSO**

<**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

set_user_audit_events - set user event list

**SYNOPSIS**

**#include <audit.h>**

**int set_user_audit_events (audit_state, user_names, event_ptr, nmemb)**
**enum { off, on, default } audit_state;**
**char **user_names;**
**aud_event_t *event_ptr;**
**size_t nmemb;**

**DESCRIPTION**

The *set_user_audit_events*() function will set the event classes or event types specified in the array pointed to by *event_ptr* to *audit_state*, for the users identified in the NULL-terminated list of user names pointed to by *user_names*.

If *user_names* is not the NULL pointer, the function sets the specified event classes or event types to *audit_state* for each named user. If the first element of the array pointed to by *event_ptr* is set to the constant AUDIT_EVENTS_ALL, all event classes and event types will be set to *audit_state*. If the value of *audit_state* is **default**, the list of audit events for each named user will be set to the system default event list. If the first string in the string list pointed to by *user_names* is set to ''*'', the event list of all current users will be updated.

If *user_names* is the NULL pointer, the function sets the specified event classes or event types to *audit_state* in the system default event list. If the first element of the array pointed to by *event_ptr* is set to the constant AUDIT_EVENTS_ALL, all event classes and event types will be set to *audit_state*. If the *audit_state* argument is set to **default**, the call will have no effect.

If the first element of *event_ptr* is not the constant AUDIT_EVENTS_ALL, *event_ptr* is defined as an event list of *nmemb* elements, where *nmemb* is an integer value in the range (1-{AUDIT_MAX_SIZE}).

A process must have appropriate privileges to call this function successfully.

**RETURN VALUE**

If successful, the *set_user_audit_events*() function will return a value of 0. Otherwise, a value of -1 will be returned and *errno* will be set to indicate the error.

**ERRORS**

The *set_user_audit_events*() function will fail if:

[EINVAL] The value of the *audit_state* or *nmemb* argument is invalid, or one of the users identified by *user_names* is not registered with the system, or the event list pointed to by *event_ptr* identifies an invalid event.

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

**SEE ALSO**

<**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

sysconf - get configurable system variables

**SYNOPSIS**

**#include <unistd.h>**

**long sysconf (name)**
**int name;**

**DESCRIPTION**

See *sysconf*() in the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interface and Headers**.

**SECURITY**

The following system variables and symbolic constants have been added for the Security Interfaces:

| Variable | Value of name |
|---|---|
| ACCOUNTABILITY | _SC_ACCOUNTABILITY |
| AUDIT_MAX_SIZE | _SC_AUDIT_MAX_SIZE |
| AUDIT_REC_SIZE | _SC_AUDIT_REC_SIZE |
| AUTHENTICATION | _SC_AUTHENTICATION |

The ACCOUNTABILITY and AUTHENTICATION variables will be defined if an implementation supports these options. In this case, the *sysconf*() function will return the value of the variable. Otherwise, -1 is returned without changing the value of *errno*.

On systems supporting the ACCOUNTABILITY option, AUDIT_MAX_SIZE gives the maximum number of elements in an audit event list and AUDIT_REC_SIZE gives the maximum number of bytes in an audit record.

**CHANGE HISTORY**

First released in the **X/Open Portability Guide**, **Issue 3**.

**This Document**

A SECURITY section has been added indicating additional variables and symbolic constants for the Security Interfaces.

**NAME**

update_audit_events - update process event list for users

**SYNOPSIS**

**#include <audit.h>**

**int update_audit_events (user_names)**
**char **user_names;**

**DESCRIPTION**

The *update_audit_events*() function will update the list of event classes and event types being audited for processes belonging to users identified in *user_names*. The *user_names* argument is a NULL terminated array, each element of which is a pointer to a string containing a user name. If the first user name in the list is the string ‘‘*’’, all user processes will be updated.

If any of the strings in the array pointed to by *user_names* identifies an invalid user name, the behaviour of this function is undefined (i.e., processes belonging to valid user names given elsewhere in the list may or may not be updated correctly).

This function allows a process with appropriate privileges to update process audit event lists so they become consistent with the current specification of user audit event lists (see *set_user_audit_events*()).

**RETURN VALUE**

If successful, the *update_audit_events()* function returns a value of 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

**ERRORS**

The *update_audit_events()* function will fail if:

[EINVAL] One of the strings in the array pointed to by *user_names* identifies an invalid user name.

[EPERM] The process does not have appropriate privileges to call this function.

[ENOSYS] The ACCOUNTABILITY option is not supported on this implementation.

**SEE ALSO**

*set_user_audit_events*(), <**audit.h**>.

**CHANGE HISTORY**

First released in this document.

**NAME**

audit.h - data types and constants for security auditing

**SYNOPSIS**

**#include <audit.h>**

**DESCRIPTION**

The following types are defined in this header through **typedef**s:

aud_event_t The type of an audit event class or audit event type identifier.

aud_rec_t The type of an audit record structure or array identifier.

aud_stat_t The type of the *status* field within an audit record header.

audit_ID_t The type of an audit identifier.

size_t Unsigned integral type returned by the **sizeof** operator.

The following structures are defined for use in querying or setting attributes in audit records:

aud_hdr_t The audit record header which contains the following members:

```
unsigned char       version;
audit_ID_t          subject;
audit_ID_t          client;
aud_event_t         event;
time_t              time;
unsigned            time_off;
aud_stat_t          status;
pid_t               pid;
struct aud_dac_t    *dac;
struct aud_mac_t    *mac;
struct aud_net_t    *net;
struct aud_priv_t   *priv;
```

aud_obj_t The audit record object which contains the following members:

```
unsigned char       version;
unsigned short      type;
unsigned short      mode;
unsigned char       namefmt;
unsigned char       namelen;
char                *name;
struct aud_mac_t    *mac;
struct aud_dac_t    *dac;
```

aud_event_info_t

The audit record event_info which contains the following members:

```
unsigned short  format;
unsigned short  len;
char            *info;
```

aud_dac_t The dac structure which contains the following members:

| | |
|---|---|
| unsigned char | version |
| uid_t | ruid |
| uid_t | euid |
| uid_t | suid |
| gid_t | rgid |
| gid_t | egid |
| gid_t | sgid |
| int | ngroups |
| gid_t | *groups |

aud_mac_t The mac structure which contains the following members:

| | |
|---|---|
| unsigned char | version |
| unsigned short | type |
| unsigned char | mt |
| size_t | size |
| char | *label |

aud_net_t The net structure which contains the following members:

(To be defined)

aud_priv_t The priv structure which contains the following members:

(To be defined)

The following symbolic constants are defined for general use:

AUDIT_EVENTS_ALL
: A value of type **aud_event_t** which is a wild card event indicating all event classes and event types.

AUDIT_NOBODY
: A value of type **audit_ID_t** which is used to specify an empty value of audit ID.

NULL The null pointer.

The following symbolic constants are defined for version values in audit record structures:

AUD_XSTD_HDR
: Version of the header structure.

AUD_XSTD_OBJ
: Version of the object structure.

AUD_XSTD_DAC
: Version of the dac structure.

AUD_XSTD_MAC
: Version of the mac structure.

The following symbolic constants are defined for values of the *format* field within the *aud_event_info_t* structure and the *namefmt* field within the *aud_obj_t* structure:

AUD_FORMAT_CHAR
AUD_FORMAT_SHORT
AUD_FORMAT_INT
AUD_FORMAT_LONG
AUD_FORMAT_STRING
AUD_FORMAT_OPAQUE

The following symbolic constants are defined for values of the *type* field within the *aud_obj_t* structure:

AUD_OBJ_FILE
AUD_OBJ_DIR
AUD_OBJ_DEV
AUD_OBJ_FIFO
AUD_OBJ_MSG
AUD_OBJ_SHM
AUD_OBJ_SEM
AUD_OBJ_STOR
AUD_OBJ_IPC

The following symbolic constants are defined for values of the *mode* field within the *aud_obj_t* structure:

AUD_OBJ_STAT
AUD_OBJ_CONTENTS
AUD_OBJ_READ
AUD_OBJ_WRITE
AUD_OBJ_EXEC
AUD_OBJ_SEARCH

The following symbolic constants are defined for values of the *mode* argument to *aud_print*():

AUD_STD_ASCII
AUD_STD_XDR
AUD_STD_NDR

The following symbolic constants are defined for the *aud_config*() function:

AUS_ABSPATH
AUS_OBJMAC
AUS_OBJDAC

The following symbolic constants are defined for status values in audit headers:

AUR_SUCCESS
AUR_FAIL_ACC
AUR_FAIL_DAC
AUR_FAIL_MAC
AUR_FAIL_PRIV
AUR_FAIL_OTHER

The following symbolic constants are defined for audit event classes:

AEC_ACCESS_CHANGE
AEC_ACCESS_DENIALS
AEC_ADMIN_OPERATOR
AEC_AUTHENTICATION
AEC_OBJECT_AVAILABLE
AEC_OBJECT_CREATION
AEC_OBJECT_DELETION
AEC_OBJECT_MODIFICATION
AEC_OBJECT_TO_SUBJECT
AEC_OBJECT_UNAVAILABLE
AEC_PRIVILEGE
AEC_PROCESS
AEC_PROCESS_CONTROL
AEC_RESOURCE_DENIALS
AEC_SYSTEM

The following symbolic constants are defined for audit event types:

AET_AUDIT SWITCH
AET_CHDIR
AET_CHMOD
AET_CHOWN
AET_CHROOT
AET_CREAT
AET_EXEC
AET_EXECE
AET_EXIT
AET_FORK
AET_KILL
AET_LINK
AET_LOGIN_USER
AET_LOGOUT_USER
AET_MKDIR
AET_MKFIFO
AET_MSGCTL
AET_MSGGET
AET_OPEN
AET_RENAME
AET_RMDIR
AET_SECURE_PUT_PASSWD_USER
AET_SEMCTL
AET_SEMGET
AET_SETGID
AET_SETUID
AET_SET_PASSWORD_AGING
AET_SET_PROCESS_AUDIT_EVENTS
AET_SET_PROCESS_AUDIT_ID
AET_SET_USER_AUDIT_EVENTS
AET_SHMCTL
AET_SHMGET
AET_SWITCH_USER
AET_UNLINK
AET_UPDATE_AUDIT_EVENTS

The following are declared as either functions or macros:

*aud_commit*()
*aud_config*()
*aud_discard*()
*aud_get_header*()
*aud_get_object*()
*aud_get_event_info*()
*aud_length*()
*aud_next*()
*aud_print*()
*aud_put_object*()
*aud_put_event_info*()
*aud_start*()
*aud_switch*()
*get_password_aging*()
*get_process_audit_ID*()
*get_process_audit_events*()
*get_user_audit_events*()
*map_audit_ID_to_user*()
*map_user_to_audit_ID*()
*secure_get_passwd_user*()
*secure_put_passwd_user*()
*set_password_aging*()
*set_process_audit_ID*()
*set_process_audit_events*()
*set_user_audit_ID*()
*set_user_audit_events*()
*update_audit_events*()

**SEE ALSO**

*aud_commit*(), *aud_config*(), *aud_discard*(), *aud_get_event_info*(), *aud_get_header*(), *aud_get_object*(), *aud_length*(), *aud_next*(), *aud_next_where*(), *aud_print*(), *aud_put_event_info*(), *aud_put_object*(), *aud_start*(), *aud_switch*(), *get_password_aging*(), *get_process_audit_events*(), *get_process_audit_ID*(), *get_user_audit_events*(), *map_audit_ID_to_user*(), *map_user_to_audit_ID*(), *secure_get_passwd_user*(), *secure_put_passwd_user*(), *set_password_aging*(), *set_process_audit_events*(), *set_process_audit_ID*(), *set_user_audit_events*(), *set_user_audit_ID*(), *update_audit_events*().

**CHANGE HISTORY**

First released in this document.

**NAME**

limits.h - implementation-specific constants

**SYNOPSIS**

**#include <limits.h>**

**DESCRIPTION**

See <**limits.h**> in the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interface and Headers**.

**SECURITY**

If the ACCOUNTABILITY option is supported, the following constants will be defined in <**limits.h**>. These constants may be made available from the *sysconf*() function and are additional to those defined in the **X/Open Portability Guide**, **Issue 3**.

| *Name* | *Description* | *Minimum Acceptable Value* |
|---|---|---|
| AUDIT_MAX_SIZE | maximum number of elements in an audit event list | * |
| AUDIT_REC_MAX | maximum number of bytes in an audit record | 65535 |

**CHANGE HISTORY**

First released in the **X/Open Portability Guide**, **Issue 1**.

**This Document**

A SECURITY section has been added indicating additional constants for the Security Interfaces.

**NAME**

unistd.h - standard symbolic constants and structures

**SYNOPSIS**

**#include <unistd.h>**

**DESCRIPTION**

See <**unistd.h**> in the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interface and Headers**.

**SECURITY**

The following symbolic constants are defined if the associated option is supported by an implementation:

- ACCOUNTABILITY
- AUTHENTICATION

The following symbolic constants are defined for the *sysconf*() function:

- _SC_ACCOUNTABILITY
- _SC_AUDIT_MAX_SIZE
- _SC_AUDIT_REC_SIZE
- _SC_AUTHENTICATION

**CHANGE HISTORY**

First released in the **X/Open Portability Guide**, **Issue 1**.

**This Document**

A SECURITY section has been added indicating additional constants for the Security Interfaces.

Chapter 6

# Audit Event Classes and Event Types

This section defines the audit event classes and event types that will be supported by a system implementing the ACCOUNTABILITY option. For each event type, this includes a definition of the information that will be recorded in an associated audit record. For each event class, this includes a list of the event types contained in that class.

The audit event type provides a method of identifying all audit records that describe the same type of event; that is, event type records differ only in the parameters supplied to an operation. For example, the event type **AET_FORK** identifies all the audit records relating to the *fork*() operation. The event type also serves as a modifier or identifier of the event-specific data. For example, a process ID in a record of type **AET_FORK** indicates a new process has been spawned, while a process ID in a record of type **AET_KILL** indicates a process has been deleted from the system. Thus the event-specific information without the event type as a modifier provides incomplete or worthless information.

The class names provide an auditor with a simple means of specifying audit criteria for selection or analysis without requiring a detailed knowledge of audit event types. For example, the **AEC_PROCESS** event class provides a grouping of events which relate to process creation and/or process deletion. This group would contain event types such as **AET_EXIT, AET_FORK** and **AET_KILL.**

The audit event type is an inherent attribute of an audit record, each audit record therefore contains the audit event type. The audit event class is a grouping of audit event types, which is inherently site-dependent and as such not known to the audit generating programs, nor is it contained in the audit records. This grouping may be reduced or expanded by the audit administrator based on site needs.

## 6.1 SUMMARY OF AUDITING OPERATIONS

Two levels of auditing are defined. First, there is auditing of operations performed by programs at the System Interface level. Second, there is auditing of operations performed by users of the system. Auditing performed at the System Interface level is inherently geared towards the system call interface, i.e., there is nearly a one-to-one mapping of event to system call. Auditing performed at the User Interface level is geared more towards a task basis; for example, logging into the system.

### 6.1.1 Auditing at the System Interface

The following interfaces, published in the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interfaces and Headers**, are defined as the minimum set of system interface functions that should be auditable by a conforming implementation. The event type is a symbolic constant defined in <**audit.h** >.

| Interface | Event Type |
|---|---|
| *chdir*() | AET_CHDIR |
| *chmod*() | AET_CHMOD |
| *chown*() | AET_CHOWN |
| *chroot*() | AET_CHROOT |
| *creat*() | AET_CREAT |
| *execl*() | AET_EXEC |
| *execle*() | AET_EXECE |
| *execlp*() | AET_EXEC |
| *execv*() | AET_EXEC |
| *execve*() | AET_EXECE |
| *execvp*() | AET_EXEC |
| *_exit*() | AET_EXIT |
| *exit*() | AET_EXIT |
| *fork*() | AET_FORK |
| *kill*() | AET_KILL |
| *link*() | AET_LINK |
| *mkdir*() | AET_MKDIR |
| *mkfifo*() | AET_MKFIFO |
| *open*() | AET_OPEN |
| *rename*() | AET_RENAME |
| *rmdir*() | AET_RMDIR |
| *setgid*() | AET_SETGID |
| *setuid*() | AET_SETUID |
| *unlink*() | AET_UNLINK |

In addition to the list defined above, the following new interfaces defined for the ACCOUNTABILITY option shall be auditable:

| Interface | Event Type |
|---|---|
| *aud_switch*() | AET_AUDIT_SWITCH |
| *set_process_audit_ID*() | AET_SET_PROCESS_AUDIT_ID |
| *set_process_audit_events*() | AET_SET_PROCESS_AUDIT_EVENTS |
| *set_user_audit_events*() | AET_SET_USER_AUDIT_EVENTS |
| *update_audit_events*() | AET_UPDATE_AUDIT_EVENTS |

If the AUTHENTICATION option is present the following interfaces should be auditable:

| Interface | Event Type |
|---|---|
| *secure_put_passwd_user*() | AET_SECURE_PUT_PASSWD_USER |
| *set_password_aging*() | AET_SET_PASSWORD_AGING |

In addition, the following IPC interfaces should be auditable if the option is present:

| Interface | Event Type |
|---|---|
| *msgctl*() | AET_MSGCTL |
| *msgget*() | AET_MSGGET |
| *semctl*() | AET_SEMCTL |
| *semget*() | AET_SEMGET |
| *shmctl*() | AET_SHMCTL |
| *shmget*() | AET_SHMGET |

Note that it may be difficult for applications to detect the presence of the IPC option, since pre- and post-selection of events may be done on remote machines.

Clearly the list of events defined above is not inclusive. Commonly used applications, such as databases and networks, are not covered by the events defined above. As such it is likely that other event types and system interface functions for auditing may be defined on a site-specific basis. The format and contents of audit records associated with non-standard event types is implementation-defined.

### 6.1.2 Auditing at the User Interface

The following are defined as the minimum set of user interface operations that should be auditable by a conforming implementation. Since user interface auditing is inherently task-oriented, these operations are defined generically. The event type is a symbolic constant defined in <**audit.h** >.

| Operation | Event Type |
|---|---|
| user login | AET_LOGIN_USER |
| user logout | AET_LOGOUT_USER |
| switch of user Id's | AET_SWITCH_USER |

Additional user interface event types are likely to be defined by individual systems. The format and contents of audit records associated with non-standard event types is implementation-defined.

## 6.2 AUDIT EVENT TYPES

This section describes the standard audit event types in greater detail. In particular, it defines the minimum set of information that shall be recorded in the audit trail for each event.

Each audit record contains a **header** in which generic attributes applicable to every event are recorded (e.g., event type, status of the event). **Section 3.2.2**, **Audit Record Contents** describes the **header** in greater detail.

Other items of information or attributes are only meaningful within the context of a specific event. For example, the pathname for the AET_CHROOT event indicates the pathname to the new root, while the pathname for the AET_CREAT event indicates the pathname to a new file system object. These additional items of information are recorded either in an **object** structure or in an **event_info** structure.

The remainder of this section details the event-specific items for each event type defined by the ACCOUNTABILITY option. It also indicates whether the information is recorded in an **object** structure or an **event_info** structure.

The ACCOUNTABILITY option defines the minimum set of attributes which shall be audited by a conforming implementation. An implementation may record additional attributes; however, the **objects** and **event_infos** defined below must appear before any additional information.

The order of **objects** in an audit record is significant. Similarly, the order of **event_infos** in a record is significant. There is, however, no relationship between the order of **objects** with respect to the order of **event_infos**.

Some attributes within **headers** and **objects** may change as a result of the event. Unless otherwise stated, the values recorded in the **headers** and **objects** are those that are current prior to the event.

Most event types are defined to contain a return value, which may be set to errno. The return value will indicate the success/failure indicator returned by the operation.

The audit records for some event types are defined to contain a ''file pathname'' attribute. This will be either a file absolute pathname or a file relative pathname, depending on the auditing style of the system. This attribute is recorded for interfaces that specify a *path* argument, both to record the setting of *path* and to describe the associated filestore object. In cases where a *path* argument contains an invalid pointer, the attribute is assigned to NULL.

Note that systems whose auditing style supports relative pathnames will have to audit all event types which record changes to the path (except *chdir*) so the absolute pathname may be rebuilt if needed.

The remainder of this section describes the event-specific information recorded for each event type. Each item of information is listed along with its storage attribute (i.e., object/event_info).

### 6.2.1 AET_AUDIT_SWITCH

| Attribute | Storage Structure |
|---|---|
| audit state | event_info |
| return value | event_info |

The **audit state** attribute records the *audit_state* argument. This is invoked whenever the state of auditing is changed.

### 6.2.2 AET_CHDIR

| Attribute | Storage Structure |
|---|---|
| file pathname | object |
| return value | event_info |

The **file pathname** attribute describes the file referenced by the *path* argument. This event type records changes in the current working directory. If a system's auditing style is such that only relative pathnames are recorded, a post-processing utility will require that this event be audited in order to rebuild the absolute pathnames.

### 6.2.3 AET_CHMOD

| Attribute | Storage Structure |
|---|---|
| file pathname | object |
| mode | event_info |
| return value | event_info |

The **file pathname** attribute describes the file referenced by the *path* argument. The **mode** attribute records the value supplied in the *mode* argument.

This event records changes in the file's access control list attributes.

### 6.2.4 AET_CHOWN

| Attribute | Storage Structure |
|---|---|
| file pathname | object |
| owner | event_info |
| group | event_info |
| return value | event_info |

The **file pathname** attribute describes the file referenced by the *path* argument, the **owner** attribute records the value supplied in the *owner* argument, and the **group** attribute records the value supplied in the *group* argument.

This event records changes in the file's owner and/or group attributes.

### 6.2.5 AET_CHROOT

| Attribute | Storage Structure |
|---|---|
| file pathname | object |
| return value | event_info |

The **file pathname** attribute describes the file referenced by the *path* argument.

This event records changes to the starting point for path searches. If a system's auditing style is such that only relative pathnames are recorded, a post-processing utility will require that this event be audited in order to rebuild the absolute pathnames.

### 6.2.6 AET_CREAT

| Attribute | Storage Structure |
|---|---|
| file pathname | object |
| mode | event_info |
| return value | event_info |

The **file pathname** attribute describes the file referenced by the *path* argument. The **mode** attribute records the value supplied in the *mode* argument.

This event tracks the introduction of new file system objects.

### 6.2.7 AET_EXEC

| Attribute | Storage Structure |
|---|---|
| file pathname | object |
| command args | event_info |
| return value | event_info |

This event is recorded for any of the functions *execl*, *execlp*, *execv* or *execvp*.

The **file pathname** attribute records the path of the file to be executed.

The **command args** records the contents of the arguments supplied to the command to be executed. If any of *arg0* ... *argn*, *argv* or any of the elements in the array pointed to *argv* contain invalid pointers, the **command args** is assigned to NULL.

### 6.2.8 AET_EXECE

| Attribute | Storage Structure |
|---|---|
| file pathname | object |
| command args | event_info |
| environment | event_info |
| return value | event_info |

This event is recorded for any of the functions *execle* or *execve*.

The **file pathname** attribute records the path of the file to be executed.

The **command args** records the contents of the arguments supplied to the command to be executed. If any of *arg0* ... *argn*, *argv* or any of the elements in the array pointed to *argv* contain invalid pointers, the **command args** is assigned to NULL.

The **environment** records the contents of the *envp* argument. If the *envp* argument or any of the elements in the array pointed to by *envp* contain an invalid pointer, the **environment** attribute is assigned to NULL.

### 6.2.9 AET_EXIT

| Attribute | Storage Structure |
|---|---|
| status | event_info |

This event is recorded for any of the functions *_exit* or *exit*.

The **status** attribute records the value of the *status* argument.

No return value is specified since there can be no return from a *_exit* or *exit* call.

### 6.2.10 AET_FORK

| Attribute | Storage Structure |
|---|---|
| process | object |
| return value | event_info |

The **process** attribute describes the child process spawned by a successful call of *fork*. If the call is unsuccessful, the **process** attribute is assigned to NULL.

This event records the introduction of new processes into the system.

### 6.2.11 AET_KILL

| Attribute | Storage Structure |
|---|---|
| pid | event_info |
| sig | event_info |
| return value | event_info |

The **pid** attribute records the *pid* argument and the **sig** attribute records the *sig* argument.

This event records the deletion of processes from the system.

### 6.2.12 AET_LINK

| Attribute | Storage Structure |
|---|---|
| file pathname | object |
| new pathname | event_info |
| return value | event_info |

The attribute **file pathname** describes the file referenced by the *path1* argument (i.e., the existing file). The attribute **new pathname** records the path described by the *path2* argument (i.e., the new directory entry to be created).

This event records the introduction of a new name into the file system name space.

### 6.2.13 AET_LOGIN_USER

| Attribute | Storage Structure |
|---|---|
| user name | event_info |
| process id | event_info |
| failure indicator | event_info |

The **user name** attribute records the username of the login operation.

For a successful login operation the value of **process id** is the process id of the process that runs the ''program to use as shell'' (see <**pwd.h**> in the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interface and Headers**). The field **failure indicator** is assigned to NULL in this case.

For an unsuccessful login operation, a human-readable message may be recorded in the **failure indicator** attribute indicating the reason for failure. This would typically be the message returned by the login operation to the user. The field **process id** is assigned to NULL in this case.

This event records the introduction of a new user identity into the system.

#### 6.2.14 AET_LOGOUT_USER

| Attribute | Storage Structure |
|---|---|
| user name | event_info |
| process id | event_info |

The attribute **user name** records the user performing the *logout* operation. The field **process id** records the *pid* of the login process.

No failure indicator is provided since there is no return from a logout.

#### 6.2.15 AET_MKDIR

| Attribute | Storage Structure |
|---|---|
| file pathname | object |
| mode | event_info |
| return value | event_info |

The **file pathname** attribute describes the path referenced by the *path* argument. The **mode** attribute records the value supplied in the *mode* argument.

This event tracks the introduction of new file system objects.

#### 6.2.16 AET_MKFIFO

| Attribute | Storage Structure |
|---|---|
| file pathname | object |
| mode | event_info |
| return value | event_info |

The **file pathname** attribute describes the file referenced by the *path* argument. The **mode** attribute records the value supplied in the *mode* argument.

This event tracks the introduction of new file system objects.

### 6.2.17 AET_MSGCTL

| Attribute | Storage Structure |
|---|---|
| message queue | object |
| cmd | event_info |
| buf | event_info |
| return value | event_info |

The **message queue** attribute describes the message queue identified by the *msqid* argument. The **cmd** attribute records the value supplied in the *cmd* argument. The **buf** attribute records the value supplied in the *buf* argument.

This event tracks the introduction of new file system objects.

### 6.2.18 AET_MSGGET

| Attribute | Storage Structure |
|---|---|
| message queue | object |
| key | event_info |
| msgflg | event_info |
| return value | event_info |

The **message queue** attribute describes the message queue obtained by the call of *msgget*. The **key** attribute records the value supplied in the *key* argument. The **msgflg** attribute records the value supplied in the *msgflg* argument.

This event tracks the introduction of new file system objects.

### 6.2.19 AET_OPEN

| Attribute | Storage Structure |
|---|---|
| file pathname | object |
| mode | event_info |
| open state | event_info |
| return value | event_info |

The **file pathname** attribute describes the file referenced by the *path* argument.

The **mode** attribute records the value supplied in the *mode* argument. The field *mode* is only recorded if the O_CREAT flag is set in *oflag*, otherwise it is assigned to NULL.

The **open state** attribute records the type of open performed (i.e., read, write, creat). As such, this enables an application to differentiate an open for read (access of a file system object) from an open for write (create / modify of a file system object).

This event may track the introduction of new file system objects (i.e., O_CREAT).

### 6.2.20 AET_RENAME

| Attribute | Storage Structure |
|---|---|
| file pathname | object |
| new pathname | event_info |
| return value | event_info |

The **file pathname** attribute describes the file referenced by the *old pathname* argument. The **new pathname** attribute records the argument of that name.

This event tracks modification of the file systems name space.

### 6.2.21 AET_RMDIR

| Attribute | Storage Structure |
|---|---|
| file pathname | object |
| return value | event_info |

The **file pathname** attribute describes the path referenced by the *path* argument.

This event tracks deletion of file system objects.

### 6.2.22 AET_SECURE_PUT_PASSWD_USER

| Attribute | Storage Structure |
|---|---|
| user name | event_info |
| return value | event_info |

The attribute **user name** records the name specified by the *username* argument. If the *username* argument contains an invalid pointer, the contents of the **user name** attribute are assigned to NULL.

### 6.2.23 AET_SEMCTL

| Attribute | Storage Structure |
|---|---|
| semaphore | object |
| semnum | event_info |
| cmd | event_info |
| arg | event_info |
| return value | event_info |

The **semaphore** attribute describes the semaphore identified by the *semid* argument. The **semnum** attribute records the value supplied in the *semnum* argument, the **cmd** attribute records the value supplied in the *cmd* argument, and the **arg** attribute records the value supplied in the *arg* argument.

This event tracks the introduction of new file system objects.

### 6.2.24 AET_SEMGET

| Attribute | Storage Structure |
|---|---|
| semaphore | object |
| key | event_info |
| nsems | event_info |
| semflg | event_info |
| return value | event_info |

The **semaphore** attribute describes the semaphore obtained by the call of *semget*. The **key** attribute records the value supplied in the *key* argument, the **nsems** attribute records the value supplied in the *nsems* argument and the **semflg** attribute records the value supplied in the *semflg* argument.

This event tracks the introduction of new file system objects.

### 6.2.25 AET_SET_PASSWORD_AGING

| Attribute | Storage Structure |
|---|---|
| user name | event_info |
| return value | event_info |

The **user name** attribute contains the name referenced by the *username* argument. If the *username* argument contains an invalid pointer, the **user name** attribute is assigned to NULL.

### 6.2.26 AET_SET_PROCESS_AUDIT_ID

| Attribute | Storage Structure |
|---|---|
| audit id | event_info |
| return value | event_info |

The **audit id** attribute contains the *audit_id* argument.

For this type of event, the **Audit ID** recorded in the *subject* field of the **header** will be the value for the current process before the function call.

This event tracks changes in the auditability of the user.

### 6.2.27 AET_SET_PROCESS_AUDIT_EVENTS

| Attribute | Storage Structure |
|---|---|
| audit state | event_info |
| events | event_info |
| return value | event_info |

The **audit state** attribute records the *audit_state* argument. The **events** attribute records the list of events referenced by the *event_ptr* argument.

If AET_SET_PROCESS_AUDIT_EVENTS is one of the events for which this function is called, the call will be audited regardless of whether the *audit_state* is on or off.

This event tracks changes in the auditability of the user.

### 6.2.28 AET_SET_USER_AUDIT_EVENTS

| Attribute | Storage Structure |
|---|---|
| audit state | event_info |
| events | event_info |
| users | event_info |
| return value | event_info |

The **audit state** attribute records the *audit_state* argument. The **events** attribute records the list of events referenced by the *event_ptr* argument. The **users** attribute records the users referenced by the *user_names* argument.

This event tracks changes in the auditability of the user.

### 6.2.29 AET_SETGID

| Attribute | Storage Structure |
|---|---|
| gid | event_info |
| return value | event_info |

The **gid** attribute records the *gid* argument.

This event tracks changes in access control.

### 6.2.30 AET_SETUID

| Attribute | Storage Structure |
|---|---|
| uid | event_info |
| return value | event_info |

The **uid** attribute records the *uid* argument.

This event tracks changes in access control.

### 6.2.31 AET_SHMCTL

| Attribute | Storage Structure |
|---|---|
| shared memory | object |
| cmd | event_info |
| buf | event_info |
| return value | event_info |

The **shared memory** attribute describes the shared memory identified by the *shmid* argument. The **cmd** attribute records the value supplied in the *cmd* argument, and the **buf** attribute records the value supplied in the *buf* argument.

This event tracks the introduction of new file system objects.

### 6.2.32 AET_SHMGET

| Attribute | Storage Structure |
|---|---|
| shared memory | object |
| key | event_info |
| size | event_info |
| shmflg | event_info |
| return value | event_info |

The **shared memory** attribute describes the shared memory obtained by the call of *shmget*. The **key** attribute records the value supplied in the *key* argument, the **size** attribute records the *size* argument, and the **shmflg** attribute records the value supplied in the *shmflg* argument.

This event tracks the introduction of new file system objects.

### 6.2.33 AET_SWITCH_USER

| Attribute | Storage Structure |
|---|---|
| user name | event_info |
| process id | event_info |
| failure indicator | event_info |

The **user name** attribute records the username of the *switch user* operation.

For a successful *switch user* operation the value of **process id** is the process id of the process that runs the ''program to use as shell'' (see <**pwd.h**> in the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interface and Headers**). The field **failure indicator** is assigned to NULL in this case.

For an unsuccessful *switch user* operation, a human-readable message may be recorded in the **failure indicator** attribute indicating the reason for failure. This would typically be the message returned by the *switch user* operation to the user. The field **process id** is assigned to NULL in this case.

This event tracks changes in user identity and access control.

### 6.2.34 AET_UNLINK

| Attribute | Storage Structure |
|---|---|
| file pathname | object |
| return value | event_info |

The **file pathname** attribute describes the path referenced by the *path* argument.

This event tracks deletion of an entry in the file systems name space.

### 6.2.35 AET_UPDATE_AUDIT_EVENTS

| Attribute | Storage Structure |
|---|---|
| users | event_info |
| return value | event_info |

The **users** attribute records the users referenced by the *user_names* argument.

## 6.3 AUDIT EVENT CLASSES

The audit event class provides a mechanism for associating a single name with a set of audit event types. The class names provide an auditor with a simple means of specifying audit criteria for selection or analysis without requiring a detailed knowledge of audit event types. It also provides a measure of portability. For example, consider the AET_OBJECT_CREATION class which contains events pertaining to object creation. While the events which constitute the class are likely to vary across different machines, (e.g., IPC not available) the class itself is a constant. As such, AET_OBJECT_CREATION can be selected on all machines regardless of the set of types each individual system supports. An additional benefit of the audit class mechanism is the ability to change the grouping of types within a given class or to add new classes. Since the grouping of events into classes is inherently arbitrary, this provides valuable functionality. It should be noted that classes which work well for specifying selection criteria do not necessarily map well to specifying analysis criteria. Thus again proving the need for a flexible audit class mechanism.

This section defines the audit event classes that will be supported by conforming systems. Our goal in defining the event classes listed below was to provide guidance as to which sets of event types should be logically grouped together to provide useful audit information for both pre- and post-processing.

An implementation may extend the list of event classes, or may define additional event types within one of the standard classes. Note, however, that numeric values assigned to event class and event type constants must not overlap.

### 6.3.1 Summary of Event Classes

The following are audit event classes which must be supported by a conforming system. Each event class is identified by a symbolic constant in <**audit.h** >.

| Event Class | Description |
|---|---|
| AEC_ACCESS_CHANGE | Change Access |
| AEC_ACCESS_DENIALS | Deny Access |
| AEC_ADMIN_OPERATOR | Administrator / Operator Actions |
| AEC_AUTHENTICATION | System Access |
| AEC_OBJECT_AVAILABLE | Make Object Available |
| AEC_OBJECT_CREATION | Create Object |
| AEC_OBJECT_DELETION | Delete Object |
| AEC_OBJECT_MODIFICATION | Modify Object |
| AEC_OBJECT_TO_SUBJECT | Map Object to Subject |
| AEC_OBJECT_UNAVAILABLE | Make Object Unavailable |
| AEC_PRIVILEGE | Use of Privilege |
| AEC_PROCESS | Process Creation and Deletion |
| AEC_PROCESS_CONTROL | Process Control |
| AEC_RESOURCE_DENIALS | Deny Resource Usage |
| AEC_SYSTEM | System Related |

The remaining sections of this chapter define which audit event types are assigned to which event class.

### 6.3.2 AEC_ACCESS_CHANGE

This event class includes audit event types that bring about changes to the access of an object.

| Event Type |
|---|
| AET_CHMOD |
| AET_CHOWN |
| AET_MSGCTL (where *cmd* is IPC_SET) |
| AET_SEMCTL (where *cmd* is IPC_SET) |
| AET_SHMCTL (where *cmd* is IPC_SET) |

### 6.3.3 AEC_ACCESS_DENIALS

This event class includes operations that have failed either because access to an object is denied, or because an object does not exist. The following list of audit event types are included for failures with *errno* values of EACCES, EISDIR, ENOENT, ENXIO, ENOTDIR, EPERM, EROFS and ETXTBSY.

| Event Type |
|---|
| AET_AUDIT_SWITCH |
| AET_CHDIR |
| AET_CHMOD |
| AET_CHOWN |
| AET_CHROOT |
| AET_CREAT |
| AET_EXEC |
| AET_EXECE |
| AET_KILL |
| AET_LINK |
| AET_MKDIR |
| AET_MKFIFO |
| AET_MSGCTL |
| AET_MSGGET |
| AET_OPEN |
| AET_RENAME |
| AET_RMDIR |
| AET_SECURE_PUT_PASSWD_USER |
| AET_SEMCTL |
| AET_SEMGET |
| AET_SET_PASSWORD_AGING |
| AET_SET_PROCESS_AUDIT_ID |
| AET_SET_PROCESS_AUDIT_EVENTS |
| AET_SET_USER_AUDIT_EVENTS |
| AET_SETGID |
| AET_SETUID |
| AET_SHMCTL |
| AET_SHMGET |
| AET_UNLINK |
| AET_UPDATE_AUDIT_EVENTS |

### 6.3.4 AEC_ADMIN_OPERATOR

This event class includes actions carried out by the System Administrator or Operator. No standard audit event types are currently defined in this class.

### 6.3.5 AEC_AUTHENTICATION

This event class includes audit event types concerned with obtaining access to the system.

| Event Type |
|---|
| AET_LOGIN_USER |
| AET_LOGOUT_USER |
| AET_SECURE_PUT_PASSWD_USER |
| AET_SET_PASSWORD_AGING |
| AET_SWITCH_USER |

### 6.3.6 AEC_OBJECT_AVAILABLE

This event class includes audit event types concerned with making objects available to a program. The objects are usually files.

| Event Type |
|---|
| AET_CREAT |
| AET_MSGGET |
| AET_OPEN |
| AET_SEMGET |
| AET_SHMGET |

### 6.3.7 AEC_OBJECT_CREATION

This event class includes audit event types that create objects. The objects are usually files.

| Event Type |
|---|
| AET_CREAT (where file is created) |
| AET_LINK |
| AET_MKDIR |
| AET_MKFIFO |
| AET_MSGGET (if message queue created) |
| AET_OPEN (where file is created) |
| AET_RENAME |
| AET_SEMGET (if semaphore created) |
| AET_SHMGET (if shared memory created) |

### 6.3.8 AEC_OBJECT_DELETION

This event class includes audit event types that delete objects. The objects are usually files.

| Event Type |
|---|
| AET_MSGCTL (if *cmd* is IPC_RMID) |
| AET_RMDIR |
| AET_SEMCTL (if *cmd* is IPC_RMID) |
| AET_SHMGET (if *cmd* is IPC_RMID) |
| AET_UNLINK |

### 6.3.9 AEC_OBJECT_MODIFICATION

This event class includes audit event types that cause objects to be modified. The objects are usually files.

| Event Type |
|---|
| AET_CHDIR |
| AET_CHROOT |

### 6.3.10 AEC_OBJECT_TO_SUBJECT

This event class includes audit event types that map objects to a subject.

| Event Type |
|---|
| AET_EXEC |
| AET_EXECE |

### 6.3.11 AEC_OBJECT_UNAVAILABLE

This event class includes actions which result in objects becoming unavailable. No standard audit event types are currently defined in this class.

### 6.3.12 AEC_PRIVILEGE

This class includes operations that succeed because the process has special privileges.

| Event Type |
|---|
| AET_AUDIT_SWITCH |
| AET_CHDIR |
| AET_CHMOD |
| AET_CHOWN |
| AET_CHROOT |
| AET_CREAT |
| AET_EXEC |
| AET_EXECE |
| AET_KILL |
| AET_LINK |
| AET_MKDIR |
| AET_MKFIFO |
| AET_MSGCTL |
| AET_MSGGET |
| AET_OPEN |
| AET_RENAME |
| AET_RMDIR |
| AET_SECURE_PUT_PASSWD_USER |
| AET_SEMCTL |
| AET_SEMGET |
| AET_SET_PASSWORD_AGING |
| AET_SET_PROCESS_AUDIT_ID |
| AET_SET_PROCESS_AUDIT_EVENTS |
| AET_SET_USER_AUDIT_EVENTS |
| AET_SETGID |
| AET_SETUID |
| AET_SHMCTL |
| AET_SHMGET |
| AET_UNLINK |
| AET_UPDATE_AUDIT_EVENTS |

### 6.3.13 AEC_PROCESS

This event class includes audit event types concerned with the creation and removal of processes.

| Event Type |
|---|
| AET_EXIT |
| AET_FORK |
| AET_KILL |

### 6.3.14 AEC_PROCESS_CONTROL

This event class includes audit event types that change the characteristics of a process.

| Event Type |
|---|
| AET_SET_PROCESS_AUDIT_ID |
| AET_SET_PROCESS_AUDIT_EVENTS |
| AET_SETGID |
| AET_SETUID |

### 6.3.15 AEC_RESOURCE_DENIALS

This event class includes audit event types that fail because the system limit has been reached on some resource. Typical failure values of *errno* are E2BIG, EMFILE, EMLINK, ENOSPC and ENFILE.

| Event Type |
|---|
| AET_CREAT |
| AET_EXEC |
| AET_EXECE |
| AET_FORK |
| AET_LINK |
| AET_MKDIR |
| AET_MKFIFO |
| AET_MSGGET |
| AET_OPEN |
| AET_RENAME |
| AET_SEMGET |
| AET_SHMGET |

### 6.3.16 AEC_SYSTEM

This event class includes audit event types that have a system wide effect. In addition to the event types defined below, this class is also likely to record such items as system startup and shutdown.

| Event Type |
|---|
| AET_AUDIT_SWITCH |
| AET_SET_USER_AUDIT_EVENTS |
| AET_UPDATE_AUDIT_EVENTS |

*Appendix A*

# *Rationale*

## A.1 INTRODUCTION

This appendix contains a rationale for the X/Open Security Specifications currently being defined by the X/Open Security Working Group (SWG). To ease cross-referencing, the section numbering in this appendix corresponds to the equivalent chapter numbers in the main body of the document. For example, **Section B.2**, **Overview** is the rationale for **Chapter 2**, **Overview** in the main document. The rationale records the dates on which decisions were made by the group, and the reasons for those decisions. This document contains a rationale for the XSI Security Specifications

### A.1.1 Document Cross-References

1. Trusted Computer System Evaluation Criteria US DoD (CSC-STD-001-83)
2. The Design of an Effective Auditing Subsystem (J.Picciotto)
3. A Guide to Understanding Audit in Trusted Systems (NCSC-TG-001)
4. Guidelines for Audit Log Mechanisms in Secure Computer Systems (R.L.Brown)
5. X/Open Portability Guide - Issue 3
6. Security Auditing for X/Open Systems (SWG)
7. Minimal Changes for the X/Open C2 Security Option (SWG)
8. P1003.6 Audit Record Format Proposal: Summary of Goals and Proposals
9. P1003.6 Proposal: Audit Record Format
10. P1003.6 Proposal: Audit Record Interfaces
11. P1003.6 Proposal: Audit Selection Class Design
12. Proposal for Audit Event Types and Audit Event Classes

## A.2 OVERVIEW

### A.2.1 General

#### TCSEC C2 Criterion

There is consensus that TCSEC C2 should form the basis for the X/Open audit work. This satisfies the technical managers and others who take a high level view that the C2 level is appropriate. It satisfies the US companies who need to have products conforming to the DoD criteria. And it is agreed by the European members of the SWG that C2 is at least a subset of the European and commercial requirements.

#### X/Open Criteria

It is implicit in the X/Open discussions that selectivity of collection is required, though selectivity of analysis would be sufficient to satisfy TCSEC. The requirement is based on the available data showing that audit trail files can grow very quickly, and thus that there must be ways of controlling the data collection to make the audit system usable.

The SWG decided at the October 1987 meeting that selectivity by event type was required, to allow the audit administrator to define that he is interested in only a subset of the possible auditable events on the system. The June 1988 meeting decided to make explicit the requirement that events should be selectable at per-user granularity; also, that for usability it should be possible to set a background level to be used for all users, including new users.

Selectivity by object identity has also been debated. This would further enhance the audit administrator's ability to reduce the volume of data collected, since he would be able to specify that only accesses to security critical objects was to be audited. The NCSC audit guideline [3] specifies that object selectivity is desirable at C2; it is not a TCSEC requirement. Balancing the gain in market acceptability provided by object level selectivity, against the implementation cost of providing it, the SWG has decided that it is not to be included in the current standard.

#### Beyond TCSEC C2

Upward compatibility of the proposed C2 audit standard to more secure systems is desirable because this will further increase portability of applications and increase acceptability as the basis of a POSIX standard.

### A.2.2 Security Auditing

This section provides a high-level statement of requirements for the auditing interfaces. These are of two main types.

Firstly, there are requirements imposed by the stated aim of the X/Open group, to ''increase the volume of applications available [on X/Open systems] ... by ensuring portability of application programs at the source code level'' [5]. These requirements lead to a definition of the types of interface that are to be standardised, in terms of the applications that will use them. Originally, two types of applications were identified as benefitting from standard interfaces. These were audit reduction utilities and applications trusted to audit themselves. At the October 1987 meeting the SWG added a

third type: general audit control packages. This is a departure from other X/Open standards, which do not normally standardise administrative interfaces. The SWG felt that security in X/Open systems would be significantly enhanced by use of good security administration packages. Production of audit administration packages should be encouraged by standardisation of audit control interfaces.

Secondly, there are technical requirements. These come from two sources:

- The technical managers have requested work on X/Open interfaces to allow them to conform to the TCSEC C2 level [1], and
- Members of the SWG are aware of some requirements in the markets to be addressed by secure X/Open systems. These requirements lead to both extra facilities and upward compatibility to more secure systems than those currently being addressed by the group.

### A.2.3 Audit Record Format

The audit record format is intended to promote application portability for two classes of trusted applications; programs that generate audit records and programs that analyse audit records.

#### Interoperability Goals

The goal of interoperability is that applications using the audit system are portable, and that audit data itself is portable among different machines, so that generation and analysis need not be tightly coupled.

One expected class of portable applications is programs that generate audit records, i.e., portable trusted applications. Another class is programs that analyse audit data. It is a major goal of the interface definition that an audit-generating program running on one machine can generate audit data which is then stored on a second machine and finally analysed by tools on a third machine, all without explicit format conversions.

It is also a goal that analysis tools operate to some degree independently of each other; that is, one analysis tool can operate on the output from another without either tool requiring a complete understanding of the records on which it is operating. For instance, a general-purpose selection tool could be used to select records generated by a trusted application and then pass them on to an analysis tool that is part of the application. The selection tool need know nothing about the application, except the event types of its records (which will be specified by the auditor performing the analysis). The analysis tool need know nothing about record selection or other types of records, but only about interpreting records produced by the trusted application.

#### Efficiency Goals

Efficiency of the auditing mechanism is very important. A system that spends all its resources auditing is obviously not a usable system.

Efficiency concerns arise in three major areas: time spent generating audit records, space used to store them, and time spent interpreting them. The proposed interfaces and data structures have been designed to reduce these costs while still retaining a programming model which meets the goal of permitting interoperable audit analysis tools. Consequently, this is not the most efficient interface imaginable; merely the most efficient

portable interface proposed so far.

One trade-off is caused by using XDR and NDR as the format of the portable audit trail. These data representation standards are not as efficient as possible, but are generally available and already understood.

Another trade-off exists in the granularity of information in the audit record. An audit record consisting of individual attributes is the more general interface but is also more inefficient. Structure-based interfaces which put and get information in large chunks are more familiar to programmers, but it may be more difficult to validate the attributes. The structure-based approach was selected to ensure a manageable number of interfaces. Initially, audit records consisted of individual tokens, but this lead to an excessive number of token types and interfaces to manipulate each token type. There was also the possibility of inconsistent use of the tokens by applications performing their own auditing.

A third trade-off is caused by offering only indirect access to the audit record, since the information must be retrieved procedurally. But the cost is minimal if these interfaces are implemented as macros and a procedural interface allows an implementation greater flexibility in defining audit trail storage and access methods.

Although this standard may cause an overall increase in the size of the system audit trail, the storage costs can be reduced through the use of standard file compression techniques.

#### Non-Goals

The interface does not address audit data storage. It is expected that each conforming implementation may have a different form of permanent storage for audit data. While some implementations may store it in files, allowing the proposed interfaces to be used directly, all that is required is that an implementation provides a program that delivers audit records on standard output so that they may be redirected into a portable analysis tool.

The interface does not address the actual mechanism for delivering audit records from a trusted application (or from the operating system itself) to a system's audit trail. However, the interfaces that an application (or the operating system) would use to perform the delivery are specified.

### A.2.4 XSI Changes and Extensions

Changes to the XSI for C2 assurance and functionality are provided in accordance with software engineering principles for object-oriented programming. This means that *information hiding* is enforced, and interfaces are provided for the handling of abstract data types.

For example, although the concept of audit IDs is introduced, there is no mechanism defined for accessing the audit ID of a process other than through a set of standard interface functions.

A number of interfaces published in the **X/Open Portability Guide**, **Issue 1** and the **X/Open Portability Guide**, **Issue 2** were determined to be affected by security, i.e., the *at* and *crontab* commands, and the *exec*(), *fork*() and *sysconf*() functions. The *sysconf*() function requires modification so that applications can interrogate system variables associated with the security option (this was agreed at the February 1989 meeting). Other

interfaces have been updated to handle process audit IDs correctly.

### A.2.5 Configuration

It was originally intended that XSI Security would be presented as a single implementation option in the **X/Open Portability Guide**. A number of delegates objected to this approach on the grounds that accountability and authentication, while both related to security, provide quite different capabilities. Further concerns were expressed that mandatory inclusion of authentication mechanisms in the Security Option might prompt some companies to vote against the option as a whole.

Given these concerns, the SWG decided at the February 1989 meeting to separate the security interface into two options: ACCOUNTABILITY and AUTHENTICATION. These would be separately configurable, their availability in a particular implementation being determined by calling the *sysconf*() function for the value of the associated system variable.

Tables were thus included in this section of the interface definition to identify which functions belong to what option.

## A.3 FUNCTION AND INTERFACE

### A.3.1 Security Auditing

Audit IDs are introduced to meet the requirement for individual accountability. In existing X/Open systems, the user has a username and a user ID. Neither of these is appropriate for use as the audit ID. The name is inconvenient to handle. The user ID is the basis of the authorisation policy of the system, which is logically distinct from the accountability policy. In particular, on some systems the system administrator allows aliasing of one user ID to several usernames which all have the same authorisations; this is incompatible with use of the user ID as an audit ID. The decision to have logically separate user ID and audit ID values for each user was taken at the October 1987 meeting. The June 1988 meeting decided to define the type **audit_ID_t**, since only by doing this could an audit file be analysed on systems of different type to the one on which it was generated. Some implementations might wish to define mappings between **audit_ID_t** values and implementation-defined identifiers, such as personnel numbers; this is not subject to standardisation.

Note that there is nothing to stop a particular implementation from implementing user ID and audit ID for each user as the same value, as long as it maintains individual accountability. However, confusion might arise from the existence of two sets of interfaces to the same value.

There is an issue connected with *su* to another username. Who should be recorded as performing the actions, subsequent to the *su*, from that terminal? If the username is a role, such as root, it is clear that accountability must still be to the logged in user. However, on *su* to another individual username, it is clear that the owner of the target user name must have done the *su*, because the *su* has to be authenticated by the password of the target user. Either the login user or the *su* user could be accountable for subsequent actions. Within the *su* session, certain commands still appear to come from the outer user (e.g., *who*, *mail*). It would be possible for either user to plant Trojan Horse programs to achieve undesirable effects within the *su* session. For example, the outer user could plant an *ls* Trojan Horse which created a writable suid file in the *su* user's name; and the *su* user could use a Trojan Horse to mail others with the apparent authorship of the outer user.

The SWG has decided the accountability policy should be that a user is accountable for everything that occurs from his login to his logout. Therefore audit must still use the login user's audit ID even after an *su*. However, the *su* user's identity is also generally available, because it is required that *su* (an authentication mechanism) is audited; the *su* audit record should contain the new username, to uniquely identify the user. Therefore in practice both identities should be available.

Note that this accountability policy extends to *at*, *batch* and *crontab* jobs fired off within an *su* session: accountability should be to the outer user, even though they run with permissions of the *su* user. This allows detection of jobs fired off by Trojan Horse programs within an *su* session.

It was pointed out at the June 1988 meeting that the arguments for unchanged accountability on *su* also apply to remote login. Although X/Open does not currently define *rlogin*(1), it would be sensible for secure systems that support *rlogin* to keep the

same accountability within the outer and inner sessions. The implications of this are beyond the scope of this document.

Because we have defined that accountability does not change after the start of a session, it is possible to increase the strength of the individual accountability by specifying that a call to the *set_process_audit_ID*() function can be used only once within a process (and its children). Thus, even if a user manages to gain superuser privileges he will not be able to change his audit ID.

If a user logs into a ''role'' username, such as root or uucp, accountability is lost; there is no identification of the individual. Secure X/Open systems must be able to detect when a user is logging in to a role username, and insist on individual accountability. Since accountability must be to individuals, secure X/Open systems should provide a way to force login to a user's account to provide individual accountability.

Note that there is no requirement that an implementation supports login to role usernames, only that accountability should be provided if it is supported. In particular, systems may require that *su*(1) is used to enter role usernames, and this does provide accountability.

There is a problem in auditing daemons: what audit ID should they have? They are not (usually) fired off by either *cron* or *getty/login*, but by a shell interpreting **/etc/rc**. The SWG agreed at the January 1988 meeting that this is not an area for standardisation, since it is of no relevance to the interfaces used by the defined types of portable applications.

**Audit Identifier Interfaces**

The set of interfaces suggested here includes the basic operations of setting and determining the audit ID for each user and for the current process, and for getting the username corresponding to a known audit ID. Details of the actual implementation of audit IDs, beyond the fact that they are arithmetic types, are concealed. Following the decision to define audit IDs to this extent, a proposed *compare_audit_ID*( ) interface was removed; there is no need to provide a special interface to compare two arithmetic values for equality.

Also at the June 1988 meeting, the *set_process_audit_ID*() interface was changed to *set_proc_audit_state*(), and defined to set both the audit ID and the corresponding events for the current process. However, this decision was reversed at the August 1988 meeting, on the grounds that the separate interfaces allowed for greater flexibility of audit policy.

Note that the *get_process_audit_ID* interface is defined to require appropriate privilege. On particular implementations this privilege may be defined to be null: it is not clear that there are any security problems implied by a process determining its audit ID.

**Audit Reduction Interfaces**

Following the April 1988 meeting, an attempt was made to define a set of interfaces for use in analysing abstract audit trails, concealing the storage method, location and format of the actual data. However, this resulted in a need to re-invent a complete IO package for such objects. Also, it did not succeed in defining any particularly useful interfaces, other than a record-oriented read function. At the June 1988 meeting, it was decided that it would be better to treat audit trails as if they are always stored in files, thus allowing

use of the standard IO packages. In fact this still does not imply that the trail actually has to be held in a file; a suitable filter can read any local form of trail and output the data to a file or pipe, thus ensuring that the data is always available through a file interface whatever its original form. Such a filter might also be a privileged program which gave selected views of the audit trail to users not privileged to see all of it, if this was permitted by the audit policy on the system. The useful record-oriented read interface was retained, operating on file descriptors. Extra functions were added to allow reading of the next record containing a given subrecord type, as well as the next sequential record.

- Reading the Audit Trail

  Initially it was envisaged that the interface for reading audit trails would need to distinguish between audit headers, records and trailers. Three functions were proposed:

  *read_audit_header*()
  *read_audit_record*()
  *read_audit_trailer*()

  However, it later became apparent that separate header and trailer records were not necessary, and that instead each audit record would contain a header and optional trailer token identifying the associated auditable event.

  At the February 1989 meeting it was agreed to replace the above interfaces with a single read function *aur_read*(). In addition, the functions *aur_next_match*() and *aur_next_type*() were included to permit searching forward in an audit trail for an audit record that contains a matching token or matching token type.

  At the August 1989 and October 1989 meetings, the token format was replaced by a structure format and the functions above were replaced by a single function, *aud_next*(). The function *aud_next*() was added to allow general searching of the audit trail. It returns a record matching the current search criteria and can also be used to alter those criteria. Using just one function simplifies the interface considerably. The decision to require system support for more general search predicates was made after considerable debate. This requirement places a considerable burden on systems (parsing SQL where clauses), but it was felt that many applications could make use of this capability.

  A record-oriented interface is provided both because this is probably more convenient for reduction utilities than a byte interface, and because it may allow an implementation to optimise the conversion of records from an internal format to the X/Open standard audit record format.

  Also at the February 1989 meeting, it was agreed to change the names of the functions in this section to align with the naming conventions proposed in [8]. New functions were added to extract individual fields from the record header (*aur_length*(), *aur_event_class*(), *aur_event_type*(), *aur_result*() and *aur_time()),* and the *aur_print*() function was included for the first time.

  At the May 1989 meeting it was decided that event classes should not be included in header tokens, as it was felt that they could not be guaranteed to provide a complete reason for an audit record. The functions *aur_event_class*(), *aut_set_header_class*() and

*aut_header_class*() were also removed from the interface definition.

At the April 1990 meeting, the record storage was changed to align with POSIX (P1003.6) and so the above interfaces were dropped, since the system now manages all storage associated with the record. The *aud_discard*() function was modified to apply to records read by the *aud_next*() function so that applications could explicitly free all storage associated with records they were no longer using.

The function *aud_print*() was also extended to permit more general translations of the audit record at this meeting. Among other things, the new translations allow the elimination of specific interfaces for appending records to file descriptors and for generating portable data.

- Manipulating Audit Records

  Previously, with the token interfaces, discrete functions were provided to extract tokens from records and then to operate on tokens. With data presented as structures, the former interfaces have been reduced to three interfaces, while the latter interfaces have been eliminated. The three new interfaces are: *aud_get_header*( ) which returns the header information, *aud_get_object*( ) which returns successive objects from the record, and *aud_get_event_info*( ) which returns successive event-specific data items from the record.

  Note that these interfaces operate on audit record descriptors as returned by *aud_next*() and *aud_start*(). The decision to use symmetric interfaces allows applications greater latitude in processing a record and allows the implementation to be considerably simplified since separate writing functions are not needed for records which are read from the trail as opposed to those that are created from scratch.

**Trusted Application Interfaces**

- Writing the Audit Trail

  Some utilities are expected to have their own audit requirements, so there is a need for a set of interfaces that add a record to the audit trail. However, these interfaces cannot be generally available, since it would provide a malicious user with a means of denying service to other users (by filling up the audit file or perhaps more subtly by seeding the audit trail with disinformation). Accordingly, utilities that access this interface must have appropriate privilege and be trusted to use it properly.

  The SWG decided at the January 1988 meeting that the parameter to an *audit_write*( ) function should be a single pointer to a structure giving, in some form, the details of the data to be recorded. It was felt by the group that a single parameter was neater than multiple parameters giving the fields from the structure.

  The April 1988 meeting heard an audit record format proposal from Sun, and decided to use this as the basis for the X/Open audit record standard. Also, it decided that it would be an advantage to increase the system's understanding of the data audited by allowing the process to produce formatted audit records. Therefore they decided that the data passed to the *audit_write*( ) routine should be in the form of audit subrecords.

  Further debate at that meeting on the format of the parameter to *audit_write( )* was superseded by changes at the June 1988 meeting. This meeting decided to adopt a proposal that there should be an interface to allow suitably privileged users to build

up audit records without having to be concerned with the details of their storage. The interface to allow writing of a collection of subrecords was also to be retained.

This strategy is expanded in [10], with the difference that separate functions are defined for creating an audit record buffer (e.g., *aur_start*()), for adding tokens to the buffer (e.g., *aur_add_***token**()), and for writing a completed buffer to the audit trail (*aur_finish*(), *aur_commit*(), etc.). The SWG accepted this proposal at the February 1989 meeting.

In adding these functions to the interface definition, it quickly became apparent that a two-stage write mechanism (i.e., **finish** and **commit**) brought with it certain problems. For example, what was to stop an application reassigning fields in a finished audit record buffer before it had been committed to the audit trail? As a result of this and other concerns, the SWG decided at the May 1989 meeting that finishing (i.e., completing the header token and optionally appending a trailer token) and committing an audit record should be done in a single operation. Thus the functions *aur_commit*(), *aur_commit_save*(), *aur_write_fd*() and *aur_get_record*() were redefined to finish an audit record, and the *aur_finish*() function was deleted.

When the data representation was changed from tokens to structures the interfaces were correspondingly changed. The *aud_start*() function is now the only interface to begin an audit record and it additionally defines the event type for the record.

The *aud_put_object*() and *aud_put_event_info*() functions may then be used to modify the record created by the *aud_start*() function by adding object and event-specific information respectively to the record.

The function *aud_commit*() is the only function for committing an audit record to the trail. As a side effect, it also discards all storage allocated as part of generating the record. Unfinished records may also be discarded with the *aud_discard*() function.

These changes have the side benefit of simplifying the interfaces for writing a record. The SWG felt that making the interfaces more programmable will make it more likely that they will be used, thus facilitating the goal of providing greater security in X/Open systems.

- Auditing Suspension and Resumption

  Any process doing its own auditing may wish to suspend standard auditing in order to save space in the audit trail. This is likely to be used mainly by processes auditing themselves at a much coarser granularity than the kernel. For example, a program that scans the filestore periodically and moves to tape files that have been unused for a long time could audit the movement of the files itself (in a more meaningful way than the kernel); it would seem unnecessary to record that it had checked the access dates of all the other files in the system, which would merely clutter the audit trail with data. Even standard utilities (with appropriate privilege) might make use of this facility to provide a higher level view of events than would be given by the kernel.

  The June 1988 meeting decided that it would also be useful to have an interface that would allow a suitable, privileged process to adjust TCB auditing of its operations at the granularity of single events (or groups of events). This interface is added as part of the later section, on event selectivity.

#### Audit Control Interfaces

It is reported that the NCSC say that use of audit control interfaces must always be audited.

### Select Users

TCSEC requires selectivity by user at C2, though it is not specified whether the selection is to be performed at audit trail generation or reduction. The SWG decided that it is desirable to permit selection at audit trail generation. This will permit the audit administrator to have better control of the size and relevance of the audit trail.

### Select Events

TCSEC does not require audit to be selective by event type. The SWG decided that this is a desirable addition for the Security Interface Specifications, because it provides a potential reduction in the amount of audit data collected unnecessarily.

### Specifying Event Lists to be Audited

The meeting in April 1988 heard a proposal for control of auditable events on a user and system wide basis. This was broadly based on the syntax used in the Sun C2 system, but was generalised to meet some additional requirements added at the January 1988 meeting. Although this interface provided flexible facilities, it was thought to be too complex. A new proposal was included for the June 1988 meeting. This was extended at that meeting to allow a suitably privileged process to ascertain and modify its audit events; this is effectively a finer granularity form of the audit_switch interface. The interface to set the events would also be used at login to set the events specified for the user.

There are at least two granularities at which an audit administrator may wish to specify the events to be audited:

- he may wish to specify events to be audited throughout the system, or
- he may wish to specify events for each individual user of the system.

The proposed solution allows both approaches.

The SWG considered at the August 1988 meeting whether there was a requirement to provide other granularities of event selection, for example by UNIX group. Noone was aware of any requirement for this, so it is not included in the interface definition.

The audit event interface functions could use username or audit ID to identify the user. We have proposed username because that is what an administrator would supply to identify a user, and what the user specifies to identify himself at login. Thus utilities using these interfaces do not have to be concerned about the audit ID at all.

At the June 1988 meeting the SWG agreed not to standardise an interface for controlling the events to be audited for daemons, since this is not relevant to application portability.

#### Protecting the Audit Trail

Of all the data in a secure computing system, the audit trail is perhaps the one item which is most important to protect against invalid manipulations, even by apparently authorised users. For instance, if an intruder can defeat a system's access control

mechanisms, and assume all the rights and powers of an authorised system administrator, it would still be extremely useful to be able to audit the intruder's activities. To any extent possible, the auditing mechanism and the audit trail should be protected against external attacks.

The SWG considered specifying a few possible mechanisms that provide elements of protection against this threat, but decided not to. The SWG took this position because any mechanism which is sufficiently general enough (not implementation-dependent) to specify for X/Open would not, itself, provide significant protection. Only a combination of mechanisms, most of them implementation-dependent and outside the scope of X/Open's work, can protect a system's audit trail to a meaningful degree beyond basic file protection.

The principal mechanism we considered was giving the administrator the ability to specify that certain events are always audited. That is, once that list of events was turned on, they could never be turned off except by a system reboot. Possibly, these could even be configured into the system, perhaps by the manufacturer, so that not even a reboot could change the state.

For this to provide real protection, however, more is required. It must be impossible to modify the running kernel or its data structures (for instance, by writing to */dev/kmem*). It must also be impossible to modify or destroy the audit files themselves. Ordinary file protection won't do, since the intruder will be able to override that. Even if the audit files are somehow special, the raw disk device offers numerous possibilities. The only real way to protect against the audit files being removed or overwritten is to write audit messages in some ''write-only'' fashion, perhaps to a tape or optical disk, or to a different host on the network which has some additional protection against the intruder. The underlying problem is that it is extremely hard to protect against privileged intruders. The only mechanisms we can invent to help are highly implementation-dependent.

If the audit file is protected using the normal filesystem protection mechanisms, the degree of protection increases with the security of the system. Thus in a DAC-based system with a single superuser, it could be read/write to superuser only. On a system with the administrative roles divided according to the principle of least privilege, it could be owned by the audit administrator, with read access available also to the security administrator. On a system with MAC controls of disclosure and integrity, it could be owned by the audit administrator with a disclosure label making it readable only to security and audit administrators, and an integrity label making it writable only to the TCB.

Of course, these access controls do not prevent the audit subsystem itself from writing to the audit trail to record actions of users, even though the users don't have write access to the audit trail file.

### A.3.2 Audit Record Format

The logical system audit trail is a stream of audit records. That is, an audit trail appears to the application program as a file of discrete, variable length records. Each record contains a complete description of an audit event. Note that the file-like appearance of records is intentional and necessary for a scheme where records may be processed by a sequence of data filters. Also note that records are intended to be largely independent entities.

During several meetings in 1988 and early 1989, a token-based format for audit records was considered. In this format, each record consisted of a stream of audit tokens. Each audit token was a self-defining entity which could be interpreted contextually. Records were constructed by building tokens and then adding them to the record. Records were read by reading the tokens sequentially from the record and then extracting the data from the tokens.

A token-based format has the advantage that it is easy to extend the system, and possible to do so granularly. Token-based formats have the disadvantage that the programming interface is more complex than with other formats. There are possible losses in computational and storage efficiency as well. Because of these disadvantages, the token format was changed during the August 1989 and October 1989 meetings to the format described below.

#### Audit Record Contents

The statement above that audit records should be largely independent is an acknowledgement that no audit data can be completely context-independent, and an encouragement that audit records contain enough context to be meaningful for analysis in most circumstances.

Each audit record contains a header, zero or more object descriptors and zero or more event-specific data descriptors. The header defines the event and the responsible subject, and includes fields for event type, event time, event status, subject identifiers and subject security characteristics.

The object descriptor include fields for the name of the object, its data format and length and object security characteristics. Object descriptors in a record are ordered so that the relative position of a descriptor may be interpreted consistently.

A data descriptor contains only the data item and its length and format. That is, a data descriptor is event-specific information with no defined semantics and self-defined syntax.

### Semantics of Audit Event Types

The Snapshot includes a set of pre-defined events with fixed interpretations (corresponding to basic system operations defined in Volumes 1, 2 and 3 of the **X/Open Portability Guide**, **Issue 3**). Applications, however, must have a way of communicating between audit producer and consumer without interference in between.

Event types are the Achilles' heel of this grand view of interoperability. Because they are the one concretely specified aspect of otherwise unconstrained records, there must be some mechanism to ensure they do not collide. No agreement for a mechanism to achieve this objective has yet been reached.

Comments would be welcome.

#### Audit Record Data Format

The physical format of an audit record is unspecified - that is, an application may make no assumptions about the format and location of the header, object and event-specific data in the record. Logically, an audit record is an opaque data object which is referred to by a handle and accessed only by operations referencing that handle.

These operations set and query structures (C) or records (Pascal). Structures contain at least the defined members, and generally include a version number which defines the members actually included. Version stamping allows for upward compatibility. On reading a structure, the application specifies the version number of the structure expected, so that the system can know which data items to return to the application. On writing a structure, the application specifies the version number of the structure it created so that the system can know which data items the application is specifying.

While this method of grouping data is neither as general nor as extensible as the token method considered earlier, it represents a more natural programming style and may be more efficient as well.

The data types of each structure member are defined by this standard, but the size and byte-ordering of the data items may vary from system to system. That is, there is no intention that the binary data returned in these structures is directly portable from system to system.

The header structure must be (logically) included in every record. This structure may only be queried. No interface is providing for setting it in total, but certain members of the structure may be set via the parameters of the functions which create and commit audit records. Each structure contains at least the members which define the event and the subject ID. When the subject is a server, the client ID may be included as well. The security characteristics of the subject may be included according to the auditing style of the system which generated the event.

The object structures in a record are ordered so that semantics may be attached to the relative positions of object descriptors. Interfaces are provided to set and query object descriptors in the record. The system must maintain a ‘‘current object pointer’’ which will determine the object read or written by the corresponding call. While allowing the objects in a record to be explicitly addressed would be more general, the SWG did not feel that the additional complication of the interface would be justified.

The event-specific information structures in a record are also ordered, for the reasons given for object descriptors. These structures are also read and written in the same way. Note that for both the object and data descriptors, reading the corresponding structure from a record logically deletes that descriptor from the record.

Note that cursor manipulation is a problem. Implicitly there is both a read and a write cursor for object descriptors and data descriptors within a record. The current definition excludes an important function, namely that of reading an audit record, modifying some attributes and writing it out. This functionality is useful to create ‘template records’ in such cases where it is likely that lots of very similar audit records will be produced. Such a record would be located, read, and then used as a template for creating other records. In practice it is likely that this type of record would not be read from the audit trail, rather it would be defined as a template, modified as needed, and then committed. Comments would be welcome

The audit records for the event types defined herein will contain well-defined sequences of object and data descriptors. For other event types, the audit records will contain application specified object and data descriptors.

**Portable Audit Record Format**

The SWG felt that it was necessary to allow analysis tools to generate portable audit trails. Among other things, a portable audit trail allows the audit data to be analysed on systems other than the systems which generated it. In earlier versions of this document, there was an implicit requirement that the system returns the audit records in a portable format. This approach, however, imposes a performance penalty on ''localx'' applications; i.e., those analysis tools which run on the generating system.

To avoid this unnecessary penalty, the data which is returned in the structures is always in the local format. An application, however, may convert any record into a portable format. These records can then be transferred to other systems portably.

There are two costs to this approach, however. The first is that each system must be prepared to read the portable format(s) defined. The second is that these records are always translated twice; once on the generating system and once on the system used for analysis.

The portable data formats are not defined in this document. That is, the size of uids, gids, mac labels, etc., is unspecified at this point. There is a necessary trade-off in specifying these sizes. If each is defined to be at least as large as the data item on any conforming system, then the data item may not be processed efficiently on machines with smaller word size. On the other hand, defining the data item as being no larger than an ''efficient'' size implies a loss of precision. A further complication arises since standards do not, in general, specify maximum data sizes. But since the loss of precision of the second approach is unacceptable due to the nature of the data being processed, the first approach is probably preferable. Comments would be welcome.

### A.3.3 Audit Event Classes and Event Types

The distinction between event types and event classes has generated considerable controversy. Reference [9] proposes that the grouping of event types into event classes is arbitrary and may differ from one system to another. The group considered this proposal at the February 1989 meeting but rejected it in favour of [12], which suggests that event types should belong to a small, fixed set of standard event classes. [12] further proposes that the event class is recorded in a header, along with the event type, thus making it the responsibility of the auditing program to fix the relationship between the two.

This latter proposal was accepted at the February 1989 meeting. However, after further reflection, it was decided that recording the event class in a header was not tenable. If an event type belongs to many classes, but only one can be recorded in a header, then the inclusion of such a value might serve to confuse rather than clarify the reason for the audit record. It was agreed at the May 1989 meeting that the event class field should be removed from the definition of header structures.

It also turns out to be very hard to define precisely when an event deserves an event type of its own. For instance, are successful and failed **open** calls the same event type? Probably so, since they can be differentiated by the result field in the record header. (Looked at another way, that really means that the result field is part of the event type, and so they are two different types).

Are open of a file for reading, and open of a file for read/write, different event types? Though they differ only in one bit of a system call argument, they certainly ought to be

different types, because they represent very different capabilities being exercised. This example leads to a circular definition of event types; two types should be separate when it would make sense to assign them to separate classes.

Although the numbering and ordering of objects and event-specific data in the record is important, it should be noted that for failed events, some objects and data in the record for the standard event types may not be included, since this information may be missing from the function or command invocation.

During the February and May 1989 meetings the SWG took a decision as to which system calls were to be made auditable as mandatory and which as optionally. The decisions were as follows:

| **Function** | **Auditability** |
|---|---|
| access | optional |
| alarm | optional |
| chdir | mandatory |
| chmod | mandatory |
| chown | mandatory |
| chroot | mandatory |
| close | optional |
| creat | mandatory |
| dup (1) | optional |
| exec (2) | mandatory |
| exit (3) | mandatory |
| fcntl | optional |
| fork | mandatory |
| fstat (4) | optional |
| fsync | optional |
| getegid | optional |
| geteuid | optional |
| getgid | optional |
| getgroups | optional |
| getpgrp | optional |
| getpid | optional |
| getppid | optional |
| getuid | optional |
| kill | mandatory |
| link | mandatory |
| lseek | optional |
| mkdir | mandatory |
| mkfifo | mandatory |
| open | mandatory |
| pause | optional |
| pipe | optional |

| Function | Auditability |
|---|---|
| read | optional |
| rename | mandatory |
| rmdir | mandatory |
| setgid | mandatory |
| setpgid | optional |
| setsid | optional |
| setuid | mandatory |
| sigaction | optional |
| sigaddset | optional |
| sigdelset | optional |
| sigemptyset | optional |
| sigfillset | optional |
| sigismember | optional |
| signal | optional |
| sigpending | optional |
| sigprocmask | optional |
| sigsuspend | optional |
| time | optional |
| times | optional |
| ulimit | optional |
| umask | optional |
| uname | optional |
| unlink | optional |
| utime | optional |
| wait | optional |
| waitpid | optional |
| write | optional |

### A.3.4 Auditing Style

Style options seem unavoidable. In most cases, they represent a simple trade-off of record size and analysis capability, and perhaps some of those options could be eliminated (by requiring trailers, file attributes, group lists, etc.). In some cases (absolute pathnames, particularly), however, they represent a trade-off that cannot be made the same way in all systems.

#### Auditing Style Interface

The *aud_config*() provides interrogation of auditing style. The interface is necessary to support portable auditing tools.

There is no equivalent support offered for portable analysis tools that may need to behave differently, or reject certain types of queries, depending on the system's auditing style.

There is a portability problem posed by portable analysis tools; it is implicitly tied to the system on which an analysis tool is running. This means that analysing one system's audit trail on another system could malfunction if the two systems have different

auditing styles. This is actually just the tip of the iceberg. Worse problems occur if the two systems have different mappings of user ID to user name, etc.. Probably there should be some explicit dependency on an environment variable to specify non-local interpretations. It is certainly not a problem unique to auditing style. In practice, however, it is thought unlikely that it will come up very often; it is not a problem for distributed systems, since many more things will fail if a distributed system has non-homogeneous auditing styles.

**Items that can be Interrogated**

Auditing style items that can be interrogated by *aud_config*() are:

a. whether audit records contain absolute pathnames or relative pathnames to identify files;

b. whether audit record object descriptors should include MAC information, and

c. whether audit record object descriptors should include DAC information.

**Items that cannot be Interrogated**

The following items describing auditing style were originally available via the *aud_config*() interface. They were subsequently removed as they are items that cannot be influenced by an auditing application:

a. whether audit record header descriptors include subject MAC information;

b. whether audit record header descriptors include subject DAC information;

c. whether audit record header descriptors include subject Privilege information, and

d. whether audit record header descriptors include origin information.

### A.3.5 XSI Changes and Extensions

From the outset the SWG decided, for stability, that changes to standard XSI interfaces for security should be kept to an absolute minimum. Nevertheless, the security aspects of various interfaces have been considered along the way.

Following is a summary of the major discussions and conclusions of the working group in this area. For further information see [7].

**New Error Codes**

It was suggested that X/Open should define a new error response, ESECUR, to indicate violation of any of the newly introduced security requirements.

The arguments in favour of this proposal were:

i. these are new error conditions and a new error response is therefore appropriate, and

ii. these errors are logically different from existing errors.

Against this proposal, it was argued that:

i. at the higher (B1+) levels of security, the ESECUR response could provide a covert channel and, when access is denied, the appropriate response is to indicate that the

object does not exist.

The SWG eventually decided not to include any new error codes.

**Auditing**

Should there be a section in each XSI entry telling the user about the auditing done on that command or system interface?

The SWG decided not to include this information in the **X/Open Portability Guide**.

**Duplication of IDs**

How and where should the X/Open recommendations or requirements about non-duplication of IDs be recorded?

The SWG decided to leave this as a site option, accordingly nothing will be said in the **X/Open Portability Guide** on the subject.

**getpwnam() and getpwuid()**

A concern was expressed that on many existing systems, the **pwd** structure returned by these functions includes the password field. There was a suggestion that on secure systems, an implementation should be required to ensure that (for non-privileged processes) this field does not contain a valid password (encrypted or otherwise). Conversely, for privileged processes, this is one way that a process could determine the password of a user.

It was decided not to change these interfaces for a number of reasons. First, **IEEE Std 1003.1-1988** and **X/Open Portability Guide**, **Issue 3** no longer define the **pwd** structure to contain a password field. And second, the SWG decided that it was better to define procedural interfaces to password data (see *secure_get_passwd_user*() and *secure_put_passwd_user*()).

**login, passwd and su**

At one time or another the SWG considered either including the above commands in the interface definition, or modifying their description as presented elsewhere in the **X/Open Portability Guide**. The *login* and *su* commands were considered necessary to satisfy requirements for user identification and authentication, while changing *passwd* was proposed for recording password construction rules.

All three were eventually dropped for different reasons: *login* because it was felt unlikely that agreement could be reached on the definition of such a mechanism (which was one of the reasons why it hadn't been included in the **X/Open Portability Guide** in the first place); *passwd* because its inclusion made no sense if *login* was not to be included; and *su* because it had been marked WITHDRAWN in the **X/Open Portability Guide**, **Issue 3**.

The decision not to include these interfaces was taken at the December 1988 meeting.

**Commands and Utilities**

The only commands affected by the Security Interfaces are schedulers that either directly or indirectly initiate other processes on behalf of the caller. It is a security requirement that, when this happens, any child processes should inherit the audit ID and audit state of the parent.

This affects the *at*, *batch* and *crontab* commands. The *cron* command itself is not presented in the **X/Open Portability Guide**, and no-one could see any value in adding to the security sections.

**System Interfaces and Headers**

A number of system interfaces are also affected by security, primarily *exec*() and *fork*(). The descriptions of these functions must be updated to define that, on secure systems, the audit ID and audit state of the caller will be inherited by child processes and new process images respectively. This functionality must also be verifiable (which raises a whole other set of questions not yet addressed by either the working group or those responsible for the verification suite).

The SWG also decided at the February 1989 meeting that the configuration state of the various security options should be made available to applications via the *sysconf*() interface. To satisfy this requirement, additional symbolic constants for security have been defined in <**unistd.h**>, which can be passed as the *name* argument to the *sysconf*() function.

The only other change to standard interfaces affects the <**limits.h**> header. Two new system limits have been defined, {AUDIT_MAX_SIZE} and {AUDIT_REC_MAX}, giving the maximum number of elements in an event list, and the maximum number of bytes in an audit record respectively. These values can also be determined at runtime by calling *sysconf*().

### A.3.6 Passwords and Password Aging

Two sets of interfaces are defined to provide a secure interface to a generic authentication database. How this database is implemented is not defined, although it could utilise existing files such as **/etc/passwd**.

The first set of interfaces includes the functions *secure_get_passwd_user*() and *secure_put_passwd_user*(). These provide a procedural interface to encrypted passwords for processes with appropriate privileges.

The second set, *get_password_aging*() and *set_password_aging*(), enable a privileged process to set and/or read the password aging rules of a named user.

Note that these functions constitute the totality of the AUTHENTICATION option. Existing System V-based systems can implement them relatively simply by manipulating entries in the *passwd* file. The fact that they are presented as a separate option simplifies things for systems that do not currently provide equivalent facilities (and for suppliers who do not want to incur the cost of implementing them).

## A.4 COMMANDS AND UTILITIES

### A.4.1 at, batch

The requirement that jobs initiated via *at* and *batch* should inherit the audit ID and audit state of the initiating process has non-trivial implications for the way the *cron* mechanism is implemented. The *at* and *batch* commands themselves already export the name of the originating user, which the *cron* daemon uses to initialise the user ID and group ID of any background processes. This mechanism must now be extended to include the audit ID of the originating process, its audit switch state and the list of process audit events.

The audit ID is okay, as this cannot be changed dynamically even by a privileged process. The audit switch state and list of process audit events is more of a problem. Unless some direct way of exporting these values from the initiating process can be devised, they can only be set to the default for the associated user. Thus any changes to the audit state of a process will not be reflected in background tasks initiated therefrom.

There is also a security issue to be considered. Current implementations of *at* and *batch* export the login name of the caller, rather than the user name associated with the effective user ID of the initiating process. Thus background jobs can acquire extra privilege if, for example, they are initiated after having switched user to a less trusted user name. This is a known weakness of the *cron* mechanism and is not made any worse by the changes proposed in this document.

### A.4.2 crontab

The situation with the *crontab* command is a lot cleaner. The interface defines that processes initiated via *crontab* will inherit the audit ID of the user who owns the crontab file, and that the audit state of the process will be initialised to the default audit state of that user. Thus there is no ambiguity in this mechanism.

## A.5 SYSTEM INTERFACES AND HEADERS

The ERRORS sections in this chapter clearly distinguish values of *errno* that ''will'' be set by a function from those that ''may'' be set. In implementation terms, a system must support the former, but it is optional whether the latter are supported. This nomenclature is consistent with the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interface and Headers**, and is important for verification.

All security functions are declared in the <**audit.**h> header. This has been done for consistency with guidelines set down in **ANS/X3.159-1989 Programming Language C Standard**, and because it permits implementations with naming restrictions to abbreviate long interface names prior to compilation, e.g.,

```
#define set_user_audit_ID      setusrID
#define set_user_audit_events  setusrEV
etc..
```

As agreed at the December 1988 meeting, the [ENOSYS] error has been defined as mandatory for all interfaces. This permits null implementations of either or both the ACCOUNTABILITY and the AUTHENTICATION options.

### A.5.1 aud_commit()

Originally, several interfaces were defined to allow a trusted application to dispose of the audit record. These have been simplified to two: *aud_commit*() and *aud_discard*(). The first function commits a finished record to the audit trail, discards the record and frees all associated storage. *aud_discard*() discards the record and frees all associated storage.

It is felt desirable, in the future, to specify changes in this area which allow a single audit buffer to be used to create several successive records pertaining to the same event type, but each with somewhat different attributes. In this way, the record would be used as a template in which attributes may be modified. Comments would be welcome.

*aud_commit*() allows an application to specify a client's audit ID in order to facilitate the creation of audit records by server processes; programs which act on the behalf of other processes. In this case, the subject's audit ID would be that of the server, which would not allow for true accountability. The third argument to *aud_commit*() allows the application to specify the status of the audit event, so that successful events may be distinguished from failures, and that different types of security relevant failures may be noted.

### A.5.2 aud_config()

This function provides an interrogative interface to the current settings of the system's auditing style options. An interface to set these options is not defined, as this functionality is considered to be implementation-dependent and therefore outside the scope of standardisation. For a complete rationale see **Section B.3.4**, **Auditing Style**.

### A.5.3 aud_discard()

This function allows an application to discard audit records and their associated storage. It may be used by both audit trail writers and audit trail readers. Audit trail writers will use it to discard records either in situations where *aud_commit*() fails or where an application calls *aud_start*() and does not call *aud_commit*(). Audit trail readers will use it to discard records which either are not interesting or for which the analysis has been

completed.

This function allows an application to manage storage since the amount of memory available to satisfy calls to the *aud_next*() and *aud_start*() functions is limited. Since the system may not discard records created by these calls, the application should free any memory which it is no longer using.

### A.5.4 aud_get_header()

This function was added at the August 1989 meeting to allow an application to retrieve the header information from the audit record. This information is returned as a structure to allow applications to access this information in standard fashion. Note that the fields which are in the returned structure are defined implicitly by the *version* field. Also, the header parameter was changed so that the caller provides a pointer to a pointer. This allows the system to do the memory allocation rather than the caller.

### A.5.5 aud_get_object()

This function was added at the August 1989 meeting to allow an application to retrieve the object information from the audit record. As with *aud_get_header*(), this information is returned as a structure to allow applications to access this information in standard fashion. And again, the fields which are in the returned structure are defined implicitly by the *version* field and storage management is done by the system.

Note that cursor manipulation is a problem. Implicitly there is both a read and a write cursor for object descriptors within a record. The current definition excludes an important function, namely that of reading an audit record, modifying some attributes and writing it out. Comments would be welcome.

The *type* field values allow for non-standard types of objects to be defined with the manifest constants AUD_OBJ_STOR and AUD_OBJ_IPC. The defined structure allows the explicit specification of the name, its format and its length, so that objects in separate naming spaces may be correctly and unambiguously named. The *mode* field in the structure allows an application to specify how an object was accessed. The function always returns the description of the 'current' object. The system must maintain this pointer and 'increment' it at each successive call to this function. Objects are retrieved in the order in which they were written so that any semantic differences between object inherent in this ordering will be preserved.

### A.5.6 aud_get_event_info()

This function was added at the August 1989 meeting to allow an application to retrieve the event-specific information from the audit record. As with *aud_get_header*(), this information is returned as a structure to allow applications to access this information in standard fashion. And again, storage management is done by the system.

Note that cursor manipulation is a problem. Implicitly there is both a read and a write cursor for data descriptors within a record. The current definition excludes an important function, namely that of reading an audit record, modifying some attributes and writing it out. Comments would be welcome.

The information returned by this function is defined only syntactically and is mostly intended to be used by filters which format the audit record.

### A.5.7 aud_length()

This function returns the length of a specified audit record.

Note that this function is defined to return a value of type **size_t**, i.e., an unsigned integral type returned by the sizeof operator. This is consistent with the definition of similar interfaces defined elsewhere in the **X/Open Portability Guide** (e.g., *strlen*()) and in the **ANS/X3.159-1989 Programming Language C Standard**.

### A.5.8 aud_next()

This function allows a trusted application to read records from the system audit trail. These records are returned in sequential order by timestamp. The function allows a predicate to be specified in SQL format which defines search conditions. The SQL format is used because it is part of the X/Open Portability Guide. The predicate specified becomes the search criteria for subsequent calls of the function where a NULL predicate is supplied. If no predicate is ever defined, the default is to read the next record.

This function only returns a descriptor for the next record. The contents of this record may be read and manipulated by the application using the *aud_get* and *aud_put* functions. The format of the contents of the record is unspecified. The record may be translated into a specific format and returned in an application managed buffer with the *aud_print*() function.

### A.5.9 aud_print()

This interface is intended to provide a way for an application to translate the audit record into a specific format and then returned in an application specified buffer. This function is mostly intended to allow an application to format audit records for viewing. It also provides an application with the means to translate the record into a format suitable for exporting to another machine. The format of the text record is left as implementation-defined, since there did not seem to be much reason to define it. A few members of the group thought that the format should be specified so that it could be parsed but could not think of a general application for this. Note that the XDR and NDR data representations were chosen for their availability more than their efficiency. Several members of the group expressed reservations about these choices. Comments would be welcome.

### A.5.10 aud_put_object()

There are two interfaces provided to add information to the audit record: *aud_put_object*() and *aud_put_event_info*(). Each of these is the ''write'' analogue of the corresponding *aud_get* function.

The *aud_put_object*() function provides a means for an application to add an object description to the audit trail for objects affected by the event. This function is distinct from *aud_put_event_info*() because the object information is common to all types of objects.

Note that no separate function is provided to define the header of an audit record. This function was omitted because the group felt that modifying many of the header fields is a different level of trust. An application given this capability would be able to fabricate an entire audit trail. Instead, only the *event*, *client* and *status* fields of the header may be

specified by the application directly. All other fields are set by the system itself.

Note that cursor manipulation is a problem. Implicitly there is both a read and a write cursor for object descriptors and data descriptors within a record. The current definition excludes an important function, namely that of reading an audit record, modifying some attributes and writing it out. Comments would be welcome.

### A.5.11 aud_put_event_info()

This interface allows an application to add event-specific information to an audit record. Since the sort of information included in an audit record varies so much, it was felt by the group that providing separate information for each semantic type of information would be neither useful enough nor general enough to justify the number of additional interfaces to be provided. The *aud_put_object*() interface is provided since this particular type of information was deemed general enough to justify a separate function. All other information can be recorded with the *aud_put_event_info*() function. Note that the application should define the syntax of this information so that it can be properly formatted.

### A.5.12 aud_start()

This function will be used by trusted applications to create an audit record to be appended to the audit trail. This function was redefined at the August 1989 meeting to subsume the function of the other interfaces used to begin audit records. The other interfaces provided for the creation of audit records with no subject or with server subjects. This functionality is offered by the client field in the header and so separate interfaces were no longer deemed necessary.

It is implementation-defined when the header fields are actually filled in. There was considerable debate over this subject, since this implies that the event time cannot be specified by the application, but there was no clear resolution to the issue of when an event actually occurs. Comments would be welcome.

### A.5.13 aud_switch()

This function only affects the audit switch state of the calling process and subsequent descendents. No mechanism is defined for changing the audit switch state of the system as a whole, nor is it defined whether the default switch state is **off** or **on.** Both of these are considered to be implementation-defined (although maybe they should be considered for standardisation at some stage?). Comments would be welcome.

### A.5.14 exec()

This interface needs updating for the ACCOUNTABILITY option, to ensure that the audit ID and audit state of the process are inherited by a new process image. On the surface, this doesn't seem to be present any particular implementation difficulties (other than those associated with implementing audit IDs and audit states in the first place).

### A.5.15 fork()

Similarly, this interface needs updating to ensure that the audit ID and audit state of a parent are inherited by child processes. There are no obvious implementation concerns here either.

### A.5.16 get_password_aging()

This interface is intended for applications that perform their own authentication. Doubt was raised about its value at the February 1989 meeting. However, despite a proposal to remove the function, the group voted 6-1 in favour of retaining it.

Also at the February 1989 meeting, discussion took place about how an implementation of this function can determine whether the caller has appropriate privilege. There are no problems in the case of ''superuser'' processes; however, at one stage this interface also defined that each user would normally have privilege to call *get_password_aging*() for there own user name. How this privilege could be determined was not defined. Theoretically it could be done by accessing the identification database, which is slow and might present implementation difficulties, or it could be done via some other, less secure mechanism (e.g., environment variables)?

The SWG decided at the May 1989 meeting that this was an implementation issue that fell outside the scope of the interface definition. Thus the text was modified to indicate that ''appropriate privileges'' are required to call this interface, without defining the nature of those privileges.

### A.5.17 get_process_audit_ID()

Simple interface to allow trusted applications to read the audit ID of the current process (i.e., similar to *getuid*() and *geteuid*() for user IDs).

### A.5.18 get_process_audit_events()

An event list was originally defined as a fixed length array of {AUDIT_MAX_SIZE} elements, containing a row of audit event numbers. This definition was challenged at the February 1989 meeting as being too rigid and unnecessarily wasteful of storage space. It was further proposed that this document was changed to define event lists as variable length rows of audit event numbers, terminated by a value of (**ad_event_t**)0.

The group voted 3-1 in favour of retaining fixed length arrays in the interface definition.

This decision was reversed at the May 1989 meeting, where the group voted in favour of passing the number of elements in an event list via an argument *nmemb*. Other interfaces that manipulate event lists were similarly affected.

### A.5.19 get_user_audit_events()

This interface is intended to allow privileged processes to read the default audit event list for named users, or for the system as a whole. Where this information should be stored by a system is an implementation matter and is not defined in this document.

### A.5.20 map_audit_ID_to_user()

Provides trusted applications with a simple way of converting audit IDs to user names. There is no ambiguity possible in this mapping as there must be a one-to-one relationship between audit IDs and user names.

The auditor must ensure consistency of **identification databases** across systems for this interface to provide consistent mapping of Audit ID to user names.

### A.5.21 map_user_to_audit_ID()

Opposite of the above, providing a way of converting user names to audit IDs. Both these functions access the identification database to make the necessary conversions.

The auditor must ensure consistency of **identification databases** across systems for this interface to provide consistent mapping of user name to Audit ID.

### A.5.22 secure_get_password_user()

At the December 1988 meeting it was proposed that the *length* argument should be defined as the number of elements in a password string in ‘‘implementation-defined units’’. This was to allow various encrypting schemes where the resulting string was not necessarily produced in units of bytes. This proposal was rejected at the February 1989 meeting on the grounds that at some level the application would still need to work in units of bytes, and therefore it was an unnecessary complication to define the interface in terms of any other data type.

Note that this interface also supports a mechanism for determining the size of an encrypted password string without reading it.

### A.5.23 secure_put_password_user()

Opposite of the above which allows a trusted application to write an encrypted password to the authentication database. Rules for encrypting passwords are not defined in the interface definition, nor is there any implication that password strings are null terminated (thus allowing 0x00 as a valid encryption character).

### A.5.24 set_password_aging()

There has been considerable debate about the set and names of arguments to this interface. The *user_name* argument has been there throughout, but initially it was proposed to have two other arguments, *max_days* and *min_elapsed*. These gave, respectively, the number of days for which a password would remain valid, and the minimum number of days that must elapse before a password could be changed.

This interface was changed at the February 1989 meeting to define the arguments (a) *user_name*, (b) *time_to_expire*, which gives the maximum number of days for which a password is valid, and (c) *warning_time*, which denotes a period in days, before the password expires, during which the user is warned to change the password. It is not defined how the user is so warned (as the login mechanism itself is implementation-defined).

It was agreed at the May 1989 meeting that if *user_name* is the NULL pointer, the function should be defined to set password information for all current users of the system, over-riding any information already set in the authentication database. It is anticipated that this form of a call will be used infrequently (e.g., during system installation).

### A.5.25 set_process_audit_ID()

This interface is provided to allow trusted applications to set a process audit ID. Note that a call to this function will fail if the audit ID of the process is already set, meaning that (once set) process audit IDs cannot be changed.

### A.5.26 set_process_audit_events()

It is assumed that the normal method of initialising a process event list will be for a trusted application to read the associated default user event list (via *get_user_audit_events*()), using the results as the set of auditable events passed to the *set_process_audit_events*() function. This interface also allows a trusted application to modify or tune a process audit event list dynamically.

### A.5.27 set_user_audit_ID()

Declarative interface to the identification database, allowing trusted applications to establish the relationship between user names and audit IDs. The SWG agreed at the February 1989 meeting that this relationship could be changed, hence there are no restrictions about *user_name* identifying an existing database entry.

### A.5.28 set_user_audit_events()

This interface allows a trusted application to establish the system default event list and default event lists for individual users. It also supports a wild-card mechanism that allows the event lists of all current users to be updated in a single call. How and where this information is stored is not defined.

### A.5.29 sysconf()

This entry was added after the February 1989 meeting. The SWG agreed that the .I sysconf () function should be updated to return various switches and values for the Security Interfaces; specifically, which options are supported (if any), and the values of the AUDIT_MAX_SIZE and AUDIT_REC_SIZE variables.

### A.5.30 update_audit_events()

This interface allows a trusted application to update process audit event lists so they become consistent with the current (possibly amended) specification of user audit event lists. One would expect it to be used after having modified a user audit event list via the *set_user_audit_events*() function. Note that the interface does not define when these changes will come into effect. In particular, it might cause some odd behaviour on multi-processor systems, unless the interface is changed to define that the system as a whole is rendered quiescent before making the changes (although this could present implementation difficulties).

### A.5.31 audit.h

This header contains all the **typedefs** and symbolic constants defined for the Security Interfaces. It also contains all the function and macro declarations, as required by **ANS/X3.159-1989 Programming Language C Standard**. This has the secondary advantage of allowing implementations with naming restrictions to map long interface names to shorter values.

### A.5.32 limits.h

Updated for the AUDIT_MAX_SIZE and AUDIT_REC_MAX variables.

### A.5.33 unistd.h

Updated for the symbolic constants ACCOUNTABILITY and AUTHENTICATION. Also contains additional constants required by the *sysconf*() function.

## A.6 AUDIT EVENT CLASSES AND EVENT TYPES

The rationale for **Chapter 6**, **Audit Event Classes and Event Types** is incorporated into the chapter itself.

*Appendix B*

# *Non-Actioned Review Comments*

Prior to the formal release of a document by X/Open, it is subjected to what is called a ''Company Review Process''. This involves all X/Open member companies and representatives from the X/Open Associate Member Councils. During this process, comments are sought on the content of the document which may or may not lead to changes.

This appendix lists the response of the X/Open Security Working Group to some of the comments that were received during the Company Review Process and had no impact on the content of either the interface specifications themselves, or the rationale. They are included here, however, for further clarification of the work contained in this document.

**Comment 1**

**Section 6.1.1**, **Auditing of the System Interface**. Some events are missing: close(), dup(), pipe(), setsid().

**Response**

ACCEPTED IN PART. (See the table in **Section A.3.3**, **Audit Event Classes and Event Types**.)

**Comment 2**

**Section 2.2**, **Security Auditing**, second bullet: ''Applications which are trusted to do their own auditing''.

We disagree that applications which are trusted to audit themselves should be able to disable TCB auditing of themselves. The disabling ability should be a special ''privilege'' afforded to only special applications. (A separate programming interface for disabling auditing and the ability to append records to the audit trail are not restrictive enough.)

A problem arises when an auditor is trying to trace the actions of a user, but trusted applications used by the user disable their auditing, thus there are no records of what the user did while in the trusted process.

Selective disabling of auditing should be a separate privilege given to only a small set of trusted processes, most of whom require it because they are part of the audit function and might generate recursive audit without it.

**Response**

ACCEPTED IN PART. The comments made are taken into consideration in the sub-section of **Chapter 3**, **Auditing Suspension and Resumption**: ''A process with appropriate privileges to insert records into the audit trail may also be given the (possibly different) privilege to advise the TCB that standard auditing of its operations should be suspended or resumed. This may be useful to avoid unnecessary detail in the audit trail. The privilege to advise the TCB in this way should be available only to fully trusted software. The TCB may or may not actually suspend its auditing of the process, depending on the audit policy currently in use''.

With this specification in mind it should be clear that processes being able to suspend the ‘‘normal’’ audit have to be a part of the TCB (fully trusted software). As members of the TCB they are not suspending the TCB audit, but replacing the ‘‘normal’’ records with their own. Many ‘‘trusted’’ processes need to disable auditing since the records generated by the process would not be meaningful. Consider the output of login, most of the audit records would simply be noise. The audit records coming from third-party applications, a network server for example, would probably not contain much useful information. In these cases the application itself is best suited to create the audit record.

It should also be kept in mind that this specification was designed to fill the gap for X/Open-compliant systems to reach the C2 level of TCSEC, as auditing is the major feature missing in standard X/Open-compliant systems to reach this level of trustworthiness. Therefore we cannot rely on any privilege mechanism other than the UID 0. With systems designed for higher security levels, this may be different.

**Comment 3**

Excluding the capability of auditing specific files seems to be a mistake. Most administrators are concerned with watching just a few files, independent of the users. In order to enable this an administrator would have to make the system default class include objects which would create a huge trail.

**Response**

ACCEPTED IN PART. This is true; it is however quite difficult to implement (e.g., requiring changes to the inode structure, etc.). As this functionality is not necessary to fullfil the C2 requirements, it was felt that auditing all opens, for instance, then reducing the data during post-processing was acceptable. This may generate quite a large audit trail. The vendor may feel free to include this functionality into his implementation for performance considerations.

**Comment 4**

Having the base set of events defined as constants would seem to increase collisions and reduce portability. Events should remain as strings which are mapped to unique constants in an implementation-defined manner. Expanding the base, therefore, reduces the possibility of collisions. X has a base set of constants for its ‘‘atoms’’ and a registration mechanism for mapping new strings into unique ‘‘atoms’’, where atoms = ulong_t.

**Response**

REJECTED. The POSIX audit group is looking at assigning numeric constant values to the event types. For example:

```
#define AET_EXIT 2
```

Since it is likely that events are stored in the record as numeric values, a name to numeric mapping, as in the example above, is needed for audit trail portability. The problem with possible collisions exists only with extensions to the audit event types and classes.

**Comment 5**

No registration mechanism is defined for increasing the base of events or classes. A different machine interpreting the trail could not resolve any base extension. Having a registration mechanism would allow the new machine to redifine its base before attempting to read the trail to resolve these new events/classes.

**Response**

REJECTED. It was felt that each application would define a way to add events and classes. Since these events, by their very nature, are non-portable, the method for defining them need not be part of a standard. The fact that the base is expandable is sufficient.

**Comment 6**

get_*_events should return names, not numbers; this is a simpler interface.

**Response**

REJECTED. It is hard to see how the return of a character string is very useful. Most applications deal with numeric values far better than they deal with strings. Additionally, with a string the problems of space and size have to be dealt with. It is hard to see what benefit could be gained by returning a name.

**Comment 7**

There should be a registration mechanism for users (clients) to register a set of audit classes/events that servers can pick up, i.e., if a client wishes all background processes to run with a particular event class it should register that with the auditing system; the server can then query that registration and run the ''client process'' with that event class.

**Response**

REJECTED. See previous 2 comments.

**Comment 8**

**Section 2.1**, **General**: ''Events should be selectable on a per-user basis and on an individual per-process basis''.

Implementation details are horrendous, but the facility for auditing selectable on a per-object basis is popular with some users (particularly Government users). **Appendix A**, **Rationale** explains the reason for not including it. Did the SWG consider as an alternative the possibility of selection at collection time based upon the optional DAC/MAC/privilege attributes of an object? This would be particularly useful, of course, at B1 and above where a selection criterion could be ''access to all TOP SECRET objects''. Our experience with audit trails to date certainly shows that the biggest single problem is volume - any mechanism to reduce audit collection size, such as the above, is worth considering.

**Response**

ACCEPTED IN PART. Adding the ability to audit based on object level (i.e., MAC) is a very useful feature, we can see applications that may wish to audit only the actions of, for example, TopSecret users. Auditing on DAC (i.e., UID) seems less useful. The functionality to audit on a per-user basis is already provided.

On the other hand, as the auditing specification is designed for C2 systems, we cannot rely on MAC functionalities. This functionality is therefore out of the scope of the current specification.

**Comment 9**

**Section 3.1.1**, **Audit Identifier Interfaces**: ''It is further required that each user of the system has a distinct, individual user name''.

The requirement for distinct user names is prescribed with reference to ''the system''. No mention is made of unique accountability in audit records where machines are part of a network. Has there been any discussion on this point? For example, the audit header contains no field to identify a machine node, so that the identification of audit records produced on one machine and analysed on another is left to manual procedure. Should audit_IDs be network-wide unique; can this be achieved by defining the tuple:

<hostname, audit_ID>

as the unique identifier of the user within the audit record? If this was the purpose of the aud_net_t field in the audit header then I would support its inclusion.

**Response**

REJECTED. There is no need for the <hostname, audit_ID> tuple, as the audit header already includes the information on the machine where the record was generated in the net structure (of type struct aud_net_t). Unfortunately, this structure is not yet defined (as is the priv structure), but the group felt that this should be provided for a future version when naming services for networks are defined in the **X/Open Portability Guide**.

**Comment 10**

Sub-section of **Section 3.1.2**, **Manipulating Audit Records**: ''It is assumed that *ard* identifies a valid audit record buffer''.

The type assigned to *ard* is not very clear from reading the document. Is it an identifier (like a file fd), or a pointer, or the address of a pointer, as some procedure argument descriptions would suggest?

**Response**

CLARIFICATION. *ard* is a handle to an audit record.

**Comment 11**

**Section 3.5.3**, **Passwords and Password Aging**.

Aging on its own does not fit the security policies of systems which I have been involved with so far. The minimum time that must elapse before a password may be changed is common. Have other password time-constants been discussed?

Also on the subject of passwords, this document says nothing about password validation rules, another very common component of a system security policy.

Has there been any consideration to an interface that will support basic validation rules in a configurable way?

For example, two functions:

```
get_password_validation (pval)
passwd_val_t *pval
put_password_validation(pval)
passwd_val_t *pval;
```

where elements in the (structure) argument may define, as a first pass:

```
pwd_val_runlength
pwd_val_length
pwd_val_sequences
pwd_val_notlogin
pwd_val_unique
pwd_val_spell
```

This gives coverage of common policy requirements, which, in order, are: maximum number allowed runs of the same character; minimum password length; maximum allowed length of a sequenced (abc..,345..,); password to be different to login name; password to be system-wide unique; password not to be a valid word in the native language.

**Response**

REJECTED. This does not seem to be something that is required for applications portability, nor does it seem something that needs to be part of a standards specification.

Furthermore, a concept like this one is dangerous for standardisation in that it may restrict the effective use of local validation policies. What about the minimum and maximum numbers of numbers and/or special characters, codeset to use, etc.? To allow local validation policies to be used, the interface could be something like a routine taking the (plaintext) password as an argument and returning a boolean value for failure or success of the validation, which is done by some local module. But then again you start to run into trouble when you wish to issue any detailed error message in case of failure. So it is better to keep such interfaces, which are non-standard by their nature, out of the specification.

**Comment 12**

aud_get_event_info() in **Chapter 5**, **System Interfaces and Headers**.

The purpose of the format type AUD_FORMAT_OPAQUE is not very transparent!

**Response**

ACCEPTED. This format is for opaque data. This data, by its very nature, has non-standard types. The purpose of the format is to provide a way of informing the application that the data is of a type that is non-standard and may not be known or interpretable by the application.

**Comment 13**

<audit.h> header in **Chapter 5**, **System Interfaces and Headers**.

Facilities are provided to create and put records to the audit trail, such that ''trusted'' applications may do their own auditing, which is essential. In the audit.h header a set of symbolic constants defining audit event classes and types is given. If an application wishes to employ a new event class or event type how is it to allocate new constant values? There is a comment about this in **Appendix A**, **Rationale**.

The problem here is similar to the allocation of error values in general. I have used a scheme for exception handling within C which mirrors the structure for error numbers used by DEC within VMS. A 32-bit longword is split into 3 sections:

<Subsystem><ErrorNum><Severity>

A unique value in the ''Subsystem'' field specifies a system error; other values are assigned to utility programs or applications. The ''ErrorNum'' field is then private to the application. The ''Severity'' field in the exception handling code was used to identify warnings, informational messages, soft errors or fatal errors. This scheme seems to work quite well.

Without considering what the ''Severity'' field might mean within the auditing context, I would suggest that some form of hierarchical audit event type structure might be beneficial.

**Response**

ACCEPTED IN PART. It is clear that some method of adding audit events needs to be defined. The group would have to hear more about the implementation of the hierarchical audit event type structure defined above before commenting on its acceptability. Problems to be taken into consideration would be:

- the same event type with different naming and/or different placement into the hierarchy, and
- a naming scheme for different subsystems (how do subsystems who don't know each other get a unique name?).

At a first glance such a scheme seems to be better suited to avoid collisions, but if there is a unique way to provide unique names for applications we should be able to provide uniqe numbers also. In this case, the scheme can be used again as is.

**Comment 14**

**Section 6.2**, **Audit Event Types**.

In the audit event type tables, no mention is made of the networking service calls (XTI) - is this deliberate? Our projects to date have ALL required some form of network session auditing. Such records could, of course, be left to trusted applications. Was this intended?

**Response**

CLARIFICATION. Networking was intended to be left out.

**Comment 15**

Sub-section of **Section A.3.2**, **Audit Record Data Format** (fourth paragraph).

This paragraph specifies that audit data contains binary fields. Is the intention that applications are to provide the service of exporting audit trails to other machines for analysis?

**Response**

CLARIFICATION. Yes. The ability to do post-processing on a remote machine is a valuable feature.

**Comment 16**

Sub-section of **Section A.3.2**, **Portable Audit Record Format**: ''Portable data formats are not defined''. What about variable length ASCII records used by ''cpio''?

**Response**

REJECTED. Comment is not clear.

**Comment 17**

The system limit {AUDIT_MAX_SIZE} seems to be a misnomer - why not use instead {AUDIT_EVENT_MAX}?

**Response**

ACCEPTED IN PART. The group agrees that AUDIT_MAX_SIZE is not a very descriptive name however AUDIT_EVENT_MAX may be ambiguous. Does it mean the MAX events supported by the system or the MAX events settable in the user's audit mask?

**Comment 18**

**Section A.5.12**, **aud_start()**: ''...implementation-defined when header fields are actually filled in''.

The event time should be written in by the TCB. Thus it depends on the definition of the TCB perimeter as to whether an application is allowed to write header information into the audit trail.

**Response**

ACCEPTED IN PART. Indeed the header of every audit record can only be manipulated by a part of the TCB. However, the problem was a bit different: The group discussed in some depth ∗when∗ the header fields should be filled in, i.e., at the time the audit record is allocated or at the time it is committed. There were arguments for both schemes, and as the group did not reach any final conclusion on the topic, this issue was left as implementation-defined.