OSF DCE SIG
Request For Comments: 29.2

S. Luan (VDG)
R. Weisz (IBM)
October 1994

DESIGN OF AN AUDIT SUBSYSTEM FOR DCE — IMPLEMENTATION SPECIFICATION

1. INTRODUCTION

Audit is one of the high-priority security features to be supported in DCE 1.1 and later versions as voted by the DCE Security Working Group [RFC 8.0]. In a companion RFC [RFC 28.1], we identify the auditable events in the DCE Security and Time Servers by interpreting the NCSC Class-C2 Requirements [NCSC 85, NCSC 87, NCSC 88], and also provide a functional specification of an audit-logging API used by servers to log the identified auditable events.

In this RFC, we present a proposed implementation design for the DCE audit subsystem, which includes the audit-logging API specified in [RFC 28.1], an audit daemon, and an audit control program. The audit daemon will be run on each server host. The daemon maintains event-selection information for the servers on the same host and provides an RPC interface to the servers for event logging. The audit control program is used as an administrator interface to the audit daemon for configuring and querying event-selection information.

The rest of this document is organized into 6 sections and 2 appendices. In Section 2, we describe the proposed DCE-audit functionalities. In Section 3, we explain why local-OS audit subsystems cannot be used to provide the proposed DCE-audit functionalities. In Section 4, we describe the encoding schemes of events and event classes. In Section 5, we specify the formats of the files used by the proposed DCE-audit subsystem. In Section 6, we explain how these files are maintained by the audit daemon. In Section 7, we describe how the proposed audit API interacts with the audit daemon. Appendices A and B contain the manual pages for the audit control program and the audit daemon.

2. PROPOSED FUNCTIONALITIES

The proposed functionalities of the DCE audit subsystem include:

(a) *Event Logging*: An audit logging API (Application Programming Interface) should be provided that is used (1) to record auditable events at some specified or default destination, (2) to retrieve and print audit records that match certain specified criteria (predicates). The API has been specified in [RFC 28.1].

(b) *Event Selection (Filtering)*: A command interface through which a system administrator can select the events of a subject (principal, group, or cell) to be logged. A filtering mechanism will determine whether an event should be logged, given the identity of the client[1] that invokes the event, and the event outcome (success, access denial, or failure).

(c) *Configurable Event Classes*: Event classes are defined to provide an efficient mechanism by which sets of events can be logically grouped and selected in an event list by means of a single value. Event classes shall be configurable; that is, a system administrator can add/remove events of an existing event class or define new event classes. Non-configurable event classes (e.g., code-point hardcoded event classes suggested in [RFC 25.0][2]) would make event-selection mechanisms impractically inefficient. System administrators will

---

1. The functionality to allow a system administrator to selectively audit the actions of any one or more users based on individual identity is a TCSEC Class C2 requirement.

2. [RFC 25.0] suggests the use of facility tables of DCE 1.1 Serviceability design for event-class definitions. Each event class has an entry in a facility table, which specifies whether that event class is selected to be audited. At an event's code-point in the server program, checks are done to determine whether any of a set of *compile-time assigned* event classes to which the event belongs, is selected. Thus, event classification is fixed at compile time. To reconfigure event classes, a system administrator has to ask the vendor to have the server modified and recompiled!

have to select multiple event classes even if they are only interested in a small subset of events in each of the selected classes.

(d) *Critical Event Notification (Alarm Raising)*: The event selection interface should also allow a system administrator to direct selected classes of critical events to the system console.

3. CONSIDERATION OF USING EXISTING OS AUDIT FACILITIES

We examined the possibility of using existing local-OS auditing facilities for DCE auditing, and identified some problems with this approach in achieving the design goals.

(a) Local-OS auditing subsystems are only able to offer audit event selection based on local subject/object identities. Thus, the DCE audit event selection cannot be based on the event selection mechanisms of local audit subsystems. For example, a local audit subsystem would not allow events to be selected on a per DCE principal, group, cell, or object basis.

(b) The separation of DCE audit from local audit simplifies the porting efforts that are required on different platforms. If the DCE audit-subsystem relies on local-OS audit sub-systems, it has to use a generic interface that can be translated into audit-system calls on all platforms where DCE servers may be installed. This is a non-trivial task, if at all possible.

(c) It would be redundant to include all local audit header information in DCE-audit event records. The header information includes local audit specific information such as the process id, local user ids, group ids, effective (local) subject privileges, etc. Recording such information for each DCE-audit event is unnecessary because DCE-audit events occur as a result of the DCE TCB interface access, not a local TCB interface access. For example, an access to a p/g/o item in the registry database generally would not cause the registry server to access any local TCB interface, because the entire registry database is read into the registry server's address space when the server starts. In case a DCE TCB interface access results in any local TCB interface access, the local TCB access will be audited by the local audit subsystem. Correlation of accesses to local TCB interface and DCE TCB interface can be established through an analysis of the two audit trails together. Recording local-audit information when there is no access to the local TCB interface is not useful.

4. ENCODING OF EVENT TYPES AND EVENT CLASSES

The goal of the event-type and event-class encoding schemes is to allow extensibility and flexibility in defining and classifying DCE auditable events. Events and event classes are represented with 32-bit integers, instead of UUID's, which require additional 12 bytes. [Also, using UUID's would make it difficult to map an event number to the identity of the server for which the event is defined. A simple mapping from event numbers to server identities facilitates the reading of event class configuration files described in Section 5.]

The schemes (described in the following two sub-sections) used for event-number and class-number assignments are similar to the IP address-assignment scheme.

4.1. Event Numbers

An event number encodes the identification of an event set as well as the identification of the unique event. A set of event numbers is assigned by the OSF (upon request) to an organization or a vendor. The organization or vendor then has the authority to use event numbers within that set.

Conceptually, each event number is a pair (`set-id`, `event-id`), where `set-id` identifies an event set, and the `event-id` identifies an event within the event set. In practice, each event number must have one of the following formats.

```
            0 1 2 3 4    8      16      24      31
+-------------------------------------------------+
| Format A |0|  set-id  |       event-id          |
+-------------------------------------------------+
| Format B |1|0|    set-id      |   event-id      |
+-------------------------------------------------+
| Format C |1|1|0|       set-id         |event-id|
+-------------------------------------------------+
| Format D |1|1|1|0|          event-id            |
+-------------------------------------------------+
| Format E |1|1|1|1|          reserved            |
+-------------------------------------------------+
```

Given an event number, its format can be determined from its four high-order bits. Format-A event numbers, which are allocated to organizations such as the OSF itself and major vendors which need more than 16 bits for event-number assignment, devote 7 bits to `set-id` and 24 bits to `event-id`. Format-B event numbers are allocated to intermediate-size vendors which need 8 to 16 bits for event-number assignment. Format-C event numbers are allocated to small-size vendors which need less than 8 bits for event number assignments. Format-D event numbers are not administered by the OSF and can be used freely for DCE cell-specific events. The use of these event numbers might not be unique across cells and should be avoided by servers which may be installed in more than one cells. Format-E event numbers are reserved for future use.

Format-A event numbers with 0 (zero) as `set-id` are assigned to the OSF. That is, all events numbers used by the OSF have a zero in the most significant byte.

```
            0 1 2 3 4    8      16      24      31
+-------------------------------------------------+
| Format A |0|0 ...... 0|     OSF event-id        |
+-------------------------------------------------+
```

The following is a sample of the proposed event number assignment for DCE base events. The event numbers shall be defined in a header file included by the Security Server and Time Server programs respectively.

```
/* Event numbers 0x00000100 to 0x000001FF are assigned to the
   Security Server. */

#define AS_Request         0x00000100
#define TGS_TicketReq      0x00000101
#define TGS_RenewReq       0x00000102
#define TGS_ValidateReq    0x00000103
...

/* Event numbers 0x00000200 to 0x000002FF are
   assigned to the Time Server. */

#define CNTRL_Create       0x00000200
#define CNTRL_Delete       0x00000201
#define CNTRL_Enable       0x00000202
#define CNTRL_Disable      0x00000203
...
```

The following is a hypothetical example of the event-number assignment by a vendor, where `vn` stands for the name of the vendor.

```
#define ET_vn_BANK_SERVER_ACCT_OPEN      0x01000000
#define ET_vn_BANK_SERVER_ACCT_CLOSE     0x01000001
#define ET_vn_BANK_SERVER_ACCT_WITHDRAW  0x01000002
#define ET_vn_BANK_SERVER_ACCT_DEPOSIT   0x01000003
```

Organizations/Vendors shall administer the event numbers assigned to them and maintain the unique assignment of event numbers to events in their products.

4.2. Event Classes and Event Class Numbers

The goal of event classes is to provide an efficient mechanism by which sets of events can be logically grouped and specified in an event list by means of a single value. Usually, event classes are defined for events that have some commonality; e.g., Authentication event class and Account-Operation event class.

Similar to event numbers, event-class numbers encode the identification of an event-class set as well as the identification of a unique event class within that set. A set of event-class numbers is assigned (upon request) by the OSF to an organization or a vendor. The organization or vendor then has the authority to use the the event-class numbers within that set.

Conceptually, each event class number is a pair (`set-id`, `class-id`), where `set-id` identifies an event-class set, and the `class-id` identifies an event class within in the set. In practice, each event-class number must have one of the following forms.

```
                0 1 2 3 4    8       16       24       31
     +-------------------------------------------------+
     | Format A |0|     set-id       |  class-id       |
     +-------------------------------------------------+
     | Format B |1|0|        set-id          |class-id|
     +-------------------------------------------------+
     | Format C |1|1|0|         class-id               |
     +-------------------------------------------------+
     | Format D |1|1|1|         reserved               |
     +-------------------------------------------------+
```

Given an event-class number, its format can be determined from its three high-order bits. Format-A class numbers, which are allocated to organizations/vendors which need more than 8 bits for event-class number assignment, devote 14 bits to `set-id` and 16 bits to `class-id`. Format-B class numbers are allocated to vendors which do not need more than 8 bits for event-class number assignment. Format-C class numbers are not administered by the OSF and can be used freely for DCE cell-specific event classes. Format-D numbers are reserved for future use.

Some event classes will be defined by vendors for product-specific events, thus: EC_*VendorA_ProductX_EventClass1*. Event classes which encompass multiple vendors will be defined by the OSF (e.g., `EC_OSF_C2_Configuration`), or by cell administrators who can tailor the definition to satisfy the specific requirements of their cells.

Format-A class numbers with 0 (zero) as the `set-id` are assigned to OSF itself. That is, all event-class numbers used by the OSF have a zero in the two most significant bytes.

```
                0 1 2 3 4    8       16       24       31
     +-------------------------------------------------+
     | Format A |0|0 .............. 0|  class-id       |
     +-------------------------------------------------+
```

The following is the proposed event-class number assignment for DCE C2 event classes [RFC 28.1], which are defined for TCSEC C2 requirements.

```
EC_OSF_C2_Authentication     0x00000000
EC_OSF_C2_Object_Deletion    0x00000001
EC_OSF_C2_Security_State     0x00000002
EC_OSF_C2_Controlled_Access  0x00000003
EC_OSF_C2_Network_Exception  0x00000004
EC_OSF_C2_Cryptographic      0x00000005
EC_OSF_C2_Configuration      0x00000006
```

5. FILE FORMAT

5.1. Event Class Configuration Files

Each event class is defined in a separate event class configuration file, which always contains an event-class number and a list of events. All event class configuration files reside in directory `/opt/dcelocal/etc/audit/ec/`, and they need to be protected by the local OS (i.e., only administrators have read and write accesses to these files). Servers need to read these files once in order to initialize an event table at runtime (described in Section 7). Optionally, an event class configuration file also contains a list of event number prefixes of the events in this class. This information is useful if the event list is very long. Servers which do not have events with one of the prefixes listed need not scan the event list. If the prefix list is not provided in the file, servers will have to read the entire file to find out if the class contains any of their events.

The following example is the configuration file of `EC_OSF_C2_Authentication` event class. The contents of file `/opt/dcelocal/etc/audit/ec/EC_OSF_C2_Authentication` are listed below.

```
# Event-class number of EC_OSF_C2_Authentication:

ECN = 0x00000000

# Event numbers in this event class have one of the
# following prefixes.  "SEP" (Server Event number Prefixes)
# information is optional.  Format: SEP = x y z,
# where x y z are the number prefixes.

# Security Server event numbers start from 0x100

SEP = 0x100

# The following is a list of events which
# belong to this event class.

# Security Daemon Events:

# AS_Request
0x00000100
# TGS_TicketReq
0x00000101
# TGS_RenewReq
0x00000102
# TGS_ValidateReq
0x00000103
```

5.2. Event Filters

Event filters are maintained by an audit daemon for event selection. Filters represent the selected event classes for principals, groups, cells, or world (wildcard). There exist eight types of filters — `principal`, `foreign_principal`, `group`, `foreign_group`, `cell`, `cell_overridable`, `world`, `world_overridable`. All filter types, except the two world types, require a key to identify an instance of filter. A key is the name of a principal, foreign principal, group, foreign group, or cell.

A filter contains one or more directives. A directive contains three elements: audit conditions, audit actions, and event classes. Essentially, a directive of a filter specifies whom (a principal/group/cell or the world), when (audit conditions), how (audit actions), and what (event classes) to audit. The following is an example.

```
Example 1:

filter type: foreign_principal
key: /.../cell_x/foo
directive 1:
     audit conditions - denial
     audit actions - log
     event classes - IBM_Confidential
directive 2:
     audit conditions - denial
     audit actions - alarm, log
     event classes - IBM_Confidential_Restricted
```

Directive 1 of this filter specifies that an audit log will be generated for event class `IBM_Confidential`, if the accessor is the foreign principal `/.../cell_x/foo` and the event failed because of access denial.

Two rules are used in filtering. The first one is an override rule: filters which are overridable (i.e., `cell_overridable` and `world_overridable` types) will be overridden by more specific filters. (A filter for a principal or a group is more specific than a filter for a cell or for the world.) A high-water-mark rule is applied after the override rule is applied. A filter is applicable to a client if its principal, groups, or cell identity matches the key of the filter. (The `world` and `world_overridable` filters have no keys and are applicable to all clients.) If there are multiple filters that are applicable to a client, then the union of the actions (log or alarm) specified by these filters is taken.

Overridable filters are used to allow negative controls described in the following scenario:

Alice in Company (cell) X is responsible to activate some operations (event class critical_transactions). Some other principals in company X are also authorized to activate the same operations, but only, for example, when Alice is not available. The system administrator wants to log an audit record no matter what the event outcome is (audit conditions = all) and no matter who activates these operations, and to generate an alarm only if the activator is not Alice. We translate this specification into the following two filters.

```
Example 2:

Filter 1:

     filter type: principal
     key: Alice
     directive 1:
          audit conditions - all
          audit actions - log
          event classes - critical_transactions

Filter 2:

     filter type: cell_overridable
     key: X
     directive 1:
          audit conditions - all
          audit actions - log, alarm
          event classes - critical_transactions
```

When Alice invokes events in the critical_transaction event class, the principal filter (filter 1) is applicable (because its key matches the Alice's identity). The principal filter is more specific than the cell filter. Although the cell filter (filter 2) is also applicable to Alice (Alice belongs to cell X), since the cell filter is overridable, it is overridden by the principal filter. For other principals in Company (cell) X, the only applicable filter is the cell filter (filter 2). Thus, these same events will not only cause a log but also raise an alarm.

Non-overridable world and cell filters are useful as well. Without them, an administrator, for example, would have to delete all filters for groups and principals of a cell in order to make a cell-wide filter effective to the whole cell. [System administrators may want to introduce a *temporary* non-overridable cell filter when a cell is suspected to be the source of a security problem.]

Figure 5-1 illustrates the override relations between different types of filters. An arrow from filter type X to filter type Y means X overrides Y.

```
+------------------------------------+
|   principal  foreign_principal     |             world
|   group      foreign_group         |                |
|   cell                             |                |
+------------------------------------+                |
        |                  |                          |
        V                  |                          |
 cell_overridable          |                          |
        |                  V                          |
        +---------> world_overridable <-------------+
```

```
Figure 5-1  Override relations between filter types
```

DCE groups are generally defined for the purpose of granting access permissions. Thus, in our design, a group filter specifies *auditing the intent to use the group's privileges*, instead of specifying *auditing the principals that belong to the group*. That is, a group filter would not have auditing effects on a member principal of the group unless the principal has the intent to use the group's privileges (by including the group in the PAC). Since group filters are defined to audit the intent to use a group's privileges, they are independent of other filters and are not overridable.

A possible future extension to the design is to define new types of group filters for naming convenience. Such filters would specify *auditing the principals that belong to a group*. These filters could be overridable by principal filters. [However, since the PAC does not necessarily contain the complete set of groups that the principal belongs to, a filter based on a group for naming convenience would require a registry-server query and have a performance impact.]

5.3. Filter Files

The audit daemon maintains the filters both in its address space and in files, these files are referred to as ESL (Event Selection List) files. The audit daemon does not always overwrite the ESL file when there is an update. Instead, it logs the update in an ESL-update file. Once the updates list in the ESL-update file becomes too long, the audit daemon incorporates the updates into the ESL file, and clears the update file. This design reduces the file I/O cost incurred during filter updates. Both ESL and ESL-update files need to be protected by the local OS, and only the audit daemon is allowed to modify these files. The audit clients on the same machine should be allowed to read (but not to modify) these files. Both ESL and ESL-update files are in a binary format (i.e., not human-readable).

The following are two examples of the contents of an ESL-update file that are maintained by an audit daemon on a server machine. The string names of principal, group, cell, event class, audit level, and action level are used here for clarity. In actual ESL and ESL-update files, the UUID's of principals, groups, and cells, and event class numbers are used (see Section 4.2). Audit conditions and audit actions are represented with 32-bit unsigned integers.

```
Example 3: (Exemplified contents of file "esl-update-princ")

entry 1:

     operation: ESL_UPDATE_ADD_FILTER
     key: Alice
     directive:
          audit conditions - all
          audit actions - alarm
```

```
            event classes - critical_transactions

     entry 2:

          operation: ESL_UPDATE_REMOVE_FILTER
          key: Joe
          directive:
               audit conditions - all
               audit actions - alarm
               event classes - controlled_access, configuration
```

6. AUDIT DAEMON

An audit daemon runs on each machine where some server(s) (including DCE components or applications) may generate audit records. The audit daemon provides two RPC interfaces: an audit-control interface provided to the audit control program used by the system administrator, and an audit-logging interface provided to the server (audit client) processes.

6.1. Audit-Control RPC Interface

The audit-control RPC interface of the audit daemon allows a (local or remote) system administrator (through a command interface: `auditcp`) to control the event filtering behavior of the servers on the machine where the audit daemon runs. The event filtering behavior is determined by the ESL (described in Section 5.2), and the audit-control RPC interface allows the caller to modify the ESL filters. The audit daemon maintains an ACL which defines three privileges. The read (`r`) privilege is required for querying the current filters, the write (`w`) privilege is required for adding/deleting/updating filters, and the control (`c`) privilege is required for administering the audit daemon itself and for modifying the daemon's ACL. The ACL is stored in a file, and the default path to the file is `/opt/dcelocal/etc/audit/ACL`. The ACL initially contains the following two default entries.

```
     user:hosts/host_name/self:rwc
     any_other:r
```

The following is the contents of the IDL file for the audit control interface (`audit_control.idl`).

```
[
uuid(006A04B4-E1C1-1C14-A5F2-02608C0FF790),
pointer_default(unique),
version(1.0)
]

interface audit_control {

    import "dce/rgynbase.idl";

    typedef enum {
        aud_e_esl_princ,
        aud_e_esl_foreign_princ,
        aud_e_esl_group,
        aud_e_esl_foreign_group,
        aud_e_esl_cell,
        aud_e_esl_cell_overridable,
        aud_e_esl_world,
        aud_e_esl_world_overridable,
        aud_e_esl_max
    } aud_esl_type_t;

    typedef unsigned32 aud_esl_cond_t;
```

```
const aud_esl_cond_t aud_c_esl_cond_success = 0x00000001;
const aud_esl_cond_t aud_c_esl_cond_failure = 0x00000002;
const aud_esl_cond_t aud_c_esl_cond_denial  = 0x00000004;
const aud_esl_cond_t aud_c_esl_cond_all     = 0x00000007;
const aud_esl_cond_t aud_c_esl_cond_unknown = 0xFFFFFFFF;
const unsigned16 aud_c_esl_cond_number = 4;

typedef unsigned32 aud_esl_act_t;

const aud_esl_act_t aud_c_esl_act_none  = 0x0;
const aud_esl_act_t aud_c_esl_act_log   = 0x00000001;
const aud_esl_act_t aud_c_esl_act_alarm = 0x00000002;
const aud_esl_act_t aud_c_esl_act_all   = 0x00000003;

/* strategies when trail storage is exhausted */
/* wrap around on old records */
const unsigned32 aud_c_trl_ss_wrap = 0x1;
/* stop collecting new records */
const unsigned32 aud_c_trl_ss_stop = 0x0;

/* is daemon accepting audit records */
const unsigned32 aud_c_dmn_state_enabled = 0x0;
const unsigned32 aud_c_dmn_state_disabled = 0x1;

typedef struct aud_esl_evt_classes {
        unsigned32  evt_class;
        struct aud_esl_evt_classes *next;
} aud_esl_evt_classes_t, *aud_esl_evt_classes_p_t;

typedef struct aud_esl_guides {
    aud_esl_evt_classes_p_t ec_list;
    aud_esl_cond_t audit_condition;
    aud_esl_act_t  audit_action;
    struct aud_esl_guides *next;
} aud_esl_guides_t, *aud_esl_guides_p_t;

typedef struct aud_esl_entry {
    uuid_t          subject_uuid;
    /* guides are link-listed */
    aud_esl_guides_p_t guides;
    struct aud_esl_entry *next;
} aud_esl_entry_t, *aud_esl_entry_p_t;

typedef struct aud_esl_foreign_entry {
    uuid_t          subject_uuid;
    uuid_t          cell_uuid;
    /* guides are link-listed */
    aud_esl_guides_p_t guides;
    struct aud_esl_foreign_entry *next;
} aud_esl_foreign_entry_t, *aud_esl_foreign_entry_p_t;

void audit_control_show_filter (
    [in]      handle_t            h,
    [in]      aud_esl_type_t      esl_type,
    [in]      sec_rgy_name_t      subject_name,
```

```
        [out]     aud_esl_guides_p_t  *guides,
        [out]     error_status_t      *st
    );

    void audit_control_add_filter (
        [in]      handle_t            h,
        [in]      aud_esl_type_t      esl_type,
        [in]      sec_rgy_name_t      subject_name,
        [in]      aud_esl_guides_p_t  *guides,
        [out]     error_status_t      *st
    );

    void audit_control_remove_filter (
        [in]      handle_t            h,
        [in]      aud_esl_type_t      esl_type,
        [in]      sec_rgy_name_t      subject_name,
        [in]      aud_esl_guides_p_t  *guides,
        [out]     error_status_t      *st
    );

    void audit_control_delete_filter (
        [in]      handle_t            h,
        [in]      aud_esl_type_t      esl_type,
        [in]      sec_rgy_name_t      subject_name,
        [out]     error_status_t      *st
    );

    void audit_control_list_filter (
        [in]      handle_t            h,
        [in]      aud_esl_type_t      esl_type,
        [in, out] unsigned16          *cursor,
        [out]     aud_esl_entry_p_t   *first_entry,
        [out]     aud_esl_foreign_entry_p_t *first_foreign_entry,
        [out]     error_status_t      *st
    );

    void audit_control_modify_sstrategy (
        [in]      handle_t            h,
        [in]      unsigned32          strgy,
        [out]     error_status_t      *st
    );

    void audit_control_show_sstrategy (
        [in]      handle_t            h,
        [out]     unsigned32          *strgy,
        [out]     error_status_t      *st
    );

    void audit_control_modify_state (
        [in]      handle_t            h,
        [in]      unsigned32          state,
        [out]     error_status_t      *st
    );

    void audit_control_show_state (
```

```
        [in]      handle_t            h,
        [out]     unsigned32          *state,
        [out]     error_status_t      *st
    );

    void audit_control_stop (
        [in]      handle_t            h,
        [out]     error_status_t      *st
    );

}
```

6.2. Audit Logging RPC Interface

The audit daemon provides the following audit-logging RPC interface (`audit_log.idl`) to audit clients, which uses this interface through the audit-logging API).

```
[
uuid(00164C16-E218-1C14-9864-02608C0FF790),
version(1.0),
pointer_default(ptr)
]

interface audit_log {

    import "dce/utctypes.idl";
    import "dce/aclbase.idl";

    typedef enum {
        audit_log_async,
        audit_log_sync_no_wait,
        audit_log_sync
    } audit_log_sync_t;

    typedef struct {
        unsigned32       format;     /* event tail data format*/
        uuid_t           server;     /* server (audit client) uuid*/
        unsigned32       event;      /* event number */
        unsigned16       outcome;    /* event outcome (failed?) */
        unsigned16       authz_st;   /* authorization status */
        uuid_t           client;
        uuid_t           cell;       /* client cell */
        unsigned16       num_groups; /* no. of client groups */
        utc_t            time;       /* record commit time */
        [string, ptr]
        char             *addr;      /* client address */
        [size_is(num_groups), ptr]
        uuid_t           *groups;   /* client groups */
    } dce_aud_hdr_t;

    const unsigned16 aud_c_evt_info_small_int   = 0;
    const unsigned16 aud_c_evt_info_short_int   = 1;
    const unsigned16 aud_c_evt_info_long_int    = 2;
    const unsigned16 aud_c_evt_info_hyper_int   = 3;
    const unsigned16 aud_c_evt_info_usmall_int  = 4;
    const unsigned16 aud_c_evt_info_ushort_int  = 5;
```

```
const unsigned16 aud_c_evt_info_ulong_int   = 6;
const unsigned16 aud_c_evt_info_uhyper_int  = 7;
const unsigned16 aud_c_evt_info_short_float = 8;
const unsigned16 aud_c_evt_info_long_float  = 9;
const unsigned16 aud_c_evt_info_boolean     = 10;
const unsigned16 aud_c_evt_info_uuid        = 11;
const unsigned16 aud_c_evt_info_utc         = 12;
const unsigned16 aud_c_evt_info_acl         = 13;
const unsigned16 aud_c_evt_info_byte_string = 14;
const unsigned16 aud_c_evt_info_char_string = 15;

typedef union switch (unsigned16 format) {
    case aud_c_evt_info_small_int:
        small int small_int;
    case aud_c_evt_info_short_int:
        short int short_int;
    case aud_c_evt_info_long_int:
        long int long_int;
    case aud_c_evt_info_hyper_int:
        hyper int * hyper_int;
    case aud_c_evt_info_usmall_int:
        unsigned small int usmall_int;
    case aud_c_evt_info_ushort_int:
        unsigned short int ushort_int;
    case aud_c_evt_info_ulong_int:
        unsigned long int ulong_int;
    case aud_c_evt_info_uhyper_int:
        unsigned hyper int * uhyper_int;
    case aud_c_evt_info_short_float:
        float short_float;
    case aud_c_evt_info_long_float:
        double * long_float;
    case aud_c_evt_info_boolean:
        boolean bool;
    case aud_c_evt_info_uuid:
        uuid_t * uuid;
    case aud_c_evt_info_utc:
        utc_t * utc;
    case aud_c_evt_info_acl:
        sec_acl_t * acl;
    case aud_c_evt_info_byte_string:
        [string] byte * byte_string;
    case aud_c_evt_info_char_string:
        [string] char * char_string;
    default:
        ;
} aud_log_ev_info_t;

typedef struct {
    unsigned32      item_count;
    [size_is(item_count)] aud_log_ev_info_t info_item[];
} aud_log_ev_info_list_t;

void audit_log_append (
    [in]      handle_t               h,
```

```
        [in]      unsigned32              options,
        [in]      dce_aud_hdr_t           *header_info,
        [in]      aud_log_ev_info_list_t  **tail_info,
        [out]     error_status_t          *st
    );

}
```

The audit-logging API passes audit records (the header and the tail) to the audit daemon through calling the `audit_log_append()` operation.[3] The `audit_log_append()` call has to be authenticated so that the identity of the caller (i.e., the audit client) can be securely determined. For the call to succeed, the caller must have the log (`l`) permission in the audit daemon's ACL. The initial ACL of an audit daemon does not contain any entry granting the log permission. Entries must be added (using the `acl_edit` command) to grant the log permission to audit clients. The purpose of having this access control is to ensure that unauthorized servers cannot flood the audit daemon with their audit records and exhaust the audit storage space (a denial-of-service attack). Since the current design only addresses local audit, an audit daemon will only receive audit records from local servers. A per-host DCE group can be created which consists of the servers which are authorized to generate audit records on a host (e.g., `group/hosts/`*`hostname`*`/audit-clients`).

6.3. Using the IDL's Pickling Service

The IDL's pickling service is used for encoding audit records before they get stored in an audit-trail file. The idl and acf files that are used for pickling are shown below. The stub function `audit_pickle_dencode_evt_info()` is used both for encoding an audit record (header and tail) into a byte string (to be stored in a trail file), and for decoding a byte string (from a trail file) into an audit header structure (header and tail).

Contents of `audit_pickle.idl`:

```
[
uuid(003dc39a-74bd-1c82-a5eb-02608c0ff790),
version(1.0),
pointer_default(unique)
]

interface audit_pickle {

    import "dce/utctypes.idl";
    import "audit_log.idl";

    void audit_pickle_dencode_evt_info (
        [in]        handle_t                h,
        [in, out]   dce_aud_hdr_t           * header,
        [in, out]   aud_log_evt_info_list_t ** info
    );

}
```

Contents of `audit_pickle.acf`:

```
interface audit_pickle {
```

3. The current design only addresses *local audit*; i.e., audit records are always written to a local file. To perform *remote audit*, the audit daemon will ship the audit records to a central audit machine.

```
    [encode, decode] audit_pickle_dencode_evt_info ();

}
```

6.4. RPC Bindings and Authentication Identity

The audit control program, `auditcp`, needs an RPC binding to the daemon before making `audit_control_*()` calls. Audit clients which perform auditing also need such a binding before making the `audit_log_append()` call. In addition, since all these calls need to be authenticated, the callers need the audit daemon's identity (a principal name) when they call the `rpc_binding_set_auth_info()` function to set authentication and authorization information for the client's binding handle. (The audit daemon is the server in these authenticated RPCs.) The audit daemon exports its RPC bindings and its identity in a local file named `/opt/dcelocal/var/audit/daemon_bindings`. The default identity of the audit daemon is the host's identity; i.e., `hosts/`*`hostname`*`/self`. The audit daemon can be assigned a different identity with a command argument of `auditd`. For example, it can be assigned the identity of `hosts/hostA/audit-daemon` on `hostA` with the following command line.

```
auditd -I hosts/hostA/audit-daemon
```

6.5. Audit Storage Size Limit

The storage size limit of audit trails on a host is determined by the space allocation for audit (e.g., the size of the audit filesystem) on the host. Vendors are expected to provide post-processing tools for backing up audit trails to tapes and/or reduce the size of the audit trails by discarding obsolete records.

The audit daemon, by default, will stop appending new audit records if the storage limit has been reached. [To avoid this situation, an alarm (a message on the screen) is raised upon new each new audit record, if some storage threshold is exceeded.] The daemon idles until it is notified through an RPC interface (by an administrator using the `auditcp` command described in the next section) that the storage has been emptied (with the content backed up). However, the option to discard old trail files to make space for new records can be specified at the command line when `auditd` is invoked. In addition, a command-line argument can be used to specify a list of authorized principals and groups who can switch the audit daemon's strategy in handling this situation or resume the audit daemon's logging operation after the storage space for old trails is freed.

6.6. Audit Control Program

Audit control program (`auditcp`) is the command interface for managing the event selection list used for filtering auditable events, and for controlling the audit daemon's behavior when the audit storage is full. The manual page for this command interface is presented in Appendix A.

7. AUDIT LOGGING API

A server utilizes the audit facility through calling the audit logging API,[4] which performs event filtering and logging. Function `dce_aud_open()` reads ESL and ESL-update files to initialize the ESL in the server's address space. The function also registers an RPC interface for filter update notifications. An RPC is made by the audit daemon to notify the audit clients whenever filters are updated. The interface is only for notification, not for transmitting the actual updates. The manager routine of the update interface maintains the timestamps for the last modification times of ESL and ESL-update files. The RPC bindings of an audit client are exported to a local file, `/opt/dcelocal/var/audit/update_notify/`*`pid`*, where `pid` is the process id of the audit client. An audit client removes its RPC-binding file before it terminates for any reason. The audit daemon periodically makes `rpc_mgmt_is_server_listening()` calls with the bindings obtained from the binding files to determine if any audit client crashed without removing its binding file, and remove such files.

Upon update notification, the audit client checks whether the ESL files or ESL-update files have been modified. If any file has been modified, the manager routine needs to refresh the audit client's in-memory copy of the ESL. It

4. The audit logging API functions (`dce_aud_*()`) are specified in [RFC 28.1].

takes the following actions.

(a) If an ESL file has been modified (the corresponding ESL-update file must have also been modified in this case), the runtime discards the server's copy of the ESL, reads the new ESL file, and applies the updates from the new ESL-update file.

(b) If only the ESL-update file has been modified, the runtime applies the updates from the file directly on its copy of the ESL.

The `dce_aud_open()` function also reads the event class configuration files (described in Section 5.1) to initialize an event table in the audit client's address space. The event table contains an entry for each of the audit client's events specifying the event classes that an event belongs to. The event class configuration files are usually static. However, to avoid the need to restart audit clients in the face of new/modified event class configurations, `dce_aud_open()` also uses the update notification interface. Upon reconfigurations of event classes, audit clients are notified through this interface. The manager routine maintains an up-to-date event table.

The `dce_aud_start()` function determines if an audit record should be logged for a specified event. From the input parameters, the function obtains the UUID's of the client principal and the client cell, and the group privileges used by the principal for the event. Based on this identity information, the server's (audit client's) ESL, the event outcome (if known), and the event classes to which this event belongs, the function determines what audit action(s) need(s) to be taken, if any. If some audit action(s) need to be taken, then an audit record descriptor is initialized and returned. If the event outcome is unknown when `dce_aud_start()` is called, it is possible that the audit action(s) cannot be determined. (In this case, the action(s) will be determined later when `dce_aud_commit` is called with a known event outcome.)

The `dce_aud_put_ev_info()` function fills the event-specific information in the audit record descriptor. The `dce_aud_commit()` function sends the audit record to the audit daemon (default) or writes to an specified audit-trail file. Figure 7-1 illustrates the runtime scenario of the audit-logging API. The data structures maintained by the server's audit runtime are displayed on the right side of the figure.

```
                                     |Server's Audit
                                ---->|Runtime Data Structures

                                           Audit-Trail
                                           Descriptor
                   initialize             +----------------+
      +---------------------------------->| Audit daemon's |
      |                                   |  RPC binding   |
      |                                   +----------------+
      |
      |                                     Event Table
      |                                   +----------------+
      |                                   | event classes  |
dce_aud_open() ------>+--------------+    |   1:   A, B    |
      |        read   | Event Class  |--->|   2:   C       |
      |               | Config Files |    | ...            |
      |               +--------------+    +----------------+
      |               +--------------+    +----------------+
      +-------------->|   ESL Files  |--->|      ESL       |
           read       +--------------+    +----------------+
                                                  |
                                                  |
                                                  |
    filter events by checking Event Table and ESL |
      +-------------------------------------------+
      |
      |
      |                                 Audit Record Descriptor
      V         initialize record buffer  +-----------------+
dce_aud_start() ------------------------->|   event name    |
                 name, header(, outcome)  |   header info.  |
                                          |   event info.   |
dce_aud_put_ev_info() ------------------->|   event outcome |
                       event-specific     +-------+---------+
                        information               |
                                                  |
                                                  |
       read audit record                          |
      +-------------------------------------------+
      |
      |           audit_log_append() RPC to audit daemon (log),
      |           or write audit record to trail file (log),
      V           or raise an alarm message (alarm)
dce_aud_commit() ---------------------------------------->
                  options, format, outcome
```

Figure 7.1 Runtime scenario of the audit-logging API

8. ACKNOWLEDGEMENTS

We would like to thank Virgil D. Gligor (University of Maryland, College Park), Ping Lin (IBM), Joe Pato (HP), Denis Pinkas (Bull), Pau-Chen Cheng (IBM), Chii-Ren Tsai (VDG), and Rajashekar Kailar (University of Maryland, College Park), for their comments on the earlier drafts of this RFC and suggestions on the design.

APPENDIX A. AUDIT DAEMON — MANUAL PAGE

A.1. Name

`auditd` — Starts the Audit Daemon

A.2. Synopsis

```
          +-----------------------------+
auditd ---|    +-------------------+    |
          +----| -I identity       |----+
             ^ | -T trail          |-+
             | | -A                | |
             | | -wrap             | |
             | +-------------------+ |
             +-----------------------+
```

A.3. Arguments

(a) `-I` identity — The identity argument specifies the server's name used for the authentication purpose. The default identity is `hosts/`*`hostname`*`/self` if this argument is not present.

(b) `-T` trail — The trail argument specifies the path to the audit trail file used by the audit daemon. The default path is `/opt/dcelocal/var/audit/trail` if this argument is not present.

(c) `-A` — Performs audit on the audit-control interface.

(d) `-wrap` — The default strategy used by the daemon is to stop collecting audit records when the trail storage is exhausted. The default behavior can be changed at the command invocation using this flag. When this flag is used, the audit daemon will wrap on the trail file when the trail file limit is reached to record new audit records.

A.4. Description

The audit daemon (`auditd`) exports RPC interfaces for audit administration (`audit_control_*`) and audit logging (`audit_log_*`).

A.5. Required Permissions

The user of auditd must be a privileged user to the local operating system.

APPENDIX B. AUDIT CONTROL PROGRAM — MANUAL PAGE

B.1. Name

`auditcp` — Starts the audit control program

B.2. Synopsis

```
auditcp [ auditcp-command ]
```

B.3. Arguments

auditcp-command specifies one of the following control program commands:

(a) `aud modify` — Changes the attributes of an audit daemon. The attributes types include `stostrategy` (storage strategy when audit-trail storage is full) and `state` (accepting audit log requests or not). Defined storage strategies include (1) the audit daemon stops servicing logging requests and waits for an instruction to resume logging service, and (2) the audit daemon wraps around the audit trail file when the trail file is exceeding its size limit. In case (1) system administrator should make the storage space available, and instruct the audit daemon to resume (by modifying the `state` attribute).

(b) `aud disable` — Disables an audit daemon. The `state` attribute is changed to `disabled`.

(c) `aud enable` — Enables an audit daemon. The `state` attribute is changed to `enabled`.

(d) `aud show` — Shows the attributes of an audit daemon.

(e) `aud stop` — Terminates an audit-daemon process.

(f) `audfilter add` — Add a directive to a specified filter.

(g) `audfilter delete` — Delete the specified filter.

(h) `audfilter catalog` — Lists the names of all filters in an audit daemon. The names are a list of a type and if necessary a key.

(i) `audfilter remove` — Removes a directive from a specified filter.

(j) `audfilter show` — Returns the directives in a specified filter.

(k) `audevents catalog` — Lists the names of all event classes.

(l) `audtrail show` — Shows an audit-trail file by converting its binary contents to a human-readable format and sends the output to a file or to the standard output (default).

(m) `exit` — Leaves an audit control program.

(n) `quit` — Leaves an audit control program.

B.4. Description

The audit control program provides a set of commands for managing the event selection list used for filtering auditable events, for controlling an audit daemon's behavior when the audit storage is full, and for terminating an audit-daemon process.

You can use control program commands from within the control program or from the system prompt (represented here as a $).

B.4.1. From inside the control program

You can start and enter the control program using the auditcp command alone, without any argument. The control program then displays the control program prompt (`auditcp>`), as follows:

```
$ auditcp
auditcp>
```

You can then enter any control program command, for example:

```
auditcp> audfilter show
```

You leave the control program and return to the system prompt using the `exit` or `quit` command.

If you enter invalid input, the control program displays the valid commands.

B.4.2. From the system prompt

Enter the `auditcp` command with a internal command of the control program as the first argument(s). For example, you can enter the `audfilter show` command as follows:

```
$ auditcp audfilter show
```

B.5. Interactive Inputs of More Command Arguments

(a) `aud modify` — This command takes in an attribute type and an attribute value interactively. The attribute types include `stostrategy` and `state`. Defined attribute values for `stostrategy` include `stop` and `wrap`, and defined attribute values for `state` include `enabled` and `disabled`.

(b) `aud show` — This command takes in an attribute type (`stostrategy` or `state`) interactively.

(c) `audfilter show`, `audfilter delete` — These two commands take in a filter type and possibly a key interactively. No key is needed for `world` and `world_overridable` filters. For other types of filters, a symbolic name (the key) that identifies a subject (i.e., a principal, a foreign principal, a group, a foreign group, or a cell) is required. The following are some examples of acceptable symbolic names:

luan, /.../toronto.ibm.com/weisz, acct-admin, /.../osf.org/dce-team, and /.../hp.com.

(d) audfilter add, audfilter remove — These two commands take in the filter type, possibly a key, and a filter directive interactively. The filter type and the key are first prompted (see audfilter show and audfilter delete commands above). Then, a directive is prompted interactively. The audit conditions, audit actions, and event classes are prompted in this order. Defined audit conditions (symbolic names) include success, failure, and denial. Defined audit actions (symbolic names) include log, alarm, and all. The response to the event-class prompt should be the name of an event-class configuration file. Only one audit condition, action, or event class should be entered at one time, and the input procedure iterates itself until an empty line is reached.

(e) audtrail show — This command takes in the names of the binary trail file and the output file interactively.

The following five auditcp commands do not have arguments:

(a) aud stop

(b) audfilter catalog

(c) audevents catalog

(d) exit

(e) quit

B.6. Required Permissions

The control (c) permission (on the audit daemon's ACL) is required for the aud modify and aud stop commands. The write (w) permission is required for the audfilter add, audfilter delete, and audfilter remove commands. The read (r) permission is required for the aud show, audfilter show, and audfilter catalog commands. audevents catalog commands. The user of auditcp who wants to invoke these commands must have a DCE login context (through dce_login).

B.7. Relation with dcecp

The functionalities provided by auditcp will aslo be provided in dcecp [RFC 42.0]. Although the two command interfaces are different (auditcp is more interactive, and dcecp is useful for writing command scripts), the implementations of sub-commands will be similar.

REFERENCES

[NCSC 85] National Computer Security Center, ''Department of Defense Trusted Computer System Evaluation Criteria'', DOD 5200.28-STD, December 1985.

[NCSC 87] National Computer Security Center, ''Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria'', NCSC-TG-005, July 31, 1987.

[NCSC 88] National Computer Security Center, ''A Guide to Understanding Audit in Trusted Systems'', NCSC-TG-001, June 1, 1988.

[RFC 8.0] M. Gasser, ''DCE SIG Security Requirements'', July 1992.

[RFC 24.0] R. Salz, ''DCE 1.1 Serviceability Proposal'', November 1992.

[RFC 25.0] E. McDermott, ''DCE 1.1 Auditing Strategy and Design Proposal'', December 1992.

[RFC 28.1] S. Luan, R. Weisz, ''A DCE Server Auditable-Event Identification and a Proposed Audit Logging API'', November 1993.

[RFC 42.0] H. Melman, ''DCE Shell Functional Specification'', June 1993.

AUTHOR'S ADDRESSES

Shyh-Wei Luan
VDG Inc.
6009 Brookside Drive
Chevy Chase MD 20815
USA

Internet email: luan@vnet.ibm.com
Telephone: +1-301-240-7385

Robert Weisz
IBM Canada Laboratory
1150 Eglinton Ave. East
North York, Ontario M3C1H7
CANADA

Internet email: weisz@torolab6.vnet.ibm.com
Telephone: +1-416-448-3131