

OpenXDAS

Version 0.8.351

User's and Developer's Manual

Document Version 0.8

August 20, 2009

John Calcote



Table of Contents

What's in an Audit Record?.....	5
Record Format.....	5
Header.....	6
Record Length.....	7
XDAS Version.....	7
Time Offset.....	7
Time Uncertainty Information (2 fields).....	7
Time Source.....	7
Event Outcome.....	7
Not An Outcome.....	7
Successful Outcome Codes.....	8
Failure Outcome Codes.....	8
Denial Outcome Codes.....	9
Event Number.....	10
Account Management.....	11
User Session.....	11
Data Item and Resource Element Management Events.....	12
Service or Application Management Events.....	12
Service or Application Utilization Events.....	13
Peer Association Management Events.....	13
Data Item or Resource Element Content Access Events.....	14
Exceptional Events.....	14
Audit Service Management Events.....	15
OpenXDAS Extended Event Codes.....	15
Originator.....	16

An Originator Information String.....	17
Initiator.....	17
An Initiator Information String	18
Target.....	18
A Target Information String.....	19
Source.....	19
A Source Information String	19
Event-Specific Data.....	19
An Event-Specific Data Information String.....	19
XDAS Name Strings.....	20
OpenXDAS Service Configuration.....	22
Service Configuration.....	22
Command Line Configuration.....	22
Win32-Specific Options.....	23
Unix/Linux-Specific Options.....	24
Configuration File Options.....	24
OpenXDAS Logger Configuration.....	26
The File Logger.....	26
The LAF Logger.....	26
The Network Stream (NetStream) Logger.....	26
The ODBC Logger.....	30
ODBC Logger Quick Start.....	31
Caveats.....	32
The Syslog Logger.....	32
The XDAS Logger.....	32
OpenXDAS Record Filters.....	34

A Simple Filter Example.....	34
Filter File Syntax.....	35
A List of Filters.....	35
A Single Filter.....	35
Match Criteria.....	36
Action Specifications.....	37
The Log Action.....	38
The Alarm Action.....	38
The Trigger Action.....	38
The OpenXDAS Application Programmer's Interface.....	40
The Basic Conformance Interface.....	40
The Audit Stream Read Conformance Interface	40
The Import Conformance Interface.....	41
The Event Submission Conformance Interface.....	41
The Management Conformance Interface.....	41
The OpenXDAS C API.....	42
The OpenXDAS Java API.....	71
What Should I Audit?.....	73
Using the OpenXDAS Client Library.....	74
Using the C Client Library.....	74
Using the Java Client Library.....	74
Writing an OpenXDAS Logger Module.....	75
The Nuts and Bolts.....	75
An Example Logger.....	77
Using Your New Logger.....	78

Using OpenXDAS

In the past, auditing has been a secondary feature of enterprise and mission critical software. Due to new legislation and SEC regulations in the US, and similar corporate requirements mandated by governments of other countries, auditing will become more important in the near future than it ever has been in the past. More than an optional feature, companies will begin to require security auditing as a critical aspect of management software. So critical, in fact, that it will begin to make the difference in the sale and use of such software, despite other perhaps more visible features that may be better designed, or more functional.

The Open Group's XDAS standard attempts to define the critical aspects of security auditing, and OpenXDAS is an effort to create an open source reference implementation of this standard. As such, the authors feel it is critical that OpenXDAS follow the standard as closely as possible. Where OpenXDAS diverges from the XDAS Preliminary Specification, documentation is provided on the rationale for each change. By far, most of these changes were made simply to make the interfaces practically implementable.

What's in an Audit Record?

Audit records should contain two types of data – information that is important to audit records in general, and information that is important to a particular type of audit event.

The XDAS standard specifies the data that is generally important to audit events, leaving event-specific data up to the application, as event-specific data is generally more specific to the application domain, and thus difficult to specify generically.

Record Format

The XDAS common record format is easy on the programmer, easy on the wire and easy on the data store. It's simply a UTF-8, length-preceded string containing colon delimited text fields. XDAS records are divided into 6 sections: **Header**, **Originator**, **Initiator**, **Target**, **Source**, and **Event-Specific Data**. Here's the complete record format:

```
HDR:
<four_digit_hex_length_in_bytes>:
<max_16_char_version_str>:
<hex_time_offset>:
<hex_time_uncertainty_interval>:
<hex_time_uncertainty_indicator>:
<time_source>:
<time_zone>:
<hex_event_number>:
<hex_outcome>:
ORG:
<org_location_name>:
<org_location_address>:
<org_service-type>:
```

```
<org_auth_authority>:  
<org_principal_name>:  
<org_principal_id>:  
INT:  
<int_auth_authority>:  
<int_domain_specific_name>:  
<int_domain_specific_id>:  
TGT:  
<tgt_location_name>:  
<tgt_location_address>:  
<tgt_service-type>:  
<tgt_auth_authority>:  
<tgt_principal_name>:  
<tgt_principal_id>:  
SRC:  
<pointer_to_source_domain>:  
EVT:  
<event_specific_information>:  
END
```

Text within angle brackets are field names which are replaced in an audit record with the contents of those fields for a given record, ALL other text in the above template is literal, with the exception of line breaks. In reality, there are no embedded carriage return or line feed characters between fields. Line breaks are added here for clarity of presentation. Fields are delimited only by intervening colon characters.

Each field is required, but some are automatically generated by the OpenXDAS client library. For example, the tags, HDR:, ORG:, INT:, TGT:, SRC:, EVT:, and END should never be part of any data submitted by application developers through the OpenXDAS API interfaces. Additionally, as discussed below, most of the fields in the header section are generated fields. Those few that are not generated, are specified by the programmer in various API parameters.

Header

Header information is generally applicable and common to all events. This data includes the following fields:

- Record Length
- XDAS Record Format Version
- Event Time Stamp
- Time Uncertainty Interval
- Time Uncertainty Indicator
- Time Source
- Time Zone
- Event Number (user-provided).
- Event Outcome (user-provided).

Unless otherwise specified above, the values for the header fields are automatically generated or calculated by the OpenXDAS client library. Only the last two fields are submitted by the application developer through the *xdas_start_record*, or *xdas_put_event_data* functions.

Record Length

In order to properly calculate the length of a record, the length field within the record is specified to be four (4) hexadecimal digits in length, allowing the record room for up to 64k characters of text. Since the length field is itself a fixed length, the record may be built with a dummy value (such as 0000) for this record field, and then populated accurately after the record is complete. The length value does not include any terminating null character. In other words, if the audit record were treated as a zero-terminated string, then the value in this field would be the result of calling the standard C library *strlen* function on the record, beginning with the first character in the first section tag (HDR).

XDAS Version

The version field contains the current XDAS record format version number (0 for this version of OpenXDAS). This field has nothing to do with OpenXDAS, but rather with the particular version of the XDAS standard record implemented by OpenXDAS.

Time Offset

The time offset field contains 8 hexadecimal digits whose value represents the number of seconds since the beginning of the epoch, as defined by the Single Unix Standard (midnight on January 1, 1970).

Time Uncertainty Information (2 fields)

The time uncertainty fields are currently left empty by OpenXDAS, as the process of obtaining this information is significantly different from platform to platform, and very difficult to calculate accurately, in any case.

Time Source

The time source field is filled in whenever it can be determined. On Win32 platforms, this is the DNS name of the time server, and is obtained by reading the Windows time server registry configuration data.

Event Outcome

A set of standardized event outcome codes is defined by the XDAS specification, and fully supported by OpenXDAS. The following standardized outcome codes are broken into four subsets.

Not An Outcome

The first set contains only the value 0xFFFFFFFF, and represents the developer's choice not to specify the outcome code at this time. This is necessary, as the outcome of an event may not be known at the time the event record is initially created.

OpenXDAS Outcome Codes Indicating Various Types of Success		
XDAS_OUT_NOT_SPECIFIED	0xFFFFFFFF	This outcome code is not really an outcome code at all, but rather a special value passed to functions in the OpenXDAS interface which accept optional parameter values, and where the user desires not to specify an outcome code. Note that all records must ultimately have a valid outcome code (other than this value) before they can be submitted to XDAS.

Successful Outcome Codes

The second set is classified as successful outcome codes and indicates success in one way or another. The first value is the primary successful outcome code, and has the special value 0 (zero), which indicates unqualified success.

OpenXDAS Outcome Codes Indicating Various Types of Success		
XDAS_OUT_SUCCESS	0x00000000	This code indicates pure success, with no caveats, or side-band qualifications. This code should be used in the absence of a better, more specific success code.
XDAS_OUT_PRIV_USED	0x00000100	This is a success code indicating that the requested privilege was successfully used in the operation.
XDAS_OUT_PRIV_GRANTED	0x00000200	This is a success code indicating that the requested privilege was successfully granted.
XDAS_OUT_PRIV_REVOKED	0x00000400	This is a success code indicating that the requested privilege was successfully revoked.
XDAS_OUT_PRESELECT_CRITERIA_SET	0x00000800	This is a success code indicating that the requested pre-selection criteria was successfully set.
XDAS_OUT_THRESHOLDS_SET	0x00001000	This is a success code indicating that the requested thresholds were successfully set.
XDAS_OUT_ACTIONS_SET	0x00002000	This is a success code indicating that the requested actions were successfully set.

Failure Outcome Codes

The third set is classified as failure outcome codes and represents various types of operation failure. Note the difference between failure and denial (documented in the final set below). Denial may be considered by some to be a special class of failure codes, and indeed, it is. But XDAS treats denial as a separate class of outcome for ease of analysis.

OpenXDAS Outcome Codes Indicating Various Types of Failure		
XDAS_OUT_FAILURE	0x00000001	This code indicates pure failure, with no caveats or side-band qualifications. This code should be used in the absence of a better, more specific failure code.
XDAS_OUT_SERVICE_UNAVAILABLE	0x00000101	This is a failure code indicating that the specified service was unavailable.
XDAS_OUT_SERVICE_FAILURE	0x00000201	This is a failure code indicating that the specified service failed to successfully complete the requested operation.
XDAS_OUT_HARDWARE_FAILURE	0x00000401	This is a failure code indicating a hardware failure of some sort, directly causing the requested operation to fail.

XDAS_OUT_LOST_ASSOCIATION	0x00000801	This is a failure code indicating that the request could not be completed due to a lost association. More specifically, this outcome code is probably meant to indicate that a trusted association between identities on disparate systems is no longer valid, and thus the operation could not be completed due to security or rights issues.
XDAS_OUT_ALREADY_ENABLED	0x00001001	This is a failure code indicating that the request to enable a service, operation, or function failed because the target is already enabled.
XDAS_OUT_ALREADY_DISABLED	0x00002001	This is a failure code indicating that the request to disable a service, operation, or function failed because the target is already disabled.
XDAS_OUT_SERVICE_ERROR	0x00004001	This is a failure code indicating that the requested operation failed because a primary or secondary service, operation, or function failed an intermediate or direct request associated with the audited operation.
XDAS_OUT_BUSY	0x00005001	This is a failure code indicating that the requested operation failed because a required intermediate or end-point service was busy.
XDAS_OUT_DISABLED	0x00010001	This is a failure code indicating that the requested operation failed because a required intermediate or end-point service was disabled.
XDAS_OUT_INVALID_INPUT	0x00020001	This is a failure code indicating that the requested operation failed because some parameter or input to an intermediate or end-point service was not valid according
XDAS_OUT_ENTITY_EXISTS	0x00040001	This is a failure code indicating that a request to create an entity failed because such an entity already exists.
XDAS_OUT_ENTITY_NON_EXISTENT	0x00080001	This is a failure code indicating that the request to query, delete, or otherwise access a resource failed because the target entity does not exist.

Denial Outcome Codes

The last set is classified as the set of denial outcome codes. Denial codes represent various types of “permission denied” responses. The first value in this set is the primary denial code and represents unqualified denial.

OpenXDAS Outcome Codes Indicating Various Types of Denial		
XDAS_OUT_DENIAL	0x00000002	This is a denial code indicating pure denial, with no caveats or side-band qualifications. This denial code should be used in the absence of a better, more specific denial code.
XDAS_OUT_INSUFFICIENT_PRIVILEGE	0x00000102	This is a denial code indicating that the requested operation failed because of insufficient privileges. This is the semantic equivalent of the more widely understood “access denied” error.

XDAS_OUT_INVALID_IDENTITY	0x00000202	This is a denial code indicating that a target identity is invalid. Note that this denial code does NOT indicate anything about the initiating identity, as such a response to a request could be considered a security risk (giving too much information about why an operation failed). In the case of an invalid initiator identity, XDAS_OUT_INSUFFICIENT_PRIVILEGE should be used instead.
XDAS_OUT_INVALID_CREDENTIALS	0x00000402	This is a denial code indicating that a set of credentials presented during the operation was invalid. This may be an intermediate set of credentials, or a primary set.

A cursory examination of the bit patterns of the values associated with each outcome type will reveal that these outcome code classes are defined by the bits of the least significant byte. The class defined by FF in this byte indicates “unspecified”, while 00 indicates success, 01 indicates failure, and 02 indicates denial. Additional outcome code classes may be defined, but they must be registered with a central authority, even if the intent is that they only be used internally to an organization. Future versions of the XDAS specification may define additional standard classes of outcome codes, which may then conflict with such internal definitions.

Event Number

XDAS provides a generic event taxonomy – a set of semantic meanings for events that are likely to meet most applications' auditing needs. This gives analysis tools the ability to focus on analysis, and spend less time on disparate event type and event data correlation. It's often the case that events generated by various applications are either not documented well, or not documented at all, making the correlation issue risky and error prone. By providing a standardized generic event taxonomy, both application developers and analysis tool writers can easily understand the intended meaning of a given generic event.

While the XDAS working group tried to define a set of generic events that would meet most needs, they also understood that this was unlikely to satisfy every auditing need, so XDAS provides the ability to register new event class and identifier spaces. New class and identifier spaces must be registered with a central authority in order to be valid, and they need to be configured in local XDAS services in order to be recognized by XDAS as valid event identifiers. As a result of this required effort on the part of application developers, there's a fair amount of pressure to generate generic events, if at all possible.

While the intended semantic meaning of a given event may seem obvious at first glance, many applications may not fit the paradigm of traditional security or identity applications, but still the developers or architects of the application feel the need to generate audit events. We submit that if such a need is felt by product architects, then there's probably a good reason for it – the application most likely manages non-traditionally structured security or identity information, making it difficult to see the connection easily. For example, many applications may not create what one might consider to be a traditional user account, and it appears quickly evident to developers of such applications that XDAS generic events simply don't fit with their particular application's needs. However, as the documentation below tries to convey, the developer should

try to mentally reclassify the act of account creation into a more generic operation, for this is in fact the intended meaning of account creation from the XDAS perspective.

The remainder of this section will attempt to provide some insight into the intended meaning of each of the generic events defined by the XDAS standard, hopefully giving application developers the ability to properly choose an XDAS event for a given instrumentation scenario. The numeric values for these named events are assigned in the master OpenXDAS interface header file, *xdas.h* (C language implementation).

Account Management

Accounts exist in application domains in order to persistently associate attributes with the set of identifiers typically associated with identities. An identity, in this context, is a token used to represent a particular user or entity to which blame (or credit) should go for a set of activities within a system. This is not necessarily a human being, but could also be considered an appropriate description for an automated identity, such as another service, which may be acting on behalf of a human or a regularly scheduled system activity. In any case, account management should be taken to mean any form of persistent account creation, wherein an identity with some limited or unlimited set of system rights is associated with attributes.

OpenXDAS Account Management Event Numbers		
XDAS_AE_CREATE_ACCOUNT	0x01000001	This event should be considered appropriate for any situation wherein an account, as defined above, is to be created.
XDAS_AE_DELETE_ACCOUNT	0x01000002	This event has the opposite semantic meaning of account creation, and should be used wherever such an account (as described above) is to be deleted.
XDAS_AE_DISABLE_ACCOUNT	0x01000003	This event should be considered relevant for any situation where a particular record in an identifier database is disabled (by an administrator or an automated security process) such that it can no longer be used until it is re-enabled.
XDAS_AE_ENABLE_ACCOUNT	0x01000004	This is the counterpart event to the disable account event defined above.
XDAS_AE_QUERY_ACCOUNT	0x01000005	Query account events should be thrown whenever a request for attribute information for a particular account is made.
XDAS_AE_MODIFY_ACCOUNT	0x01000006	Modify account events should be thrown whenever a request to change attribute information for a particular account is made.

User Session

The abstract concept of a session can be explained as the association an initiator with a stream of communication. A session may represent a user's connection to server (as in the case of logging into a Unix or Windows host), or a set of related transactions in a connection-less environment (as in the case of using a cookie to maintain persistence between transactions between a browser client and a web server).

OpenXDAS User Session Event Numbers

XDAS_AE_CREATE_SESSION	0x01000007	This event should be reported whenever a new session (as defined above) is created.
XDAS_AE_TERMINATE_SESSION	0x01000008	This event should be reported whenever an existing session (as defined above) is terminated.
XDAS_AE_QUERY_SESSION	0x01000009	This event should be reported whenever attribute information is requested on an existing session.
XDAS_AE_MODIFY_SESSION	0x0100000A	This event should be reported whenever attribute information is modified on an existing session.

Data Item and Resource Element Management Events

This set of events relate to the creation and management of data items and resource elements within a domain. The type of data item or resource element is dependent upon the domain. For example, files and directories, device special files, and shared memory segments within an operating system, tables and records within a database, messages within an email system. The term data item is used in this context to refer to any type of resource element.

OpenXDAS Data Item and Resource Element Management Event Numbers		
XDAS_AE_CREATE_DATA_ITEM	0x0100000B	This event should be reported whenever a security-relevant data item is created.
XDAS_AE_DELETE_DATA_ITEM	0x0100000C	This event should be reported whenever a security-relevant data item is deleted.
XDAS_AE_QUERY_DATA_ITEM_ATT	0x0100000D	This event should be reported whenever a security-relevant data item is queried – either for value, or for an attribute of the data item.
XDAS_AE_MODIFY_DATA_ITEM_ATT	0x0100000E	This event should be reported whenever a security-relevant data item is modified – either the value, or an attribute of the data item.

Service or Application Management Events

This set of events relates to the management of services or applications. For example, the RPM package manager may throw these events as packages are installed or removed from a Linux system. Win32 service control manager (SCM) events sent to the Win32 System Event Log may be translated into these events as they are imported into OpenXDAS. This set of events could also be much more domain-specific, including concepts such as installing, removing, or configuring installable executable modules within a single application domain. The key idea is to ensure that reported events have security significance.

OpenXDAS Service or Application Management Event Numbers		
XDAS_AE_INSTALL_SERVICE	0x0100000F	This event should be reported when a service or application has been installed.
XDAS_AE_REMOVE_SERVICE	0x01000010	This event should be reported when a service or application has been removed.
XDAS_AE_QUERY_SERVICE_CONFIG	0x01000011	This event should be reported when service or application configuration information is requested.
XDAS_AE_MODIFY_SERVICE_CONFIG	0x01000012	This event should be reported when service or application configuration information is modified.
XDAS_AE_DISABLE_SERVICE	0x01000013	This event should be reported when a service, operation or function is disabled.

XDAS_AE_ENABLE_SERVICE	0x01000014	This event should be reported when a service, operation or function is enabled.
------------------------	------------	---

Service or Application Utilization Events

This class of events relates to the use of services and applications. They typically map to the execution of a program or a procedure and manipulation of the processing environment.

OpenXDAS Service or Application Utilization Event Numbers		
XDAS_AE_INVOKE_SERVICE	0x01000015	This event should be reported when a security-relevant service is invoked.
XDAS_AE_TERMINATE_SERVICE	0x01000016	This event should be reported when a service is terminated.
XDAS_AE_QUERY_PROCESS_CONTEXT	0x01000017	This event should be reported when any attributes of a process context are queried – this event is somewhat specific to operating systems, but some use might be found for it in other domain-specific applications.
XDAS_AE_MODIFY_PROCESS_CONTEXT	0x01000018	This event should be reported when any attributes of a process context are modified – this event is somewhat specific to operating systems, but some use might be found for it in other domain-specific applications.

Peer Association Management Events

Peer association events are related to the association of user or identity with a group, or the association of two users in some domain-specific context. An example might be adding an LDAP user to a group, or associating two users for a domain-specific purpose in an application's identity association database.

These events are also related to the association of identities within disparate authentication domains for purposes of federation. For example, when an identity in domain A makes a request to a service governed by domain B, then a peer association is required between these domains – often this is called a trust relationship. From an implementation perspective, setting up a trust relationship is often done by establishing an identity in domain B to be used as a proxy for any request coming from any identity in domain A. Trust relationships can be much more complex, however, as individual identities in domain A can have individual associations with specific domain B identities.

OpenXDAS Peer Association Management Event Numbers		
XDAS_AE_CREATE_PEER_ASSOC	0x01000019	This event should be reported when a new peer association is created.
XDAS_AE_TERMINATE_PEER_ASSOC	0x0100001A	This event should be reported when an existing peer association is destroyed.
XDAS_AE_QUERY_ASSOC_CONTEXT	0x0100001B	This event should be reported when the attributes of a peer association are queried.
XDAS_AE_MODIFY_ASSOC_CONTEXT	0x0100001C	This event should be reported when the attributes of a peer association are modified.

XDAS_AE_RECEIVE_DATA_VIA_ASSOC	0x0100001D	This event should be reported when data is received from a service in an authentication domain specifically via a trust relationship or peer association.
XDAS_AE_SEND_DATA_VIA_ASSOC	0x0100001E	This event should be reported when data is sent to a service in an authentication domain specifically via a trust relationship or peer association.

Data Item or Resource Element Content Access Events

Resource content access events are related to access of any data files protected by an authentication domain. This could be file system files, database records, web pages, etc. The important thing to consider while instrumenting applications that protect access to resources is that resource access can be a high-bandwidth process, and so only security-relevant events should be reported, and such instrumentation should be configurable at the application level by the application administrator, and thus policy driven. This may mean that such applications add additional infrastructure and user interface to allow administrators to manage which resource access events are audited, and which are unimportant within the security context.

OpenXDAS Data Item or Resource Element Content Access Event Numbers		
XDAS_AE_CREATE_DATA_ITEM_ASSOC	0x0100001F	This event should be reported when rights are granted to an identity to a specific data item – when a trust relationship is established between an identity and a data item.
XDAS_AE_TERMINATE_DATA_ITEM_ASSOC	0x01000020	This event should be reported when rights are revoked from an identity to a specific data item – when a trust relationship is revoked between an identity and a data item.
XDAS_AE_QUERY_DATA_ITEM_ASSOC	0x01000021	This event should be reported when rights are queried for an identity on a specific data item – when trust relationship attributes are queried for a specific identity and data item.
XDAS_AE_MODIFY_DATA_ITEM_ASSOC	0x01000022	This event should be reported when rights are modified on the previously established relationship between an identity and specific data item.
XDAS_AE_QUERY_DATA_ITEM_CONTENTS	0x01000023	This event should be reported when a data item is read on behalf of an identity.
XDAS_AE_MODIFY_DATA_ITEM_CONTENTS	0x01000024	This event should be reported when a data item is written on behalf of an identity.

Exceptional Events

Exceptional events are events which don't happen often, and are considered important simply because they happened. For instance, shutting down an enterprise-critical server is exceptional because it shouldn't happen without someone's permission.

OpenXDAS Exceptional Event Numbers		
XDAS_AE_START_SYS	0x01000025	This event should be reported when a server, system, or mission critical application starts up.
XDAS_AE_SHUTDOWN_SYS	0x01000026	This event should be reported when a server, system, or mission critical application shuts down.

XDAS_AE_RESOURCE_EXHAUST	0x01000027	This event should be reported when a server, system, or mission critical application runs out of some critical resource (like memory or disk space). Note that it is often difficult to report such events because often the critical resource in question is required in order to report the event.
XDAS_AE_RESOURCE_CORRUPT	0x01000028	This event should be reported when a server, system, or mission critical application detects a resource corruption (memory, disk file, etc).
XDAS_AE_BACKUP_DATASTORE	0x01000029	This event should be reported when a server, system, or mission critical application backs up a critical data store.
XDAS_AE_RECOVER_DATASTORE	0x0100002A	This event should be reported when a server, system, or mission critical application restores a critical data store.

Audit Service Management Events

For a variety of reasons, audit services have traditionally been classified by themselves. This is probably because auditing represents a lower level activity than the security events which themselves are being audited. By classifying audit events separately, an entire category of endless-loop defects can be avoided. Developers instrumenting applications will probably never need to report these events, as they are generally reported by systems like OpenXDAS itself. However, they are documented here for the sake of completeness.

OpenXDAS Audit Service Management Event Numbers		
XDAS_AE_AUD_CONFIG	0x0100002B	Configuration data has been changed for an audit subsystem. OpenXDAS reports this event when a SIGHUP is received, indicating that the xdasd configuration file has been modified and should be re-read.
XDAS_AE_AUD_DS_FULL	0x0100002C	This event is reported by OpenXDAS when an audit log is full, and can no longer accept additional audit records. Where possible, space is reserved for this event, in case it must be reported.
XDAS_AE_AUD_DS_CORR	0x0100002D	This event is reported by OpenXDAS when the data store reports that an audit log has been corrupted. Generally, this condition is not detected unless a request is made to read an audit stream, and the audit log reports that it cannot be read due to corruption.

OpenXDAS Extended Event Codes

Besides the standard XDAS event codes, OpenXDAS also defines the following extended set of event codes. These codes were discovered to be important to current users of the OpenXDAS implementation, as they tested and attempted to classify events they were already using in their previous auditing services.

These extensions to the standard include events that pertain to three specific areas: Password modification, workflow, and role management. Password modification is actually a special case of the more generic account attribute modification event, XDAS_AE_MODIFY_ACCOUNT.

Password modification, workflow and role management events similar to those defined here

will be added to the next XDAS standard through the Update-XDAS working group of the Open Group Security Forum.

OpenXDAS Extended Event Numbers		
XDAS_AE_MODIFY_AUTH_TOKEN	0x02000001	An authentication token may be a password, or any other type of authentication materials associated with a user account. Here, a user account means any type of account by which a user, application or system service may authenticate.
XDAS_AE_APPROVAL_RECEIVED	0x02000002	Workflow: Approval for a workflow item has been received by appropriate authority.
XDAS_AE_APPROVAL_REQUESTED	0x02000003	Workflow: Approval for a workflow item has been requested.
XDAS_AE_REQUEST_ESCALATED	0x02000004	Workflow: A workflow request has been escalated.
XDAS_AE_NOTIFICATION_SENT	0x02000005	Workflow: A workflow change notification has been sent.
XDAS_AE_CREATE_ROLE	0x02000006	Role Management: A new role has been created.
XDAS_AE_DELETE_ROLE	0x02000007	Role Management: An existing role has been deleted.
XDAS_AE_DISABLE_ROLE	0x02000008	Role Management: An existing role has been disabled.
XDAS_AE_ENABLE_ROLE	0x02000009	Role Management: A new role has been enabled, or a previously disabled role has been re-enabled.
XDAS_AE_QUERY_ROLE	0x0200000A	Role Management: Role attributes have been queried.
XDAS_AE_MODIFY_ROLE	0x0200000B	Role Management: Role attributes have been modified.

As mentioned earlier, extensions to this generic standard event taxonomy are allowed, but discouraged. OpenXDAS takes exception for its own extensions to the standard event number set, primarily because work is being done to add such events to the standard via the Open Group Security Forum's Update-XDAS working group. System architects should consider carefully the complete ramifications of extending the standard taxonomy.

Originator

The originator of an audit record is the service that is detecting the audit-worthy event, and requesting the recording of the event. In the context of submitting events from within an instrumented application, the originator then is the identity under which the submitting application is acting, relative to the authority or security realm from which that identity is obtained.

Within the originator information, the following data fields are defined in specified order:

- Originator Location Name
- Originator Location Address
- Originator Service Type
- Originator Authentication Authority (required)

Originator Principal Name (optional)
Originator Principal ID (required)

The *Originator Location Name* is the name of the host or service that is reporting the event. This name is an XDAS composite name, which is composed of a hierarchical representation of services. For instance, it may be as simple as host/service, as in the example that follows. The format of this name and all other names in the XDAS record format is defined below in the section entitled, *XDAS Name Strings*.

The *Originator Location Address* is a communication service end point address, which may be a URL, or other type of address that fully specifies a connection point.

The *Originator Service Type* indicates the protocol used by the *Originator Location Address*. One should, under ideal circumstances, be able to establish a communication channel to the originator using the combined data in this field and in the previous field.

The *Originator Authentication Authority* is the name of a server, or domain and realm that provides the identities involved in the associated events. A Unix host name is a good example of an authentication authority. Any service that manages an identity database with authoritative control of a security domain is a valid authentication authority. This name field follows the same formatting rules as the *Originator Location Name* field.

The *Originator Principal Name* is the user name relative to the *Originator Authentication Authority*. For example, if the application is the Apache web server, then the *Originator Principal Name* might be the name of the Unix user account under which Apache is running, or simply "root" if the Apache server is not running under any specific user account. The *Originator Principal Name* is optional, and may be left blank.

The *Originator Principal ID* is the user id of the principal, relative to the authentication authority. In the example above using Apache, the root user ID might just be 0. On Win32 systems, this might be the GUID of the user under which IIS is running.

An Originator Information String

A sample originator string passed to the *xdas_initialize_session* function might look like this:

```
char * org_info =  
    "waldo/apache:http%://waldo.novell.com:"  
    "http:waldo:root:0";
```

In the sample above, note that the embedded percent sign in the URL is escaped.

Initiator

The initiator is the user or identity that causes the audit-able event. A user logging in through a web server for instance, is the initiator. (The account under which the web service is running is the originator in this case.) The following fields are associated with the initiator string, and are specified in the following order, separated by colon characters:

Initiator Authentication Authority
Initiator Domain-Specific Name (optional)
Initiator Domain-Specific ID

The *Initiator Authentication Authority* is the host, service, or domain and realm name of the authentication service that provides the identity attributes for the initiator of an audit-able action.

The *Initiator Domain-Specific Name* is the name of the user or identity that is initiating an audit-able action, relative to the *Initiator Authentication Authority*.

The *Initiator Domain-Specific ID* is the identifier (uid, guid, etc) of the user or identity that is initiating the audit-able action, relative to the *Initiator Authentication Authority*.

An Initiator Information String

Here's an example of an initiator information string, as it might be passed to *xdas_start_record*, or *xdas_put_event_info*:

```
char * initiator_info = "waldo/nfs:joe:15";
```

Target

The target is the object being acted upon during the audit-able event. For example, a user whose rights are being modified through a web management interface is a target. Be aware that some events don't have a discernible target. The following fields are associated with the target information fields of an XDAS record, defined in the following order:

Target Location Name
Target Location Address
Target Service Type
Target Authentication Authority
Target Principal Name (optional)
Target Principal ID

These fields are similar in structure and content to the Originator fields defined above, except these fields indicate information about the object being acted upon, not the actor.

The *Target Location Name* is the name of the target resource, host or service of the event that is being reported. This name is an XDAS composite name, which is composed of a hierarchical representation of services or resources. The format of this name and all other names in the XDAS record format is defined below in the section entitled, *XDAS Name Strings*.

The *Target Location Address* is a communication service end point address, which may be a URL, or other type of address that fully specifies a connection point.

The *Target Service Type* indicates the protocol used by the *Target Location Address*.

The *Target Authentication Authority* is the name of a server, or domain and realm that provides the identities involved in the associated events. The target authentication information may be left blank in this case. A Unix host name is a good example of an authentication authority. This name field follows the same formatting rules as the *Target Location Name* field.

IMPORTANT: Not all events have target with associated identities. In this case, the last three fields, related to target identity are optional, and should all be left blank.

The *Target Principal Name* is the user name relative to the *Target Authentication Authority*. For example, if the application is the Apache web server, then the *Target Principal Name* might be the name of the Unix user account under which Apache is running, or root if not running under any user account. The *Target Principal Name* is optional, and may be left blank.

The *Target Principal ID* is the user id of the target principal, relative to the authentication authority.

A more interesting example is perhaps a file being downloaded from an ftp site. This is also a target, but note that there is no identity information associated with such a target, so the *Target Authentication Authority*, *Target Principal Name*, and *Target Principal ID* fields would be left empty in this case.

A Target Information String

The following is an example of a target information string:

```
char * tgt_info = "waldo/apache:http%://waldo.novell.com/index.html";
```

Source

The source field contains a pointer to a source domain in case the event was imported from a domain-specific logging service (such as Win32 Event Logs, or the Unix Syslog service). This allows the XDAS record to contain only security-relevant information without losing the ability to reference the additional information logged with the original event.

A Source Information String

The following example source string might be seen in an event record created on a Windows machine whose application event log has been streamed into OpenXDAS by an agent importing event records from the Windows application event log:

```
char * source = "jmc-ia32x/applog/{<event-guid>}";
```

Event-Specific Data

The event data field is designed to carry additional information that is specific to the application domain. The XDAS specification indicates that this field is designed to be used in environments

where XDAS is the primary audit system, and events should carry domain-specific information, as well as XDAS generic security event information.

An Event-Specific Data Information String

The format of this string is text-only, comma-delimited `<name>=<value>` pairs:

```
char * event_information = "laf-id=1701,my_attribute=32";
```

XDAS Name Strings

An *XDAS name* used within an XDAS audit event record consists of an ordered list of zero or more components. This is termed a *composite name*. Each component is a string name from the name space of a single naming system and uses the naming syntax of that naming system. A component may be an atomic or a compound name from that name space.

A composite name is the concatenation of the components of the name from left to right with the XDAS component separator character (`'/'`) separating each component. Special characters used in the XDAS composite name syntax, such as the component separator or escape characters, have the same encoding as they would in UTF-8. The minimum requirement for all XDAS implementations is to support UTF-8 for communication of name strings.

This section defines the standard string form of XDAS composite names in BNF. Note that all the characters of the string representation of one name must uniformly use the same encoding and locale information.

The XDAS composite name syntax in BNF is as follows:

```

NULL ::= // Empty set
<PCS> ::= // Portable Character Set
// The set consists of the glyphs:
// !"#$%&'()*+,-./0123456789:;<=>?
// @ABCDEFGHIJKLMNPOQRSTUVWXYZ[\]^_
// `abcdefghijklmnopqrstuvwxyz{|}~

<CharSet> ::= <PCS>
| Chars from the repertoire of a string.

<SimpleChar> ::= // <CharSet> minus /, ", and '
// %, %/, %", or %' is substituted for the
// corresponding unescaped character.

<Component> ::= <SimpleChar>*
| <SimpleChar>+ { " | ' | <SimpleChar> }*
| " <CharSet>* { % }* <CharSet>* "
// <CharSet> must not contain unescaped "
// (note that ' can appear unescaped)
| ' <CharSet>* { % ' }* <CharSet>* '
// <CharSet> must not contain unescaped '
// (note that " can appear unescaped)

<CompositeName> ::= NULL
| <Component> { / <Component> }*

```

This language is a fairly complicated way of saying that all XDAS record field strings are composite names – names built up from lists of component names separated by a forward slash character. Component names are quoted or unquoted strings composed of characters from the UTF-8 character set. The remaining rules simply indicate that embedded forward slashes, and double and single quotes must be escaped by the XDAS escape character (%), which must also be escaped in order to use this character literally.

OpenXDAS Service Configuration

OpenXDAS is implemented entirely in C as client library and service executable components. On Unix systems, `xdasd` is a daemon that starts at system startup time. On Win32 systems, `xdasd` is a service installed to start at system boot time. This executable is about 98 percent common code between Windows and Unix, differing only where system differences mandate. The service/daemon listens on a local interprocess communications (IPC) channel for requests submitted through the client library by instrumented applications. The client library is a shared library on Unix/Linux systems and the equivalent dynamic link library (DLL) on Windows systems.

This separation of duty between the client library and the service/daemon generally makes for a more secure and robust total auditing service. In addition to security, this separation provides less-than-obvious language-specific features to client applications. For instance, OpenXDAS also provides a pure Java client library implementation. This architecture makes Java security people very happy. Since no native code exists in the Java client library, it's easier to get Sun Java certification for applications instrumented to OpenXDAS when using the OpenXDAS Java client library (`openxdas.jar`).

Service Configuration

The service is configured primarily through two mechanisms: The command line, and a configuration file called `xdasd.conf`. On Windows systems, `xdasd.conf` is found in `%SystemRoot%` (usually `c:\windows`), and on Unix/Linux systems, it's found in `/etc`.

Command Line Configuration

The following is a list of current configuration options supported via the service command line. Generally, these are unimportant to the user, as the service is configured to use the command line parameters required by the installation, but we document them here for completeness:

```
USAGE: xdasd [-cefhklmv] [[-irsx] | [-d]]
```

Common options:

```
-c <file> use <file> as the configuration file.
-e <level> set the logging level to <level>.
-f <file> use <file> as the filter specification file.
-h        display this help screen and terminate.
-k <file> write dynamic listen socket number to <file>.
-l <file> use <file> as the log file.
-m <dir>  use <dir> as the message cache directory.
-p <dir>  use <dir> as the run path (for ipc files).
-v        display version information and terminate.
```

Windows-specific options:

```
-i [auto] install as a Windows service (auto: start on boot).
-r        remove the previously installed Windows service.
-s        start the previously installed Windows service.
-x        stop the previously installed Windows service.
```

Unix-specific options:

```
-d        detach process from console (daemonize).
```

This syntax was copied directly from the command line output of the following command:

```
c:> xdasd --help
```

As a matter of fact, any option text that is not recognized by *xdasd* will generate this help text.

As you can see, the configuration file itself may be specified on the command line by using a *-c* command line option, followed by a fully-qualified file path, allowing the administrator to move the configuration file to a more convenient location. For the most part, however, the default locations on various platforms will probably be sufficient.

Most of these options are self-explanatory. The *-k* option however probably requires a little explanation. This option is designed for unit testing by OpenXDAS developers, and has no direct use by end-users. Its purpose is to allow *xdasd* to start up listening on a dynamic port, rather than the default well-known XDAS port assigned by IANA (7629). This allows unit tests to start a test copy of XDAS and communicate with it from a test client without disturbing or interacting with an officially installed version of OpenXDAS on the test machine.

Another interesting option is the *-l* option, used for specifying a non-default location for the *xdasd.log* file. By default, this file is created and maintained in the *%SystemRoot%* directory on Win32 systems, and in the */var/log* directory on Unix/Linux systems. This log file can provide valuable insight into mis-configuration issues with the *xdasd* service, especially when filter files and unusual configuration options are involved.

The message cache is used to store intermediate data between record submission and record commission. This is normally stored in a directory called *xdasd.mcache* under *%SystemRoot%* on Win32 systems, and under */var/log* on Unix/Linux systems.

Win32-Specific Options

On Win32 systems, additional options are provided to help install, remove, start or stop the *xdasd* service manually. This can be useful for debugging, but with normal installations, the Win32 msi package performs these tasks for you.

Unix/Linux-Specific Options

On Unix/Linux systems, the *-p* and *-d* options are helper options for the system init scripts. These are generally not useful for the end-user directly, as the installer packages use these options as necessary for proper installation of the service.

Configuration File Options

The following is a list of current configuration options supported via the *xdasd.conf* file. A sample configuration file is installed into the */etc* directory on Unix/Linux systems, and into the *%ProgramFiles%\OpenXDAS\Etc* directory (by default) on Win32 systems. This sample configuration file lists all of the configuration file options available in configuration file comments, along with their default values if not specified.

Generally, text in the *xdasd.conf* file contain lines of configuration options or comments. Comments may be placed anywhere. They begin with hash (#) marks, and continue until the end of the line.

Configuration options contain *name = value* pairs. The names are hierarchical by convention only, meaning that there is no implied functionality in the hierarchy, only name space conventions, where *xdasd* is the root name space in the hierarchy. This is a common convention in Unix configuration files. Hierarchy levels are separated by periods.

Configuration lines may be continued using standard Unix line continuation syntax. That is, any line whose last character is a back-slash (\) will be joined with the next line before the line is interpreted by *xdasd*. This means that you shouldn't end a value with a trailing back-slash, or it will be interpreted as line continuation, rather than the last character of the value. One sadly ridiculous way around this problem is to place a space character after the trailing back-slash character. Trailing value spaces will be removed by *xdasd* anyway.

On Win32 systems, configuration data may be written to the registry rather than the configuration file. A similar hierarchy is used in the registry under the *Parameters* key of the service configuration stored in *HKLM\SYSTEM\CurrentControlSet\Services\xdasd*. Note that the configuration file will override these registry configuration values.

The following represents an exhaustive list of existing configuration options:

```
xdasd.loggers
xdasd.loggers.file.alogname
xdasd.loggers.netstream.secure
xdasd.loggers.netstream.server
xdasd.loggers.netstream.port
xdasd.loggers.netstream.trustfile
xdasd.loggers.netstream.keyfile
xdasd.loggers.netstream.keypwd
xdasd.loggers.netstream.ciphers
xdasd.loggers.odbc.connstr
```

The *xdasd.loggers* option allows a comma-separated list of executable logger modules to be specified. A logger module is nothing more than a shared library with a few well-known entry points, which are designed to be dynamically imported at load time. The *xdasd* service loads these shared libraries (dynamic link libraries on Windows systems), and then dynamically imports these well-known entry points, calling them at appropriate times during the operation of the service. Development of a logger module is documented in a later section.

Each logger module owns an entire sub-arch (sub-name-space) in the configuration file. A logger module's name space begins with *xdasd.loggers.xyz*, where *xyz* is the base file name of the logger module itself. For example, one logger module currently provided by OpenXDAS is the file logger module. It logs all audit records to a simple file. On Unix/Linux systems, the name of this module is typically *libxdm_file.so* (although this is not necessarily so). The base file name is simply *file*, so the configuration name-space owned by this module is *xdasd.loggers.file*.

All supported options for each of the existing loggers are described in detail in the following section, in a subsection dedicated to each logger. As new loggers are added, new documentation will be added for each new logger, its use and its supported configuration options.

OpenXDAS Logger Configuration

OpenXDAS is basically a logger. But it's designed to be flexible and expandable. The job of actually committing data to a data store is relegated to logger modules. There are 5 logger modules packaged with the 0.4.226 version of OpenXDAS. Each of these 5 loggers will be described in the following sections.

The loggers specified in the *xdasd.loggers* configuration parameter (specified in the *xdasd.conf* file) are loaded by *xdasd* on system initialization:

```
xdasd.loggers = C:\Program Files\OpenXDAS\loggers\xdm_file.dll, \  
C:\Program Files\OpenXDAS\loggers\xdm_syslog.dll
```

On Unix/Linux systems, this same configuration would look something like this:

```
xdasd.loggers = /usr/lib/openxdas/libxdm_file.dll, \  
/usr/lib/openxdas/libxdm_syslog.dll
```

The File Logger

The file logger is the simplest of the loggers provided in OpenXDAS. Its job is to write all audit records to a local file system file. The name and location of the file may be specified in the *xdas.conf* configuration file in a parameter called *xdasd.loggers.file.logfile* as follows:

```
xdasd.loggers.file.logfile = /var/log/xdas_audit.log
```

In fact, the audit log name and location specified in the above example is the default name and location of the audit file if this parameter is not specified. The default name and location on Win32 systems is *%SystemRoot%\xdas_audit.log*.

The LAF Logger

The LAF logger commits XDAS audit records to the Linux *Lightweight Auditing Framework* user-space API. There are no configuration parameters currently required or used by the LAF logger. If you're running Linux, and you want XDAS records to go to LAF, simply add the LAF logger to the *xdasd.loggers* configuration option.

The Network Stream (NetStream) Logger

The netstream logger is designed to log to a specified server address (by number or DNS name) and port (by number or service tag), over a clear-text or SSL connection using a TCP channel.

The netstream logger acts as a network client to servers that accept event records in the form of CRLF-terminated log lines.

This logger supports the following configuration options (with default values specified):

```

xdasd.loggers.netstream.secure = No/False
xdasd.loggers.netstream.server = localhost
xdasd.loggers.netstream.port = 1468/syslog
xdasd.loggers.netstream.trustfile = <empty>
xdasd.loggers.netstream.keyfile = <empty>
xdasd.loggers.netstream.keypwd = <empty>
xdasd.loggers.netstream.ciphers = <empty>

```

As of OpenXDAS, release 0.7.320, the netstream logger fully supports SSL connections to secure log-line servers. The *trustfile*, *keyfile*, *keypwd*, and *ciphers* options are ignored by the netstream logger if the *secure* option is not “True” or “Yes”.

The *trustfile* option refers to the file system path of a standard OpenSSL CA trust file in PEM format, containing one or more certificates and/or CA certificates used to verify the identity of servers to which the netstream logger connects. If no *trustfile* option is specified, then server authentication is disabled.

Server certificates and/or certificate chains are usually obtained from the server itself. That is, a server provides its public certificate chain so that clients can verify that they are who they say they are. This verification occurs as the client scans the chain ensuring that the server certificate is actually signed by an authority, or at least by a trusted entity. The server sends a signed certificate during the SSL handshake, at which point the client verifies that the certificate chain is valid, the trust list is in order, and that the server's host name is in the CN field of the certificate. The comparison of host name to CN field is borrowed from RFC 2818, wherein the rules for validating HTTPS servers are outlined.

The *keyfile* and *keypwd* options specify the file system path and password of the client-side key file to be used if client-side authentication is desired. The type of key file expected here is a standard OpenSSL PEM file containing the required keys *and* certificates to establish a secure connection with the server. If the *keypwd* option is left empty, then the client-side key should not be generated with a password. The netstream logger will fail to initialize if the keys are configured with a password and no password is given in the *keypwd* option, or the password given is incorrect. If the *keyfile* option is left empty, then SSL client-side authentication is disabled. If the server requests client-side authentication, and it's disabled in the netstream logger, then the SSL handshake will fail.

A client-side RSA private key can be generated using the `openssl` utility in the following manner:

```

$ openssl genrsa -des3 -out privkey.pem 2048
Generating RSA private key, 2048 bit long modulus
.....+++
.....+++
e is 65537 (0x10001)
Enter pass phrase for privkey.pem:*****
Verifying - Enter pass phrase for privkey.pem:*****
$

```

To generate a private key without a password, just leave out the `-des3` option, as follows:

```

$ openssl genrsa -out privkey.pem 2048
Generating RSA private key, 2048 bit long modulus
.....+++
.....
.....+++
e is 65537 (0x10001)
$

```

A valid client-side key file must also contain a client certificate if it's to be of any value to the server during client authentication. Client certificates may either be requested from a certificate signing authority (CA), or generated locally and self-signed. Generating a certificate signing request is done in the following manner, using the previously generated private key file:

```

$ openssl req -new -key privkey.pem -out cert.csr
You are about to be asked to enter information that will be
incorporated into your certificate request.
What you are about to enter is what is called a Distinguished Name
or a DN. There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Utah
Locality Name (eg, city) []:Provo
Organization Name (eg, company) [Widgits Pty Ltd]:Engineering
Organizational Unit Name (eg, section) []:Novell Inc
Common Name (eg, YOUR name) []:appsrv1.provo.novell.com
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
$

```

The signing request file (*cert.csr*) may now be sent off to a public or company CA to be signed. The CA returns a CA-signed certificate that may be used by your client. Client certificates are rarely signed by public CA's, however. Instead, servers are usually manually configured with a set of certificates that represent permissible clients. A server may also be configured with a company CA certificate, so that all client's with certificates signed by the company CA will be allowed by the server.

Client-side certificates will often be self-signed. A self-signed certificate may be generated from the previously generated private key as follows:

```

$ openssl req -new -x509 -key privkey.pem -out cacert.pem -days 1095
You are about to be asked to enter information that will be
incorporated into your certificate request.
What you are about to enter is what is called a Distinguished Name or
a DN. There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Utah
Locality Name (eg, city) []:Provo
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Novell
Organizational Unit Name (eg, section) []:Engineering
Common Name (eg, YOUR name) []:appsrv1.provo.novell.com
Email Address []:
$

```

The certificate file (*cacert.pem*) may then be used to configure the server for client-side verification of this client.

Additionally, the certificate should be added to the key file, so that both components will be available to the netstream logger via the *keyfile* option. Since these are both simply text files, this can be done as follows:

```

$ cat cacert.pem >> privkey.pem
$ cat privkey.pem
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAu8breh7eIvmi8el2LPIz5Ly6vidKnYbVmkFCqUaSEQcC980o
U+ixHdIGacNQUA4TKMdNZh2+L54EkNVQgSXXr+uft7r9dUKTUfov2HuDVW4SztJW
...
eX092RUXowBN2aaHwmZpi4oS7aTcxOisWwxJv/0Nm1SS9vCqD4AO5t3q0iCRoaTx
8n0ogDEhH6paIjLGx+4W3HQyVa1BArL64YbBM7wI/jMyCBLAGfSQKA==
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
MIIEJjCCA0agAwIBAgIJAOUuhOFRk43/MA0GCSqGSIb3DQEBBQUAMHwxCzAJBgNV
BAYTA1VMTMQ0wCwYDVQQIEwRVdGFoMQ4wDAYDVQQHEwVQcm92bzEVMBMGAlUEChMM
...
mZ85TKbSnDXaX2uGRNLBBhhMdszORGH/vieCkIWQul3Vxp0ynn3XpAdfwIr/t8U/
z8VRAvB9ikHNxxKD59AqfG5v
-----END CERTIFICATE-----
$

```

If both the *trustfile* and *keyfile* options are left unspecified, then an SSL channel is still used for data encryption, but no authentication is performed by either end of the channel.

The *ciphers* option allows the administrator to configure the set of ciphers required by the netstream logger to be supported by the server. This is a standard set of comma-separated OpenSSL cipher names. If the *ciphers* option is left unspecified, then OpenSSL's default cipher suite is used.

The netstream logger sends each record to the specified server and port followed by a CRLF pair for line termination.

The ODBC Logger

The ODBC logger commits XDAS audit records to SQL data bases through the *Open Data Base Connectivity* API. This logger may be configured with a single configuration parameter in the *xdasd.conf* file using the *xdasd.loggers.odbc.connstr* parameter as follows:

```
xdasd.loggers.odbc.connstr = DSN=xdas;
```

This string, “DSN=xdas;” is the default connection string used if this parameter is not given in the *xdasd.conf* file. The format and contents of the ODBC connection string is documented well in any good ODBC reference manual. Basically, however, a DSN is a *Data Source Name*, and represents a locally configured data source (which can also be a data sink, as it is, in fact, actually used in this context). If you choose not to configure the ODBC logger using this configuration parameter, then you will at least have to setup a DSN in your local ODBC configuration called “xdas”.

The connection string can be much more complex, allowing you to configure remote servers, ports, database names, user name and password, and table names in-line within the string itself. How you configure your connection string is completely up to you as the system administrator.

Besides configuring the connection string, you also need to setup your database schema before the ODBC logger can write to it

The examples in the following sections assume the use of MySQL as your DBMS. To use a data base management system other than MySQL, simply determine the ODBC driver schema mapping between the ODBC data types currently used by the OpenXDAS ODBC logger:

SQL_VARCHAR
SQL_INTEGER

and your desired database's native types. For instance, in the case of MySQL, SQL_VARCHAR maps to the MySQL 'varchar' data type, while SQL_INTEGER maps to the MySQL 'int unsigned' type.

Note also that the field names are statically defined by the OpenXDAS ODBC logger and must be defined and maintained as specified in the following table:

OpenXDAS ODBC Schema Definition	
Field Name	Data Type
xdasver	varchar(16)
timestamp	int(10) unsigned
tmuncert_int	int(10) unsigned

MySQL, MyODBC and unixODBC must be updated to at least the following versions in order to keep from crashing during reconnect after a connection timeout between xdas and the MySQL database:

- * unixODBC-2.2.12-13.i586.rpm
- * mysql-5.0.26-12.i586.rpm
- * mysql-client-5.0.26-12.i586.rpm
- * mysql-shared-5.0.26-12.i586.rpm
- * MyODBC-unixODBC-3.51.12-33

A Note About MySQL

tmuncert_ind	int(10) unsigned
timesrc	varchar(256)
tz	varchar(10)
event	int(10) unsigned
outcome	int(10) unsigned
org_loc	varchar(512)
org_addr	varchar(512)
org_type	varchar(512)
org_auth	varchar(512)
org_pname	varchar(512)
org_pid	varchar(512)
int_auth	varchar(512)
int_dsname	varchar(512)
int_dsid	varchar(512)
tgt_loc	varchar(512)
tgt_addr	varchar(512)
tgt_type	varchar(512)
tgt_auth	varchar(512)
tgt_pname	varchar(512)
tgt_pid	varchar(512)
esrc	varchar(1024)
edata	varchar(1024)

ODBC Logger Quick Start

Complete the following steps to configure your SQL database and ODBC driver manager to accept input from the OpenXDAS ODBC logger:

1. Configure your system's ODBC driver manager with a driver for your SQL database of choice (Oracle, SyBase, MySQL, Postgress, SQLite, etc.). This driver may reference a local or remote database.
2. Run the MySQL command-line client and create the xdas database using the following command:

```
mysql> create database xdas;
```

3. Create the xdas table using the command text specified above.
4. Configure your system's ODBC driver manager with a SYSTEM DATA SOURCE such that the name of the database to be populated with XDAS records is specified. Give that Data Source a name (DSN) such as "xdas".

5. If you've chosen a DSN other than "xdas", add a line to the *xdasd.conf* file indicating the name of the DSN to be used (as defined at the beginning of this section).
6. Enable the ODBC logger for *xdasd* by adding the name and path of the ODBC logger library to the *xdasd.conf* file as described above.

For MySQL data bases, create an ODBC logger table in new “xdas” database you created in step 2 above, using the following MySQL command:

```
mysql> create table xdas (xdasver varchar(16),
  timestamp int unsigned, tmuncert_int int unsigned,
  tmuncert_ind int unsigned, timesrc varchar(512),
  tz varchar(10), event int unsigned,
  outcome int unsigned, org_loc varchar(512),
  org_addr varchar(512), org_type varchar(512),
  org_auth varchar(512), org_pname varchar(512),
  org_pid varchar(512), int_auth varchar(512),
  int_dsname varchar(512), int_dsid varchar(512),
  tgt_loc varchar(512), tgt_addr varchar(512),
  tgt_type varchar(512), tgt_auth varchar(512),
  tgt_pname varchar(512), tgt_pid varchar(512),
  esrc varchar(1024), edata varchar(1024));
```

If you wish, you may simply cut and paste this text onto your MySQL command line. For more information on using the OpenXDAS ODBC logger, please refer to the README.ODBC file in the *xdasd/modules/odbc* directory of the source OpenXDAS distribution.

Caveats

Ensure that the user specified in your ODBC driver configuration has rights to the *xdas* database and table, and that you've specified the password in the ODBC configuration (if you've specified a password). Strange errors may come back on commit or logger initialization if *openxdas* doesn't have the necessary rights to insert data into the database.

You can view logger initialization and append errors in the *xdasd.log* file.

The Syslog Logger

The Syslog logger does just what it's name implies: It logs to syslog. However, in reality, the syslog logger is a short name for the “system log” logger. It logs to the system log on whatever system you happened to be running. On Unix/Linux systems, this is the syslog facility. However, on Windows, the syslog logger logs to the Windows event service Application Event Log. There are currently no configuration parameters for the syslog logger.

The XDAS Logger

The XDAS Logger is currently a null device logger. That is, it currently does nothing with audit records sent to it. The intent is that in the future, the XDAS logger will send audit messages to remote *xdasd* servers using the same wire protocol used internally between the XDAS client

library and the `xdasd` service.

There are several security issues to be worked out before this functionality can be incorporated into XDAS, such as channel encryption, authentication, etc.

Look for more on the XDAS logger as new versions become available in the future.

OpenXDAS Record Filters

XDAS supports the concept of filtering, and the OpenXDAS implementation supports this feature beginning in version 0.4.226.

Filtering is defined in XDAS such that audit records that match a given filter according specified match criteria, have certain actions carried out relative to those records. That is to say, any audit record that matches a given filter will have the associated actions carried out on that record.

XDAS (and thus OpenXDAS) supports three types of filter actions: logging, alarms and triggers (the XDAS standard calls triggers “actions” – OpenXDAS renames this third type of action to “trigger” in order to free up the word “action” to be used to specify the entire category of filtering actions).

Logging means committing a record to an audit log. OpenXDAS allows a filter to be defined in order to filter a particular class of audit records *out* of audit logs. This type of filter can ensure that valuable audit log space is not consumed by useless information (as defined by system administrators). Filters can also be defined in OpenXDAS to cause a particular class of audit records to be logged. While this is the default behavior for all audit records, when this sort of filter is defined, matching records will NOT be filtered out by subsequent filter definitions.

Triggering means executing external processes. OpenXDAS defines trigger actions as host operating system shell scripts. These scripts may do anything that a host shell script may do.

The concept of an alarm is not well defined by the XDAS preliminary specification. This is done on purpose, we believe. The concept of an alarm can have dramatically different meanings on different systems. A Unix host may use alarm to simply mean a separate and distinct audit record is sent to the logging subsystem when an alarm is triggered. An embedded system controlling the space shuttle, on the other hand, may truly set off an audible siren when an alarm is indicated. Since alarm is such a slippery concept, and can be implemented in terms of triggers for a particular application, we've chosen to simply leave alarms unimplemented at this time. However, the framework is in place within OpenXDAS to execute an alarm, and alarm actions may be defined within filters (though record matches are effectively benign).

The default file name for the filter file used to specify a filter set within OpenXDAS is *xdasd.filter*, and is found in *%SystemRoot%\Windows* on Windows systems, and in the */etc* directory on Unix and Linux systems.

If no filter file is defined, OpenXDAS will log a note in the *xdasd.log* file indicating that no filter file was found, and then continue as if no filters were defined (which is true).

A Simple Filter Example

The following is fairly trivial example of an OpenXDAS XML filter file. This example filter will cause a trigger script to execute, which sends a mail message to the system administrator for all matching audit records:

```

<filter-list version='OX1'>
  <filter name='Account Created' status='On' type='Submit, Import'>
    <match>
      <allow attr='HdrEvent' op='EQ'>0x01000001</allow>
      <deny attr='HdrOutcome' op='NE'>0</deny>
    </match>
    <action>
      <trigger>
        <![CDATA[
mailx -s "Account $XDAS (TgtName) created by $XDAS (IntName) "
admin@someplace.com <<InputFromHERE
$XDAS
InputFromHERE
]]>
      </trigger>
    </action>
  </filter>
</filter-list>

```

In this example, a mail message is sent to “admin@somplace.com” each time the filter conditions are met by an audit record. The filter conditions, in this case, indicate matching records if the records' event type (*HdrEvent*) matches the value of 0x01000001 (“Create Account” in the XDAS standard taxonomy) AND whose outcome (*HdrOutcome*) is NOT non-zero. The logic here seems a bit convoluted at first glance, but is easily understood once the intent is made clear. Simply put, we want all successful account creation records to trigger an email message sent to the system administrator.

Filter File Syntax

Let's consider the filter file syntax a section at a time. The overall syntax is XML, and in fact, a SAX parser (eXpat) is used to extract the data from the filter file as the *xdasd* service initializes. So general XML rules must be adhered to by filter file authors.

A List of Filters

The root-most element in a filter file is the “filter-list” element:

```
<filter-list version='OX1'>
```

This element contains only a single attribute, the “version” attribute, which specifies the version of the filter list. In this release of OpenXDAS, the initial filter version is “OX1”, exactly the same as the OpenXDAS record version. The version attribute **MUST** exist. The *xdasd* service will not load if the version attribute is missing or is not set to the value “OX1”.

A filter-list may contain zero or more “filter” elements, each defining a single filter with a set of match criteria and a set of associated actions.

A Single Filter

This element acts as a container for multiple named “filter” elements:

```
<filter name='Account Created' status='On' type='Submit, Import'>
```

The filter element specifies a filter by name. The filter name is used to manipulate the filter through the filter management API (not yet implemented in version 0.7.320). The filter status indicates whether or not this filter is currently active. Values of “On”, “Off”, “True”, “False”, and “Yes”, or “No” are all acceptable, and case is irrelevant. The default status is “On” if the attribute is missing.

The filter type indicates which input service this filter applies to: events that are submitted by instrumented applications, events that are imported via XDAS system agent, or both. Values of “Submit” and “Import” may be used, and again, case is irrelevant. The default filter type is “Submit” if the type attribute is missing.

Match Criteria

The filter element acts as a container for two types of sub-elements, “match” elements and “action” elements. Match elements indicate match criteria for a filter, containing “allow” and “deny” statements. As stated earlier, the order of allow and deny statements is important in defining the ultimate outcome of a match:

```
<match>
  <allow attr='HdrEvent' op='EQ'>0x01000001</allow>
  <deny attr='HdrOutcome' op='NE'>0</deny>
</match>
```

The filter criteria act as a state machine where the three possible states of a given record with respect to a given filter are “default”, “include” and “exclude”. The default state of a record is defined by the action. The default state of the “log” action, for instance, is to include the message in the audit log. That is, if a filter doesn't match a given record, the default state for logging is to log the message. The default state of the trigger or alarm action is to ignore all records. In this case, any record that doesn't match a particular filter is ignored for purposes of triggers and alarms associated with that filter.

A filter's match criteria is specified as a set of allow and deny statements that each affect the state of the match outcome, in the order that these statements are specified. Allow statements add to the set of messages that pass through the filter – acting as logical OR constructs. Deny statements act as constraints on the set of events that have already passed through previous allow statements – having the ultimate effect of logical AND statements.

In the above example, the initial allow statement passes (or matches) records that have the “Create Account” event type, but filters out previously matched records whose outcome is not successful (zero). Once a record has been matched by an allow statement, it can no longer qualify for the default state – only for the include or exclude states.

It should be noted that the most effective filtering may be implemented based primarily on event type and outcome. These fields and the other two fields allowed for submitted events are specified in the OpenXDAS filter file using the following tags: *HdrVer*, *HdrEvent*, *HdrOutcome*, and *IntAuth*. The case is irrelevant in these tag values.

When *importing* events from system event generation sources, many more fields may be used as filtering criteria. This is done for performance reasons, as the XDAS filter specification can cause unacceptable performance bottlenecks if over-applied or mis-applied.

When matching an audit record, the following match criteria are defined by the XDAS specification: “equal”, “not equal”, “greater than”, “less than”, “greater than or equal”, “less than or equal”, “bit test”, and “substring”. These are defined by using the following tags (respectively): “EQ”, “NE”, “GT”, “LT”, “GE”, “LE”, “BT”, “SS”.

All but substring (SS) may be used on numeric fields, and all but bit test (BT) may be used on string fields. The semantic meaning of these operations is fairly obvious, except possibly for bit test, which is a bit-wise AND operation between the specified value and the specified record field. If the result of the operation is anything but zero, the logical result is “match”. The result of a substring operation is true only if the specified match value is a substring of the specified audit record field.

The XDAS specification indicates which fields may be filtered for submitted and imported records. Since OpenXDAS has not yet implemented import agents, we'll focus on submitted records in this version of this document. The XDAS specification indicates that only the following XDAS audit record fields may be used to filter submitted records: Version, Event Number, Outcome, and Initiator Authentication Authority. The XDAS specification uses some interesting language to define rationale for the use of these (and only these) fields. The reader should refer to this document for rationale, and argue on behalf of, or against various fields in the community forums.

Action Specifications

Action statements specify the actions to be associated with audit records that match this filter according to the previously defined match criteria. The order of match and action elements is unimportant, however only one match or action element should be associated with a given filter element:

```

<action>
  <log />
  <alarm severity='3'>
    <trigger>
      <![CDATA[
mailx -s "Account $XDAS(TgtName) created by $XDAS(IntName)"
admin@someplace.com <<InputFromHERE
$XDAS
InputFromHERE
]]>
    </trigger>
  </action>

```

Multiple action statements are allowed within an “action” element. Multiple instances of “log” have no side-effects. Multiple instances of “alarm” has the effect of using only the highest valued “severity” value. Multiple instances of “trigger” store all trigger scripts to be executed serially by external processes for matching records.

The Log Action

The “log” element indicates that the match criteria of this filter should be applied to logging of matching records. Remember that a record may match (be included, or excluded), or not match – that's three states: default (no match), include, and exclude – a given filter specification. If a record matches and is allowed, then the “log” action indicates that this record must be logged, regardless of the outcome of other filter matches and actions. If a records matches and is denied, it is filtered out, as long as another filter does not specify that this record must be logged. If a record doesn't match the filter match criteria, then the default logging action is applied to the record – which means that the record will be logged – assuming no other filters interfere with this state of affairs for this record, that is.

The Alarm Action

The “alarm” element maybe specified, but does nothing at this time, as mentioned previously.

The Trigger Action

The “trigger” element specifies a script to be executed by the system shell with respect to all matching records. As shown in this example, the script may be instrumented with variables representing various fields of the associated record. Audit record variables are defined as either `$XDAS(field name)` or simply `$XDAS`, which represents the entire audit record. The following list represents the entire set of field names that may be used in this syntax:

HdrLen	OrgLoc	IntAuth
HdrVer	OrgAddr	IntName
HdrTMOff	OrgType	IntID
HdrTUInt	OrgAuth	
HdrTUInd	OrgName	
HdrTMSrc	OrgID	
HdrTMZone		
HdrEvent		
HdrOutcome		
TgtLoc	SrcDPtr	EvtInfo
TgtAddr		
TgtType		
TgtAuth		
TgtName		
TgtID		

The script text is propagated to the system shell by OpenXDAS exactly as formatted in the filter file, including all internal white space. Leading and trailing white space is stripped off before processing, however.

Be sure that your scripts work correctly and return a valid completion code in all cases. A completion code of zero represents successful operation of a shell script, and OpenXDAS will detect non-zero trigger script status codes, and log them as errors in the `xdasd log` file.

On Unix and Linux systems the system shell is detected through the `SHELL` environment variable. The value of this variable is often `/usr/bin/sh`. As such, this is also the default value if the `SHELL` variable is not set. On Windows systems, the system shell is detected through the

COMPSEC environment variable, which is often specified as *%SystemRoot%\system32\cmd.exe*, and so this is the default value on Windows systems.

Trigger scripts are shell-syntax-specific. That is, a script on Linux that's intended for bash, will likely be different than a script that does the same thing on a Windows system, as *cmd.exe* is not likely to understand bash syntax. Since the contents of the script is passed verbatim to the shell for execution, it should be left-justified in the filter file, as shown in the example above, to avoid any extraneous leading whitespace issues.

NOTE: Prior to version 0.7.320, there was a bug in the script parsing code that caused the last line of trigger scripts to be ignored. If you have an older version of OpenXDAS, this bug can be worked around by adding an extra blank line at the end of your trigger scripts. This defect has been fixed in the 0.7.320 release.

The OpenXDAS Application Programmer's Interface

The OpenXDAS API is divided into 5 conformance levels: **basic**, **read**, **submit**, **import**, and **manage**. Each of these conformance levels, except for basic, stands alone. The basic conformance level includes only the ability to initialize a OpenXDAS session, which is not actually enough to do anything practical.

The Open Group's XDAS preliminary specification actually defines the basic conformance level to contain both the audit session initialization and the audit stream read API's. However, previous implementations of XDAS generally did not implement the audit stream read interface, or at least considered it a lower priority than other interfaces. We felt it made more sense to define basic conformance as containing both initialization and event submission API's, as these were more generally required than the audit stream read interface. However, it's clear that, when considered from a purely theoretical perspective, it makes little sense to create a write-only interface! Pragmatically, it's not really so – when submitting to a file or a database, there are plenty of existing ways to read the data submitted through OpenXDAS.

While each of these interfaces is documented in this manual, application developers desiring to instrument their applications for event submission will wish to pay particular attention to the basic and submit interfaces.

The Basic Conformance Interface

The basic interface contains two functions:

xdas_initialize_session, and
xdas_terminate_session

These two functions are used to manage an audit session, and must be used by anyone wanting to use any of the other OpenXDAS interfaces.

The Audit Stream Read Conformance Interface

The read interface contains 5 functions:

xdas_close_audit_stream,
xdas_get_next,
xdas_open_audit_stream,
xdas_parse_record, and
xdas_rewind_audit_stream

These five functions are used to access a stream of audit records. Currently OpenXDAS does

not support the Read conformance level, but these functions are documented here for completeness, and because it will eventually be completed as documented here.

The Import Conformance Interface

The import interface contains only one function:

xdas_import_event_records

This routine should be used to write system agents for OpenXDAS that import domain-specific event records into XDAS format.

The Event Submission Conformance Interface

The event submission interface is of particular importance to application developers wishing to instrument their applications for audit event submission. The Submit interface contains the following 5 functions:

xdas_commit_record,

xdas_discard_record,

xdas_put_event_info,

xdas_start_record, and

xdas_timestamp_record

There are five functions in the submission interface, however most application developers will only use two of these functions on a regular basis, *xdas_start_record*, which is used to create a new record and populate it, and *xdas_commit_record*, which is used to actually submit the record.

The XDAS event submission interface is flexible however, in that developers may begin a record, and then add information to it gradually, as a complex system function is carried out, and finally committing the record once it's been fully populated. The developer may even specify the exact point in this process that the time-stamp should be recorded in a record built in this manner using the *xdas_timestamp_record* function. The *xdas_commit_record* function will only add a time-stamp to a record if it sees this has not already been done manually.

The Management Conformance Interface

The management interface is used to specify filter information for a local host OpenXDAS service. The Management interface contains the following six functions:

xdas_create_filter,

xdas_delete_filter,

xdas_disable_filter,

xdas_enable_filter,

xdas_get_filter, and

xdas_list_filters

The OpenXDAS C API

The C API follows the documented specification to the degree that it can. Where it deviates from the XDAS preliminary specification, the Doxygen documentation clearly documents the rationale for these changes.

The following pages documents each function defined in the C application programmer's interface. The C-language header file *xdas.h* contains the prototypes for these functions.

xdas_initialize_session

Basic

Initialize an audit client library session.

Synopsis

```
XDASXPC int XDASAPI xdas_initialize_session(  
    int *                minor_status,  
    const char *         org_info,  
    xdas_audit_ref_t *   das_ref);
```

Parameters

<i>minor_status</i>	(optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE .
<i>org_info</i>	Originator field information specified as colon-separated text fields in a zero-terminated string. Embedded colons must be escaped with '%'
<i>das_ref</i>	Return storage for the newly created OpenXDAS session handle.

Return Values

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_ORIG_INFO** if the originator information supplied has a syntax error.

Remarks

The *xdas_initialize_session* function is a member of the Basic XDAS conformance class. On completion, the caller must terminate the OpenXDAS session by calling *xdas_terminate_session*.

All callers must initiate a session with the OpenXDAS service before they can use any of the services it provides. The initialization of the session supports the mutual authentication of the audit client and audit service components and establishes the audit client's OpenXDAS authorities. The caller is returned a handle to the OpenXDAS service which is then used for all OpenXDAS API functions.

The function initiates a session between the caller, identified by *org_info*, and the distributed audit service. The *org_info* data is inserted by the client library into every audit record submitted by the caller through subsequent calls to the OpenXDAS submit API within the OpenXDAS session. The function validates the security context implicit in the caller's process address space to ensure that the caller is authorized to use the OpenXDAS service. The use of this function is itself be audited by the OpenXDAS service.

The caller must have the **XDAS_AUDIT_SERVICE** authority.

xdas_terminate_session**Basic**

Terminate an existing audit client library session.

Synopsis

```
XDASXPC int XDASAPI xdas_terminate_session(  
    int *          minor_status,  
    xdas_audit_ref_t das_ref);
```

Parameters

<i>minor_status</i>	(optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE .
<i>das_ref</i>	The handle of the OpenXDAS session to be terminated.

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid OpenXDAS audit service session.

Remarks

The *xdas_terminate_session* function is a member of the Basic XDAS conformance class.

The function closes a session between the caller and the distributed audit service. Upon return, the handle *das_ref* is no longer a valid OpenXDAS session handle.

The caller must have the **XDAS_AUDIT_SERVICE** authority.

xdas_close_audit_stream

Read

Close a previously opened OpenXDAS audit stream.

Synopsis

```
XDASXPC int XDASAPI xdas_close_audit_stream(
    int *                minor_status,
    xdas_audit_ref_t     das_ref,
    xdas_audit_stream_t  audit_stream_ref);
```

Parameters

<i>minor_status</i>	(optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE .
<i>das_ref</i>	The handle of the OpenXDAS session to use for this operation, obtained through a previous call to <i>xdas_initialize_session</i> .
<i>audit_stream_ref</i>	The audit stream handle to be closed.

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_AUDIT_STREAM** if the specified audit stream handle is not valid.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid audit service session.

Remarks

The *xdas_close_audit_stream* function is a member of the Read XDAS conformance class.

The function closes the audit stream, previously opened for reading, specified by the *audit_stream_ref* handle. Once an audit stream is closed, that audit stream is no longer valid for use in any OpenXDAS function call.

The caller must possess the **XDAS_AUDIT_READ** authority.

xdas_get_next**Read**

Fill an output buffer with the next set of records in an OpenXDAS audit stream.

Synopsis

```

XDASXPC int XDASAPI xdas_get_next (
    int *                minor_status,
    xdas_audit_ref_t    das_ref,
    xdas_audit_stream_t audit_stream_ref,
    unsigned            max_records,
    xdas_buffer_t       audit_record_buffer,
    unsigned *          no_of_records);

```

Parameters

<i>minor_status</i>	(optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE .
<i>das_ref</i>	The handle of the OpenXDAS session to use for this operation, obtained through a previous call to <i>xdas_initialize_session</i> .
<i>audit_stream_ref</i>	The audit stream handle to be read.
<i>max_records</i>	Specifies the maximum number of records to be returned in <i>audit_record_buffer</i> . If <i>max_records</i> is zero then the buffer is filled to capacity.
<i>audit_record_buffer</i>	The buffer to which audit records are to be copied.
<i>no_of_records</i>	Return storage for the number of records returned in <i>audit_record_buffer</i> .

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_BUFF_TOO_SMALL** if the buffer specified by the caller is not large enough to return a single record.
- ◆ **XDAS_S_END** if the end of the audit stream has been reached.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_AUDIT_STREAM** if the specified audit stream handle is not valid.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid audit service session.

Remarks

The *xdas_close_audit_stream* function is a member of the Read XDAS conformance

class.

The function closes the audit stream, previously opened for reading, specified by the *audit_stream_ref* handle. Once an audit stream is closed, that audit stream is no longer valid for use in any OpenXDAS function call.

The caller must possess the `XDAS_AUDIT_READ` authority.

xdas_open_audit_stream**Read**

Opens an OpenXDAS audit stream and associates it with an OpenXDAS session.

Synopsis

```
XDASXPC int XDASAPI xdas_open_audit_stream(  
    int *                minor_status,  
    xdas_audit_ref_t     das_ref,  
    xdas_audit_stream_t * audit_stream_ref);
```

Parameters

<i>minor_status</i>	(optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE .
<i>das_ref</i>	The handle of the OpenXDAS session to use for this operation, obtained through a previous call to <i>xdas_initialize_session</i> .
<i>audit_stream_ref</i>	Return storage for the new audit stream handle value.

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid audit service session.

Remarks

The *xdas_open_audit_stream* function is a member of the Read XDAS conformance class.

The function opens the audit stream for reading and returns a handle to the audit stream in *audit_stream_ref*. A caller may obtain more than one handle to the audit stream, each of which is independent of any other handles.

The caller must possess the **XDAS_AUDIT_READ** authority.

xdas_parse_record**Read**

Parse a specified OpenXDAS record from an XDAS record buffer.

Synopsis

```

XDASXPC int XDASAPI xdas_parse_record(
    int *                minor_status,
    xdas_audit_ref_t    das_ref,
    xdas_buffer_t       audit_record_buffer,
    unsigned             record_number,
    xdas_audit_record_t audit_record);

```

Parameters

<i>minor_status</i>	(optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE .
<i>das_ref</i>	The handle of the OpenXDAS session to use for this operation, obtained through a previous call to <i>xdas_initialize_session</i> .
<i>audit_record_buffer</i>	A pointer to a buffer containing the audit records to be parsed, filled by a previous call to <i>xdas_get_next</i> .
<i>record_number</i>	Indicates which record should be parsed and returned in <i>audit_record</i> . The first record number is zero.
<i>audit_record</i>	The audit record structure to be populated with buffer record information.

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid audit service session.
- ◆ **XDAS_S_INVALID_EVENT_NO** if the specified record number is not valid.

Remarks

The *xdas_parse_record* function is a member of the Read XDAS conformance class.

The function parses and decomposes the record numbered *record_number* in *audit_record_buffer*, which was filled with a number of records by a previous call to *xdas_get_next*. Records are extracted from *audit_record_buffer* by starting with record number 0 and iterating through one less than the number of records returned by *xdas_get_next*. If *record_number* does not match a record within *audit_record_buffer* then **XDAS_S_INVALID_RECORD_NUMBER** is returned.

The caller must possess the **XDAS_AUDIT_READ** authority.

xdas_rewind_audit_stream

Read

Rewind an OpenXDAS audit stream read pointer.

Synopsis

```
XDASXPC int XDASAPI xdas_rewind_audit_stream(
    int *                minor_status,
    xdas_audit_ref_t     das_ref,
    xdas_audit_stream_t  audit_stream_ref);
```

Parameters

<i>minor_status</i>	(optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE .
<i>das_ref</i>	The handle of the OpenXDAS session to use for this operation, obtained through a previous call to <i>xdas_initialize_session</i> .
<i>audit_stream_ref</i>	The audit stream handle to be rewound.

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_AUDIT_STREAM** if the specified audit stream handle is not valid.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid audit service session.

Remarks

The *xdas_rewind_audit_stream* function is a member of the Read XDAS conformance class.

The function rewinds the audit stream referred to by *xdas_stream_ref* so that the read cursor associated with the *xdas_stream_ref* points to the first record in the audit stream.

The caller must possess the **XDAS_AUDIT_READ** authority.

xdas_import_event_records

Import

Import event records from an external service into the XDAS common format.

Synopsis

```

XDASXPC int XDASAPI xdas_import_event_records (
    int *                minor_status,
    xdas_audit_ref_t    das_ref,
    xdas_buffer_t       audit_record_buffer,
    size_t *            position_in_buffer);

```

Parameters

<i>minor_status</i>	(optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE .
<i>das_ref</i>	The handle of the OpenXDAS session to use for this operation, obtained through a previous call to <i>xdas_initialize_session</i> .
<i>audit_record_buffer</i>	A pointer to a buffer containing the properly formatted sequence of audit records to be imported.
<i>position_in_buffer</i>	Return storage for the zero-based buffer position at which an import failed, in case a syntax error is detected during the import process.

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid audit service session.

Remarks

The *xdas_import_event_records* function is a member of the Import XDAS conformance class.

This function allows a caller to import audit event records in the XDAS format directly to the OpenXDAS service. The caller places one or more complete audit event records into the buffer referred to by *audit_record_buffer*, from which they are copied and integrated into the OpenXDAS audit stream. The function reads audit records until the start of a next record is not found.

The implementation may select the records that are actually imported based upon some selection criteria. The caller is not advised of the disposition of the audit records it submits.

Records specified in *audit_record_buffer* may be placed end-to-end with no intervening space or separation characters, however this routine will parse records starting with the **HDR** tag and ending with the **END** tag, so intervening characters, or white space are simply ignored.

The caller must possess the **XDAS_AUDIT_IMPORT** authority.

xdas_commit_record**Submit**

Write a completed audit record to the OpenXDAS audit stream.

Synopsis

```
XDASXPC int XDASAPI xdas_commit_record(
    int *                minor_status,
    xdas_audit_ref_t     das_ref,
    xdas_audit_rec_desc_t audit_record_descriptor);
```

Parameters

- | | |
|--------------------------------|--|
| <i>minor_status</i> | (optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE . |
| <i>das_ref</i> | The handle of the OpenXDAS session to use for this operation, obtained through a previous call to <i>xdas_initialize_session</i> . |
| <i>audit_record_descriptor</i> | The handle of the audit record to be committed, obtained through a previous call to <i>xdas_start_record</i> . |

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INCOMPLETE_RECORD** if the audit record has not been fully populated by the caller.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid audit service session.
- ◆ **XDAS_S_INVALID_RECORD_DESCRIPTOR** if the specified audit record descriptor is not valid.
- ◆ **XDAS_S_NOT_SUPPORTED** if the called function is not supported by this implementation.
- ◆ **XDAS_S_SERVICE_FAILURE** if there has been an audit service failure.
- ◆ **XDAS_S_STORAGE_FAILURE** if the audit record cannot be written to stable storage.

Remarks

The *xdas_commit_record* function is a member of the Submit XDAS conformance class.

The function writes the audit record identified by *audit_record_descriptor* to the current audit stream controlled by the audit service and accessed by *das_ref*. The OpenXDAS client library adds the time information to the audit record unless a previous call to *xdas_timestamp_record* has been made using *audit_record_descriptor*. The caller must have the **XDAS_AUDIT_SUBMIT** authority.

If any of the *event_number*, *outcome*, *initiator_information*, *target_information* and *event_information* parameters to *xdas_start_record* and *xdas_put_event_info* have not been completed in at least one such call, even when component fields are empty, then this call returns **XDAS_S_INCOMPLETE_RECORD**.

IMPORTANT: If the record is successfully committed, this routine closes this record handle and the releases any associated system resources.

xdas_discard_record

Submit

Discard a previously created audit record.

Synopsis

```
XDASXPC int XDASAPI xdas_discard_record(
    int *                minor_status,
    xdas_audit_ref_t     das_ref,
    xdas_audit_rec_desc_t audit_record_descriptor);
```

Parameters

- | | |
|--------------------------------|--|
| <i>minor_status</i> | (optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE . |
| <i>das_ref</i> | The handle of the OpenXDAS session to use for this operation, obtained through a previous call to <i>xdas_initialize_session</i> . |
| <i>audit_record_descriptor</i> | The handle of an existing record to be discarded, obtained through a previous call to <i>xdas_start_record</i> |

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid audit service session.
- ◆ **XDAS_S_INVALID_RECORD_DESCRIPTOR** if the specified audit record descriptor is not valid.
- ◆ **XDAS_S_NOT_SUPPORTED** if the called function is not supported by this implementation.

Remarks

The *xdas_discard_record* function is a member of the Submit XDAS conformance class.

The function clears the buffer specified by *audit_record_descriptor* and releases the memory used by it.

The caller must have the **XDAS_AUDIT_SUBMIT** authority.

xdas_put_event_info**Submit**

Add specific event information to an existing audit record.

Synopsis

```

XDASXPC int XDASAPI xdas_put_event_info (
    int *                minor_status,
    xdas_audit_ref_t    das_ref,
    xdas_audit_rec_desc_t audit_record_descriptor,
    unsigned            event_number,
    unsigned            outcome,
    const char *        initiator_information,
    const char *        target_information,
    const char *        event_information);

```

Parameters

<i>minor_status</i>	(optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE .
<i>das_ref</i>	The handle of the OpenXDAS session to use for this operation, obtained through a previous call to <i>xdas_initialize_session</i> .
<i>audit_record_descriptor</i>	The handle of the OpenXDAS audit event record in which to store new or overwrite existing event data, obtained through a previous call to <i>xdas_start_record</i> . Note this is not event-specific data, but rather any event data fields may be populated or repopulated with this function.
<i>event_number</i>	(optional) The event number or event identifier to be associated with this event record. Use zero (0) to indicate “not specified”.
<i>outcome</i>	(optional) The outcome of this event. Use XDAS_OUT_NOT_SPECIFIED to indicate “not specified”.
<i>initiator_information</i>	(optional) A pointer to a zero-terminated string containing UTF-8 colon-delimited string fields of the initiator portion of an XDAS event record. Embedded colon characters must be escaped by preceding it with '%’.
<i>target_information</i>	(optional) A pointer to a zero-terminated string containing UTF-8 colon-delimited string fields of the target portion of an XDAS event record. Embedded colon characters must be escaped by preceding it with '%’.
<i>event_information</i>	(optional) A pointer to a zero-terminated string containing UTF-8 comma-delimited name/value pairs. Embedded commas must be escaped with '%’.

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid audit service session.
- ◆ **XDAS_S_INVALID_EVENT_INFO** if the event specific information given is not valid or not formatted correctly.
- ◆ **XDAS_S_INVALID_EVENT_NO** if the event number specified is not valid.
- ◆ **XDAS_S_INVALID_INITIATOR_INFO** if the initiator information given has a syntax error.
- ◆ **XDAS_S_INVALID_OUTCOME** if the outcome supplied is not valid.
- ◆ **XDAS_S_INVALID_RECORD_DESCRIPTOR** if the specified audit record descriptor is not valid.
- ◆ **XDAS_S_INVALID_TARGET_INFO** if the target information given has a syntax error.
- ◆ **XDAS_S_NO_AUDIT** if the specified event does not need to be audited.
- ◆ **XDAS_S_NO_DECISION_YET** if the audit service has insufficient information to decide if the event requires auditing.

Remarks

The *xdas_put_event_info* is a member of the Submit XDAS conformance class.

The function adds event information to an audit record or overwrites existing information. If the combination of information submitted and already present in the audit record referred to by *audit_record_descriptor* is insufficient to evaluate applicable pre-selection criteria, the function returns **XDAS_S_NO_DECISION_YET** to the caller. If there is sufficient information for evaluation of applicable pre-selection checks the **XDAS_S_COMPLETE** or **XDAS_S_NO_AUDIT** are returned to the caller. Multiple calls to *xdas_put_event_info* may be made. For any individual parameter, information supplied in this call will overwrite any previous information supplied.

Although several parameters are optional in this call, a caller must have populated all of the XDAS record fields, even when empty, in one or more sequences of calls to *xdas_start_record* and *xdas_put_event_info* before a call to *xdas_commit_record* will succeed.

The caller must have the **XDAS_AUDIT_SUBMIT** authority.

If successful, the function returns **XDAS_S_COMPLETE**, **XDAS_S_NO_DECISION_YET** or **XDAS_S_NO_AUDIT**. If **XDAS_S_NO_AUDIT** is returned, then *audit_record_descriptor* is no longer a valid reference to an audit record. If **XDAS_S_NO_DECISION_YET** is returned, then the caller should continue to construct the audit record by making additional calls to *xdas_put_event_info*.

xdas_start_record**Submit**

Create a new OpenXDAS event record, and return a handle to the new record.

Synopsis

```

XDASXPC int XDASAPI xdas_start_record(
    int *                minor_status,
    xdas_audit_ref_t     das_ref,
    xdas_audit_rec_desc_t * audit_record_descriptor,
    unsigned             event_number,
    unsigned             outcome,
    const char *         initiator_information,
    const char *         target_information,
    const char *         event_information);

```

Parameters

<i>minor_status</i>	(optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE .
<i>das_ref</i>	The handle of the OpenXDAS session to use for this operation, obtained through a previous call to <i>xdas_initialize_session</i> .
<i>audit_record_descriptor</i>	Return storage for the handle of a new OpenXDAS record.
<i>event_number</i>	(optional) The event number or event identifier to be associated with this event record. Use zero (0) to indicate “not specified”.
<i>outcome</i>	(optional) The outcome of this event. Use XDAS_OUT_NOT_SPECIFIED to indicate “not specified”.
<i>initiator_information</i>	(optional) A pointer to a zero-terminated string containing UTF-8 colon-delimited string fields of the initiator portion of an XDAS event record. Embedded colon characters must be escaped by preceding it with '%’.
<i>target_information</i>	(optional) A pointer to a zero-terminated string containing UTF-8 colon-delimited string fields of the target portion of an XDAS event record. Embedded colon characters must be escaped by preceding it with '%’.
<i>event_information</i>	(optional) A pointer to a zero-terminated string containing UTF-8 comma-delimited name/value pairs. Embedded commas must be escaped with '%’.

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.

- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid audit service session.
- ◆ **XDAS_S_INVALID_EVENT_NO** if the event number specified is not valid.
- ◆ **XDAS_S_INVALID_INITIATOR_INFO** if the initiator information given has a syntax error.
- ◆ **XDAS_S_INVALID_OUTCOME** if the outcome supplied is not valid.
- ◆ **XDAS_S_INVALID_TARGET_INFO** if the target information given has a syntax error.
- ◆ **XDAS_S_INVALID_EVENT_INFO** if the event specific information given is not valid or not formatted correctly.
- ◆ **XDAS_S_NO_AUDIT** if the specified event does not need to be audited.
- ◆ **XDAS_S_NO_DECISION_YET** if the audit service has insufficient information to decide if the event requires auditing.

Remarks

The *xdas_start_record* function is a member of the Submit XDAS conformance class.

The function returns the handle of a new audit record to the caller. If the optional parameters are not specified in the call, then the audit record is initialized, but requires further population by subsequent calls to *xdas_put_event_info*.

If the optional parameters are specified, the function determines whether a specified event should be audited, given the *event_number*, *outcome* and *initiator_information* supplied. If the event should be audited a valid record handle is returned to the caller. If the audit event does not require auditing then *audit_record_descriptor* is returned containing **NULL**. The caller must have the **XDAS_AUDIT_SUBMIT** authority.

Although several parameters are optional in this call, a caller must have populated all of the parameters, even when empty, in one or more sequences of calls to this function and *xdas_put_event_info* before a call to *xdas_commit_record* will succeed.

xdas_timestamp_record**Submit**

Record the current time as a time stamp on an existing OpenXDAS event record.

Synopsis

```
XDASXPC int XDASAPI xdas_timestamp_session(
    int *                minor_status,
    xdas_audit_ref_t     das_ref,
    xdas_audit_rec_desc_t audit_record_descriptor);
```

Parameters

- | | |
|--------------------------------|---|
| <i>minor_status</i> | (optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE . |
| <i>das_ref</i> | The handle of the OpenXDAS session to use for this operation, obtained through a previous call to <i>xdas_initialize_session</i> . |
| <i>audit_record_descriptor</i> | The handle of the OpenXDAS audit event record to be stamped with the current time, obtained through a previous call to <i>xdas_start_record</i> . |

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid audit service session.
- ◆ **XDAS_S_INVALID_RECORD_DESCRIPTOR** if the specified audit record descriptor is not valid.

Remarks

The *xdas_timestamp_record* function is a member of the Submit XDAS conformance class.

The function puts a time stamp on the audit record supplied.

The *xdas_commit_session* function will not time-stamp a record that has already been time-stamped using this function. However, it will always time-stamp a record that has not been time-stamped already. This function may also be used to re-record the time stamp of an existing OpenXDAS record that has already been stamped through a previous call to this function.

The caller must have the **XDAS_AUDIT_SUBMIT** authority.

xdas_create_filter**Manage**

Create a new named OpenXDAS audit filter.

Synopsis

```

XDASXPC int XDASAPI xdas_create_filter(
    int *                minor_status,
    xdas_audit_ref_t    das_ref,
    const char *        name,
    unsigned            filter_type,
    const char *        filter_exp,
    const char *        filter_act);

```

Parameters

<i>minor_status</i>	(optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE .
<i>das_ref</i>	The handle of the OpenXDAS session to use for this operation, obtained through a previous call to <i>xdas_initialize_session</i> .
<i>name</i>	The name of the filter to be created.
<i>filter_type</i>	The type of the filter to be created. This may be either XDAS_C_SUBMIT or XDAS_C_IMPORT .
<i>filter_exp</i>	The expression list which defines the criteria for detection of the event for the new filter.
<i>filter_act</i>	The list of actions associated with this filter which are to be taken when the event is submitted or imported.

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_ACTION_LIST** if the action list specified is not valid.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid audit service session.
- ◆ **XDAS_S_INVALID_FILTER** if the filter name specified already exists or the name parameter is **NULL**.
- ◆ **XDAS_S_INVALID_FILTER_EXP** if the filter expression supplied is not valid or the *filter_exp* parameter is **NULL**.
- ◆ **XDAS_S_INVALID_FILTER_TYPE** if the filter type specified is not recognized.
- ◆ **XDAS_S_INVALID_FILTER_ACTION** if the filter action specified is not recognized or the *filter_act* parameter is **NULL**.

Remarks

The *xdas_create_filter* function is a member of the Manage XDAS conformance class.

The function creates a filter for the name specified. If a filter with the specified name already exists, the call fails. On creation, the filter is in a disabled state.

The caller must possess the `XDAS_AUDIT_CONTROL` authority.

xdas_delete_filter**Manage**

Delete an existing OpenXDAS audit filter by name.

Synopsis

```
XDASXPC int XDASAPI xdas_delete_filter(
    int *                minor_status,
    xdas_audit_ref_t    das_ref,
    const char *        name);
```

Parameters

<i>minor_status</i>	(optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE .
<i>das_ref</i>	The handle of the OpenXDAS session to use for this operation, obtained through a previous call to <i>xdas_initialize_session</i> .
<i>name</i>	The name of the filter to be deleted.

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid audit service session.
- ◆ **XDAS_S_INVALID_FILTER** if the filter name specified already exists.

Remarks

The *xdas_delete_filter* function is a member of the Manage XDAS conformance class.

The function deletes the filter defined by *name* from the OpenXDAS system. This may involve deleting copies of the filter from all agents managed via a particular instance of the XDAS interface. The function does not wait upon the successful deletion of all instances of the filter maintained by OpenXDAS agents.

The caller must possess the **XDAS_AUDIT_CONTROL** authority.

xdas_disable_filter**Manage**

Disable an existing OpenXDAS audit filter by name.

Synopsis

```
XDASXPC int XDASAPI xdas_delete_filter(  
    int *                minor_status,  
    xdas_audit_ref_t     das_ref,  
    const char *         name);
```

Parameters

<i>minor_status</i>	(optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE .
<i>das_ref</i>	The handle of the OpenXDAS session to use for this operation, obtained through a previous call to <i>xdas_initialize_session</i> .
<i>name</i>	The name of the filter to be disabled.

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid audit service session.
- ◆ **XDAS_S_INVALID_FILTER** if the filter name specified already exists.

Remarks

The *xdas_disable_filter* function is a member of the Manage XDAS conformance class.

The function disables the filter defined by *name* from the OpenXDAS system. It sets the state of the filter to disabled. If necessary the disabled state of the filter may require propagation to all XDAS agents managed by a particular instance of the XDAS interface. The function does not wait upon the successful disabling of all instances of the filter maintained by XDAS agents.

The caller must possess the **XDAS_AUDIT_CONTROL** authority.

xdas_enable_filter**Manage**

Enable an existing OpenXDAS audit filter by name.

Synopsis

```
XDASXPC int XDASAPI xdas_enable_filter(  
    int *                minor_status,  
    xdas_audit_ref_t    das_ref,  
    const char *        name);
```

Parameters

<i>minor_status</i>	(optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE .
<i>das_ref</i>	The handle of the OpenXDAS session to use for this operation, obtained through a previous call to <i>xdas_initialize_session</i> .
<i>name</i>	The name of the filter to be enabled.

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid audit service session.
- ◆ **XDAS_S_INVALID_FILTER** if the filter name specified already exists.

Remarks

The *xdas_enable_filter* function is a member of the Manage XDAS conformance class.

The function enables the filter corresponding to the name specified. If necessary the enabled state of the filter may require propagation to all OpenXDAS agents managed by a particular instance of the XDAS Interface. The function does not wait upon the successful enabling of all instances of the filter maintained by OpenXDAS agents.

The caller must possess the **XDAS_AUDIT_CONTROL** authority.

xdas_get_filter**Manage**

Get an existing OpenXDAS audit filter by name.

Synopsis

```

XDASXPC int XDASAPI xdas_get_filter(
    int *                minor_status,
    xdas_audit_ref_t    das_ref,
    const char *        name,
    unsigned *          filter_type,
    xdas_buffer_t       filter_exp,
    xdas_buffer_t       filter_act,
    unsigned *          filter_status);

```

Parameters

<i>minor_status</i>	(optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE .
<i>das_ref</i>	The handle of the OpenXDAS session to use for this operation, obtained through a previous call to <i>xdas_initialize_session</i> .
<i>name</i>	The name of the filter whose attributes should be returned.
<i>filter_type</i>	(optional) Return storage for the type of the filter. This may either be XDAS_C_SUBMIT or XDAS_C_IMPORT . If this value is not desired, pass zero.
<i>filter_exp</i>	(optional) Return storage for the contents of the filter expression that determines the events to be selected by this filter. If this value is not desired, pass zero.
<i>filter_act</i>	(optional) Return storage for the contents of the filter action list that contains the actions to be carried out for events selected by this filter. If this value is not desired, pass zero.
<i>filter_status</i>	(optional) Return storage for the enabled or disabled state of the filter. If the filter is enabled a boolean value of true (1) is returned, otherwise a boolean value of false (0) is returned. If this value is not desired, pass zero.

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid audit service session.

- ◆ `XDAS_S_INVALID_FILTER` if the filter name specified already exists.

Remarks

The *xdas_get_filter* function is a member of the Manage XDAS conformance class.

The function returns the components of the filter referred to by name.

The caller must possess the `XDAS_AUDIT_CONTROL` authority.

xdas_list_filters**Manage**

Return a list of all existing OpenXDAS audit filter names as a null-terminated character pointer list.

Synopsis

```

XDASXPC int XDASAPI xdas_list_filter(
    int *                minor_status,
    xdas_audit_ref_t    das_ref,
    char **             filter_name_list,
    size_t *           buffer_size);

```

Parameters

<i>minor_status</i>	(optional) Return storage for an implementation-specific minor status code, if the return value is XDAS_S_FAILURE .
<i>das_ref</i>	The handle of the OpenXDAS session to use for this operation, obtained through a previous call to <i>xdas_initialize_session</i> .
<i>filter_name_list</i>	Return storage for the filter name list, formatted as a zero-terminated list of names. This buffer will contain both the names and an array of pointers to the names. The caller may simply free the buffer when done if it was allocated originally on the heap.
<i>buffer_size</i>	On entry, contains the size in bytes of the <i>filter_name_list</i> buffer. Also acts a return storage for the number of bytes consumed or required for the complete name list.

Return Value

- ◆ **XDAS_S_COMPLETE** on successful completion.
- ◆ **XDAS_S_AUTHORIZATION_FAILURE** if the caller is not authorized to initialize an XDAS session.
- ◆ **XDAS_S_FAILURE** if an implementation specific error or failure has occurred.
- ◆ **XDAS_S_INVALID_DAS_REF** if the audit service handle supplied does not represent a valid audit service session.
- ◆ **XDAS_S_INVALID_FILTER_LIST** if the filter name list buffer is **NULL**, but the size is NOT zero.
- ◆ **XDAS_S_BUFF_TOO_SMALL** if the *filter_name_list* buffer is too small to return all of the available filter names.

Remarks

The *xdas_list_filters* function is a member of the Manage XDAS conformance class.

The function yields a zero-terminated array of pointers to filter names. The memory for holding the array of pointers and the name filter buffers is allocated by the caller and passed in the *filter_name_list* parameter. If the buffer specified in this pointer is

insufficient, as specified in the *buffer_size* parameter on input, the function will return the required buffer size in *buffer_size*. The caller should reallocate the buffer and call this routine again.

This routine will fill the buffer to capacity regardless of whether there is sufficient space or not for all filter names. However, an effective way to manage this function call is to call it twice, first with a null *filter_name_list* buffer pointer and a valid *buffer_size* parameter containing zero, and then again with a *filter_name_list* buffer allocated to the size specified in *buffer_size* on return from the previous call.

NOTE: The proper use of this function is not exactly obvious. The *filter_name_list* parameter actually points to a raw buffer of bytes on entry. On return, this buffer may be interpreted as its true type - a zero-terminated array of character pointers. The space in this buffer that follows the array is used to return the actual string data. The pointers in the pointer array refer to the space at the end of the buffer on return.

The caller must possess the `XDAS_AUDIT_CONTROL` authority.

The OpenXDAS Java API

The Java client API is implemented entirely in Java. **No Java Native Interface (JNI) code is used in the Java client library.** The Java client library uses the same local inter-process communication (IPC) mechanism that the C client library uses to communicate with the *xdasd* daemon/service running on the local host.

The Java API is designed in the same spirit as the C API, except where it makes more sense to follow general interface conventions defined by the Java community. Since the XDAS preliminary specification does not specifically define a Java API, OpenXDAS has license to define this interface according to its authors' desires, within the boundaries of the spirit of the XDAS specification.

The Java jar file used by interface clients is named simply *openxdas.jar*. The most accurate interface documentation is found in the javadoc html pages accessible at the OpenXDAS web site at <http://openxdas.sourceforge.net/documentation>. To access the javadoc documentation directly use this link:

<http://openxdas.sourceforge.net/javadoc/doc/index.html>

The general usage of the openxdas Java interfaces is very simple: The caller creates a new *XDasSession*, either with a single pre-escaped originator string by using

```
XDasSession(java.lang.String sOriginator);
```

or with individual originator string components, which need not be escaped, through the more complex:

```
XDasSession(java.lang.String sOriginatorLocationName,  
             java.lang.String sOriginatorLocationAddress,  
             java.lang.String sOriginatorServiceType,  
             java.lang.String sOriginatorAuthAuthority,  
             java.lang.String sOriginatorPrincipalName,  
             java.lang.String sOriginatorPrincipalIdentity);
```

Once a session has been established (and this is generally done once in the application's initialization code), then the *XDasStartRecord* method of the *XDasSession* object is used to create *XDasRecord* objects:

```
XDasStartRecord(int iEventNumber,  
                 int iOutcome,  
                 java.lang.String sInitiatorInfo,  
                 java.lang.String sTargetInfo,  
                 java.lang.String sEventInfo);
```

This is done once for each audit event that needs to be recorded. The parameters are optional for this factory method, and may be passed as 0 or NULL in order to specify them later. They may also be changed later at any point with additional methods on the *XDasRecord* object.

When the application is ready to commit the record, the application should call the commit method of the XDasRecord object:

```
void commit ();
```

This method pushes the record out to the local *xdasd* service.

Within the XDasRecord object, there are two kinds of setter methods. If the application writer wishes to use a more compact format for setting data, he or she may use the pre-escaped string setters. These setters are:

```
void putEventInfo (int iEventNumber,  
                 int iOutcome,  
                 java.lang.String sInitiatorInfo,  
                 java.lang.String sTargetInfo,  
                 java.lang.String sEventInfo);
```

```
void setEventInfo (java.lang.String sEventInfo);
```

```
void setInitiatorInfo (java.lang.String sInitiatorInfo);
```

```
void setTargetInfo (java.lang.String sTargetInfo);
```

These method require that you pre-escape the strings passed such that there are no unescaped embedded colon characters. In addition, the `setEventInfo` method requires that you escape the `sEventInfo` parameter such that there are no unescaped equal (=) or comma (,) characters, as these characters are used as delimiters within the event info data string.

If you wish to use unescaped strings for the initiator and target information strings, you may use these more verbose forms of the `setInitiatorInfo` and `setTargetInfo` methods:

```
void setInitiatorInfo (java.lang.String sAuthAuthority,  
                    java.lang.String sDomainSpecificName,  
                    java.lang.String sDomainSpecificId);
```

```
void setTargetInfo (java.lang.String sTargetLocationName,  
                 java.lang.String sTargetLocationAddress,  
                 java.lang.String sTargetServiceType,  
                 java.lang.String sTargetAuthAuthority,  
                 java.lang.String sTargetPrincipalName,  
                 java.lang.String sTargetPrincipalIdentity);
```

What Should I Audit?

If, as you read the descriptions of the XDAS generic events and event classes, you felt that these events seem to be heavily tailored toward operating system management, then you are right. Recall that the purpose of XDAS is to provide a distributed auditing service – a system of auditing an entire security domain, as opposed to a single authentication domain or realm (perhaps a corporate or enterprise network, for example) for security-relevant events. Most of the security-relevant events within a networked environment revolve around system access issues for systems that manage protected resources.

This doesn't mean that OpenXDAS can't or shouldn't be used to audit non-system applications, but bear in mind, as you instrument your applications, which events are of security relevance, and which are just nice to know about.

Using the OpenXDAS Client Library

OpenXDAS provides client libraries in both C and native Java. The C library provides a great access point for multiple other languages that have natural bindings to C, and the Java client library, being pure Java code provides Java developers with that special secure feeling that comes from using only Java code in an application. This section discusses the proper use of these libraries by C and Java developers.

Using the C Client Library

The C library comes in the forms of either shared or static link libraries. On Unix/Linux systems these libraries are usually named *libxdas.so* (shared) and *libxdas.a* (static). On Win32 systems, the libraries are named *libxdas.dll* and *libxdas.lib*.

Be careful on Win32 to not confuse the DLL stub library (*libxdas.lib*) with the static library (*libxdas.lib*). The stub library is just a catalog of names exported from the DLL, and as such is much smaller than the static library. This is really the only quick way to tell them apart. The stub library is used to link your application to the DLL, and is usually around 2 or 3 kilobytes in size.

On Windows, in order to use the static library, you will also have to define **XDAS_STATIC** on your compiler command line for each source file that includes the *xdas.h* header file. Since this variable will likely only affect the inclusion of the *xdas.h* header file, it's therefore simpler to just define **XDAS_STATIC** on the command line for all of your source files, but it's really your choice. This definition tells the compiler the correct names to use for the *xdas* client library functions for the type of library to which you will be linking your application (static or dynamic)..

Using the Java Client Library

The Java client library comes in the form of a standard *.jar* file or, (OpenXDAS being open source software) the *.java* source files may also be used directly.

The name of this *jar* file changes with each new version of OpenXDAS that is released. It will generally be called after this pattern: *openxdas-0.4.226.jar*. This presents a problem to users because they will be expecting to install OpenXDAS from a late distribution, and have your Java application simply work against the *jar* file. But if your application was linked against *openxdas-0.4.226.jar*, and they've installed a later version – say *openxdas 0.5.376*, they will have a *jar* file called *openxdas-0.5.376.jar*, and the JVM will complain.

To solve this problem on Unix/Linux systems, we recommend that you drop *openxdas-0.4.226.jar* (or whatever version you have currently) into your */usr/share* directory, and then create a link in the same directory using the following command:

```
ln -s openxdas-0.4.226.jar /usr/share/openxdas.jar
```

On Win32 systems, you will need to rename or copy the *openxdas-0.4.226.jar* file at installation time to *openxdas.jar*.

Writing an OpenXDAS Logger Module

OpenXDAS provides a modular expandable interface for flexibly sending audit records to just about any form of data sink.

Currently, OpenXDAS provides 5 distinct logger modules whose base file names are **file**, **laf**, **odbc**, **syslog**, and **xdas**. The file logger is designed to log audit records to a simple file, the name and location of which may be specified using an *xdasd* configuration file option. The laf logger is designed to log audit records to the Linux Lightweight Auditing Framework API. The odbc logger logs audit records to an SQL database via the *Open Data Base Connectivity* API. The syslog logger logs messages to the system log. On Unix/Linux systems, this is the syslog facility. On Windows systems, the system log is the Win32 Event System Application Log. Finally, the xdas logger is designed to use the xdasd wire protocol to forward audit records to a remote xdas service. This aspect of OpenXDAS is currently still in the design phase, and so the xdas logger doesn't currently actually do anything – it acts as a null logger for the time being.

The Nuts and Bolts

Writing a logger module is a fairly trivial process for the seasoned C programmer. There are three well-known functions which must be exported from the Unix/Linux shared library or the Win32 dynamic link library (dll):

```
XDMEXP int XDMAPI xdm_append(const char ** msgflds);
XDMEXP int XDMAPI xdm_init(
    void (*logmsg)(int level, const char * msg, ... ),
    char * (*getcnfstr)(const char *, char *, size_t *));
XDMEXP void XDMAPI xdm_exit(void);
```

These three functions provide all of the required access to logger functionality to xdasd. The *xdm_init* function is called once at the time xdasd loads the logger module. The *xdm_exit* function is called once as the logger module is being unloaded—usually at the time xdasd itself is being stopped. These allow the module a chance to perform any global setup and tear down that may be required.

The *xdm_init* function accepts some important parameters that help a module interact with its environment within the xdasd service process. The *logmsg* parameter is a pointer to a log function that acts like *printf*—in fact, *exactly* like *printf*, except for the leading log-level parameter. The level parameter is used to specify the logging level for which a particular message should be logged. The system administrator can specify the logging level on the xdasd command line, or in the environment with the XDASD_LOG_LEVEL variable. The default logging level is zero, so all messages logged with a log level of zero are always logged.

The *logmsg* function pointer may be stored by the logger module in a module static or global variable so that it can be called later by the other functions if required. The *logmsg* function will send all formatted information directly to the *xdasd log* (as long as the specified message log level value is less than or equal to the current system log level). It's a good idea to preface your log messages with your module's name so that a reader will know where the message came

from, if it's not clear from the context of the message.

The second parameter, *getcnfstr*, is a pointer to a function that returns a configuration parameter value from a configuration parameter name. This can be used to allow your logger to access configuration parameters in the *xdasd* configuration file.

The third function – *xdm_append* – is called once for each message that should be logged by the logger. The *xdasd* service passes an array of 34 character pointers to this function in the *msgflds* parameter. Each character pointer in the array points to a field of the audit record to be logged. There are 33 fields in a standard XDAS audit record. The first field points to the very beginning of the record. For instance:

```
msgflds[0] → HDR:
msgflds[1] → 00A8:
msgflds[2] → OX1:
msgflds[3] → 2234934821:
msgflds[4] → tm_uncert_int:
msgflds[5] → tm_uncert_ind:
msgflds[6] → time.nist.gov:
msgflds[7] → MST7MDT:
msgflds[8] → 0x10000001:
msgflds[9] → 0:
msgflds[10] → ORG:
msgflds[11] → org_location_name:
msgflds[12] → org_location_address:
msgflds[13] → org_service_type:
msgflds[14] → org_auth_authority:
msgflds[15] → org_principal_name:
msgflds[16] → org_principal_id:
msgflds[17] → INT:
msgflds[18] → int_auth_authority:
msgflds[19] → int_domain_specific_name:
msgflds[20] → int_domain_specific_id:
msgflds[21] → TGT:
msgflds[22] → tgt_location_name:
msgflds[23] → tgt_location_address:
msgflds[24] → tgt_service_type:
msgflds[25] → tgt_auth_authority:
msgflds[26] → tgt_principal_name:
msgflds[27] → tgt_principal_id:
msgflds[28] → SRC:
msgflds[29] → pointer_to_source_domain:
msgflds[30] → EVT:
msgflds[31] → event_specific_information:
msgflds[32] → END
msgflds[33] →
```

Contrary to the way this diagram looks, these fields are NOT null terminated. In fact, if you were to print out the value of the string pointed to by *msgflds[0]*, you'd get the entire audit record, right up to the end of *msgflds[32]*.

Additionally, `msgflds[32]` IS null-terminated. So if your logger merely needs a null-terminated UTF-8 string, you can just use the value of `msgflds[0]` as a reference to the start of the null-terminated audit record.

So if `msgflds[0]` through `msgflds[32]` point to each of the record fields, then what's `msgflds[33]` for? It points to the byte *following* the null-terminator after the last field. The reason for this is so that all fields can be treated the same by a field processor. If you need to parse and examine various fields of the record in your logger, you can treat the last field (32) just like the first (0). That is, you can rest assured that the relationship between field 1 and field 0 is exactly the same as the relationship between field 33 and field 32. The algorithm for accessing field data is simple: The length of field X can be calculated as `msgflds[X + 1] - msgflds[X] - 1`. We subtract one to remove the trailing separator, or in the case of the last field, the trailing null-terminator.

An Example Logger

The following code snippet is a complete logger skeleton, which you may cut and paste as a starting point for writing your own loggers:

```

/* Source for the name xdasd logger module. */

#ifdef _WIN32
# define XDMAPI __cdecl
# define XDMEXP __declspec(dllexport)
#else
# define XDMAPI
# define XDMEXP
#endif

static void (*s_fplogmsg)(const char * msg, ... );
static char * (*s_fpgetcnfstr)(const char *, char *, size_t *);

XDMEXP int XDMAPI xdm_append(const char ** msgflds)
{
    return 0; /* return zero to indicate success */
}

XDMEXP int XDMAPI xdm_init(void (*logmsg)(const char * msg, ... ),
    char * (*getcnfstr)(const char *, char *, size_t ))
{
    s_fplogmsg = logmsg; /* save these off to be used later */
    s_fpgetcnfstr = getcnfstr;

    return 0; /* return zero to indicate success */
}

XDMEXP void XDMAPI xdm_exit(void) { }

```

When you link your module, name it **libxdm_name.so** on Unix/Linux systems (unless you happen know your system is different with respect to shared library naming conventions – in this case, check out the *README* file in the `xdasd/modules` source directory for more specific information regarding your Unix system type). On Win32 systems, name your module **xdm_name.dll**. Set *name* to anything you desire (as long as it doesn't conflict with other

existing logger module names).

Using Your New Logger

To load your module, use the *xdasd.loggers* configuration variable in the *xdasd.conf* file, like this:

```
xdasd.loggers = c:\program files\openxdas\loggers\xdm_name.dll, \  
c:\program files\openxdas\loggers\xdm_odbc.dll, \  
c:\program files\openxdas\loggers\xdm_syslog.dll, ...
```

You can add up to 16 loggers to this list, separated by commas. Note that we're using the line wrapping feature of the *xdasd* configuration file in this example to make the text more readable in the file. The backslash characters terminating the first two lines transform these three lines into a single configuration parameter.