DCE 1.2.2 Application Development Guide—Core Components

OSF® DCE Product Documentation

The Open Group

OSF® DCE Product Documentation:

DCE 1.2.2 Application Development Guide—Core Components, (Volume 1)
ISBN 1–85912–192–6
Document Number F203A

DCE 1.2.2 Application Development Guide—Core Components, (Volume 2)
ISBN 1–85912–154–3
Document Number F203B

Published in the U.K. by The Open Group, 1997.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:
OGPubs@opengroup.org

OTHER NOTICES

Contents

Part 3. DCE Remote Procedure Call

Part 4. DCE Distributed Time Service

List of Figures

List of Tables

Preface

The Open Group

The Open Group is the leading vendor-neutral, international consortium for buyers and suppliers of technology. Its mission is to cause the development of a viable global information infrastructure that is ubiquitous, trusted, reliable, and as easy-to-use as the telephone. The essential functionality embedded in this infrastructure is what we term the IT DialTone. The Open Group creates an environment where all elements involved in technology development can cooperate to deliver less costly and more flexible IT solutions.

Formed in 1996 by the merger of the X/Open Company Ltd. (founded in 1984) and the Open Software Foundation (founded in 1988), The Open Group is supported by most of the world's largest user organizations, information systems vendors, and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to meet its new mission, as well as to assist user organizations, vendors, and suppliers in the development and implementation of products supporting the adoption and proliferation of systems which conform to standard specifications.

With more than 200 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- consolidating, prioritizing, and communicating customer requirements to vendors
- conducting research and development with industry, academia, and government agencies to deliver innovation and economy through projects associated with its Research Institute
- managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- adopting, integrating, and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- licensing and promoting the Open Brand, represented by the "X" mark, that designates vendor products which conform to Open Group Product Standards
- promoting the benefits of IT DialTone to customers, vendors, and the public.

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development, and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trademark on behalf of the industry.

The Development of Product Standards

This process includes the identification of requirements for open systems and, now, the IT DialTone, development of CAE and Preliminary Specifications through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product. There are currently two forms of Product

Standard, namely the Profile Definition and the Component Definition, although these will eventually be merged into one.

The "X" mark is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the Open Brand they guarantee, through the X/Open Trade Mark License Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

Open Group Publications

The Open Group publishes a wide range of technical documentation, the main part of which is focused on specification development and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys, and business titles.

There are several types of specification:

CAE Specifications

CAE (Common Applications Environment) Specifications are the stable specifications that form the basis for our Product Standards, which are used to develop X/Open branded systems. These specifications are intended to be used widely within the industry for product development and procurement purposes.

Anyone developing products that implement a CAE Specification can enjoy the benefits of a single, widely supported industry standard. Where appropriate, they can demonstrate product compliance through the Open Brand. CAE Specifications are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

Preliminary Specifications

Preliminary Specifications usually address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is not a draft specification; rather, it is as

stable as can be achieved, through applying The Open Group's rigorous development and review procedures.

Preliminary Specifications are analogous to the trial-use standards issued by formal standards organizations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a CAE Specification. While the intent is to progress Preliminary Specifications to corresponding CAE Specifications, the ability to do so depends on consensus among Open Group members.

Consortium and Technology Specifications

The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE, OSF/Motif, and CDE.

Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as CAE Specifications, in which case the relevant Technology Specification is superseded by a CAE Specification.

In addition, The Open Group publishes:

Product Documentation

This includes product documentation—programmer's guides, user manuals, and so on—relating to the Prestructured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

Guides

These provide information that is useful in the evaluation, procurement, development, or management of open systems, particularly those that relate to the CAE Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

Technical Studies

Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Program. They

are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

Versions and Issues of Specifications

As with all live documents, CAE Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new Version indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it replaces the previous publication.
- A new Issue indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at *http://www.opengroup.org/public/pubs*.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at *http://www.opengroup.org/public/pubs*.

This Book

The *DCE 1.2.2 Application Development Guide* provides information about how to program the application programming interfaces (APIs) provided for each OSF® Distributed Computing Environment (DCE) component.

Audience

This guide is written for application programmers with UNIX operating system and C language experience who want to develop and write applications to run on DCE.

Applicability

This revision applies to the OSF® DCE Release 1.2.2 offering and related updates. See your software license for details.

Purpose

The purpose of this guide is to assist programmers in developing applications that use DCE. After reading this guide, you should be able to program the Application Programming Interfaces provided for each DCE component.

Document Usage

The *DCE 1.2.2 Application Development Guide* consists of three books, as follows:

- *DCE 1.2.2 Application Development—Introduction and Style Guide*
  Document Number F202, ISBN 1–85912– 187–X
- *DCE 1.2.2 Application Development Guide—Core Components*

  - — Volume 1
    Document Number F203A, ISBN 1–85912–192–6

      Part 1. DCE Facilities

      Part 2. DCE Threads

      Part 3. DCE Remote Procedure Call

  - — Volume 2
    Document Number F203B, ISBN 1–85912–154–3

      Part 4. DCE Distributed Time Service

      Part 5. DCE Security Service

- *DCE 1.2.2 Application Development Guide—Directory Services*
  Document Number F204, ISBN 1–85912–197–7

  - — Part 1. DCE Directory Service
  - — Part 2. CDS Application Programming
  - — Part 3. GDS Application Programming
  - — Part 4. XDS/XOM Supplementary Information

Related Documents

For additional information about the Distributed Computing Environment, refer to the following documents:

- *DCE 1.2.2 Introduction to OSF DCE*
  Document Number F201, ISBN 1–85912–182–9
- *DCE 1.2.2 Command Reference*
  Document Number F212, ISBN 1–85912–138–1
- *DCE 1.2.2 Application Development Reference*
  Document Number F205A, ISBN 1–85912–103–9 (Volume 1)
  Document Number F205B, ISBN 1–85912–108–X (Volume 2)
  Document Number F205C, ISBN 1–85912–159–4 (Volume 3)
- *DCE 1.2.2 Administration Guide—Introduction*
  Document Number F207, ISBN 1–85912–113–6

- *DCE 1.2.2 Administration Guide—Core Components*
  Document Number F208, ISBN 1–85912–118–7
- *DCE 1.2.2 DFS Administration Guide and Reference*
  Document Number F209A, ISBN 1–85912–123–3 (Volume 1)
  Document Number F209B, ISBN 1–85912–128–4 (Volume 2)
- *DCE 1.2.2 GDS Administration Guide and Reference*
  Document Number F211, ISBN 1–85912–133–0
- *DCE 1.2.2 File-Access Administration Guide and Reference*
  Document Number F216, ISBN 1–85912–158–6
- *DCE 1.2.2 File-Access User's Guide*
  Document Number F217, ISBN 1–85912–163–3
- *DCE 1.2.2 Problem Determination Guide*
  Document Number F213A, ISBN 1–85912–143–8 (Volume 1)
  Document Number F213B, ISBN 1–85912–148–9 (Volume 2)
- *DCE 1.2.2 Testing Guide*
  Document Number F215, ISBN 1–85912–153–5
- *DCE 1.2.2 File-Access FVT User's Guide*
  Document Number F210, ISBN 1–85912–189–6
- *DCE 1.2.2 Release Notes*
  Document Number F218, ISBN 1–85912–168–3

Typographic and Keying Conventions

This guide uses the following typographic conventions:

Bold — **Bold** words or characters represent system elements that you must use literally, such as commands, options, and pathnames.

Italic — *Italic* words or characters represent variable values that you must supply. *Italic* type is also used to introduce a new DCE term.

`Constant width` — Examples and information that the system displays appear in `constant width` typeface.

[ ] — Brackets enclose optional items in format and syntax descriptions.

{ }
: Braces enclose a list from which y ou must choose an item in format and syntax descriptions.

|
: A vertical bar separates items in a list of choices.

< >
: Angle brackets enclose the name of a key on the keyboard.

...
: Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.

This guide uses the following keying conventions:

<Ctrl-*x***>** or ^*x*
: The notation **<Ctrl-** *x* **>** or ^*x* followed by the name of a key indicates a control character sequence. For example, **<Ctrl-C>** means that you hold down the control key while pressing **<C>**.

<Return>
: The notation **<Return>** refers to the key on your terminal or workstation that is labeled with the word Return or Enter, or with a left arrow.

Problem Reporting

If you have any problems with the software or vendor-supplied documentation, contact your software vendor's customer service department. Comments relating to this Open Group document, however, should be sent to the addresses provided on the copyright page.

Pathnames of Directories and Files in DCE Documentation

For a list of the pathnames for directories and files referred to in this guide, see the *DCE 1.2.2 Administration Guide—Introduction* and *DCE 1.2.2 Testing Guide*.

Trademarks

Motif®, OSF/1®, and UNIX® are registered trademarks and the IT DialTone™, The Open Group™, and the "X Device"™ are trademarks of The Open Group.

DEC, DIGITAL, and ULTRIX are registered trademarks of Digital Equipment Corporation.

DECstation 3100 and DECnet are trademarks of Digital Equipment Corporation.

HP, Hewlett-Packard, and LaserJet are trademarks of Hewlett-Packard Company.

Network Computing System and PasswdEtc are registered trademarks of Hewlett-Packard Company.

AFS, Episode, and Transarc are registered trademarks of the Transarc Corporation.

DFS is a trademark of the Transarc Corporation.

Episode is a registered trademark of the Transarc Corporation.

Ethernet is a registered trademark of Xerox Corporation.

AIX and RISC System/6000 are registered trademarks of International Business Machines Corporation.

IBM is a registered trademark of International Business Machines Corporation.

DIR-X is a trademark of Siemens Nixdorf Informationssysteme AG.

MX300i is a trademark of Siemens Nixdorf Informationssysteme AG.

NFS, Network File System, SunOS and Sun Microsystems are trademarks of Sun Microsystems, Inc.

PostScript is a trademark of Adobe Systems Incorporated.

Microsoft, MS-DOS, and Windows are registered trademarks of Microsoft Corp.

NetWare is a registered trademark of Novell, Inc.

Part 4

DCE Distributed Time Service

Chapter 20

Introduction to the Distributed Time Service API

This chapter describes the DCE Distributed Time Service (DTS) programming routines. You can use these routines to obtain timestamps that are based on Coordinated Universal Time (UTC). You can also use the DTS routines to translate among different timestamp formats and perform calculations on timestamps. Applications can use the timestamps that DTS supplies to determine event sequencing, duration, and scheduling. Applications can call the DTS routines from any host that has the **libdce**. The **dtsd** need not be running.

DTS routines are written in the C programming language. You should be familiar with basic DTS concepts before you attempt to use the application programming interface (API). The DTS chapters of the *DCE 1.2.2 Administration Guide—Core Components* provide conceptual information about DTS.

The DTS API routines offer the following basic functions:

- Retrieving timestamp information
- Converting between binary timestamps that use different time structures

- Converting between binary timestamps and ASCII representations
- Converting between UTC time and local time
- Manipulating binary timestamps
- Comparing two binary time values
- Calculating binary time values
- Obtaining time zone information

The sections that follow describe how DTS represents time, discuss the DTS time structures, discuss the DTS API header files, and briefly describe the DTS API routines.

20.1 DTS Time Representation

UTC is the international time standard that has largely replaced Greenwich Mean Time (GMT). The standard is administered by the International Time Bureau (BIH) and is widely used. DTS uses opaque binary timestamps that represent UTC for all of its internal processes. You cannot read or disassemble a DTS binary timestamp; the DTS API allows applications to convert or manipulate timestamps, but they cannot be displayed. DTS also translates the binary timestamps into ASCII text strings, which can be displayed.

20.1.1 Absolute Time Representation

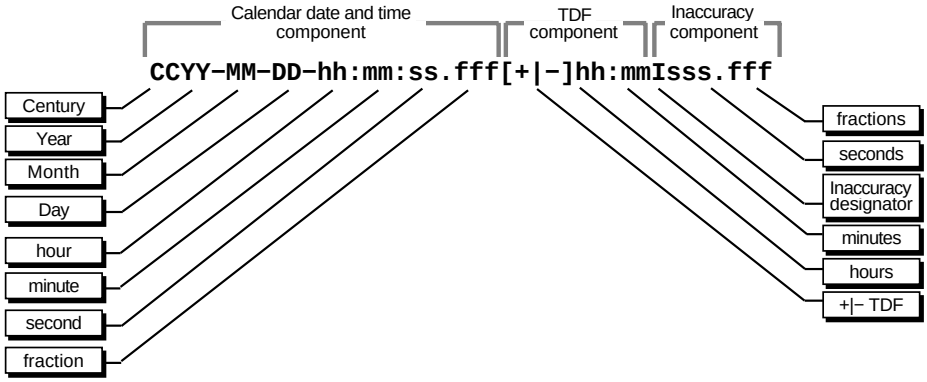
An absolute time is a point on a time scale. For DTS, absolute times reference the UTC time scale; absolute time measurements are derived from system clocks or external time-providers. When DTS reads a system clock time, it records the time in an opaque binary timestamp that also includes the inaccuracy and other information. When you display an absolute time, DTS converts the time to ASCII text as shown in the following display:

```
1990-11-21-13:30:25.785-04:00I000.082
```

DTS displays all times in a format that complies with the International Organization for Standardization (ISO) 8601 (1988) standard. Note that the inaccuracy portion of the time is not defined in the ISO standard; times that do not include an inaccuracy are accepted.

Figure 20-1 explains the ISO format that generated the previous display.

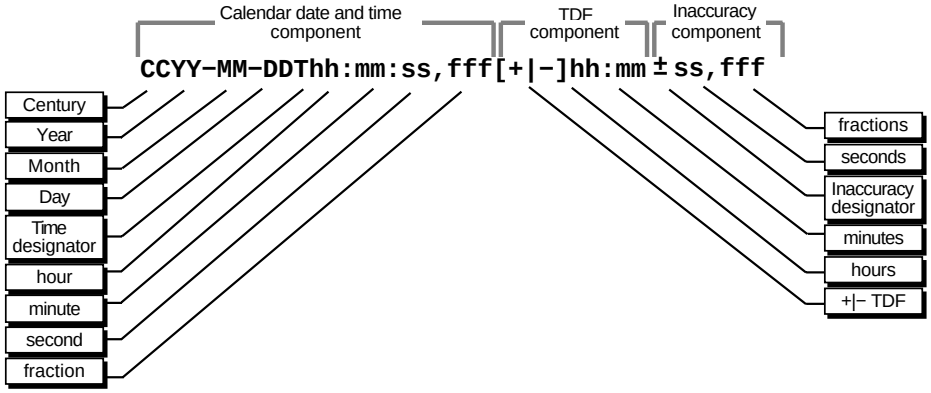
Figure 20–1. ISO Format for Time Displays



In this figure, the relative time preceded by the + (plus) or – (minus) character indicates the hours and minutes that the calendar date and time are offset from UTC. The presence of this time differential factor (TDF) in the string also indicates that the calendar date and time are the local time of the system, not UTC. Local time is UTC plus the TDF. The Inaccuracy (**I**) designator indicates the beginning of the inaccuracy component associated with the time.

Although DTS displays all times in the previous format, variations to the ISO format shown in Figure 20-2 are also accepted as input for the ASCII conversion routines.

Figure 20–2. Variations to the ISO Time Format



In this figure, the Time (**T**) designator separates the calendar date from the time, a **,** (comma) separates seconds from fractional seconds, and the **+** or – indicates the beginning of the inaccuracy component.

The following examples show some valid time formats.

The following represents July 4, 1776 17:01 GMT and an unspecified inaccuracy (default):

```
1776-7-4-17:01:00
```

The following represents a local time of 12:01 (17:01 GMT) on July 4, 1776 with a TDF of –5 hours and an inaccuracy of 100 seconds:

```
1776-7-4-12:01:00-05:00I100
```

Both of the following represent 12:00 GMT in the current day, month, and year with an unspecified inaccuracy:

```
12:00 and T12
```

The following represents July 14, 1792 00:00 GMT with an unspecified inaccuracy:

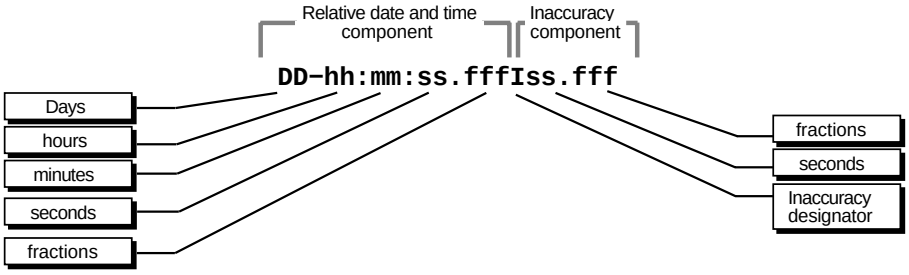
```
1792-7-14
```

20.1.2 Relative Time Representation

A relative time is a discrete time interval that is usually added to or subtracted from another time. A TDF associated with an absolute time is one example of a relative time. A relative time is normally used as input for commands or system routines.

Figure 20-3 shows the full syntax for a relative time.

Figure 20–3. Full Syntax for a Relative Time



The following example shows a relative time of 21 days, 8 hours, and 30 minutes, 25 seconds with an inaccuracy of 0.300 seconds:

```
21-08:30:25.000I00.300
```

The following example shows a negative relative time of 20.2 seconds with an unspecified inaccuracy (default):

```
-20.2
```

The following example shows a relative time of 10 minutes, 15.1 seconds with an inaccuracy of 4 seconds:

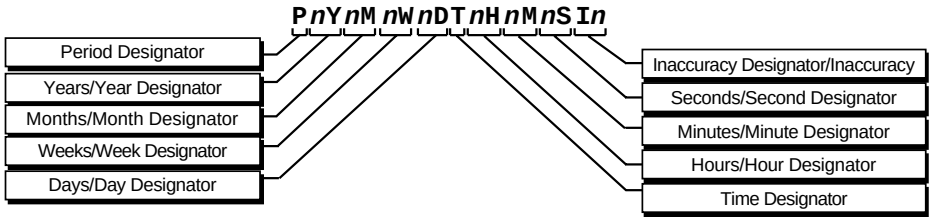
```
10:15.1I4
```

Notice that a relative time does not use the calendar date fields, since these fields concern absolute time. A positive relative time is unsigned; a negative relative time is preceded by a – (minus) sign. A relative time is often subtracted from or added to another relative or absolute time. Relative times that DTS uses internally are opaque binary timestamps. The DTS API offers several routines that can be used to calculate new times by use of relative binary timestamps.

20.1.2.1 Representing Periods of Time

A given duration of a period of time can be represented by a data element of variable length that uses the syntax shown in Figure 20-4.

Figure 20–4. Syntax for Representing a Duration



20.1.2.2 The Data Element Parts

The data element contains the following parts:

- The designator **P** precedes the part that includes the calendar components, including the following:
  - — The number of years followed by the designator **Y**

  - The number of months followed by the designator **M**
  - The number of weeks followed by the designator **W**
  - The number of days followed by the designator **D**
- The **T** designator precedes the part that includes the time components, including the following:
  - The number of hours followed by the designator **H**
  - The number of minutes followed by the designator **M**
  - The number of seconds followed by the designator **S**
- The designator **I** precedes the number of seconds of inaccuracy.

The following example represents a period of 1 year, 6 months, 15 days, 11 hours, 30 minutes, and 30 seconds and an unspecified inaccuracy:

```
P1Y6M15DT11H30M30S
```

The following example represents a period of 3 weeks and an inaccuracy of 4 seconds:

```
P3WI4
```

20.2 Time Structures

DTS can convert among several types of binary time structures that are based on different base dates and time unit measurements. DTS uses UTC-based time structures and can convert other types of time structures to its own presentation of UTC-based time. The DTS API routines are used to perform these conversions for applications on your system.

Table 20-1 lists the absolute time structures that the DTS API uses to modify binary times for applications.

Table 20–1. Absolute Time Structures

Structure	Time Units	Base Date	Approximate Range
utc	100-nanosecond	15 October 1582	A.D. 1 to A.D. 30,000
tm	second	1 January 1900	A.D. 1 to A.D. 30,000
timespec	nanosecond	1 January 1970	A.D. 1970 to A.D. 2106

Table 20-2 lists the relative time structures that the DTS API uses to modify binary times for applications.

Table 20–2. Relative Time Structures

Structure	Time Units	Approximate Range
utc	100-nanosecond	+/– 30,000 years
tm	second	+/- 30,000 years
reltimespec	nanosecond	+/- 68 years

The remainder of this section explains the DTS time structures in detail.

20.2.1 The utc Structure

UTC is useful for measuring time across local time zones and for avoiding the seasonal changes (summer time or daylight savings time) that can affect the local time. DTS uses 128-bit binary numbers to represent time values internally; throughout this guide, these binary numbers representing time values are referred to as *binary timestamps*. The DTS **utc** structure determines the ordering of the bits in a binary timestamp; all binary timestamps that are based on the **utc** structure contain the following information:

- The count of 100-nanosecond units since 00:00:00.00, 15 October 1582 (the date of the Gregorian reform to the Christian calendar)

- The count of 100-nanosecond units of inaccuracy applied to the preceding item
- The TDF, expressed as the signed quantity
- The DTS version number

The binary timestamps that are derived from the DTS **utc** structure have an opaque format. This format is a cryptic character sequence that DTS uses and stores internally. The opaque binary timestamp is designed for use in programs, protocols, and databases.

Note: Applications use the opaque binary timestamps when storing time values or when passing them to DTS.

The API provides the necessary routines for converting between opaque binary timestamps and character strings that can be displayed and read by users.

20.2.2 The tm Structure

The **tm** structure is based on the time in years, months, days, hours, minutes, and seconds since 00:00:00 GMT (Greenwich Mean Time), 1 January 1900. The **tm** structure is defined in the **time.h** header file.

The **tm** structure declaration follows:

```
struct tm {
     int tm_sec;    /* Seconds (0 - 59)                  */
     int tm_min;    /* Minutes (0 - 59)                  */
     int tm_hour;   /* Hours (0 - 23)                    */
     int tm_mday;   /* Day of Month (1 - 31)             */
     int tm_mon;    /* Month of Year (0 - 11)            */
     int tm_year;   /* Year - 1900                       */
     int tm_wday;   /* Day of Week (Sunday = 0)          */
     int tm_yday;   /* Day of Year (0 - 364)             */
     int tm_isdst;  /* Nonzero if Daylight Savings Time  */
                    /*  is in effect                     */
          };
```

Not all of the **tm** structure fields are used for each routine that converts between **tm** structures and **utc** structures. (See the parameter descriptions contained in the reference pages in the *DCE 1.2.2 Application Development Reference* for additional information about which fields are used for specific routines.)

20.2.3 The timespec Structure

The **timespec** structure is normally used in combination with or in place of the **tm** structure to provide finer resolution for binary times. The **timespec** structure is similar to the **tm** structure, but the **timespec** structure specifies the number of seconds and nanoseconds since the base time of 00:00:00 GMT, 1 January 1970. You can find the structure in the **dce/utc.h** header file.

The **timespec** structure declaration follows:

```
struct timespec {
   time_t  tv_sec;       /*  Seconds since 00:00:00 GMT,  */
                         /*   1 January 1970              */
   long    tv_nsec;      /*  Additional nanoseconds since */
                         /*   tv_sec                      */
             }    timespec_t;
```

20.2.4 The reltimespec Structure

The **reltimespec** structure represents relative time. This structure is similar to the **timespec** structure, except that the first field is *signed* in the **reltimespec** structure. (The field is *unsigned* in the **timespec** structure.) You can find the **reltimespec** structure in the **dce/utc.h** header file.

The **reltimespec** structure declaration follows:

```
struct reltimespec {
         time_t tv_sec;   /*  Seconds of relative time    */
```

```
    long   tv_nsec;  /*  Additional nanoseconds of   */
                     /*   relative time              */
         }   reltimespec_t;
```

20.3 DTS API Header Files

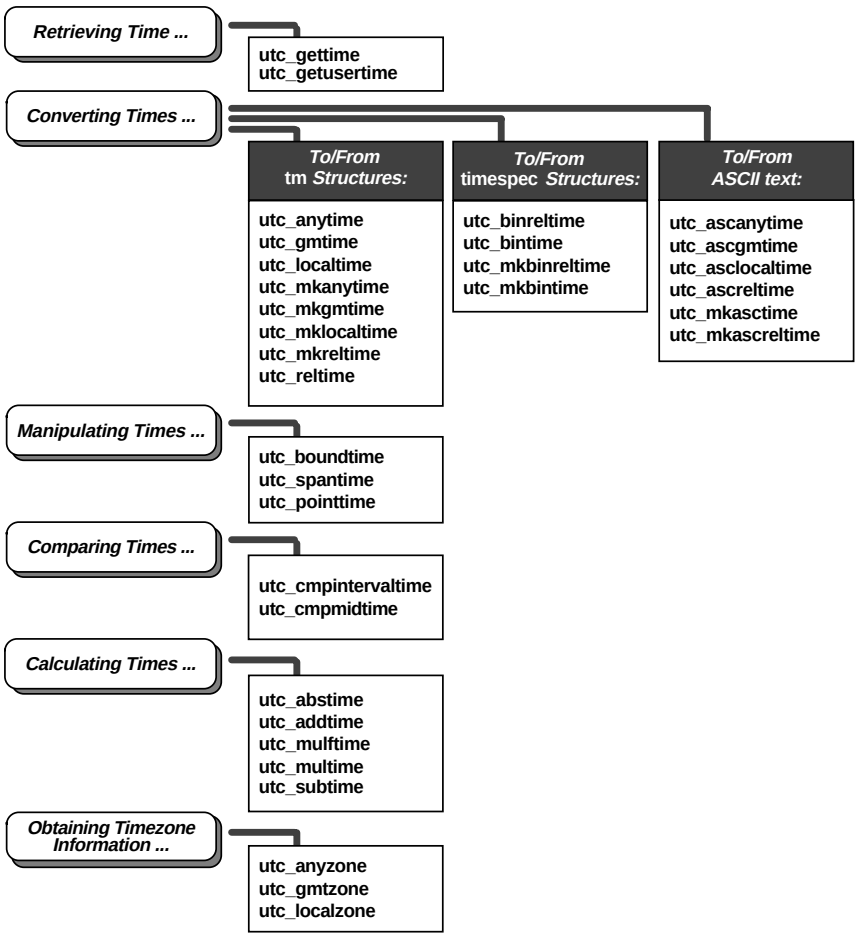
The **time.h** and **dce/utc.h** header files contain the data structures, type definitions, and define statements that are referenced by the DTS API routines. The **time.h** header file is a standard UNIX file. The **dce/utc.h** header file includes **time.h** and contains the **timespec**, **reltimespec**, and **utc** structures.

These header files are located in **/usr/include/dce**.

20.4 DTS API Routine Functions

Figure 20-5 categorizes the DTS portable interface routines by function.

Figure 20–5. DTS API Routines Shown by Functional Grouping



An alphabetical listing of the DTS portable interface routines and a brief description of each one follows:

- **utc_abstime**: Computes the absolute value of a binary relative timestamp
- **utc_addtime**: Computes the sum of two binary timestamps; the timestamps can be two relative times or a relative time and an absolute time

- **utc_anytime**: Converts a binary timestamp into a **tm** structure by using the TDF information contained in the timestamp to determine the TDF returned with the **tm** structure
- **utc_anyzone**: Gets the time zone label and offset from GMT by using the TDF contained in the input **utc**
- **utc_ascanytime**: Converts a binary timestamp into an ASCII string that represents an arbitrary time zone
- **utc_ascgmtime**: Converts a binary timestamp into an ASCII string that expresses a GMT time
- **utc_asclocaltime**: Converts a binary timestamp to an ASCII string that represents a local time
- **utc_ascreltime**: Converts a binary timestamp that expresses a relative time to its ASCII representation
- **utc_binreltime**: Converts a relative binary timestamp into two **timespec** structures that express relative time and inaccuracy
- **utc_bintime**: Converts a binary timestamp into a **timespec** structure
- **utc_boundtime**: Given two UTC times, one before and one after an event, returns a single UTC time whose inaccuracy includes the event
- **utc_cmpintervaltime**: Compares two binary timestamps or two relative binary timestamps
- **utc_cmpmidtime**: Compares two binary timestamps or two relative binary timestamps, ignoring inaccuracies
- **utc_gettime**: Returns the current system time and inaccuracy as an opaque binary timestamp
- **utc_getusertime**: Returns the time and process-specific TDF, rather than the system-specific TDF
- **utc_gmtime**: Converts a binary timestamp into a **tm** structure that expresses GMT or the equivalent UTC
- **utc_gmtzone**: Gets the time zone label, given **utc**
- **utc_localtime**: Converts a binary timestamp into a **tm** structure that expresses local time
- **utc_localzone**: Gets the time zone label and offset from GMT, given **utc**

- **utc_mkanytime**: Converts a **tm** structure and TDF (expressing the time in an arbitrary time zone) into a binary timestamp
- **utc_mkascreltime**: Converts a null-terminated character string, which represents a relative timestamp, to a binary timestamp
- **utc_mkasctime**: Converts a null-terminated character string, which represents an absolute timestamp, to a binary timestamp
- **utc_mkbinreltime**: Converts a **timespec** structure expressing a relative time to a binary timestamp
- **utc_mkbintime**: Converts a **timespec** structure into a binary timestamp
- **utc_mkgmtime**: Converts a **tm** structure that expresses GMT or UTC to a binary timestamp
- **utc_mklocaltime**: Converts a **tm** structure that expresses local time to a binary timestamp
- **utc_mkreltime**: Converts a **tm** structure that expresses relative time to a binary timestamp
- **utc_mulftime**: Multiplies a relative binary timestamp by a floating-point value
- **utc_multime**: Multiplies a relative binary timestamp by an integer factor
- **utc_pointtime**: Converts a binary timestamp to three binary timestamps that represent the earliest, most likely, and latest time
- **utc_reltime**: Converts a binary timestamp that expresses a relative time into a **tm** structure
- **utc_spantime**: Given two (possibly unordered) binary timestamps, returns a single UTC time interval whose inaccuracy spans the two input timestamps
- **utc_subtime**: Computes the difference between two binary timestamps that express either an absolute time and a relative time, two relative times, or two absolute times

Chapter 21

Time-Provider Interface

This chapter describes the Time-Provider Interface (TPI) for DCE Distributed Time Service software. The chapter provides a brief overview of the TPI, explains how to use external time-providers with DTS, and describes the data structures and message protocols that make up the TPI.

Coordinated Universal Time (UTC) is widely used and is disseminated throughout the world by various standards organizations. Several manufacturers supply devices that can acquire UTC time values via radio, satellite, or telephone. These devices can then provide standardized time values to computer systems. Normally, one device is connected to a computer system; the device runs a process that interprets signals and translates them to time values, which can either be displayed or be provided to the server process running on the connected system.

To synchronize its system clock with UTC using an external time-provider device, a DTS server needs a software interface to the device to periodically obtain UTC. In effect, this interface serves as an intermediary between the DTS server and external time-provider processes. The DTS server requires the interface to obtain UTC time values and to determine the associated inaccuracy of each value. The interface between

the DTS server process and the time-provider process is called the *Time-Provider Interface*.

The remainder of this chapter describes the TPI and its attendant processes in detail. The following section describes the control flow between the DTS server process, the TPI, and the time-provider process.

21.1 General TPI Control Flow

When you use a time-provider with a system running DTS, the external time-provider is implemented as an independent process that communicates with a DTS server process through remote procedure calls (RPCs). A remote procedure call is a synchronous request and response between a main calling program and a procedure executing in another process. RPC applications are based on the client/server model. In this context, the following processes act as the client and server components in the RPC-based application:

- The DTS daemon is the client.
- The Time-Provider process (TP process) is the server.

Both the RPC-client (DTS daemon) and the server (TP process) must be running on the same system.

Applications running on RPC communicate through an interface that is well known to both the client and the server. The RPC interface consists of a set of procedures, data types, and constants that describe how a client can invoke a routine running on the server. The server offers the interface to the clients through the Interface Definition Language (IDL) file.

The IDL file defines the syntax for an operation, including the following:

- The name of the operation
- The data type of the value that the operation returns (if any)
- The order and data types of the operation's parameters (if any)

The TP process offers two procedures that DTS calls to obtain time values. These procedures are **ContactProvider** and **ServerRequestProviderTime**.

At each system synchronization, DTS makes the initial remote procedure call (**ContactProvider**) to a TP process that is assumed to be running on the same node.

If the TP process is active, the RPC call returns the following arguments:

- A successful communication status message
- A control message that DTS uses for further processing

If the TP process is not active, the RPC call either returns a communication status failure or a time-out occurs. DTS then synchronizes with other servers instead of with the external time-provider.

If the initial call (**ContactProvider**) is successful, DTS makes a second call (**ServerRequestProviderTime**) to retrieve the timestamps from the external time-provider. The control message sent by the TP process in the first RPC call specifies the length of time DTS waits for the RPC call to complete. The TP process returns the following parameters in the procedure call:

- A communication status message.
- A time structure that contains timestamps collected from the external time-provider. (DTS then uses these timestamps to complete its synchronization.)

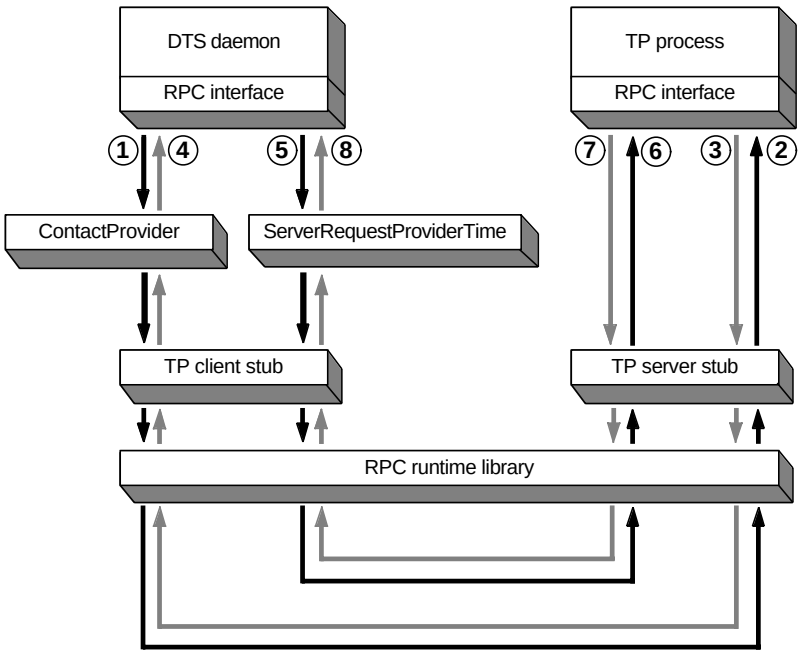
Figure 21-1 illustrates the RPC calling sequence between DTS and the TP process. Note that solid black lines represent the path followed by input parameters; dashed lines represent the path followed by output parameters and return values.

The following steps describe the process shown in Figure 21-1:

1. At synchronization time, DTS calls the **ContactProvider** remote procedure. Input parameters are passed to the TP client stub, dispatched to the RPC runtime library, and then passed to the TP server stub.
2. The TP process receives the call and executes the **ContactProvider** procedure.
3. The procedure terminates and returns the results through the TP server stub, the RPC runtime library, and the TP client stub.
4. The procedure terminates in the DTS call, where the returned parameters are examined.

5. DTS then calls the **ServerRequestProviderTime** remote procedure. Input parameters are passed to the TP client stub, dispatched to the RPC runtime library, and then passed to the TP server stub.

Figure 21–1. DTS/Time-Provider RPC Calling Sequence



6. The TP process receives the call and executes the **ServerRequestProviderTime** procedure.
7. The procedure terminates and returns the results through the TP server stub, the RPC runtime library, and the TP client stub.
8. The DTS remote procedure call terminates and the timestamps are returned as an output parameter. DTS then synchronizes using the timestamps returned by the external time-provider.

The following section describes the remote procedures that are exported by the TP process during the previous sequence.

21.1.1 ContactProvider Procedure

ContactProvider is the first routine called by DTS. The routine is called to verify that the TP process is running and to obtain a control message that DTS uses for subsequent communications with the TP process and for synchronization after it receives the timestamps. The parameters passed in the **ContactProvider** procedure call consist of the following elements:

- Binding Handle

  An input parameter that establishes the relationship between DTS and the TP process. A binding handle enables the client (DTS) to recognize and find a server (the TP process) that offers the same interface.

- Control Message

  An output parameter that contains information used by DTS for subsequent processing. The control message consists of the following elements:

  TPstatus One of the following values:

  - **K_TPI_SUCCESS**
  - **K_TPI_FAILURE**

  nextPoll A time value that tells DTS when to contact the TP process again. For example, once a day through dial-up, radio, or satellite.

  timeout A value that tells DTS how long to wait for a response from the TP process.

  noClockSet A value that specifies whether or not DTS is allowed to alter the system clock. If *noClockSet* is specified as 0x01 (TRUE), DTS does not adjust or set the clock during the current synchronization. This option is useful for systems whose system clock is known to be accurate (such as systems equipped with special hardware) or systems that are managed by some other time service (such as Network Time Protocol (NTP)), but which still wish to function as a DTS server.

- Communication Status

  An output parameter that contains a status code returned by the DCE RPC runtime library. The status **rpc_s_ok** is returned if the TP process is successfully contacted.

21.1.2 ServerRequestProviderTime Procedure

After the TP process is successfully contacted, DTS makes the **ServerRequestProviderTime** procedure call to obtain the timestamps from the external time-provider. The parameters passed in the **ServerRequestProviderTime** procedure call consist of the following elements:

- Binding Handle

  An input parameter that establishes the relationship between DTS and the TP process. A binding handle enables the client (DTS) to recognize and find a server (the TP process) that offers the same interface.

- Time Response Message

  An output parameter that contains a TP process status value (**K_TPI_SUCCESS** or **K_TPI_FAILURE**), a count of the timestamps that are returned, and the timestamps obtained from the external time-provider. The timestamp count is an integer in the range **K_MIN_TIMESTAMPS** to **K_MAX_TIMESTAMPS**. Each timestamp consists of three **utc** time values:

  — The system clock time immediately before the TP process polls the external time source. (The TP process normally obtains the time from the **utc_gettime()** DTS API routine.)

  — The time value returned to the TP process by the external time source.

  — The system clock time immediately after the external time source is read. (The TP process obtains the time from the **utc_gettime()** DTS API routine.)

- Communication Status

  An output parameter that contains a status code returned by the DCE RPC runtime library. The status **rpc_s_ok** is returned if the TP process is successfully contacted.

21.2 Time-Provider Process IDL File

A remote procedure call can only work if an interface definition that clearly defines operation signatures exists. Operation signatures define the syntax for an operation, including its name and parameters (input and output) that are passed as part of the procedure call. The TP process interface exports the two operation signatures that have been previously explained. The interface is provided in the file **examples/dts/**

dtsprovider.idl. When building the TP process application, this file must be compiled using the IDL compiler, which creates three files:

- **dtsprovider.h** (header file)
- **dtsprovider_sstub.c** (server stub file)
- **dtsprovider_cstub.c** (client stub file)

The Time-Provider program (TP program) must be compiled along with the **dtsprovider_sstub.c** code and then linked together. The TP program must also include the stub-generated file **dtsprovider.h**. The following sample code shows the structure of this interface.

```
/*
 *              Time Service Provider Interface
 *
 * This interface is defined through the Network Interface
 * Definition Language (NIDL).
 */
[uuid (bfca1238-628a-11c9-a073-08002b0dea7a),
    version(1)
]

interface time_provider
{

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

/* Minimum and Maximum number of times to read time source at
 * each synchronization
 */
const long K_MIN_TIMESTAMPS   = 1;
const long K_MAX_TIMESTAMPS   = 6;

/* Message status field return values
 */
const long K_TPI_FAILURE        = 0;
const long K_TPI_SUCCESS        = 1;
```

```
/* This structure contains one reading of the TP wrapped in
 * the timestamps of the local clock.
 */
typedef struct TimeResponseType
{
   utc_t beforeTime;  /* local clk just before getting UTC */
   utc_t TPtime;      /* source UTC; inacc also supplied   */
   utc_t afterTime;   /* local clk just after getting UTC  */
} TimeResponseType;

/* Time-provider control message.  This structure is returned
 * in response to a time service request.  The status field
 * returns TP success or failure.  The nextPoll gives the
 * client the time at which to poll the TP next.  The timeout
 * value tells the client how long to wait for a time response
 * from the TP.  The noClockSet will tell the client whether
 * or not it is allowed to alter the system clock after a
 * synchronization with the TP.
 */
typedef struct TPctlMsg
{
    unsigned long       status;
    unsigned long       nextPoll;
    unsigned long       timeout;
    unsigned long       noClockSet;
} TPctlMsg;

/* TP timestamp message.  The actual time-provider
 * synchronization data.  The status is the result of the
 * operation (success or failure).  The timeStampCount
 * parameter returns the number of timestamps being returned
 * in this message.  The timeStampList is the set of
 * timestamps being returned from the TP.
 */
typedef struct TPtimeMsg
{
    unsigned long       status;
    unsigned long       timeStampCount;
    TimeResponseType    timeStampList[K_MAX_TIMESTAMPS];
```

```
} TPtimeMsg;

/* The Time-Provider Interface structures are described here.
 * There are two types of response messages from the TP:
 * control message and data message.
 *
 *         <<<< TPI CONTROL MESSAGE >>>>
 *
 * 31                                          0
 * +-------------------------------------------+
 * |            Time-Provider Status           |
 * +-------------------------------------------+
 * |            Next Poll Delta                |
 * +-------------------------------------------+
 * |            Message Time Out               |
 * +-------------------------------------------+
 * |            NoSet Flag                     |
 * +-------------------------------------------+
 *
 *      <<<< a single timestamp >>>>
 *
 * 128                                         0
 * +-------------------------------------------+
 * |            Before Time                    |
 * +-------------------------------------------+
 * |            TP Time                        |
 * +-------------------------------------------+
 * |            After Time                     |
 * +-------------------------------------------+
 *
 *      <<<< TPI DATA MESSAGE >>>>
 *
 * 31                                          0
 * +-------------------------------------------+
 * |            Time-Provider Status           |
 * +-------------------------------------------+
 * |            Timestamp Count                |
 * +-------------------------------------------+
 * |                                           |
```

```
 * |            <timestamp one>                |
 * |                                            |
 * +--------------------------------------------+
 * |                   .                        |
 * |                   .                        |
 * |                   .                        |
 * |                   .                        |
 * |                   .                        |
 * +--------------------------------------------+
 * |                                            |
 * |      <timestamp K_MAX_TIMESTAMPS>          |
 * |                                            |
 * +--------------------------------------------+
 */

/* The RPC-based Time-Provider Program (TPP) interfaces are
 * defined here.  These calls are invoked by a Time Service
 * daemon running as a server (in this case it makes an RPC
 * client call to the TPP server).
 */

/* CONTACT_PROVIDER
 * Send initial contact message to the TPP.  The TPP server
 * responds with a control message.
 */
void ContactProvider
        (
        [in]    handle_t        bind_h,
        [out]   TPctlMsg        *ctrlRespMsg,
        [out]   error_status_t  *comStatus
        );

/* SERVER_REQUEST_PROVIDER_TIME
 * The client sends a request to the TPP for times.  The
 * TPP server responds with an array of timestamps obtained
 * by querying the Time-Provider hardware that it polls.
 */
void ServerRequestProviderTime
        (
        [in]    handle_t        bind_h,
```

```
        [out]   TPtimeMsg       *timesRspMsg,
        [out]   error_status_t  *comStatus
        );
}
```

21.3 Initializing the Time-Provider Process

Initializing the RPC-based TP process prepares it to receive remote procedure calls from a DTS daemon requesting the timestamps. The following steps are involved:

1. Include the header file (**dtsprovider.h**) that is created by compiling **/usr/include/dce/dtsprovider.idl**, which contains the interface definition.
2. Register the interface with the DCE RPC runtime.
3. Select one or more protocol sequences that are compatible with both the interface and the runtime library. It is recommended that the TP process application selects all protocol sequences available on the system. Available protocol sequences are obtained by calling an RPC API routine, described in the example that follows. This ensures that transport independence is maintained in RPC applications.
4. Register the TP process with the endpoint mapper service of the DCE daemon (**dced**) running on the system. This makes the TP process available to the DTS daemon.
5. Obtain the name of the machine's principal and then register an authentication service to use with authenticated remote procedure calls coming from the DTS daemon. Note that DTS and the TP program are presumed to be running in an authenticated environment.
6. Listen for remote procedure calls.

The following shows these steps, including the sequence of calls needed:

```
/* Register the TP server interface with the RPC runtime.
 * The interface specification time_provider_v1_0_ifspec
 * is obtained from the generated header file dtsprovider.h
 * The entry point vector is normally defined at the top of
 * the TP source program similar to this:
```

```
 *
 *    globaldef time_provider_v1_0_epv_t time_provider_epv =
 *    {
 *       ContactProvider,
 *       ServerRequestProviderTime
 *    };
 */
rpc_server_register_if (time_provider_v1_0_s_ifspec,
                        NULL,
                        (rpc_mgr_epv_t) &time_provider_epv,
                        &RPCstatus);

/*
 * This call tells the DCE RPC runtime to listen for remote
 * procedure calls using all supported protocol sequences.
 * To listen for a specific protocol sequence, use the
 * rpc_server_use_protreq call.
 */
rpc_server_use_all_protseqs (max_calls,
                             &RPCstatus);

/* This routine is called to obtain a vector of binding
 * handles that were established with registration of
 * protocol sequences.
 */
rpc_server_inq_bindings (&bind_vector,
                         &RPCstatus);

/* This routine adds the address information of the binding
 * handle for the TP server to the endpoint mapper database.
 */
rpc_ep_register (time_provider_v1_0_s_ifspec,
                 bind_vector,
                 NULL,
                 "Time-Provider",
                 &RPCstatus);

/* Obtain the name of the machine's principal and register an
 * authentication service to use for authenticated remote
 * procedure calls coming from the time service daemon.
```

```
 */
dce_cf_prin_name_from_host (NULL,
                            &machinePrincipalName,
                            &status);

rpc_server_register_auth_info (machinePrincipalName,
                               rpc_c_authn_dce_private,
                               NULL,
                               NULL,
                               &RPCstatus);

/* This routine is called to listen for remote procedure calls
 * sent by the DTS client.  Possible RPC calls coming from DTS
 * client are ContactProvider and ServerRequestProviderTime.
 */
rpc_server_listen (max_calls,
                   &RPCstatus);
```

21.4 Time-Provider Algorithm

The time-provider algorithm assumes that the two remote procedure calls will come in the following order: **ContactProvider** followed by **ServerRequestProviderTime**. The algorithm to create a generic time-provider follows:

1. Initialize the TP process, as described previously. Listen for RPC calls.
2. If the **ContactProvider** procedure is invoked, perform the following steps:
   a. Initialize the control message to the appropriate values (status value to **K_TPI_SUCCESS**; *nextPoll*, *timeout*, and *noClockSet* to valid integer values).
   b. Set the communication status output parameter to **rpc_s_ok**.
   c. Return from the procedure call. (The DCE RPC runtime returns the values to DTS.)
3. If the **ServerRequestProviderTime** procedure is run, perform the following steps:
   a. Initialize the timestamp count to the appropriate number.
   b. Use the **utc_gettime()** DTS API routine to read the system time.

c. Poll the external time source and read a UTC value. Use the **utc_gmtime()** routine to convert the UTC time value to a binary timestamp.

d. Use the **utc_gettime()** routine to read the system time.

e. Repeat steps b, c, and d the number of times specified by the values of **K_MIN_TIMESTAMPS** and **K_MAX_TIMESTAMPS**.

f. If steps b, c, or d return erroneous data, initialize the TP process status field (*TPstatus*) of the data message to **K_TPI_FAILURE**; otherwise, initialize the data message timestamps.

g. Set the communication status output parameter to **rpc_s_ok**.

h. Return from the procedure call. (The DCE RPC runtime sends the values back to DTS.)

4. The TP process continues listening for RPC calls.

21.5 DTS Synchronization Algorithm

DTS performs the following steps to synchronize with an external time-provider:

1. At startup time, creates the binding handle for the TPI. The binding handle is obtained from the list of available protocol sequences on the system.
2. At synchronization time, makes the remote procedure call **ContactProvider**, assuming that a TP process is running on the system. If the procedure call fails, examine the RPC communication status, checking the availability of the server. If the server is unavailable, synchronize with peer servers; otherwise, continue.
3. Waits for the procedure call to return the control message in the output parameter. If the procedure call does not return within the specified LAN timeout interval, synchronizes with peer servers. Otherwise, go to step 4.
4. If the procedure call returned successfully (communication status is **rpc_s_ok**), reads the data in the control message.
5. Makes the remote procedure call **ServerRequestProviderTime** to obtain the timestamps from the external time-provider. If the procedure does not return within the elapsed time specified by the control message (*timeout*), then synchronizes with peer servers. Schedules the next synchronization based upon the applicable DTS management parameters, ignoring *nextPoll*.

6. If the procedure returns successfully, verifies that the TP process status is **K_TPI_SUCCESS**. Otherwise, synchronizes with peer servers and schedule the next synchronization.
7. Extracts the timestamps from the data message and synchronizes using the timestamps.
8. Schedules the next synchronization time by adding the value of *nextPoll* seconds to the current time. At the next synchronization, goes to step 2.

Note: Application developers do not have to perform these steps; DTS performs these steps internally during synchronization with an external time-provider.

21.6 Running the Time-Provider Process

Both the TP process and the DTS daemon must run on the same system. The TP process must be started up under the login context of the machine's principal, which has root privileges. The DTS daemon and the TP process are started independently. However, before starting the TP process, ensure that **dced** is running on the system. If it is not running, start it. The TP process can always exit without affecting the DTS daemon. DTS dynamically reestablishes communications with the TP process when it creates binding handles.

21.7 Sources of Additional Information

Refer to the following for additional information:

- See **/examples/dts** for examples of time-provider programs that you can use with several different types of external time-provider devices.
- See the *DCE 1.2.2 Administration Guide—Core Components* for commercial sources of external time-providers.
- See the *DCE 1.2.2 Application Development Reference* for reference pages describing the RPC API and DTS API routines.

Chapter 22

DTS API Routines Programming Example

This chapter contains a C programming example showing a practical application of the DTS API programming routines. The program performs the following actions:

- Prompts the user to enter two sets of time coordinates corresponding to the timestamps of two ‘‘events.’’
- Stores those coordinates in a **tm** structure.
- Converts the **tm** structure to a **utc** structure.
- Prints out the **utc** structure in ISO text format.
- Determines which event occurred first.
- Determines if Event 1 may have caused Event 2 by comparing the intervals.

```
#include time.h       /* time data structures             */
#include dce/utc.h    /* utc structure definitions        */
```

```
void ReadTime();
void PrintTime();

/* This program requests user input about events, then prints
 * out information about those events.
 */
main()
{
    struct utc event1,event2;
    enum utc_cmptype relation;

    /* Read in the two events.
     */
    ReadTime(&event1);
    ReadTime(&event2);

    /* Print out the two events.
     */
    printf("The first event is : ");
    PrintTime(&event1);
    printf("\nThe second event is : ");
    PrintTime(&event2);
    printf("\n");

    /* Determine which event occurred first.
     */
    if (utc_cmpmidtime(&relation,&event1,&event2))
        exit(1);

    switch( relation )
    {
        case utc_lessThan:
        printf("comparing midpoints: Event1 < Event2\n");
        break;
        case utc_greaterThan:
        printf("comparing midpoints: Event1 > Event2\n");
        break;
        case utc_equalTo:
        printf("comparing midpoints: Event1 == Event2\n");
        break;
```

```
        default:
        exit(1);
        break;
    }

    /* Could Event 1 have caused Event 2?  Compare the
     * intervals.
     */
    if (utc_cmpintervaltime(&relation,&event1,&event2))
        exit(1);

    switch( relation )
    {
        case utc_lessThan:
        printf("comparing intervals: Event1 < Event2\n");
        break;
        case utc_greaterThan:
        printf("comparing intervals: Event1 > Event2\n");
        break;
        case utc_equalTo:
        printf("comparing intervals: Event1 == Event2\n");
        break;
        case utc_indeterminate:
        printf("comparing intervals: Event1 ? Event2\n");
        default:
        exit(1);
        break;
    }
}

/* Print out a utc structure in ISO text format.
 */
void PrintTime(utcTime)
struct utc *utcTime;
{
    char   string[50];

    /* Break up the time string.
     */
    if (utc_ascgmtime(string,      /* Out: Converted time  */
```

```
                    50,          /* In:  String length   */
                    utcTime))    /* In:  Time to convert */
        exit(1);
    printf("%s\n",string);
}

/* Prompt the user to enter time coordinates.  Store the
 * coordinates in a tm structure and then convert the tm
 * structure to a utc structure.
 */
void ReadTime(utcTime)
struct utc *utcTime;
{
struct tm tmTime,tmInacc;
    (void)memset((void *)&tmTime,  0, sizeof(tmTime));
    (void)memset((void *)&tmInacc, 0, sizeof(tmInacc));
    (void)printf("Year? ");
    (void)scanf("%d",&tmTime.tm_year);
    tmTime.tm_year -= 1900;
    (void)printf("Month? ");
    (void)scanf("%d",&tmTime.tm_mon);
    tmTime.tm_mon -= 1;
    (void)printf("Day? ");
    (void)scanf("%d",&tmTime.tm_mday);
    (void)printf("Hour? ");
    (void)scanf("%d",&tmTime.tm_hour);
    (void)printf("Minute? ");
    (void)scanf("%d",&tmTime.tm_min);
    (void)printf("Inacc Secs? ");
    (void)scanf("%d",&tmInacc.tm_sec);

    if (utc_mkanytime(utcTime,
                    &tmTime,
                    (long)0,
                    &tmInacc,
                    (long)0,
                    (long)0))
        exit(1);
}
```

Part 5

DCE Security Service

Chapter 23

Overview of Security

This chapter provides a brief overview of the two security services available in DCE:

- DCE Security Service
- Generic Security Services (GSS)

Refer to the *DCE 1.2.2 Application Development Reference* for detailed information on the Application Program Interfaces (APIs) discussed in the security chapters of this guide.

23.1 Purpose and Organization of the Security Chapters

This part of the guide explains the major features of DCE security so that you can decide what, if anything, you need to do to ensure that your DCE application is sufficiently secure. A lot of security is built into DCE, so in many cases you will need to do nothing, or very little, to secure your DCE application. Furthermore, you do not

need to understand all of the details of the DCE security services in order to use them effectively.

Following the overview of the DCE Security Service in this chapter are two chapters that contain conceptual discussions of authentication and authorization. The remaining chapters in this part of the guide discuss the DCE Security Service APIs—registry, login, extended registry attribute (ERA), extended privilege attribute (EPA), key management, access control list (ACL), password management, and ID map—and GSS credentials.

23.2 About Authenticated RPC

Perhaps the most important security facility is the authenticated remote procedure call (RPC) facility. Authenticated RPC enables distributed applications to participate in authenticated network communications. Applications using the authenticated RPC routines may select the authentication protocol and the authorization protocol to be used, and set various protocol-independent protection levels for communicating with remote entities (users, servers, and computers).

The use of authenticated RPC is explained in Chapters 13 and 14. Chapter 14 contains information about a number of RPC routines that relate directly to security issues, such as **rpc_binding_set_auth_info()**.

These security chapters, however, contains conceptual information that is useful for understanding the authentication and authorization protocols that authenticated RPC routines use; for this information, we recommend that you read Chapters 24 and 25, as well as this one.

23.3 About the GSSAPI

The GSS provides an alternate way of providing DCE security to distributed applications that handle network communications by themselves. With GSSAPI, you can include established applications in DCE and ensure the security and integrity of the applications and their data. In peer-to-peer communications, the application that establishes the secure connection is the *context initiator* or simply *initiator*. The context initiator is like a DCE RPC client. The application that accepts the secure

connection is the *context acceptor* or simply *acceptor*. The context acceptor is like a DCE RPC server.

The GSS available with DCE includes two sets of routines:

- Standard GSSAPI routines, which are defined in the Internet RFC 1509 ‘‘Generic Security Service API: C-bindings.’’ These routines have the prefix **gss_**.
- OSF DCE extensions to the GSSAPI routines. These are additional routines that enable an application to use DCE security services. These routines have the prefix **gssdce_**.

The chapters that follow provide information about how the GSSAPI routines use the authentication and authorization protocols. Chapter 26 provides information about GSS credentials, which are used to establish an application’s identity in DCE.

23.4 UNIX System Security and DCE Security

UNIX system security mostly presumes that a computer’s backplane can be trusted because computing operations are assumed to be local, and because the computer itself can be physically secured. In a distributed environment, the logical equivalent of the single system’s backplane is the network itself. Network computing means distributed, rather than localized, computing operations and, in the case of an open network (which DCE assumes), little of the network is physically secure. Thus, the nature of distributed systems poses special security risks, in addition to those posed by nondistributed systems. Unlike UNIX system security, DCE security is designed specifically to address those risks.

These considerations notwithstanding, network security is ultimately dependent on the security features that are local to the individual computers in the network and, what is more important, the manner in which those features are used and administered. Since any compromise to the local security of a computer in the distributed environment may introduce opportunities for compromising network security, DCE security does not diminish the importance of local security. In fact, the relative importance of local system security is greater in the distributed environment because the consequences of a local security breach may not be local. Finally, while DCE security does nothing to enhance local security, neither does it introduce any new avenues for compromising local security.

In the discussions in this guide, we assume you are familiar with the authentication and authorization features that UNIX systems provide: **/etc/passwd** and **/etc/group** file processing, routines that return or change file attributes, routines that return or change real or effective user IDs (UIDs) and group IDs (GIDs), and data encryption and decryption.

23.5 What Authentication and Authorization Mean

There are two questions that DCE security can answer for a principal about another principal with which it might want to communicate:

- Is this principal really who it says it is?
- Does it have the right to do what it wants to do?

Depending on the answers to these questions, a security-sensitive principal takes different courses of action with respect to a principal with which it is communicating.

To authenticate a principal means to verify that the principal is representing its true identity. To authorize a principal means to grant permission for the principal to perform an operation. While distinct, the concepts of authentication and authorization are also intertwined. For one thing, a principal's authorization is explicitly linked to its identity. For another, there is the possibility that authorization data concerning an authenticated principal can be falsified, which raises the additional question, ''Should the authorization data concerning this principal be believed?'' To this question also, DCE security can provide an answer to a principal for which this issue is a concern.

The discussions of authenticated RPC refer to the specific mechanisms by which authentication and authorization are performed as authentication and authorization protocols. Authenticated RPC supports at least one of each. However, RPC documentation refers to authentication and authorization protocols as services. The security chapters use the term *protocol* instead of *service* in this context to prevent confusion between the protocol-independent DCE authentication and authorization services and the various authentication and authorization protocols that those services support.

The GSSAPI combines authentication and authorization under a single security concept called a *mechanism*. The security mechanism provides applications a choice

of either Kerberos security or Privilege Attribute Certificate (PAC) authorization under DCE security.

23.6 Authentication, Authorization, and Data Protection in Brief

When one principal talks to another in a distributed computing environment, there is a risk that communications between the two will provide a means for compromising the security of one or the other. For example, a client may attack a server, or a server may set a trap for clients. An attack is most likely to succeed if the malevolent principal can convince its victim that it is something other than what it really is (an attacker), and/or that it possesses authorization that it does not really have. A counterfeit identity and/or authorization data grants an attacker access that it presumably would not otherwise have, and so provides an opportunity for the attacker to do damage.

One way an attacker might obtain counterfeit credentials is to intercept network transmissions between a client and a server, and then attempt to decipher (and perhaps modify) the transmitted data. If the attacker is able to intercept *and decipher* a principal's authentication or authorization information, it can later use this data to masquerade as an authentic principal with proper authorization.

DCE security protects against these kinds of attacks. It contains features that enable principals to

- Detect whether data they receive has been modified in transit
- Be certain that an attacker will be unable to decipher any authentication and authorization data it may succeed in intercepting

DCE security gives DCE principals confidence that the identity and authorization of principals they communicate with are authentic.

Figure 23-1 is an extremely condensed and highly stylized representation of the essentials of DCE security in terms of the DCE shared-secret authentication protocol and the DCE authorization protocol. Unless we note otherwise, assume that discussions in the security chapters of this guide refer to these two protocols, used in conjunction with one another.

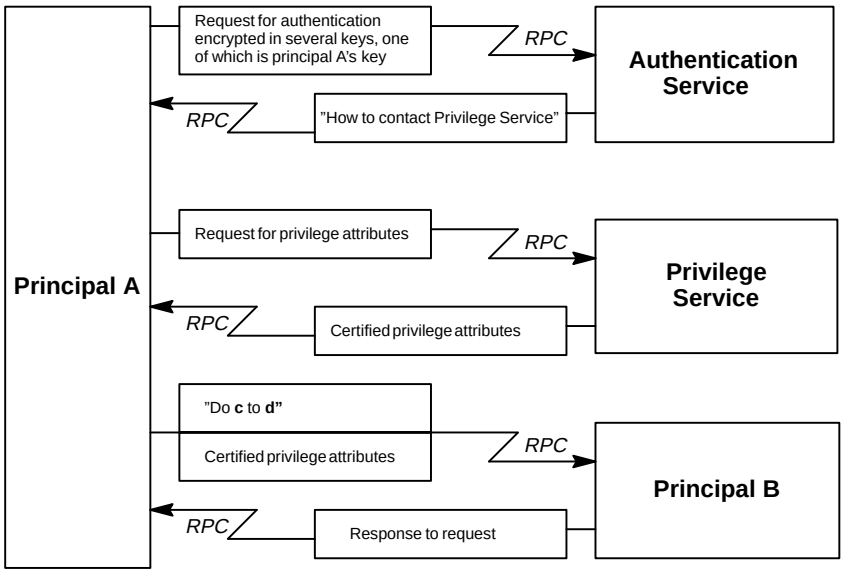
The following is a description of the events depicted in the illustration:

1. Principal A (which could be an attacker masquerading as Principal A) requests authentication of its identity from the authentication service. This request is encrypted using several keys, one of which is a key derived from the password supplied by Principal A. A copy of Principal A's key also exists in the registry database, having been stored there when the principal's account was created (or when the password was changed). It is thus available to the authentication service.

   The authentication service then obtains the registry's copy of Principal A's key and uses it to decrypt Principal A's authentication request. If the decryption succeeds, the keys are the same; Principal A is therefore authenticated and the authentication service replies with information that enables Principal A to ask the privilege service to authenticate its privilege attributes. (Privilege attributes are data used in making authorization decisions; they consist of the principal's name and group memberships.) If Principal A fails to get authenticated privilege attributes (also referred to as *credentials*), it may simply assert its privilege attributes to Principal B.

2. Principal A now makes a request to Principal B to perform some operation that requires the **c** permission to object **d**, and presents its certified privilege attributes. Principal B may grant or deny **c** access to **d** after examining the ACL that protects object **d**. (An ACL associates the privilege attributes of principals with permissions to an object.) If**c** is one of the permissions listed in the ACL granted to Principal A, then Principal A is allowed to perform the operation; if the **c** permission is not granted, A is denied access.

Figure 23–1. Shared-Secret Authentication and DCE Authorization in Brief



Had the authentication service been unable to decrypt the principal's authentication request, the principal would have been unauthenticated and, as a consequence, unable to acquire certified privilege attributes from the privilege service. In that case, Principal A might have simply asserted its privilege attributes to B; that is, claimed them for itself, without the benefit of having the privilege service certify this data as being genuine. Had Principal A then presented asserted privilege attributes to Principal B, then B might have denied the requested permission or granted it, depending on whether B grants permissions to unauthenticated principals, and whether **c** is among the permissions that B grants to such principals.

If Principals A and B are especially sensitive to security concerns, they may request that transmitted data be checked for integrity to establish whether it has been modified in transit, and possibly also encrypted to ensure that the data is unintelligible to any party other than Principals A and B.

23.7 Summary of DCE Security Services and Facilities

The DCE Security Service consists of services and facilities. The security services are

- The registry service, which maintains a database of principals, groups, organizations, accounts, and administrative policies.
- The authentication service, which verifies the identity of a principal and issues tickets that the principal uses to access remote services. (A ticket is data about a principal that is presented to the entity providing the service.)
- The privilege service, which certifies a principal's privilege attributes (that is, its name and group memberships, which are represented as UUIDs).

The three security services are implemented in a single daemon, the security server.

The DCE Security Service facilities are

- The login facility, which enables a principal to establish its network identity.
- The ERA facility, which extends the registry database to maintain attribute types and instances.
- The EPA facility, which provides access to the information in extended privilege attribute certificates (EPACs)
- The ACL facility, which enables a principal's access to an object to be determined by a comparison of the principal's privilege attributes to the object's permissions.
- The key management facility, which enables noninteractive principals (most frequently, servers) to manage their secret keys.
- The ID map facility, which maps cell-relative principal names to global principal names, and global principal names to cell-relative principal names. This facility is used in connection with the transmission of information about principals that are members of different DCE cells.
- The password management facility, which enables principal's passwords to be generated, and to be subjected to strength-checks beyond those defined in DCE standard policy.

For UNIX system compatibility with DCE, the DCE Security Service also provides implementations of UNIX system C library interfaces to the **/etc/passwd** and **/etc/group** files.

23.7.1 Interfaces to the Security Server

Following are the user interfaces to the security server itself (see the *DCE 1.2.2 Administration Guide—Core Components* and the *DCE 1.2.2 Command Reference*):

- **secd**

  The security daemon (a replicated server)

- **sec_create_db**

  Creates the security databases

- **sec_admin**

  Administers instances of the security daemon

- **sec_salvage_db**

  Converts the security database from one version of DCE to another

  Salvages a corrupted security database

- The security validation service of **dced**

  Enables clients of the security server to communicate with it

All other interfaces to the security server are more precisely characterized as interfaces to its three services: registry, authentication, and privilege.

23.7.1.1 Registry Service Interfaces

User interfaces to the registry service are described in the *DCE 1.2.2 Administration Guide—Core Components* and the *DCE 1.2.2 Command Reference*. Following is a summary of them:

- **rgy_edit**

  Edits registry database entries

- **passwd_import**

  Creates registry database entries from UNIX system **/etc/passwd** and **/etc/group** files

- **passwd_export**

Creates local registry information that corresponds to network registry database entries

- **chpass**

  Changes a user's password in a registry database entry

23.7.1.2 Authentication Service Interfaces

Following is a summary of the user interfaces to the authentication service when the default authentication protocol is in effect (the default protocol is DCE shared-secret, which is based on the Kerberos Version 5 network authentication system).

- **kinit**

  Obtains a login session's ticket(s) to remote services (the**login** and **su** tools also perform this service)

- **klist**

  Lists a login session's tickets to remote services

- **kdestroy**

  Destroys a login session's tickets to remote services

There are two security APIs that distributed applications are most likely to call to use the authentication service:

- Authenticated RPC facility
- GSSAPI

Although an application that uses GSSAPI may not make explicit calls to RPC routines, the GSSAPI implementation itself uses DCE RPC to communicate with the DCE registry.

23.7.1.3 Privilege Service Interfaces

There are no user interfaces or APIs to the privilege service. The login facility and authenticated RPC or GSSAPI encapsulate interactions between a principal and the privilege service.

23.7.2 Interfaces to the Login Facility

User interfaces to the login facility consist of the following tools:

- **dce_login**

  Enables an interactive principal to log into DCE, but does not change the principal's local identity

- **login**

  Enables an interactive principal to log in

- **su**

  Enables a logged-in interactive principal to assume a different principal identity

The API to the login facility consists of calls that are prefixed with **sec_login_**. This API enables application processes to assume their network identities. Network login and system login programs are examples of applications that call this API.

23.7.3 Interfaces to the Extended Registry Attribute Facility

The user interface to the ERA facility consists of DCE control program (**dcecp**) commands that allow users to modify the registry schema to create and maintain attribute types and to create and maintain instances of those types.

The API to the ERA facility consists of calls that are prefixed with **sec_rgy_attr_**.

23.7.4 Interfaces to the Extended Privilege Attribute Facility

There are no user interfaces to the EPA facility. The API to this facility consists of calls that are prefixed with **sec_cred_**. These routines extract data from EPACs.

23.7.5 Interfaces to the Key Management Facility

For a distributed application, it may be important for a server to have a network identity that is distinct from the principal identity it inherits from the user who invokes it or the host on which it runs. The key management facility provides features that enable noninteractive principals to manage their secret keys.

The user interface to the key management facility consist of a few **rgy_edit** subcommands that enable an administrator to maintain a key table. A remote interface allows users and administrators to maintain key tables on remote machines through the **dcecp keytab** verbs. A subset of local operations is also available though this interface. These subcommands call the key management API, which consists of several calls with the prefix **sec_key_**.

23.7.6 Interfaces to the ID Map Facility

There are no user interfaces to the ID map facility. The API to this facility consists of calls that are prefixed wht **sec_id_**. These routines map a global principal or group name into a cell name and a cell-relative principal or group name, and generate a global principal or group name from a cell name and a cell-relative principal or group name. This API also converts between the internal (UUID) representation of a name and the human-readable string.

23.7.7 Interfaces to the Access Control List Facility

The only user interface to the ACL facility is the **dcecp** ACL object **acl_edit**. This tool edits an object's ACL, the entries of which specify the permissions to the object that may be granted to principals possessing specified privilege attributes.

The ACL API consists of routines that are prefixed with **sec_acl_**. This is the same API that **acl_edit** calls, so an ACL editor or browser that is intended to replace **acl_edit** would call this API. A different case is that of an application server that needs to store and retrieve application-specific, access-control information for its clients. Such an application needs to implement its own ACL manager by using the DCE ACL library. (Refer to Chapter 32 for more information on ACL managers).

23.7.8 DCE Implementations of UNIX System Program Interfaces

DCE security provides implementations of UNIX system C library interfaces related to security. These are **getpwent()** and the related program interfaces to the **/etc/passwd** file, and **getgrent()** and the related program interfaces to the **/etc/group** file. Applications that bind with **libdce.a** are bound with the DCE security implementations of these interfaces.

23.7.9 Interfaces to the Password Management Facility

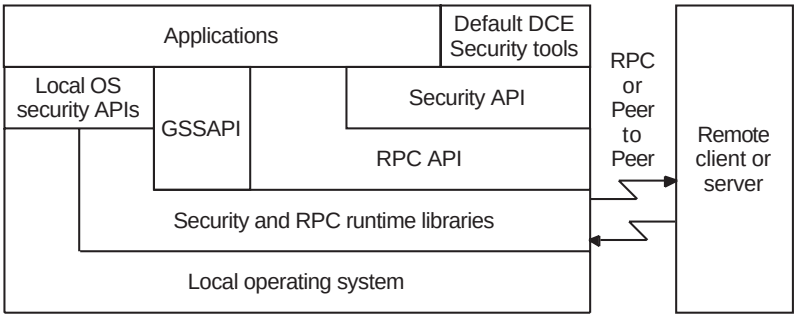
The user interface to the password management facility is provided by subcommands to the **rgy_edit** and **dcecp** commands. These subcommands enforce password management policy for principals and enable them to request generated passwords. See the **rgy_edit(8sec)** and **dcecp(8dce)** reference pages and the *DCE 1.2.2 Administration Guide—Core Components* for information on using these commands to create and change principal passwords.

The API to the password management facility consists of routines that are prefixed with **sec_pwd_mgmt_**. See the appropriate reference pages and Chapter 36 for information on these routines.

23.8 Relationships Between the DCE Security Service and DCE Applications

Figure 23-2 is a schematic illustration of the relationships among the interfaces to the DCE Security Service, and the relationship of security interfaces to DCE applications.

Figure 23–2. DCE Security and the DCE Application Environment



23.9 DTS, the Cell Namespace, and Security

The following subsections discuss the dependencies of DCE security on the Distributed Time Service (DTS), and the relationship between the security namespace and the Cell Directory Service (CDS) namespace. For information about how DCE components such as CDS use features of DCE security, refer to the documentation on the component of interest (for example, the section of the *DCE 1.2.2 Administration Guide—Core Components* on CDS).

23.9.1 DTS and Security

The DCE Security Service depends on a relatively close synchronization of network clocks, a service provided by DTS. When network clocks become too skewed, unexpired tickets to services may be regarded as invalid, and/or expired tickets considered valid. Excessive skewing can inconvenience users and introduce opportunities for security breaches; in the latter case, administrative intervention is required.

23.9.2 The Cell Namespace and the Security Namespace

The registry database maintains three security namespaces: the principal, group, and organization (PGO) namespaces. These namespaces are distinct from the cell namespace maintained by CDS. Security names take the following form:

/.../cell_name/pgo_name

CDS names take the following form:

/.../cell_name/pathname/object_name

Since the security namespace is rooted in the CDS namespace, security names have equivalent CDS names. Thus, for example, an entry for a principal in the registry database has the first of the following forms in the security namespace and the second of the following forms in the CDS namespace:

/.../cell_name/principal_name

/.../cell_name/security_mount_point/principal/principal_name

Note: The security mount point (*security_mount_point* as shown in the preceding syntax) is determined when DCE is configured. Therefore, the name may differ at individual sites.

There is no ambiguity about the security namespace to which a name refers because security names are always used in contexts that identify the namespace in question. For example, logging into DCE requires a principal name to be supplied.

However, an ACL is an object that is referenced not directly, but by the name of the object it protects. Since protected objects are not always security objects (and therefore may be registered *only* in the CDS namespace), ACL management interfaces always take CDS names rather than security names as input, whether or not it is the ACL of a security object (such as a registry database entry) that is being read or modified.

Chapter 24

Authentication

This chapter describes the authentication process of users and applications, as well as of principals in other cells.

Note: The authenticated RPC facility may also be referred to as the *protected* RPC facility, as it involves services beyond authentication.

24.1 Background Concepts

The following concepts, as they relate to this chapter, are described within this section:

- Principals, which are the subjects of authentication
- The shared-secret authentication protocol, which is the mechanism by which authentication is effected when applications specify this protocol via the authenticated RPC facility
- Cells, which are the environment where authentication takes place

- Protection levels, which are the various degrees to which transmitted application-level data may be protected
- Data encryption/decryption (cryptographic) algorithms, which are the mechanisms that the security server and client and server runtimes use to encrypt and decrypt data exchanged between principals

24.1.1 Principals

For the purposes of this discussion, the term *principal* may be precisely defined as an entity that is capable of believing it can communicate securely with another entity. In DCE security, principals are represented as entries in the registry database. DCE principals include the following:

- Users, who are also referred to as *interactive principals*
- Instances of DCE (system-level) servers
- Instances of application-level servers
- Computers (hosts) in a DCE cell
- Key distribution service (KDS) surrogates (these are used for cross-cell authentication; see Section 24.3)

The DCE security server itself comprises three principals that correspond to the three services that it provides: KDS, registry service, and privilege service. The KDS in turn provides two subservices: the authentication (sub)service and the ticket-granting (sub)service (TGS).

Note: As used in the literature, the term *authentication service* is sometimes ambiguous. This name may be, in places, associated with at least three distinct entities: the authentication (sub)service of the KDS, the KDS itself (comprising its authentication and ticket-granting subservices), and the entire DCE Security Service (comprising the KDS, the registry service, and the privilege service).

These three servers (KDS, registry service, and privilege service) comprise the main part of the DCE *network trusted computing base*. The KDS, registry service, and privilege service servers are commonly all implemented in a single process called the security server or security daemon.

24.1.2 The Shared-Secret Authentication Protocol

The registry service maintains a database, which contains an entry representing every principal, identifying the principal by its name and a secret key *bound* to it. It is this binding of the principal identity to a secret key shared with the registry that is at the root of the DCE shared-secret authentication protocols, as will be seen in this chapter. In the case of an interactive principal, the secret key is derived from the user's password (at login time). In order to establish its identity as a principal, a *noninteractive principal*, such as a server or computer, must store its secret key in a data file or hardware device, or rely on a system administrator to enter it. The secret keys of servers are considered to be stronger than those of users/clients, because they are truly random (as opposed to being derived from a password, which greatly restricts their randomness).

DCE shared-secret authentication implements an extended version of the Kerberos Version 5 system as its authentication protocol. Namely, the part of the DCE security server that corresponds to Kerberos is the KDS. The other parts (registry service and privilege service) do not occur in Kerberos. The Kerberos system was originally developed at the Massachusetts Institute of Technology as part of Project Athena, and provides a trustworthy, shared-secret authentication system. The walkthrough of the authentication protocol in this chapter describes the protocol in general terms.

Note: The KDS is an exceptional principal in that it does not share its key with any other principal. KDS surrogates (see Section 24.3) are also exceptional in that they are not autonomous participants in authenticated communications, as other kinds of principals are.

In the theory of shared-secret authentication all principals are initially considered to be untrusted, except for those in the trusted computing base itself (KDS, registry service, privilege service). A security-sensitive application must make use of the trusted computing base to convince itself of the level of trust it may place in all other principals. How that is done is the subject of this chapter.

24.1.3 Cells and Realms

The cell is the basic unit of configuration and administration in DCE. In terms of security, a cell is the set of principals that share a secret key with an instance of

the registry service. Therefore, each instance of a security server (together with its replicas) defines a separate cell.

From the perspective of security only, a cell is sometimes also known as a *realm* or *security domain*. (The term realm is often used in Kerberos documentation, and so may be more familiar to some readers than is the term *cell*.) A security cell is always configured to coincide with a corresponding CDS cell, and perhaps Distributed File System (DFS) cell as well. DCE documentation always refers to such a collective configuration of services as a cell.

24.1.4 Protection Levels

Protection levels specify how much of the information in network messages exchanged by principals is encrypted. As a rule, the higher the protection level, the greater the negative impact on performance. An application can set a protection level by using either authenticated RPC or GSSAPI.

24.1.4.1 Authenticated RPC and Protection Levels

The authenticated RPC facility provides several levels of protection so that applications can control tradeoffs between security and performance. Following is a summary of some of the protection levels that an application using authenticated RPC may specify:

- Connect level

  Performs authentication only when a client and server establish a relationship (or connection)

- Call level

  Attaches a verifier to each client call and server response that protects the system-level *metadata* of every remote call (but not the application-level data)

- Packet-integrity level

  In addition to protecting metadata, ensures the integrity of the application-level data (RPC and return parameters) transferred between two principals, that is, that none of it has been modified in transit

- Packet-privacy level

In addition to protecting metadata and integrity, encrypts all application-level data, thus guaranteeing its confidentiality

Refer to the discussion of authenticated RPC in Chapters 13 and 14 for complete information about protection levels.

24.1.4.2 GSSAPI and Protection Levels

Unlike authenticated RPC, where the client chooses a protection level that is then applied automatically to all data transferred in either direction, applications that use GSSAPI must explicitly protect data on a message-by-message basis. This allows an application the option of protecting only particularly sensitive messages, and avoids the overhead of security processing for other messages. (That is possible with RPC too, of course, provided that the programmer is willing to specify security attributes on an RPC call-by-call basis.)

GSSAPI offers two distinct types of protection through the **gss_sign()**/ **gss_verify()** routines and the **gss_seal()**/**gss_unseal()** routines, as follows:

- The **gss_sign()** routine creates a token containing an encrypted signature to protect the integrity of the message data. The token contains only the signature (not the message data). The application must send both the token and the message to which it applies to the peer application for verification. The receiving application calls the **gss_verify()** routine to check the signature.
- The **gss_seal( )**routine creates a token containing both an encrypted signature and the message data, and may optionally encrypt the message data. Only the token need be sent to the peer application, which processes it by using the **gss_unseal()** routine to verify the signature and extract the message data.

Three distinct signature algorithms are supported by the per-message protection routines. An algorithm may be requested by providing one of several constants to the **qop_request** parameter (**qop** stands for quality of protection) of either the **gss_sign()** or the **gss_seal()** routine. The constants are as follows:

GSSDCE_C_QOP_DES_MAC
Conventional DES MAC. Slow but well understood.

GSSDCE_C_QOP_DES_MD5
DES MAC of an MD5 (Message Digest #5) signature. Faster than DES MAC.

GSSDCE_C_QOP_MD5
MD5 signature. Fastest supported signature algorithm. The default.

24.1.5 Data Encryption Mechanisms

Authentication protocols assume the availability of a data encryption mechanism, parameterized by a so-called *crypto-variable* or *key*. In fact, it is the knowledge of such a key that is the concrete manifestation of the abstract notion of authentication. One mechanism that is frequently used is the Data Encryption Standard (DES), though the DCE security architecture supports other cryptographic algorithms. Your version of DCE security may use DES for data privacy or for principal authentication and data-integrity checking; or it may use another encryption mechanism, or no encryption at all. Consult the documentation supplied by your DCE vendor for specific information.

24.2 A Walkthrough of Shared-Secret Authentication Protocols

This section walks you through the following topics:

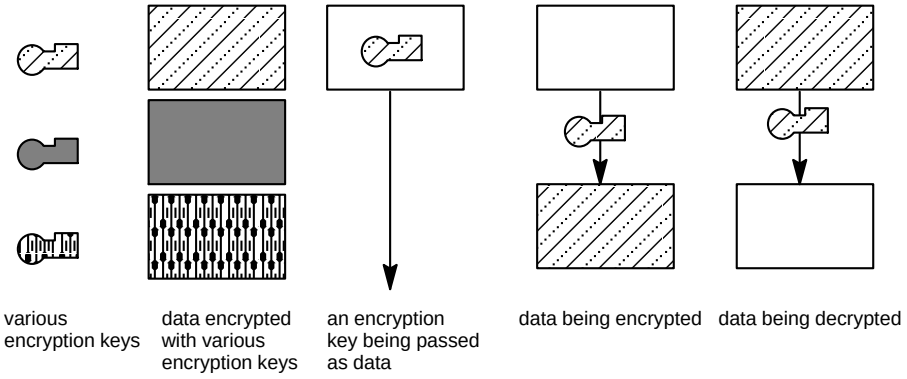
- Authentication of the user during login, in Section 24.2.1.
- Authentication of applications, in Section 24.2.2.

The walkthrough is seen primarily from the user and the associated application-client side. The illustrations in this chapter show only a high-level view (not low-level details) of what happens when a user logs in and runs an authenticated application; they are intended only to provide a general understanding of the protocol. (See the *Security Volume* of the *Application Environment Specification/Distributed Computing* for full details.)

In these figures, fill patterns represent encryption key values and encrypted data. The key symbol within a box indicates that a key is being passed as data. The key symbol on a line indicates that encryption or decryption is taking place, depending on whether

the resulting data is represented as encrypted or not. These conventions are shown in Figure 24-1.

Figure 24–1. Conventions Used in Authentication Walkthrough Illustrations



Note: All computer-to-computer communications initiated by DCE security are processed through the RPC mechanism, although the integration of security with client and server RPC runtimes are not illustrated or explained in any detail here.

Finally, note that to use shared-secret protocol, you do not need to understand how it works. It is described here so that application developers can determine whether it provides sufficient security for their needs. The discussion that follows is technical and detailed and may not be of interest to every reader.

24.2.1 Authenticating a User

This section explains how DCE security authenticates a user/client. DCE authentication basically consists of two successive procedures:

1. Acquisition by the security client of a ticket-granting ticket (TGT) for the user.
2. Acquisition by the security client of a privilege-ticket-granting ticket (PTGT) for the user.

These procedures are described in the following two subsections.

24.2.1.1 How the Client Obtains a TGT for the User

This section describes the acquisition, by the security client, of the user's TGT. It is the authentication service of the KDS that issues TGTs. Acquisition of the user's TGT is the first of the two parts of DCE user authentication. The other part is the acquisition of service tickets, which are issued by the TGS of the KDS.

Authentication protocols used by DCE security clients and servers to obtain TGTs for users, which is the first part of the user-authentication process, are:

- The *public key* protocol, which provides the highest level of security
- The *third-party* protocol, which is less secure than public key protocol
- The *timestamps* protocol, which is less secure than the third-party protocol
- The DCE Version 1.0 protocol, which is the least secure of the three and is provided solely to enable DCE Version 1.1 security servers to process requests from pre-DCE Version 1.1 clients

The protocol used by the security client when it makes a login request to the authentication service is determined as follows:

1. Pre-DCE Version 1.1 clients always use the DCE Version 1.0 protocol.
2. DCE Version 1.1 clients always use the third-party protocol, unless the host machine's session key, which the client uses to construct the request, is unavailable. It then uses the timestamps protocol.
3. DCE Version 1.2.2 clients always attempt to use the public key authentication protocol. If a client is unable to use the protocol, the client reverts to DCE Version 1.1 behavior.

The protocol used by the authentication service to respond to the client is determined by the following:

- The protocol used by the client making the login request
- The value of any **pre_auth_req** ERA attached to the requesting principal

The authentication service always attempts to reply by using the same protocol used by the client making the request, unless the value of the ERA forbids it to do so. (See the discussion of DCE Version 1.1 authentication in the *DCE 1.2.2 Administration*

Guide—Core Components for more detailed information on how security clients and the authentication service determine which protocol to use.)

For a general discussion of the security aspects of these protocols, and of security administration and security ERAs, see the *DCE 1.2.2 Administration Guide—Core Components*. The following subsections explain how the three protocols operate.

24.2.1.1.1 The Public Key Authentication Protocol

Public key authentication protocol works via public and private key-pairs. A user's identity is proven to the DCE Key Distribution Center (KDC) through a signature in the user's private authentication key. The KDC verifies the request through the user's authentication public key, which must be contained in the DCE registry. If the request is verified, the KDC replies with a TGT that is first signed by the KDC using its own private authentication key, and then is encrypted by the KDC using the client's key encipherment public key, which must be stored in the DCE registry. Because the KDC does not know the user's private keys, a compromise of the KDC cannot reveal the private keys. Therefore, public key users will not have any identifying information exposed to an intruder. This method of public and private key pair usage constitutes the public key protocol.

The public key protocol invokes routines **sec_login_validate_identity()**, **sec_login_valid_and_cert_ident()**, and **sec_login_validate_first()** as follows:

1. The user logs in.
2. The client process sends a message to the KDC. The message consists of a timestamp and nonce signed by the client's private digital signature key. An optional certificate of the client can also be sent along.
3. The KDC checks the timestamp and signature of the client's message. If the information is valid, the KDC sends a reply key to the client. The reply consists of a message signed by the KDC's digital signature key and then encrypted by the client's key encipherment key. The reply key is for encrypting the encrypted portion of the Kerberos **KRB_AS_REP** message, which includes the symmetric session key associated with the TGT. The session key used in association with the TGT is returned in the standard **EncKDCRepPart** field of the **KRB_AS_REP** message.

   If the KDC is unable to authenticate the user's supplied public key data, the KDC returns an error indicating why the authentication failed and whether the user is

required to use the public key authentication protocol. The KDC determines this from the **pre_auth_req** ERA attached to the user principal.

If the public key login attempt fails, the **sec_login** code instead falls back to the use of existing password-based authentication unless the KDC error information indicates that the principal is required to use public key login authentication. Preventing fall back is done by giving each principal a **pre_auth_req** ERA value of **PADATA_ENC_PUBLIC_KEY**.

Authentication information is transmitted as data types:

- **KRB5_PADATA_PUBKEY_REQ**
- **PADATA_ENC_PUBKEY_REP**

4. The client checks the signature on the reply to make sure it is from the KDC. The sesion key can be decrypted only by the legitimate client that possesses the private key needed to decrypt. The client then uses the TGT and associated session key.

24.2.1.1.2 Storage of the Private Key

Private key information is stored either in a local file or by the DCE private key storage server (PKSS). If the principal's **DCEPKPrivateKeyStorage** ERA value is not set, the login program assumes the private key is stored in a local file. If the principal's **DCEPKPrivateKeyStorage** ERA value is set, the login program obtains the private key from the private key storage mechanism associated with the UUID contained in the ERA. The currently supported storage mechanisms and their associated UUID's are the following:

- Local file — The UUID is **8687c5b8-b01a-11cf-b137-0800090a5254**.
- Private key storage server (PKSS) — The UUID is **72053e72-b01a-11cf-8bf5-0800090a5254**.
- Registry Database — The UUID is **adb48ed4-e94d-11cf-ab4b-08000919ebb5**. (This mechanism is supported for internal security server purposes only.)

The PKSS stores private keys in records that have the following information:

- The user's principal name.
- The user's public key.
- The key version of the user's public key (key v.n).

- The application domain. (Currently, private keys are used only in the context of a DCE login.)
- Key usage flags. (Currently, private keys are used only for authentication and for key encipherment.)
- Password hash value 2 (H2) derived from the user's password.
- The user's private key encrypted under the user's password hash value 1 (H1).

The PKSS cannot directly access the user's private key because it does not have the user's password H1 value. An ACL protects user records from unauthorized access, allowing access to only the **sec_admin** principal.

The following two descriptions depict the initial message exchange between the login client and the PKSS, and the second (final) exchange in which the PKSS returns the private key to the client.

Client Initiation of Private Key Acquisition from PKSS

The client DCE login program begins the process of key acquisition from the PKSS. Refer to Figure 24-2 as you read the following steps.

1. The login client sends a message to the PKSS that consists of the following components:
   - The user's principal name.
   - The application domain.
   - Key usage flags.
   - The key version number (key v.n).
   - An exponentiated Diffie-Hellman value (S^C) used for establishing a Diffie-Hellman key.
   - An algorithm list (alg list), which is a list of secret key encryption algorithms supported by the client (currently only DES). The client and the PKSS use this algorithm with the Diffie-Hellman key and the session key.
2. Upon receipt of this message from the login client, the PKSS generates a Diffie-Hellman value of its own (S^S). Using this value along with the client's Diffie-Hellman value, the PKSS computes a Diffie-Hellman key.

The PKSS determines whether it supports any of the algorithms listed in the client message. If so, it can communicate securely with the client and the PKSS selects one of the supported algorithms for use. (Currently, OSF DCE clients and servers support only DES.)

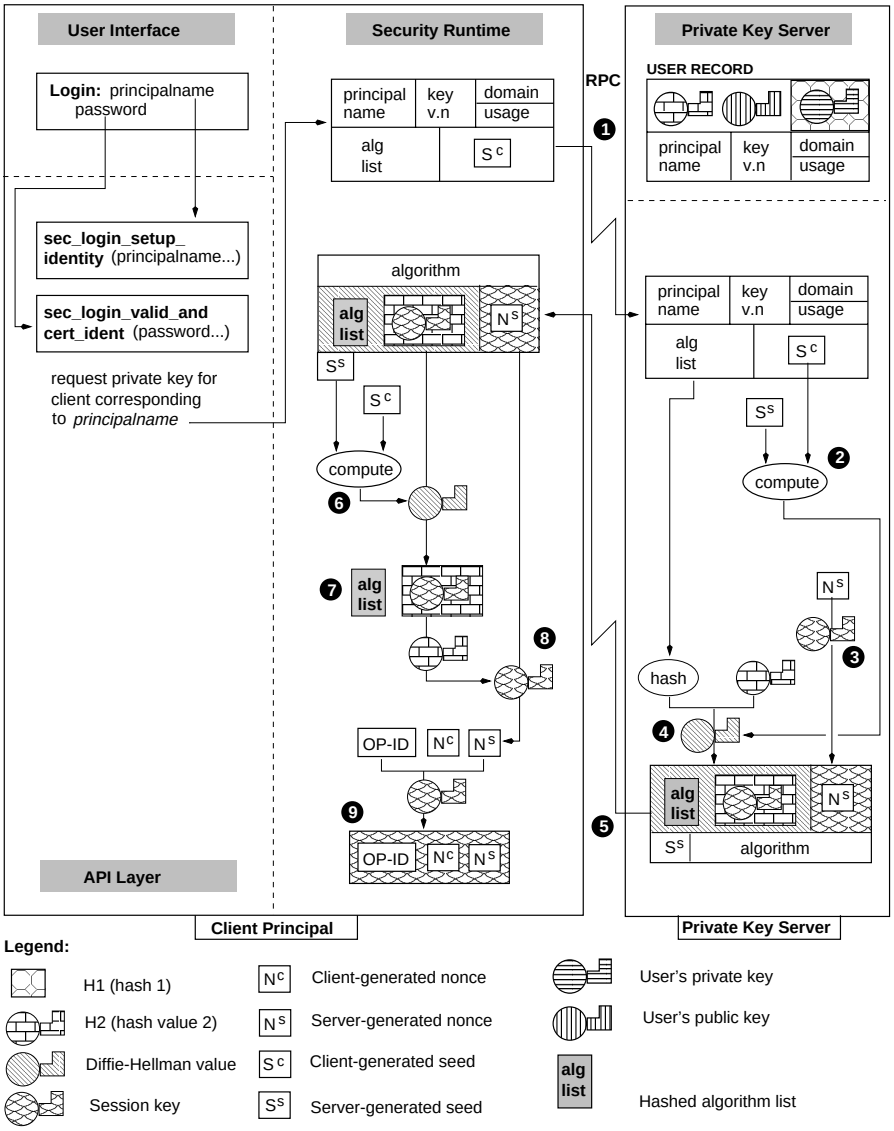
3. The PKSS generates a random session key and a nonce (N^S). The session key will be used to encrypt messages between the client and server. The PKSS encrypts the nonce (N^S) with the session key, then encrypts the session key with the user's password H2 value taken from the user record.
4. The PKSS computes a hash on the algorithm list provided by the client. It encrypts both the hashed algorithm list and the encrypted session key (see step 3) under the Diffie-Hellman key generated in step 2.
5. The PKSS composes and sends the client a message consisting of
   - The nonce (N^C) encrypted under the random session key.
   - The session key encrypted by the user's password H2 value and then encrypted under the Diffie-Hellman key.
   - The hashed algorithm list further encrypted under the Diffie-Hellman key.
   - The algorithm to be used for the session key. (The algorithm is chosen from the client's algorithm list.)
   - The PKSS-generated Diffie-Hellman value.
6. Upon receipt, the login client extracts the PKSS-generated Diffie-Hellman value and combines it with its own Diffie-Hellman value to obtain its copy of the Diffie-Hellman key.
7. The client uses the encryption algorithm specified by the PKSS, along with its Diffie-Hellman key to obtain the hashed algorithm list and the session key (still encrypted under the user's password H2 value).

   The login client computes a hash on its own algorithm list and compares it with the hashed algorithm list returned from the PKSS. The two lists must match. Otherwise, the client determines that the PKSS is invalid and returns an error.
8. The login client decrypts the session key by using H2 derived from the user's password. The client uses the session key to decrypt the PKSS-generated nonce (N^S). The session key will be used to authenticate the current session communications between the client and PKSS.

Note: A PKSS imposter would not know the user's password H2 value. The resulting session key would differ from the imposter's session key, preventing further communications between the client and the imposter.

9. The client composes a message encrypted under the session key. The message consists of
   - The PKSS-generated nonce (N^S)
   - A client-generated nonce (N^C)
   - An operation identifier that indicates private key acquisition

Figure 24–2. Client Initiation of Private Key Acquisition



Client Completion of Private Key Acquisition from PKSS

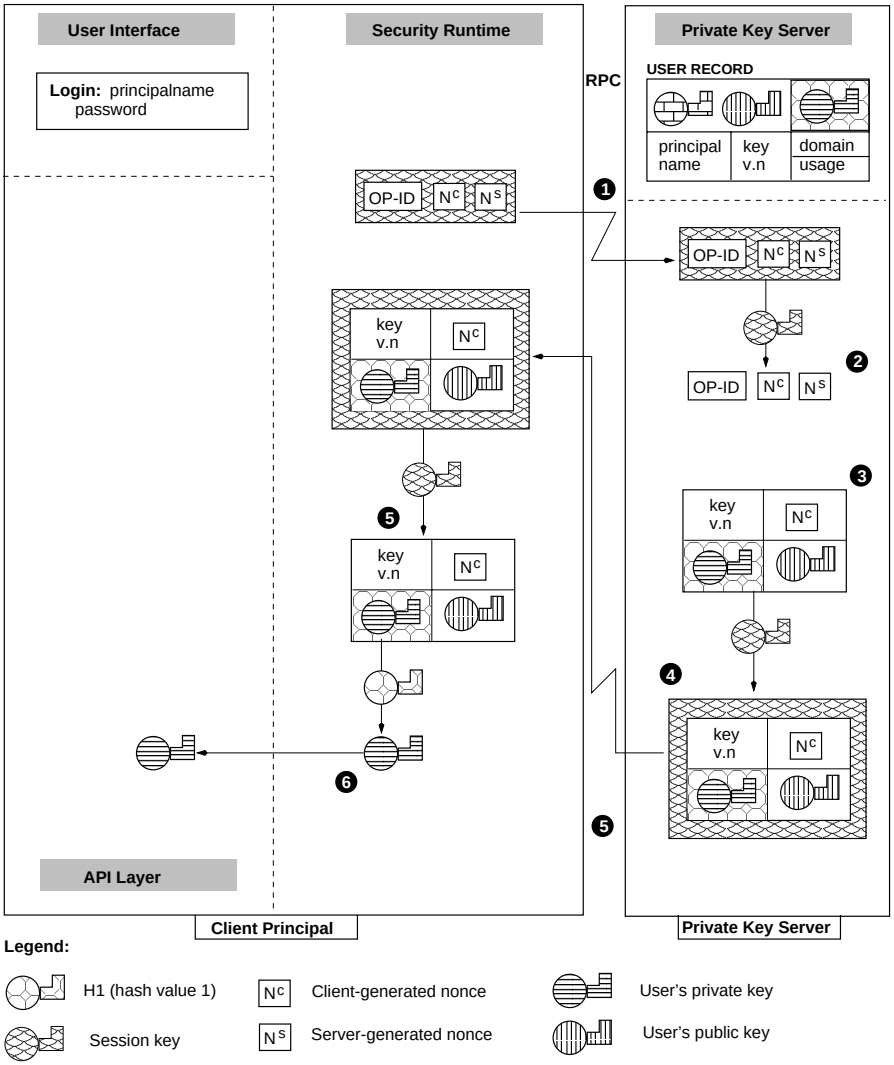
The client DCE login program completes the process of key acquisition from the PKSS. Refer to Figure 24-5 as you read the following steps.

1. The client sends the PKSS the composed message encrypted under the session key (see Step 9 in the preceding discussion).
2. Upon receipt of the client message, the PKSS uses the session key to obtain the operation ID, the client-generated nonce (N^C), and the PKSS-generated nonce (N^S).

   The PKSS compares the client's copy of N^S with its original nonce (N^S). A match proves the client had knowledge of the user's secret password which was needed to obtain the session key. The client used the session key to obtain N^S.
3. The PKSS composes a message consisting of
   - The user's private key encrypted under the user's password H1 value
   - The user's public key
   - The key version number
   - The client-generated nonce (N^C)
4. The PKSS encrypts this message with the session key and sends it to the client.
5. Upon receipt, the client uses the session key to obtain the user's private key (encrypted under the user's password H1 value), the user's public key, the key version number, and the client-generated nonce (N^C).

   The client compares the PKSS's copy of N^C with its own nonce. A match proves the authenticity of the PKSS because only the true PKSS could have used the correct user password H2 value to properly encrypt the session key passed to the client in the first message.
6. The client uses its password H1 value to decrypt the private key. The client's security runtime program returns the authenticated private key to the calling routine.

Figure 24–3. Client Acquisition of Private Key from PKSS



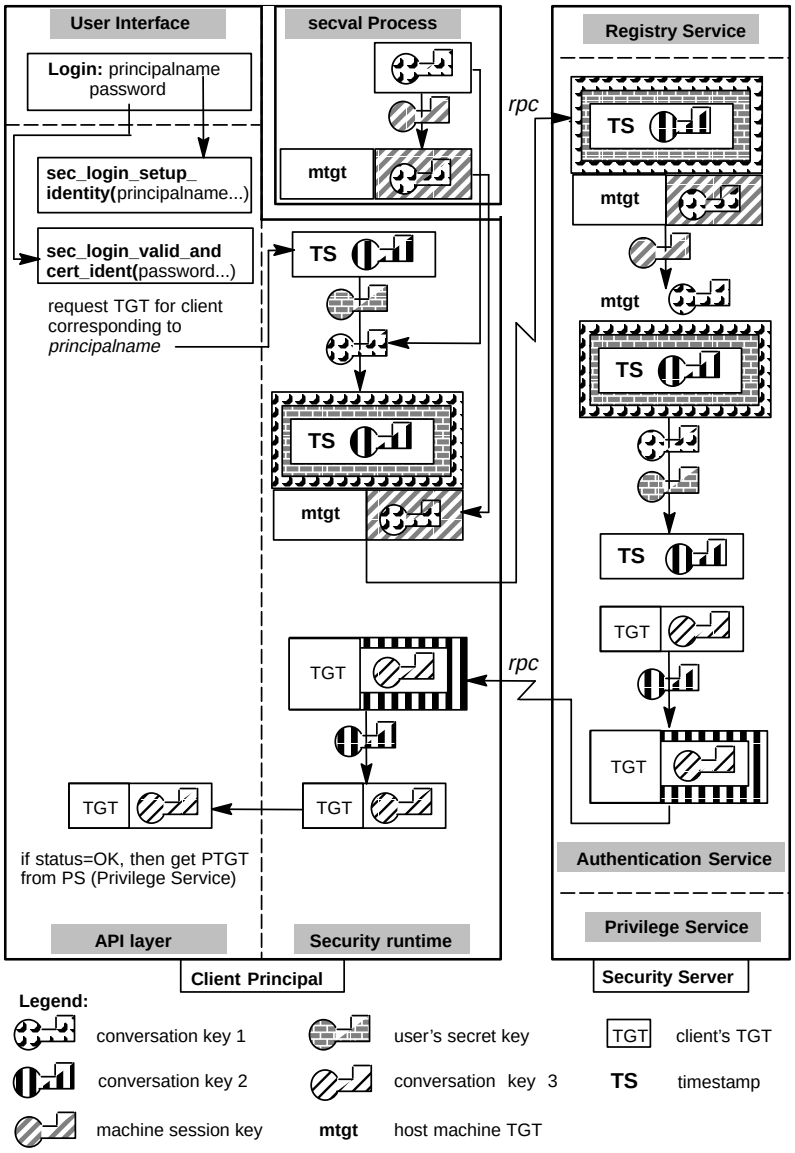
24.2.1.1.1 The Third-Party Authentication Protocol

The DCE Authentication Service can use the third-party authentication protocol to provide a user with a TGT. Refer to Figure 24-2 as you read the following steps.

1. The user logs in, entering the correct user name. The login program invokes **sec_login_setup_identity()**, which takes the user's principal name as one of its arguments, and **sec_login_valid_and_cert_ident()**, which has the user's password as one of its arguments. The **sec_login_valid_and_cert_ident()** routine causes the security runtime to request a TGT from the authentication service of the KDS. (The client principal will later present the TGT to the TGS, to acquire service tickets to other servers.) The client's security runtime performs the following steps to construct the TGT request to the authentication service:

   a. It requests, from the **secval** service, a random key, say conversation key 1, which the client will later use to encrypt its request to the authentication service. Two copies of conversation key 1 are passed to the client: one unencrypted and one encrypted in the machine session key (a copy of which is sealed inside the machine ticket-granting ticket, or MTGT). (In order to do this securely, the request to **secval** must be done over a secure local communications channel on the host machine.) It then concatenates the encrypted copy of conversation key 1 with the MTGT.

Figure 24–4. Client Acquires TGT Using Third-Party Protocol



b. It generates another random key, conversation key 2, which the authentication service will later use to encrypt the TGT it returns to the client. It then concatenates it to a timestamp string.

c. It derives, from the password input by the user, the user's secret key, a copy of which also exists in the registry service database. It then encrypts the timestamp/conversation key 2 twice: first by using the user's secret key, and then by using conversation key 1.

d. Finally, it completes constructing the authentication service request message by concatenating the encrypted conversation key 1 (obtained from **secval** in Step 1a) with the doubly encrypted timestamp and conversation key 1.

2. The client's security runtime then forwards the constructed request to the authentication service of the KDS. (This corresponds to the first step of the DCE Version 1.0 protocol, described in Section 24.2.1.1.3.)

3. The authentication service receives the request and performs the following steps to verify the user and prepare the user's TGT:

   a. It decrypts the MTGT (by using the KDS's secret key), and obtains the machine session key from it. (This decryption is not shown pictorially in Figure 24-2.)

   b. Using the machine session key, it decrypts the package containing conversation key 1.

   c. It obtains the user's secret key from the registry service and then decrypts the doubly encrypted package containing the timestamp and conversation key 2 by using the user's secret key and conversation key 1.

      If this decryption fails, the user's secret key that was used by the login program to encrypt the package differs from the one stored in the registry service, and therefore the password supplied to the login program by the user was incorrect. In this case, the user is not authenticated, and an error code is returned to the login program.

      If the decryption succeeds, and if the decrypted timestamp is within an allowable clock skew (5 minutes) of the current time, the user has been authenticated (that is, the user knows the correct principal password and this isn't a replay attack), and the authentication service proceeds with preparation of the user's TGT.

4. The authentication service then prepares the user's TGT, encrypts it in the KDS's secret key, encrypts the conversation key 3 contained in the TGT (to be used later

by the client to acquire service tickets) in conversation key 2, and returns this data to the client.

5. The client security runtime decrypts the reply from the authentication service by using conversation key 2, obtaining the conversation key 3 from the TGT, and it becomes part of the client's login context.

Note the following security safeguards inherent in the structure of this protocol:

- All network transmissions between the security client and the authentication service are encrypted by using strong random keys (not weak keys derived from passwords), placing even offline decryption attempts at the outer limits of practical possibility.
- The timestamp and conversation key 2 are encrypted by using the user's secret key, which is derived from the user's password (and subsequently reencrypted by using conversation key 1). This enables the authentication service to verify that the requesting client knows the user's password. (It does this by decrypting the package via the registry service's copy of the user's secret key; if the decryption succeeds, the keys are the same, that is, they were derived from the same password.)
- The authentication service actively verifies whether the requesting client knows the user's password. Contrast this with the DCE Version 1.0 protocol, where the authentication service blindly issues TGTs without requiring any evidence that the requestor knows the user's password. It is therefore aware of, and can manage, persistent login failures for a given user, eliminating active password-guessing attacks.
- The authentication service's reply is encrypted by using conversation key 2, which was provided by the client. This verifies to the client that the authentication service itself is authentic since, if it were not, it would not have been able to obtain the machine session key and user's secret key it needed to decrypt conversation key 2.

These safeguards provide assurance to both server and client that the entity with which each is communicating is, in fact, what it claims to be.

Having acquired the user's TGT, the login program proceeds with the next step in the authentication procedure (described in Section 24.2.1.2).

24.2.1.1.2 The Timestamps Authentication Protocol

This section describes how the DCE Authentication Service uses the timestamps authentication protocol to provide a user with a TGT.

Since the timestamps protocol is largely identical to the DCE Version 1.0 protocol, which is fully explained in the next section, this section describes only the differences between the two.

The timestamps protocol proceeds exactly as the DCE Version 1.0 protocol described in Section 24.2.1.1.5, with these additions:

- In Step 1, the client security runtime sends to the authentication service, in addition to the user's stringname, the current timestamp encrypted in the user's secret key.
- In Step 2, the authentication service, before preparing the user's TGT, verifies the user's authenticity (albeit not as strongly as in the third-party protocol) as follows:
  1. It decrypts the timestamp by using the copy of the user's key it obtained from the registry service.
  2. If the decryption succeeds, and the timestamp is within an allowable clock skew (5 minutes) of the current time, the user is authenticated, and the authentication service proceeds to prepare the TGT. If the decryption fails, or if the timestamp is not within the allowable clock skew, the authentication service rejects the login request.

With this protocol, the authentication service can verify the following:

- That the client login request is timely; that is, that the authentication service is communicating with the client now (within the allowable clock skew)
- That the requesting client knows the user's password

The authentication service is therefore aware of, and can manage, persistent login failures for a given user, eliminating passive password-guessing attacks.

From this point, the timestamps protocol continues as the DCE Version 1.0 protocol described in the next section, and then proceeds with the next step in the authentication procedure, described in Section 24.2.1.2.

Note: Encrypted timestamps (under the name authenticators) are passed in several places in the protocols, to guarantee fresh communications (within the

allowable clock skew) and thereby guard against replay attacks. This has been shown explicitly in the preceding, but will be omitted in the remainder of this chapter.

24.2.1.1.3 The DCE Version 1.0 Authentication Protocol

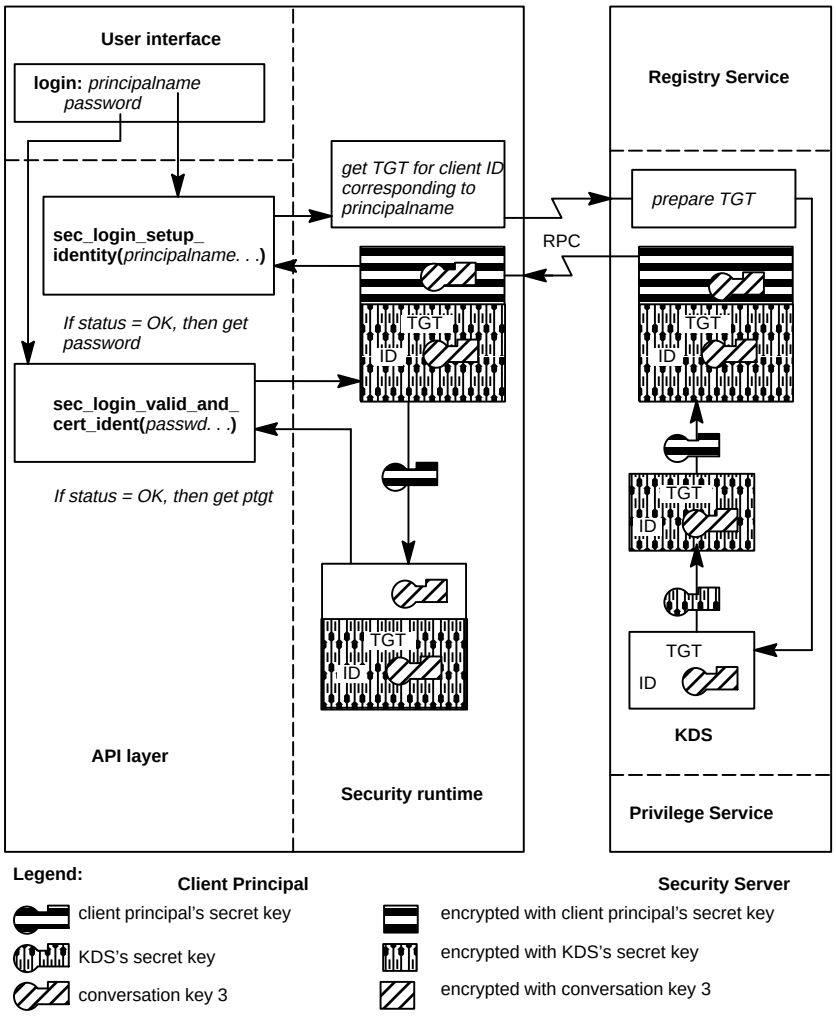
This section explains how the DCE Authentication Service uses the DCE Version 1.0 protocol to authenticate a user. This protocol exists in DCE Version 1.1 solely to provide interoperability between DCE Version 1.1 servers and pre-DCE Version 1.1 clients; *only* pre-DCE Version 1.1 clients transmit DCE Version 1.0 login requests, and the authentication service returns DCE Version 1.0 responses *only* to pre-DCE Version 1.1 clients.

The DCE Version 1.0 protocol lacks the security features previously described for the third-party and timestamps protocols, hence this protocol is more vulnerable to attacks. You should keep this in mind when you are considering the inclusion of pre-DCE Version 1.1 clients in your DCE Version 1.1 cell.

The DCE Version 1.0 protocol proceeds as follows. Refer to Figure 24-5 as you read these steps.

1. The user logs in, entering the correct user name. The login tool invokes **sec_login_setup_identity()**, which takes the user's principal name as one of its arguments. This call causes the client security runtime to request a TGT and passes the user's name (represented as a string, not a UUID) to the authentication service. The TGT will later be used by the client to acquire service tickets to other services; the first such usage will be to acquire a service ticket to the privilege service (see Section 24.2.1.2).

Figure 24–5. Client Acquires TGT Using the DCE Version 1.0 Protocol



2. Upon receiving the request for a TGT, the authentication service obtains the user's secret key from the registry service database (where the secret keys of all principals in the cell are stored). Using its own secret key (that is, that of the KDS), the authentication service encrypts the user's identity, along with a

conversation key 3 (this conversation key 3 is the same as conversation key 3 in the discussion of the third-party protocol, earlier in this chapter), in a TGT. The authentication service separately encrypts a copy of conversation key 3 with the user's secret key and returns this data to the client.

3. When this data arrives at the client, the login tool prompts the user for the password and invokes **sec_login_valid_and_cert_ident()** . This call passes the password to the client's security runtime library. The security runtime derives the user's secret key from the password (using a well-known algorithm), and uses it to decrypt conversation key 3. (If the user enters the wrong password, this decryption fails.) The client's security runtime cannot decrypt the TGT since it does not know the KDS's secret key. The TGT is the client principal's certificate of identity—it is usable by the client precisely because the client knows the conversation key 3 carried in it.

Note: One of the functions of **sec_login_valid_and_cert_ident()** is to authenticate the authentication service itself to the host machine's login program, by demonstrating that the (purported) authentication service really knows the secret key of the host computer. (The mere fact that the purported authentication service knew the user's secret key is not convincing to the host's login program, because that purported authentication service could have been a bogus server working in league with a bogus user—the host doesn't trust any of these things.) The way in which this is accomplished is not illustrated here but is explained in Chapter 30.

Having acquired the user's TGT, the login program proceeds with the next step in the authentication procedure, described in Section 24.2.1.2.

24.2.1.3 How the Client Obtains a PTGT for the User

This section describes the acquisition, by the client's security runtime, of the user's PTGT. Acquisition of the user's PTGT is the second of the two parts of DCE user authentication.

From this point on, the client principal uses four different conversation keys to talk with other principals. Use of multiple short-lived keys makes an attacker's task far more difficult, since there are more encryption keys to discover and less encrypted material and time with which to crack them.

Refer to Figure 24-6 as you read the following steps.

1. When the client's security runtime has succeeded in decrypting conversation key 3, it next wants to acquire a PTGT from the privilege service. Before a request for a PTGT can even be formulated, however, a service ticket to the privilege service must be acquired. The client's security runtime therefore begins by requesting such a service ticket from the TGS. The security runtime encrypts this request by using the conversation key 3 (which is also sealed in the client's TGT); it also sends along the client's TGT.

2. The TGS decrypts the TGT (which was encrypted in the KDS's secret key), learning conversation key 3, and verifies that the request was properly encrypted by using conversation key 3. This convinces the TGS that the identity of the requesting client is authentic; that is, no other principal could have sent a message so encrypted because no other principal knows conversation key 3. (The reader should review the preceding steps if necessary to be convinced that this is true.) Since the user has demonstrated to the TGS knowledge of the key, the TGS allows the user to talk to the privilege service, and so prepares a service ticket to that service. This ticket contains the identity of the user (and a conversation key 4), encrypted under the secret key of the privilege service (which the TGS retrieves from the registry service). The TGS separately encrypts conversation key 4 under conversation key 3, and returns this data to the client.

   Note: Beginning with Figure 24-6, the illustrations do not emphasize all the TGS's encryption and decryption activities (such emphasis would be redundant since the TGS knows all of the keys).

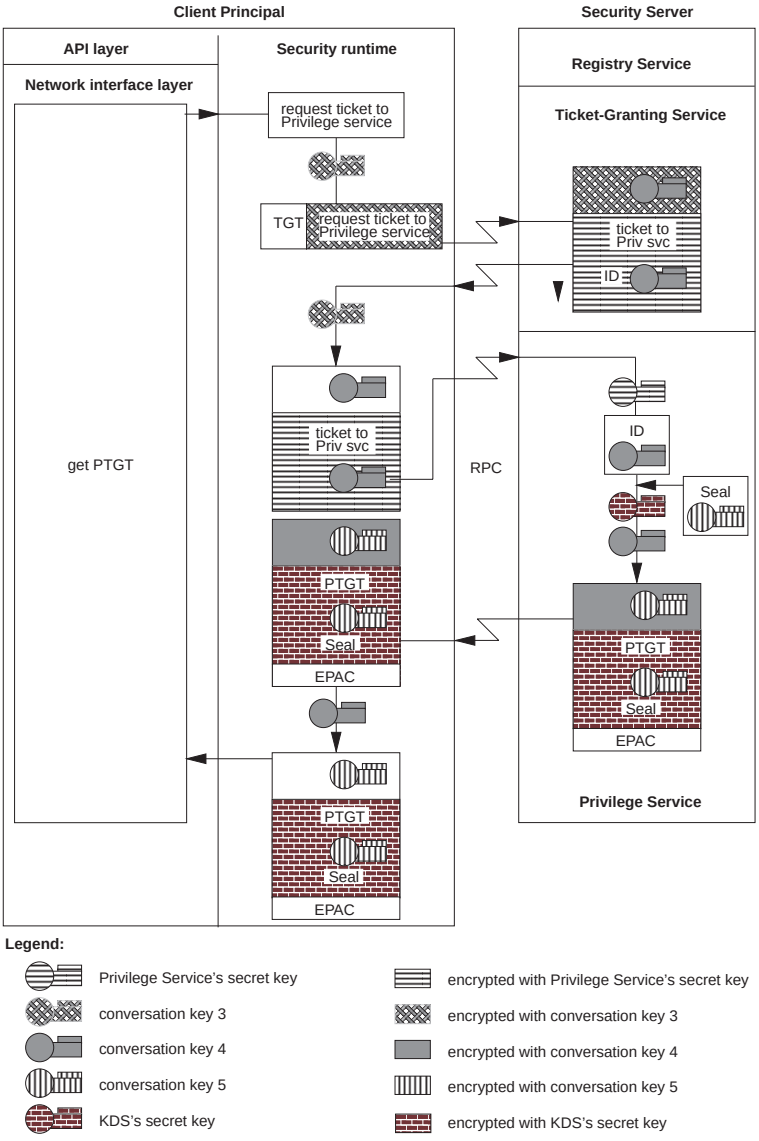
3. Upon receipt of this data, the client's security runtime uses conversation key 3 to decrypt conversation key 4. The client then formulates a request for a PTGT, encrypting it with conversation key 4, and sends this together with the service ticket it just received from the TGS, to the privilege service.

4. The privilege service decrypts the service ticket sent to it (using its secret key), thereby learning the identity of the client and the conversation key 4 it will use to decrypt the request and to encrypt its response. The privilege service is convinced of the authenticity of this request because the information was encrypted under its own secret key, and no principal other than the KDS (acting as the TGS) could have encrypted the information by using this secret key. Because the privilege service believes the authenticity of the client's identity, it prepares an extended privilege attribute certificate (EPAC) to issue to the client. (Actually, in the pure DCE Version 1.0 protocol this would be a PAC, not an EPAC, but since this is a high-level description intended for both releases we'll just talk about EPACs

without fear of confusion. So what we're really describing here is an extended privilege TGT, or EPTGT, though we'll continue to call it a PTGT.)

The EPAC describes the user's privilege attributes (identity information and group membership) and any extended attributes that are associated with the user—all represented as UUIDs (not strings). The EPAC (or EPAC chain, in case of a delegated operation) is sealed with an MD5 checksum. (Delegation is described in Chapter 27.) The privilege service constructs a PTGT, which is a ticket that contains the EPAC, the EPAC seal, another copy of the EPAC seal encrypted in the secret key of the privilege service (this is called a *delegation token*), and a conversation key 5 (which is actually generated by the KDS, though the illustration doesn't show this detail). All this information except for the EPAC itself is encrypted in the secret key of the KDS (thus, the delegation token is doubly encrypted). (The KDS and privilege service cooperate to prepare the PTGT, although the illustration only shows the privilege service preparing it.) The EPAC seal inside the PTGT binds the EPAC to the PTGT, guaranteeing its integrity even though it isn't encrypted. The conversation key 5 is encrypted in conversation key 4, and all this data is returned to the client.

Figure 24–6. Client Acquires PTGT



5. The client's security runtime uses conversation key 4 to decrypt conversation key 5. It cannot decrypt the PTGT itself, since the PTGT is encrypted under the secret key of the KDS.

24.2.1.4 The Login Context

At this point, the security service has authenticated the user's identity (that is, has verified that the user knows its password), and the user has acquired (trusted) information about its privilege attributes from the privilege service. The client now calls **sec_login_set_context()** to set the login context (a handle to this user's network identity and privilege attributes that have been established). Henceforth, processes invoked by this user inherit the user's login context, and among these processes is the client side of distributed applications — those are the subject of the rest of the walkthrough.

24.2.1.5 Identities in a Delegation Chain

When a user who has initiated delegation (with **sec_login_become_initiator()**) makes an authenticated RPC to the next member in a delegation chain (the first intermediary), the initiator passes its PTGT (including EPAC, seal and delegation token) to the TGS, and receives an extended privilege service ticket (again containing EPAC, seal and delegation token) to the intermediary. This is passed to the intermediary. The intermediary then invokes either routine **sec_login_become_delegate()** or **sec_login_become_impersonator()** , passing to the privilege service the authorization information it received from the initiator (EPAC and delegation token), together with the intermediary's own PTGT (including the intermediary's EPAC, seal and delegation token).

The privilege service uses the two delegation tokens, which are seals over the initiator's and intermediary's EPAC encrypted in the privilege service's own secret key, to verify the authenticity of the EPACs. If these are valid, the privilege service creates an EPAC chain, consisting of the initiator's and intermediary's EPACs, and generates a new seal and delegation token for this EPAC chain, and returns to the intermediary a new PTGT containing this information. Thus, the intermediary's authorization information now includes both EPACs in the delegation chain and a PTGT that contains the EPAC chain's seal and delegation token. The subsequent additions of identities to

the delegation chain are handled in the same manner, resulting in PTGTs with each intermediary's identity being added to the EPAC chain. Any such PTGT can be used to continue the delegation chain or to acquire a service ticket to the ultimate target server.

24.2.2 Authenticating an Application

Applications that are run between client and server must also be authenticated. For specific information about using the authenticated RPC routines see Chapters 13 and 14. For information about the GSSAPI, see Chapters 23 and 26.

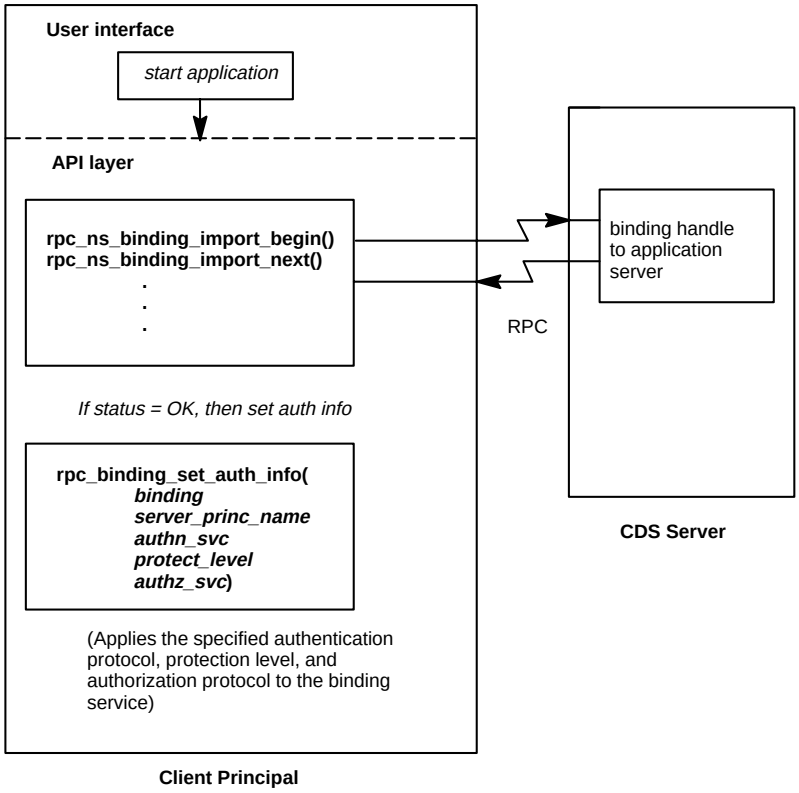
24.2.2.1 Authentication Using Authenticated RPC

This section explains how DCE security authenticates an application, to which the application developer has added authenticated RPCs.

Note: The authenticated RPC facility may also be referred to as the *protected* RPC facility, as it involves services beyond authentication. Authenticated RPC may also be referred to as the protected RPC facility,

Refer to Figure 24-7 as you read the following steps.

Figure 24–7. Client Sets Authentication and Authorization Information



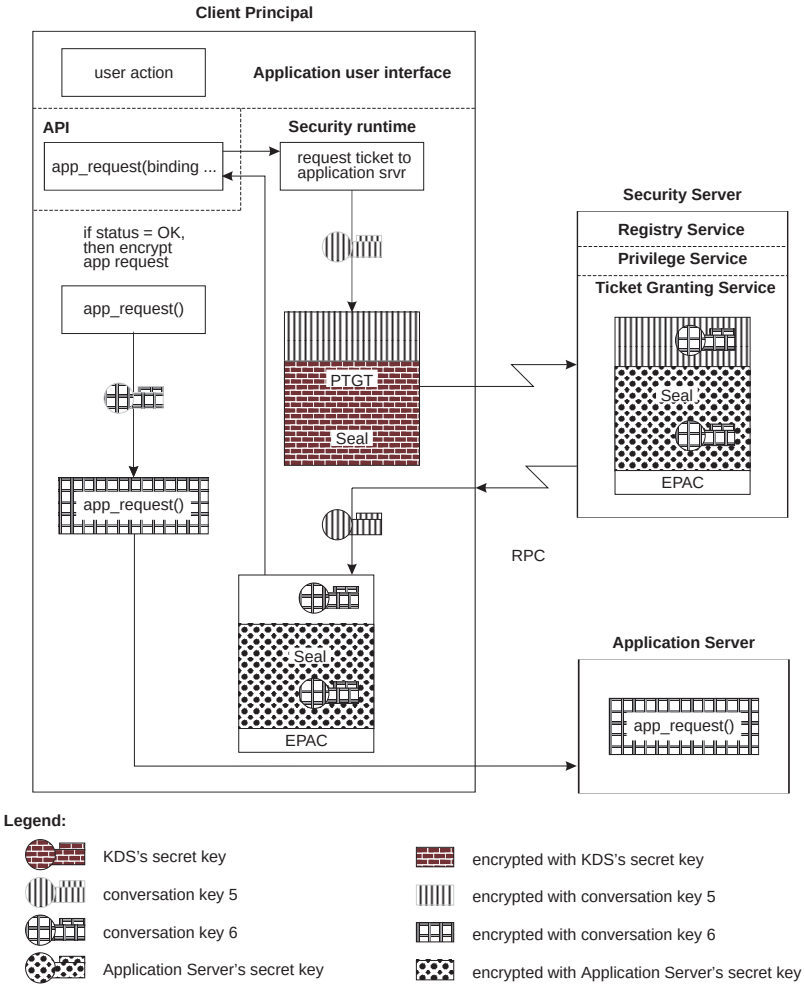
1. Having been authenticated and having acquired a PTGT, the user invokes an application. The client side of the application makes calls to routines **rpc_binding_import_begin()**, **rpc_binding_import_next()** , and the like. These calls specify the remote interfaces required by the client for the application.
2. The CDS returns the client binding handles to the specified interfaces. (For simplicity in this example, we consider the simple binding model in which the client consults the CDS for the server's RPC binding name.)
3. The client *annotates the binding handle*—that is, it sets security information for the binding handle by calling **rpc_binding_set_auth_info()**. Among other parameters, this routine sets the authentication protocol, the protection level, and authorization protocol for the binding handle corresponding to the remote interface. It also sets the server's principal name, which the client must know

securely (it may be the same or different than the server's RPC binding name). In this example, assume that the authentication protocol (*authn_svc* parameter) is DCE shared-secret authentication, the protection level (*protect_level*) is packet privacy (all RPC argument values are encrypted), and the authorization protocol (*authz_svc*) is DCE authorization. (DCE authorization means that an EPAC chain, containing UUIDs representing the client's or delegation chain's privilege attributes, will be sent to the server, which will compare this information with the ACLs protecting the objects of interest in order to determine whether the principal is to be granted or denied access.)

Refer to Figure 24-8 as you read the following steps.

4. The client requests some operation (using the annotated binding handle) to be performed by the server. The client RPC runtime requests from the TGS a service ticket to the server (identified by the server principal name with which the binding handle has been annotated). To acquire the ticket, the client security runtime formulates a request to the TGS. The request includes the server's principal name, which the client security runtime encrypts under conversation key 3. Also sent along with the request is the principal's PTGT, including EPAC and seal.

Figure 24–8. Client Principal Makes Application Request



5. The TGS decrypts the PTGT (which was encrypted in the KDS's secret key), thereby recovering conversation key 5, and uses conversation key 5 to decrypt the rest of the TGS request message. The TGS then constructs a service ticket, including the EPAC chain information and conversation key 6. By default, the key that is used to encrypt the service ticket is the application server's secret key.

For server principals that must use the user-to-user authentication protocol, the service ticket granted must be encrypted using the session key obtained from the server's current TGT, which the client must pass in with the ticket request. If the client had used the server-key-based request, and the server requires user-to-user protocol, the TGS will respond with an error instructing the client-side runtime to ask the server for its current TGT and to reissue the request with this TGT.

The service ticket is returned to the client, together with conversation key 6 encrypted under conversation key 5.

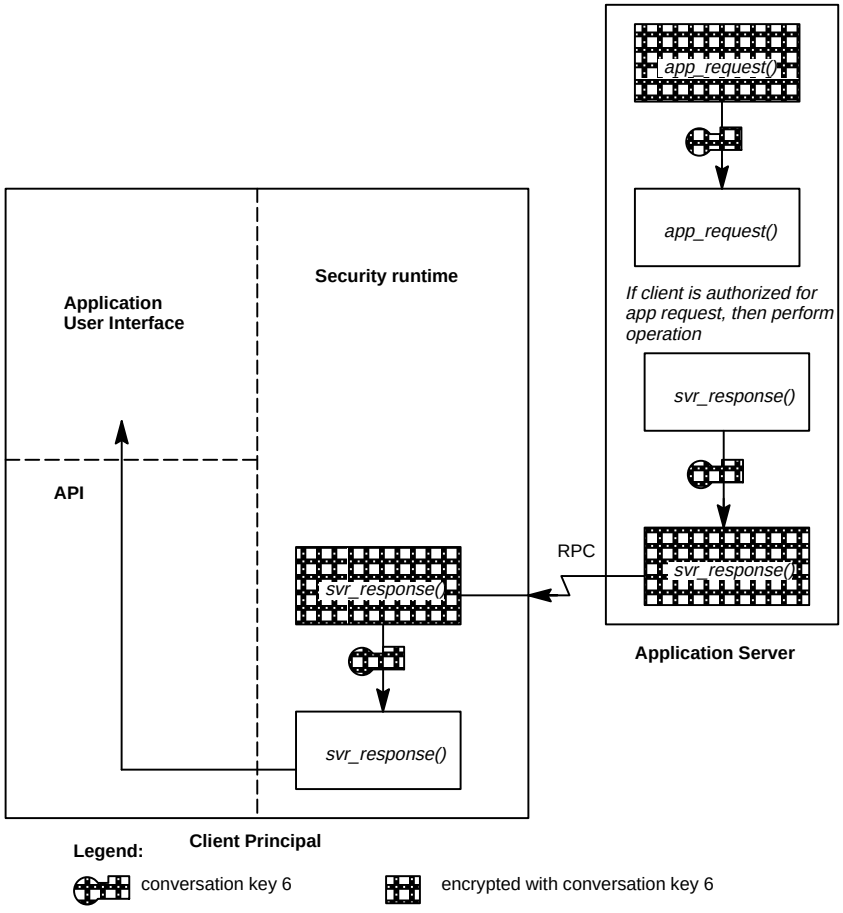
6. The client's security runtime uses conversation key 5 to decrypt conversation key 6, and then uses conversation key 6 to encrypt the application-level RPC request to the server. The client's RPC runtime sends the encrypted application request to the application server, together with the service ticket.

   Refer to Figure 24-9 as you read the following steps.

7. The application server's security runtime receives the client's request and decrypts the service ticket by using its secret key, or the TGT session key if user-to-user based authentication is used. In this way, the server's security runtime learns conversation key 6 and uses it to decrypt the RPC request. If the server determines from the client's authorization information (EPAC chain) that the request is granted, it performs the requested operation and prepares a response. The server's runtime encrypts the response by using conversation key 6 and sends it back to the client.

8. The client runtime receives and decrypts the response, and returns data to the application (by returning from the RPC).

Figure 24–9. Application Server Responds to Client's Request



The preceding walkthroughs have focused on the security aspect of authenticated RPC in DCE, not on its communications aspect. The technical details of integrating security with RPC lie beyond the scope of this chapter. However, the following remarks apply:

- In the CO (virtual circuit) RPC protocol, the client's security credentials (ticket with conversation key 6 and EPAC) are *pushed* from client to server at connection

establishment time, that is, at the time of the first remote procedure call from client to server (and the remote call is of course protected with conversation key 6). In the CL (datagram) RPC protocol, on the other hand, while the first remote call from client to server is protected as previously described (with conversation key 6), the credentials themselves are not sent with the remote call. Instead, the server itself *pulls* the credentials, by performing a *callback*, that is, a reverse (system-level) RPC back to the client, requesting the credentials. Once it receives these credentials, the server proceeds as if the credentials had been transferred with the original application-level RPC (from client to server) itself, as in the preceding walkthroughs.

- Once the application client and server have established conversation key 6, they cache it and continue to use it for subsequent RPCs, until it *expires*. All tickets and their conversation keys are accompanied by an expiration time, beyond which a new conversation key must be established (via a new service ticket, or perhaps even a new TGT if that expires, as described in the preceding walkthroughs). Thus, the security overhead of these subsequent RPCs is minimal, namely, it is reduced merely to the overhead of encryption/decryption processing itself, without the protocol message-passing.

24.2.2.2 Authentication Using GSSAPI

This section describes the process by which applications that perform their network communications via a mechanism other than DCE RPC can use GSSAPI and DCE security to authenticate and otherwise protect their communications. (These alternative communications mechanisms are called *peer-to-peer* , to distinguish them from RPC.)

In peer-to-peer communications, the application component that establishes the secure connection is called the *context initiator* or simply *initiator*. The context initiator is analogous to a DCE RPC client. The application component that accepts the secure connection is called the *context acceptor* or simply *acceptor*. The context acceptor is analogous to a DCE RPC server.

The peer application components establish a secure connection in the following way. (The reader will notice that the underlying security aspects are identical to those of the preceding RPC case, the only differences being in the explicit routine-invocation and communications aspects.)

1. The context initiator uses the **gss_init_sec_context()** routine to request from the DCE security server a service ticket (as previously described) that will allow the initiator to talk to the context acceptor.

   The initiator's security runtime creates an envelope that contains:

   - The initiator's PTGT

     Note: It is assumed that the initiator's security runtime already possesses a PTGT; that is, GSSAPI itself does not handle login.

   - The acceptor's principal name, protected under conversation key 5

   The initiator's security runtime sends the envelope to the TGS. (As in Section 24.2.2, step 4, this communication happens via RPC, but this use of RPC is hidden from the application because it's an implicit RPC being made by the security runtime, not an explicit RPC by the application initiator itself.) The TGS issues a service ticket to the initiator, encrypted in the acceptor's secret key, exactly as described in Section 24.2.2, step 5.

2. The initiator's security runtime recovers conversation key 6 as described in Section 24.2.2, step 6, and then hands to the GSSAPI the service ticket (including EPAC chain) and conversation key 4.

3. GSSAPI holds onto conversation key 6 and creates a GSSAPI *token* containing the service ticket.

   This GSSAPI token is then returned to the initiator, which forwards it to the acceptor (via the application's chosen communications mechanism). (Compare this with Section 24.2.2, step 6.)

4. The acceptor calls the **gss_accept_sec_context()** routine, which passes the token to the acceptor's security runtime.

5. The acceptor's security runtime processes the token, in particular recovering conversation key 6, exactly as described in Section 24.2.2, step 7.

6. The acceptor's GSSAPI holds onto conversation key 6 and the EPAC chain, and creates a GSSAPI token containing the success message. It passes the token to the acceptor. (Again, refer to Section 24.2.2, step 7.)

7. The acceptor forwards the GSSAPI token to the initiator.

8. The initiator passes the token to its GSSAPI, which sends it to the security runtime by calling the **gss_init_sec_context()** routine again.

9. The initiator's security runtime tries to decrypt the message. If this succeeds, it returns a success status to the GSSAPI that the acceptor's identity is authenticated. If not, it returns a failure status to the GSSAPI. (Compare this to Section 24.2.2, step 8.)

The context acceptor and context initiator can then use conversation key 6 in future communications by calling the **gss_sign()** and **gss_seal()** routines. (Compare this scenario with the RPC remarks following Section 24.2.2, step 8.) The context acceptor can get the initiator's EPAC chain in the form of an **rpc_authz_cred_handle_t** object so it can perform a DCE ACL check by calling the **gssdce_extract_creds_from_sec_context()** routine. If the context initiator wants to talk to a different context acceptor, it must acquire a ticket to that context acceptor.

24.3 Intercell Authentication

While the intercell authentication model is an extension of intracell authentication, certain concepts are particular to intercell authentication. The following subsections discuss those concepts.

24.3.1 KDS Surrogates

A principal trusts the DCE Security Service (registry service/KDS/privilege service) to authenticate other principals in its cell because it trusts the cryptographic algorithms and protocols, and the security of the code and data of the security service itself (which is trusted because it is part of the DCE network trusted computing base). The DCE Security Service can authenticate all principals in its cell because it shares a secret key with each of them. A client principal that wants to talk to a *foreign* server principal (that is, a principal in another cell) must acquire a ticket targeted to that server. As always, such a ticket must be encrypted in the secret key of the foreign server, else the server will not trust the ticket. The client cannot get such a ticket from its own local security service, because only the foreign security service, not local security service, knows the secret key of the foreign server. Therefore, some means must be devised by which the two instances of the security service can securely convey information about their respective principals to one another (without actually divulging secret keys of principals to foreign security services, which would be a security risk).

Besides the fact that it is trusted *a priori*, a cell's KDS is an exceptional principal in this other respect: other kinds of principals share their secret keys with the local security service, whereas the KDS's key is private to the KDS; that is, it is known to no other principal. Thus, one problem that intercell authentication must overcome is the means by which the KDS in one cell may trust that in another cell without either of them having to share their private keys (which would again introduce an unacceptable security risk).

Note: With respect to cryptographic keys, the term *secret* refers to keys that are (securely) shared between a bounded set of two (or more) principals, while *private* refers to keys that are known to only a single principal, and *public* to keys that are known to an unbounded set of principals (potentially to all principals). The cryptographic algorithms and protocols that are currently supported by DCE all depend on secret key technology (typified by DES), even though a small number of private keys (those of KDSs) are used.

The solution to this problem is a small extension of the shared-secret authentication model previously discussed in this chapter. Namely, a new principal is invented specifically for cross-cell authentication, and two entries for this principal are made, one each in the registry service databases of the two mutually authenticating cells. The two entries have the *same* secret key. These two special registry service database entries are known as *mutual authentication surrogates*, and the two cells that maintain mutual authentication surrogates are called *trust peers*. It is through their surrogates that the two instances of the KDS can convey information about their respective principals to one another (though the two KDSs never communicate directly with one another, nor do the surrogates), thus enabling a client principal from one cell to acquire a ticket to a server principal in another cell.

An authentication surrogate is a true principal in the sense that it is represented by an entry in a registry service database, but it is not an autonomous participant in authenticated communications in the same sense that, for example, a client or a server is. Rather, it is more like an *alias* that is assumed by a cell's KDS when it communicates with foreign clients. The establishment, via surrogates, of a trust peer relationship between two cells is an *explicit* expression of mutual trust in the two KDSs on the part of the cell administrators who establish the relationship. Administrators use the **rgy_edit** tool to create surrogates and establish the trust relationship. Administrators who do not trust one another's cells must not establish such a relationship.

24.3.2 Intercell Authentication by Trust Peers

This section explains how a client principal in one cell is authenticated by the KDS in a peer cell, so that the client principal may communicate with a server principal that is a member of the foreign cell. The style of description is the same as in the walkthroughs earlier in this chapter, though no figures are used here.

1. A client principal, having already been authenticated in the normal way by the KDS and privilege service in its home cell and acquired its PTGT, requests its local TGS for a service ticket targeted to a server in a foreign cell. The client specifies the server principal by its fully qualified principal name, which includes the name of the foreign cell.
2. The client's security runtime makes a request to the client's local TGS for a service ticket to the foreign server. The TGS recognizes by the server's principal name that it is foreign, so this TGS cannot directly issue the desired service ticket. Instead, it issues a so-called cross-cell TGT (XTGT), which is targeted to the *surrogate* shared between the two cells (that is, it is encrypted in the surrogate's secret key). The EPAC data in the client's PTGT is copied into the XTGT, and the local TGS returns the XTGT to the client. (For simplicity, we deal here only with simple case of EPAC data, not a delegation EPAC chain.)
3. The client receives the XTGT, recognizes that it is not targeted to the application server it had requested, and proceeds to send a request to the foreign TGS for a service ticket to the foreign privilege service, this time presenting the XTGT (instead of its original TGT) as proof of authentication. Upon receiving this request, the foreign TGS decrypts it by using the surrogate's secret key, and returns to the client a service ticket to the foreign privilege service. (Note how knowledge of the surrogate's shared key makes it possible for the two TGSs to cooperate in this way.)
4. The client's security runtime sends this service ticket to the foreign privilege service, to obtain a cross-cell privilege TGT (XPTGT). This XPTGT contains the client's original EPAC, and is encrypted with the secret key of the foreign privilege service.
5. After the client principal receives the XPTGT, it sends it to the foreign TGS, requesting a service ticket to the foreign server principal it was originally interested in. From this point on, the protocol goes exactly as it would in the case of a client principal in the server's cell requesting a service ticket to that server (as previously described). Similarly, the client principal may reuse the XPTGT to acquire service tickets to any other servers in the foreign cell.

Chapter 25

Authorization

This chapter explains concepts related to authorization. The authenticated RPC facility enables you to select the authorization protocol that your application uses. Among the authorization protocols supported by the DCE Security Service for use by authenticated RPC is DCE authorization (the default), and name-based authorization.

This chapter first discusses DCE authorization, and more particularly, DCE access control lists (ACLs). At the end of this chapter, we also briefly discuss the name-based authorization protocol.

25.1 DCE Authorization

The DCE authorization protocol is based in part on the UNIX file-protection model, but is extended with ACLs. An ACL is a list of access control entries that protects an object. Each entry in the ACL specifies a set of permissions. Usually, most of the entries in the ACL specify a privilege attribute (such as membership in a group) and the set of permissions that may be granted to the principal(s) that possesses that

privilege attribute. Some other entries specify a set of permissions that may mask the permission set in a privilege attribute entry.

Every ACL is managed by an ACL manager type. An ACL manager type determines a principal's authorization to perform an operation on an object by reading the object's ACL to find the appropriate entry (or entries) that matches some privilege attribute possessed by the principal. If the type of access requested by the principal is one of the permissions listed in the matching entry, and assuming no applicable mask entry denies that permission, then the ACL manager type allows the principal to perform the requested operation. If the requested permission is not listed in the matching ACL entry, or is denied by a mask, permission to perform the operation is denied. Permission to perform the operation is also denied if the ACL contains no matching privilege attribute entry.

Unlike UNIX file permissions, DCE ACLs are not limited to the protection of file system objects such as is, files, directories, and devices. ACLs may also control access to nonfile-system objects, such as the individual entries in a database.

Note: The implementation of DCE ACLs is aligned with POSIX P1003.6 Draft 12.

In the discussions in this chapter, we use the general term *name* to refer to a principal, group, or cell identifier; but readers should always bear in mind that these names have two representations: as UUIDs in ACL program interfaces and as print strings in user interfaces.

25.1.1 Object Types and ACL Types

The ACL facility distinguishes between two types of objects: container objects and simple objects. Container objects contain other objects, which may be simple and/or other container objects. Simple objects do not contain other objects. Examples of container objects include file-system directories and databases; examples of simple objects include files and database entries.

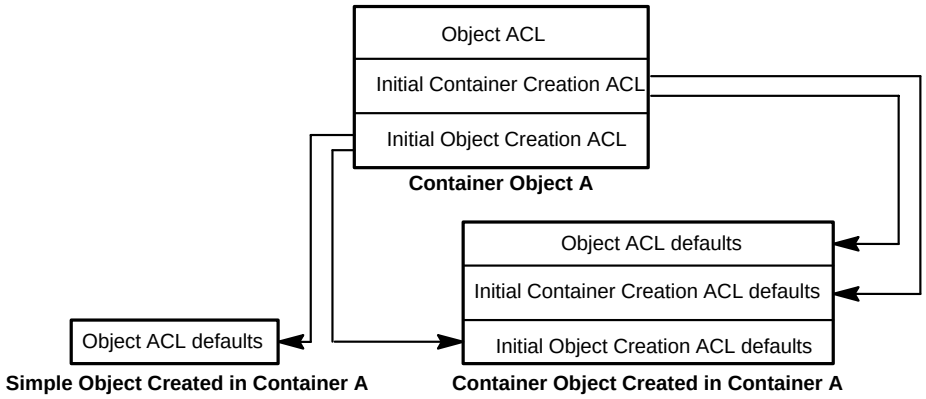
To protect both object types, and to enable newly created objects to inherit default ACLs from their parent container objects, the ACL facility supports two basic kinds of ACLs:

- An Object ACL is associated with either a container or a simple object, and controls access to it.

- A Creation ACL is associated with a container object only. Its function is not to control access to the container but to supply default values for the ACLs of objects created in the container. There are two types of Creation ACLs:
  - — An Initial Object Creation ACL supplies default values for a simple object's Object ACL and for a container object's Initial Object Creation ACL.
  - — An Initial Container Creation ACL supplies default values for both a container object's Object ACL and its Initial Container Creation ACL.

Figure 25-1 illustrates how ACL defaults are derived from Creation ACLs.

Figure 25–1. Derivation of ACL Defaults



Aside from the distinctions previously described, there are no differences between Object ACLs and Creation ACLs; therefore, the information about ACLs in the rest of this chapter does not differentiate between them.

25.1.2 ACL Manager Types

A separate ACL manager type manages the ACLs for each class of objects for which permissions are uniquely defined. The manager type defines the permissions for those objects whose ACLs it manages, which are the number of permissions, the meanings of the permissions, and the tokens that represent the permissions in user interfaces to ACL manipulation tools.

For example, for the purpose of access control, five classes of objects are defined in the registry database, and five ACL manager types manage the ACLs for the registry database objects (the five registry manager types run in a single security server process). Other DCE components implement their own manager types, and applications implement manager types for the objects that the applications protect.

Refer to the *DCE 1.2.2 Administration Guide* and the *DCE 1.2.2 Command Reference* for information about standard DCE ACL manager types and the permissions they implement. Refer to Part 1 and Chapter 32 of this guide for information about implementing ACL manager types for distributed applications.

25.1.3 Access Control Lists

An ACL consists of the following:

- An ACL manager type identifier, which identifies the manager type of the ACL.
- A default cell identifier, which specifies the cell of which a principal or group identified as local is assumed to be a member. A DCE global pathname is necessary to specify a principal or a group from a nondefault cell; this consists of a pair of UUIDs representing the principal or group, and the cell of which it is a member. It is necessary to use the ID Map API to convert the global print string names of foreign principals and groups to the UUID representations that DCE ACL managers use. (Refer to Chapter 33 for more information on this subject.)
- At least one ACL entry.

The rest of this chapter discusses ACLs primarily from a user-interface point of view, since this perspective provides an orientation to the discussion of the ACL API in this part.

25.1.4 ACL Entries

DCE authorization defines two basic kinds of ACL entries:

- Those that associate a specified privilege attribute with a permission set; these are privilege attribute entries.

- Those that specify a permission set that masks a permission set specified in a privilege attribute entry; these are mask entries.

The following subsections describe the two kinds of ACL entries in detail.

25.1.4.1 Privilege Attribute Entry Types

The privilege attributes of a principal are based on identity and include the principal's name, its group membership(s), and native cell. Note that not all ACL manager types implement all privilege attribute entry types. For example, the ACL manager type of a database object probably would not support the **user_obj** and **group_obj** entry types.

Note: The term *local cell* means the cell specified in the ACL header; this is not necessarily the cell in which the protected object resides.

The descriptions of the ACL entry types that specify privilege attributes are as follows:

- **user_obj**

  The **user_obj** entry establishes the permissions for the object's ''user'' (in the established UNIX sense). An ACL may contain only one entry of this type. The identity of the principal to which this ACL entry refers is assumed to be local and is specified somewhere other than in this entry. In the case of a file, for example, the identity is attached to the file's inode.

- **user**

  The **user** entry establishes the permissions for the local principal named in this entry. An ACL may contain a number of entries of this type, but each entry must be unique with respect to the principal it specifies.

- **foreign_user**

  The **foreign_user** entry establishes the permissions for the foreign principal named in this entry. An ACL may contain a number of entries of this type, but each entry must be unique with respect to the foreign principal it specifies. This entry type is exactly like the **user** entry type, except that this entry explicitly names a cell. (For the entry type**user**, the principal inherits the cell specified by the default cell identifier in the ACL header.)

- **group_obj**

The **group_obj** entry establishes the permissions for the object's ''group'' (in the established UNIX sense). An ACL may contain only one entry of this type. As is the case with the **user_obj** entry, the identity of the group is assumed to be local and is specified elsewhere than in the **group_obj** entry itself.

- **group**

  The **group** entry establishes the permissions for the local group named in this entry. An ACL may contain a number of entries of this type, but each entry must be unique with respect to the group it specifies.

- **foreign_group**

  The **foreign_group** entry establishes the permissions for the foreign group named in this entry. An ACL may contain a number of entries of this type, but each entry must be unique with respect to the foreign group it specifies. This entry type is exactly like the **group** entry type, except that this entry explicitly names a cell (for the entry type**group**, the principals inherit the default cell identifier).

- **other_obj**

  The **other_obj** entry establishes the permissions for local principals whose identities do not correspond to any entry type that explicitly names a principal or group; an ACL may contain only one entry of this type.

- **foreign_other**

  The **foreign_other** entry establishes the permissions for all principals that are members of a specified foreign cell and whose identities do not correspond to any **foreign_user** or **foreign_group** entry. An ACL may contain a number of entries of this type, but each entry must specify a different foreign cell.

- **any_other**

  The **any_other** entry establishes the permissions for principals whose privilege attributes do not match those specified in any other entry type. An ACL may contain only one entry of this type.

The following additional ACL entry types are supplied for delegated identities:

- **user_obj_delegate**
- **user_delegate**
- **foreign_user_delegate**
- **group_obj_delegate**

- **group_delegate**
- **foreign_group_delegate**
- **foreign_other_delegate**
- **other_obj_delegate**
- **foreign_other_delegate**
- **any_other_delegate**

These ACL entry types are described in detail in Chapter 27, along with the extensions to the ACL checking algorithm for delegation.

ACL entries for privilege attributes consist of three fields in the following form:

*entry_type***[:***key***]:***permissions*

Following are descriptions of the fields:

- The ACL *entry_type* specifies an ACL entry type as described in the previous list.
- The *key* field specifies the privilege attribute to which the permissions listed in the entry apply. The key field for the ACL entry types **user**, **group**, **foreign_user**, **foreign_group**, and **foreign_other** explicitly names a principal, group, or cell. For the entry types **foreign_user**, **foreign_group**, and **foreign_other**, the key field must contain a global DCE pathname of the forms **/.../***cellname/principalname*, **/.../***cellname/groupname*, or **/.../***cellname*, respectively. The entry types **user_obj**, **group_obj**, **other_obj**, and **any_other** do not use the key field.
- The *permissions* field lists the permissions that may be granted to the principal possessing the privilege attribute specified in the entry, unless a mask (or masks) further restricts the permissions that may be granted to the principal. As noted previously, the number and meaning of the permissions that may protect an object are defined by the object's ACL manager type. Therefore, the permissions that an ACL entry may specify must be the set, or a subset, of the permissions implemented by the manager type of the ACL in which the entry appears.

  A principal is denied access when a **user** or **foreign_user** entry that names the principal contains an empty permission set.

25.1.4.2 Mask Entry Types

Following are descriptions of the ACL entry types that specify masks:

- **mask_obj**

  The **mask_obj** entry establishes the permission set that masks all privilege attribute entry types except the **user_obj** and **other_obj** types.

- **unauthenticated**

  The **unauthenticated** entry establishes the permission set that masks the permission set in a privilege attribute entry that corresponds to a principal whose privilege attributes have not been certified by an authority such as the privilege service.

The two masks are similar in that the permission set specified in the mask entry is intersected (logically ANDed) with the permission set in a privilege attribute entry. This masking operation yields the effective permission set (the permissions that may be granted to the principal) for the principal possessing the privilege attribute. For example, if a privilege attribute entry specifies the permissions **ab**, and a mask entry that specifies the permissions **bc** masks that privilege attribute entry, the effective permission set is **b**. Similarly, a mask entry that specifies the empty permission set means that none of the permissions in any privilege attribute entry that mask entry masks is granted to the principal possessing the privilege attribute.

The two masks are dissimilar in one notable respect. Adding an **unauthenticated** mask entry with an empty permission set to an ACL is equivalent to omitting the **unauthenticated** mask entry from the ACL; in both cases, the set of effective permissions for principals possessing unauthenticated privilege attributes is empty. However, adding a **mask_obj** entry with an empty permission set to an ACL is different from having no **mask_obj** entry in the ACL. In the first case, the effective permission set is empty; in the second case, the effective permission set is identical to the permission set in the privilege attribute entry.

ACL entries for masks consist of two fields in the following form:

entry_type:permissions

Following are descriptions of the fields:

- The *entry_type* field specifies one of the two masks entry types: **mask_obj** or **unauthenticated.**
- The *permissions* field specifies the permission set that masks the permission set in any privilege attribute entry masked by the mask entry.

25.1.4.3 The Extended ACL Entry Type

The ACL entry type **extended** is a special entry type for ensuring the compatibility of ACL data created by different software revisions. It enables old application clients to copy ACLs from one newer revision object store to another without losing data. It also enables obsolete clients to manipulate ACL data that they understand without corrupting the extended entries that they do not understand.

25.1.5 Access Checking

Standard DCE ACL manager types use a common access-check algorithm to determine the permissions they grant to a principal. Access checking is executed in up to six stages, in the following order:

1. The **user_obj** entry check
2. The check for a matching **user** or **foreign_user** entry
3. The **group_obj** entry check and the check for matching **group** or **foreign_group** entries
4. The **other_obj** entry check
5. The check for a matching **foreign_other** entry
6. The **any_other** check

If during any stage of access checking an ACL manager type finds a privilege attribute entry that matches a privilege attribute possessed by a principal, then the manager type does not execute any subsequent stages, even though the principal may possess other privilege attributes for which there are other matching entries. See the *Security Volume* of the *Application Environment Specification/Distributed Computing* for descriptions of the algorithms used at each stage of access checking.

25.1.6 Examples of ACL Checking

The following subsections provide some examples that illustrate ACLs and the access-check algorithms. The examples use the arbitrary convention of ordering entries as follows: masks, principals, groups, and ‘‘other’’ entries. However, the access check algorithm disregards the order in which entries appear in an ACL. Also note that the permissions in these examples do not refer to any particular permissions implemented by any ACL manager type.

25.1.6.1 Example 1

Following is an ACL that protects an object to which three principals, **janea**, **/.../cella/fritzb**, and **mariac**, seek access:

```
mask_obj:ab
user_obj:abc
user:janea:abdef
foreign_user:/.../cella/fritzb:abc
group:projectx:abcf
group:projecty:bcg
```

Note: The numbered lists in the discussions that follow correspond to stages 1, 2, 3, 4, 5 and 6 of the access-check algorithm referred to in Section 25.1.5.

The principal **janea** requests permission **c** to the object protected by the ACL. Assume that the principal **janea** has the privilege attributes of being a member of the groups**projectx** and **projecty** (as well as having a **user** entry that names her) and that **janea** is the principal to which the **user_obj** entry refers. Assume also that this principal’s privilege attributes are certified:

1. The **user_obj** check yields the permissions **abc**.

The result of this check is that the effective permission set for **janea** is **abc**. Because a matching entry is found during the first stage of access checking, none of the remaining stages of access checking is executed, even though there are three other matching entries. The**mask_obj** entry does not mask the **user_obj** entry, so **janea**’s

effective permissions are the permissions in the **user_obj** entry. Since **janea** requested a permission that is a member of the effective permission set, her request is granted.

The second principal seeking access to the protected object is **/.../cella/fritzb**. This principal requests permission **b**. Assume that **user_obj** resolves to some identity other than **/.../cella/fritzb,** and that this principal's privilege attributes are uncertified:

1. The **user_obj** check yields no permissions because **/.../cella/fritzb**'s identity does not match that of the **user_obj** (no foreign principal can ever match this entry).
2. The **foreign_user** entry for **/.../cella/fritzb** specifies the permissions **abc**. The application of the **mask_obj**, which specifies the permissions **ab** to this permission set, yields the permissions **ab**. Since the **unauthenticated** mask entry is missing from the ACL, all permissions for unauthenticated identities are masked, yielding an empty effective permission set.

The result of these checks is that **/.../cella/fritzb**'s request is denied (and would be denied, regardless of the permission requested). In this case, only the first two stages of access checking are executed.

The third principal seeking access is **mariac**, who requests permission **a**. Assume that the privilege attributes of **mariac** are certified, that **mariac** is not the principal that corresponds to the **user_obj** entry, and that **mariac** is a member of the groups **projectx** and **projecty**:

1. The **user_obj** check yields no permissions.
2. There is no matching user entry.
3. The group check finds two matching entries. The permissions associated with **projectx** (**abcf**) when masked by the **mask_obj** entry (**ab**) yield the permissions **ab**. The permissions associated with **projecty** (**bcg**) when masked by the **mask_obj** entry yield the permission **b**. The union of the permission sets **ab** and **b** is the set **ab**.

The effective permission set for **mariac** is **ab** and since the requested permission (**a**) is a member of that set, **mariac**'s request is granted. The remaining stages of access checking are not executed.

25.1.6.2 Example 2

Following is the ACL for an object to which two principals, **ugob** and **/.../cellb/lolad**, seek access:

```
mask_obj:bcde
unauthenticated:ab
user_obj:abcdef
user:ugob:abcdefg
group:projectz:abh
foreign_other:/.../cellb/:abc
```

Note: The numbered lists in the discussions that follow correspond to stages 1, 2, 3, 4, 5 and 6 of the access check algorithm referred to in Section 25.1.5.

The principal **ugob** requests permission **b.** Assume that **ugob** is not the principal to which the **user_obj** entry refers. Assume also that the privilege attributes of **ugob** include membership in the group **projectz**, in addition to the **user** entry that names him. In this case, the principal has failed to acquire certified privilege attributes:

1. The **user_obj** check yields no permissions.
2. The matching entry among the user entries specifies the permissions **abcdefg.** Applying **mask_obj** (**bcde**) yields the permission set **bcde**. Applying the **unauthenticated** mask (**ab**) to the permission set **bcde** yields the effective permission set **b**.

Since the principal **ugob** requests a permission (**b**) that is a member of the effective permissions set, this principal's request is granted.

A case that illustrates how access is determined for otherwise undifferentiated members of a specified foreign cell is that of the principal **/.../cellb/lolad**, who requests permission **e**. Assume that the privilege attributes of this principal are certified:

1. The principal is foreign, so the **user_obj** check cannot be a match.
2. There are no **foreign_user** entries.
3. There are no **foreign_group** entries.

4. The principal **lolad** is a member of **cellb**, meaning that the privilege attributes match those in the **foreign_other** entry for **cellb.** The permissions specified by the **foreign_other** entry for **cellb** (**abc**) as masked by **mask_obj** (**bcde**) yields the effective permission set **bc**.

The permission requested (**e**) is not a member of the effective permission set (**bc**), so the request is denied.

25.1.6.3 Example 3

Following is the ACL for an object to which one principal, **silviob** seeks access.

```
unauthenticated:a
user:jeand:abcde
user:denisf:-
group:projectx:abcd
foreign_other:/.../cella:-
foreign_other:/.../cellc:abc
any_other:ab
```

Note: The **user** entry for **denisf** and the **foreign_other** entry for **cella** both specify an empty permission set with the notation **-** (dash), meaning that identities corresponding to these entries are explicitly denied all permissions. Also, the numbered lists in the discussions that follow correspond to stages 1, 2, 3, 4, 5 and 6 of the access-check algorithm referred to in Section 25.1.5.

The principal **silviob** requests permission **a**. Assume that this principal's privileges include membership in the group **projecty** and that they are not certified:

1. There is no **user_obj** entry, so this check can yield no permissions.
2. There is no **user** entry for this principal, so this check yields no permissions.
3. There is no entry for the group **projecty**, so this check yields no permissions.
4. There is no **other_obj** entry, so this check can yield no permissions.
5. The principal is local, so no **foreign_other** entry can be a match; this check yields no permissions.

6. Having failed to match any entry examined in the preceding checks, the principal matches the **any_other** entry, which yields the permission set **ab**. There is no **mask_obj** entry, but there is the **unauthenticated** mask entry, which specifies the permission set **a**. Applying the **unauthenticated** mask to this privilege attribute entry yields the effective permission **a**.

The permission requested (**a**) is a member of the effective permission set (**a**), so this principal's request is granted.

25.2 Name-Based Authorization

The Kerberos authentication service, upon which the DCE shared-secret authentication protocol is based, authenticates the string name representation of a principal. The DCE Security Service converts these string representations to UUIDs, and it is these UUIDs that an ACL manager uses to make authorization decisions. However, since some existing (non-DCE) applications implement Kerberos authentication, DCE security supports an authorization protocol based on principal string names: name-based authorization.

It is assumed that applications that use name-based authorization have a means to associate string names with permissions, since the DCE Security Service offers no such facility. Because in name-based authorization there is no UUID representation of privilege attribute data, and because DCE ACL managers recognize only UUIDs, if an application uses name-based authorization, then a principal's privilege attributes are represented as an anonymous PAC. Such PAC data can only match the ACL entry types **other_obj**, **foreign_other**, or **any_other**, and are masked by the **unauthenticated** mask.

Also note that there is essentially no intercell security for an application that uses the name-based authorization protocol because such applications never communicate with the privilege service, which evaluates intercell trust.

Chapter 26

GSSAPI Credentials

A GSSAPI *credential* is a data structure that provides proof of an application's claim to a principal name. An application uses a credential to establish its global identity. The global identity can be, but is not necessarily, related to the local user name under which the application (either the initiator or the acceptor) is running.

A credential can consist of either of the following:

- DCE login context
- Principal name

There are three types of credentials, as shown in Table 26-1.

Table 26–1. Credential Types

Credential	Content
INITIATE	A login context only. This credential identifies applications that only initiate security contexts.
ACCEPT	Principal name and an associated entry key table. This credential identifies applications that only accept security contexts.
BOTH	A login context and principal name with a key table entry. This credential identifies applications that can either initiate or accept security contexts.

Credentials are maintained internally to GSSAPI. When they establish a security context, applications use credential handles to point to the credentials they need.

When an application initiates or accepts a security context, it can use GSSAPI routines with either a default credential or a specific credential handle. This chapter discusses how applications do the following:

- Use default credentials
- Create credential handles to refer to specific credentials
- Delegate credentials

For detailed information on the GSSAPI routines referred to in this chapter, see the *DCE 1.2.2 Application Development Reference*.

26.1 Using Default Credentials

A *default credential* is a credential that is

- Generated by either of the following routines:
  - **gss_init_sec_context()**
  - **gss_accept_sec_context()**
- Based on the following information:

— The DCE default login context for the application (for **INITIATE** type credentials)

— The registered principal name in the token (for **ACCEPT** or **BOTH** type credentials).

When an application calls the GSSAPI routine to either initiate (**gss_init_sec_context()**) or accept (**gss_accept_sec_context()**) a security context, it can specify the use of its default credential.

Use default credentials to help ensure the portability of your applications.

26.1.1 Initiating a Security Context

To use a default credential when initiating a security context, an application calls the **gss_init_sec_context()** routine and specifies **GSS_C_NO_CREDENTIAL** as the input claimant credential handle to the routine. The routine uses the initiator's DCE default login context to generate the default credential. The credential is an **INITIATE** type credential.

You can change the default login context by calling the DCE **sec_login_ *()** routines. For information on these routines, see see the appropriate **sec_login_ *(3sec)** reference page.

26.1.2 Accepting a Security Context

To use a default credential when accepting a security context, an application calls the **gss_accept_security_context()** routine and specifies **GSS_C_NO_CREDENTIAL** as the verifier credential handle to the routine. The GSSAPI uses a principal name registered for the context acceptor to generate the default credential handle. The credential is an **ACCEPT** credential type.

26.2 Creating New Credential Handles

An application can create a new credential handle to pass to the **gss_init_sec_context()** routine or the **gss_accept_sec_context()** routine. An application might create a credential handle rather than use the default credential for the following reasons:

- Limit the identities the application can use
- Provide an additional identity for the application

26.2.1 Initiating a Security Context with New Credential Handles

To create a credential handle for an **INITIATE** credential type, the application calls the **gssdce_login_context_to_cred()** routine and specifies its login context as input to the routine. The routine creates a credential handle that points to the credential consisting of that login context.

An application can also use a **BOTH** type credential to initiate a security context. Use the **gss_acquire_cred()** routine to create a **BOTH** type credential, as explained in the next section.

When the application uses a **BOTH** credential, the **gss_acquire_cred()** routine creates a login context from the key table information. Then, it uses the login context to create the credential. For more details, see the **gss_acquire_cred(3sec)** reference page.

26.2.2 Accepting a Security Context Using New Credential Handles

To create new credential handle for an **ACCEPT** or **BOTH** type credential, an application calls the **gss_acquire_cred()** routine.

The **gss_acquire_cred()** routine uses a principal name and its entry in the key table to generate the credential handle. If the principal name has not yet been registered (using **gssdce_register_acceptor_identity()** or the **rpc_server_register_auth_info()** routines), the **gss_acquire_cred()** routine automatically registers it.

26.3 Delegating Credentials

In delegation, an initiator forwards its identity to an acceptor so that the acceptor can use the identity to act as an agent for the initiator. There are two forms of delegation:

- Impersonation delegation
- Traced delegation

26.3.1 Initiating a Security Context to Delegate Credentials

An application indicates that it wants to delegate credentials when it calls the **gss_init_sec_context()** routine and sets the **GSS_C_DELEG_FLAG** flag to TRUE. Notes added to the initiator's login context can indicate the type of delegation used and any restrictions in effect (for traced delegation only). If no delegation notes are included with the login context and the **GSS_C_DELEG_FLAG** flag is set, impersonation delegation is used.

26.3.2 Accepting a Security Context with Delegated Credentials

If the **GSS_C_DELEG_FLAG** flag has been set when the security context was intiated, the **gss_accept_sec_context()** routine will pass a credential to the acceptor. The routine does the following:

1. Uses information from the input token to create the appropriate delegated credential
2. Creates an impersonation or traced delegation credential with an **INITIATE** credential type
3. Passes the delegated **INITIATE** credential to the acceptor

The principal named in the delegated **INITIATE** credential is the name of the initiator (for impersonation delegation) or the acceptor acting *for* the initiator (for traced delegation). The acceptor uses the credential to act for the initiator, initiating security contexts as appropriate.

Chapter 27

The Extended Privilege Attribute API

This chapter describes the extended privilege attribute (EPA) API. The EPA facility addresses the requirements of complex distributed systems by allowing clients and servers to invoke secure operations via one or more intermediate servers.

In a simple client/server distributed environment, most operations involve two principals: the initiator of the operation and the target of the operation. The target of the operation makes authorization decisions based on the identity of the initiator. However, in distributed object-oriented environments, there is frequently a need for server principals to perform operations on behalf of a client principal. In these cases, it may not be enough for authorization decisions to be based simply on the identity of the initiator since the initiator of the operation may not be the principal that requests the operation.

To handle these cases, the EPA API provides routines that allow principals to operate on objects on behalf of (as delegates of) an initiating principal. The collection of the delegation initiator and the intermediaries is referred to as a *delegation chain*.

Using the EPA API and related **sec_login_ *()** calls, an application may be written that allows client Principal A to invoke an operation on server Principal C via server

Principal B. The DCE Security Service will know the true initiator of the operation (Principal A) and can distinguish the delegated operation from the same operation invoked directly by Principal A.

The EPA interface consists of the security credential calls (**sec_cred_ *()**) that extract privilege attributes and authorization data from an opaque binding handle to authenticated credentials. In addition, the following **sec_login_ *()** calls of the login API are used to establish delegation chains and to perform other delegation related functions.

- **sec_login_become_initiator()**
- **sec_login_become_delegate()**
- **sec_login_become_impersonator()**
- **sec_login_cred_get_delegate()**
- **sec_login_cred_get_initiator()**
- **sec_login_cred_initialize_cursor()**
- **sec_login_disable_delegation()**
- **sec_login_set_extended_attrs()**

27.1 Identities of Principals in Delegation

The identity of principals in a delegation chain is maintained in extended privilege attribute certificates (EPACs), as are the identities for all DCE principals. Each EPAC contains the name and group memberships of a principal in the delegation chain and any extended attributes that apply to the principal. The delegation chain includes an EPAC for each member of the delegation chain.

When delegation is in use, the target server receives the delegation chain, and thus knows the privilege attributes of the delegation chain initiator and each intermediary (delegate) in the chain. Authorization decisions can then be made based on the identities of all principals involved in the operation.

27.1.1 ACL Entry Types for Delegation

When a server's ACL manager is presented with credentials to use as a base of an authorization decision, the manager evaluates the privilege attributes of each principal involved in the delegation chain. The ACL manager grants access for the requested operation only if all principals in the delegation chain have the necessary permissions on the object that is the eventual target of the operation.

For the initiator of the delegation chain, permission on the target object must be granted directly using any of the following standard ACL entry types:

- **user_obj**
- **user**
- **foreign_user**
- **group_obj**
- **group**
- **foreign_group**
- **foreign_other**
- **other_obj**
- **foreign_other**
- **any_other**
- **extended**

For intermediaries in a delegation chain, permissions to a target object can be granted directly to the intermediary with the standard ACL entry type previously described, or permissions can be granted by delegate ACL entries. Delegate ACL entries grant permissions to principals only if they are acting as delegates. The following delegate ACL entry types are available:

- **user_obj_delegate**
- **user_delegate**
- **foreign_user_delegate**
- **group_obj_delegate**
- **group_delegate**

- **foreign_group_delegate**
- **foreign_other_delegate**
- **other_obj_delegate**
- **foreign_other_delegate**
- **any_other_delegate**

Note that, to perform an operation, all delegates in the chain must have the appropriate permissions. For example, assume a delegation chain consists of Principal A (the initiator) and Principal's B and C (the intermediaries). To perform the operation, the delegation chain requires **Mrw** permissions on Server X. One way of granting these permission is to grant them directly to each member of the delegation chain, as shown in the following:

```
user:Principal A:Mrw
user:Principal B:Mrw
user:Principal C:Mrw
```

Providing access directly also allows each intermediary in the chain to perform the operation of their own initiative, a consequence that may or may not be desired. To specify that Principals B and C may only be intermediaries operating on behalf of an authorized initiating principal without granting them the ability to perform the operation on their own, use delegation entries. In this case, the Server X's ACL would contain the following entries:

```
user:Principal A:Mrw
user_delegate:Principal B:Mrw
user_delegate:Principal C:Mrw
```

27.1.2 ACL Checking for Delegation

To determine permissions, the ACL manager first uses the standard access-check algorithm (described in Chapter 25) to determine the permissions to grant to the delegation initiator. If the requested permission is not granted, access is denied.

If the requested permission is granted, the ACL manager then checks the permissions granted to the delegates in the chain. This checking is similar to the standard access-check algorithm, but it takes into account any additional delegate permissions granted to the delegates. If the requested permission is not granted to all delegates, access is denied. If the requested permission is granted to all delegates, access is granted.

27.2 Calls to Establish Delegation Chains

The following **sec_login_* ()** API calls set up a delegation chain:

- **sec_login_become_initiator()**

  Enables delegation for a client. The principal that executes this call is known as the *delegation initiator*.

- **sec_login_become_delegate()**, **sec_login_become_impersonator()**

  Cause an intermediate server to become a delegate in a delegation chain. The principals that execute these calls are known as *intermediaries* in the delegation chain.

The **sec_login_become_delegate()** call should be used if the traced delegation has been enabled. The **sec_login_become_impersonator()** call should be used if simple delegation has been enabled. See Section 27.2.1 for more information about delegation types.

The following subsections describe the information supplied to the calls that establish delegation chains.

27.2.1 Types of Delegation

When a client application calls **sec_login_become_initiator()** to enable delegation, that application specifies the type of delegation that should be enabled. The delegation type can be any of the following:

- Traced Delegation

Includes the identities of all members of the delegation chain in the credentials used for authorization. To become an intermediary in a traced delegation chain, server principals use the **sec_login_become_delegate()** call.

Note that ACLs on objects that are targets of traced delegation must grant the requested permission (or delegate permission) to each member of the delegation chain.

- Impersonation

  Includes only the identity of the initiator of the delegation chain used for authorization. All intermediaries ‘‘impersonate’’ the delegation initiator. To become an impersonator, principals use the **sec_login_become_impersonator()** call.

  Note that ACLs on objects that are targets of impersonation need list only the delegation initiator, not each delegate in the chain.

Generally, traced delegation is the preferred method. The high degree of location transparency inherent in simple delegation greatly increases the risk of a client being compromised by a Trojan horse application.

When server principals run the **sec_login_become_delegate()** or **sec_login_become_impersonator()** call to become an intermediary in a delegation chain, they must also specify the delegation type as input to the call. The type they specify must be the same type as the delegation type specified by the initiator of the chain (unless they specify no delegation).

27.2.2 Target and Delegate Restrictions

When a principal enables delegation or becomes an intermediary in a delegation chain, the principal may specify target and delegate restrictions. Target restrictions identify the server principals (by UUID) to which the identities in a delegation chain can be projected. Delegate restrictions identify the server principals that can further project the delegation chain.

If a target restriction prohibits a server from seeing an identity in a delegation chain, the security runtime replaces that identity with the identity of the *anonymous principal*. If a delegate restriction prohibits a principal from being an intermediary in a chain, then the security runtime replaces that principal’s identity with the identity of the anonymous

principal. This replacement with the anonymous identity allows the authenticated RPC call to complete. Whether the operation requested by the delegation chain is performed can be controlled by ACL entries that grant permission to the anonymous principal on the objects that are the targets of the delegated operation.

If no delegate restrictions are supplied, any principal can be an intermediary in the delegation chain. If any delegate restrictions are supplied, then only those supplied can further transmit the delegation chain.

Note: In the current release of DCE, there is no way for a server to register its DCE credentials with the RPC runtime. Only a server name and key table can currently be registered. Because of this limitation, target restrictions are currently implemented so that *all* target servers see anonymous credentials for any EPAC that contains any target restriction regardless of the identity specified in the restriction.

27.2.2.1 The Anonymous Principal

The DCE Security Service replaces those identities in the delegation chain that are not allowed to be seen by target or delegate restrictions with the UUIDs associated with the anonymous principal's identity. These UUIDs are as follows:

- Anonymous principal UUID: **fad18d52-ac83-11cc-b72d-0800092784e9**
- Anonymous group UUID: **fc6ed07a-ac83-11cc-97af-0800092784e9**

The **other_obj**, **any_other**,**other_obj_deleg**, and **any_other_deleg** ACL entries define the anonymous principal's access to objects. The entries must be set up just as for any other principal. The appropriate direct or delegate permissions must be granted to the anonymous principal or the delegated operation will fail.

27.2.2.2 Target and Delegate Restriction Syntax

Target and delegate restrictions are expressed as a list of values of type **sec_id_restriction_t**. This data type consists of a UUID and an entry type. The entry type specifies whether the UUID identifies a principal, a group, or ''any other'' principals (in a manner similar to the **any_other** ACL entry type). As in ACL entry

types, the target restriction entry types can refer to principals and groups from the local cell or from foreign cells.

The possible delegation entry types are as follows:

- **sec_rstr_e_type_user**

  The target or delegate is a local principal identified by UUID.

- **sec_rstr_e_type_group**

  The target or delegate is any member of a local group identified by UUID.

- **sec_rstr_e_type_foreign_user**

  The target or delegate is a foreign principal identified by principal and cell UUID.

- **sec_rstr_e_type_foreign_group**

  The target or delegate is any member of a foreign group identified by group and cell UUID.

- **sec_rstr_e_type_foreign_other**

  The target or delegate is any principal that can authenticate to the foreign cell identified by UUID.

- **sec_rstr_e_type_any_other**

  The target or delegate is any principal that can authenticate to any cell.

- **sec_rstr_e_type_no_other**

  No principal can act as a target or delegate.

27.2.3 Optional and Required Restrictions

When a principal calls **sec_login_become_initiator()** to enable delegation, or **sec_login_become_delegate()** or **sec_login_become_impersonator()** to become an intermediary, the principal can specify optional and required restrictions. Optional and required restrictions are provided for use by applications that have specific authorization requirements. These restrictions, which are defined by the application, can be set by initiators or intermediaries, and are interpreted and enforced by application target servers. Servers can ignore optional restrictions that they cannot interpret, but they must reject requests associated with a required restriction that they

cannot interpret. Both optional and required restrictions are supplied as values of type **sec_id_opt_req_t**. They are inserted in an EPAC by the privilege server and evaluated by the target server application.

27.2.4 Compatibility Between Version 1.1 and Pre-Version 1.1 Servers and Clients

Prior to DCE Version 1.1, a principal's privilege attributes were stored in a privilege attribute certificate (PAC). At Version 1.1, the PAC was renamed to EPAC and extended to include the following:

- Target, delegate, optional, and required restrictions.
- Extended registry attributes (ERAs), as described in Chapter 29.

Additionally, authorization credentials can now consist of multiple EPACs, as in delegation chains, instead of a single PAC.

When a pre-Version 1.1 client interacts with a Version 1.1 server or vice versa, the Version 1.1 server requires an EPAC and the pre-Version 1.1 server requires a PAC.

For Version 1.1 servers, the security runtime automatically converts the PAC supplied by a pre-Version 1.1 client to an EPAC. For pre-Version 1.1 servers, the security runtime automatically extracts PAC data from the credentials supplied by the Version 1.1 client. However, because an EPAC for a delegation chain contains the privilege attributes of multiple principals and a PAC contains only one set of privilege attributes, the principals engaged in delegation must specify how to handle this issue of multiple versus single identities.

When a principal initiates delegation or becomes an intermediary in a delegation chain, that principal can specify whether to use the privilege attributes of the chain initiator or the last intermediary in the chain to construct the PAC required by a pre-Version 1.1 server. This compatibility decision is specified as a value of type **sec_id_compatibility_mode_t**, which is set to one of the following three values:

- **sec_id_compat_mode_none**

  Compatibility mode is off. The security runtime supplies the application server with an unauthenticated PAC.

- **sec_id_compat_mode_initiator**

Compatibility mode is on. The pre-Version 1.1 PAC data is extracted from the EPAC of the delegation initiator.

- **sec_id_compat_mode_caller**

  Compatibility mode is on. The pre-Version 1.1 PAC data extracted from the EPAC of the last intermediary in the delegation chain.

27.3 Calls to Extract Privilege Attribute Information

The EPA API **sec_cred_* ()** and login API **sec_login_cred_* ()** calls extract privilege attribute information. These calls return information associated with an opaque handle to an authenticated identity.

The **sec_cred_* ()** calls are used by servers that have been called by a client with authenticated credentials. The calls and the information they return are as follows:

- **sec_cred_get_authz_session_info()**

  Returns a client's authorization information

- **sec_cred_get_client_princ_name()**

  Returns the principal name of the client

- **sec_cred_get_deleg_restrictions()**

  Returns delegate restrictions

- **sec_cred_get_delegate()**

  Returns a credential handle to the privilege attributes of a delegate in a delegation chain

- **sec_cred_get_delegation_type()**

  Returns the delegation type

- **sec_cred_get_extended_attrs()**

  Returns extended attributes

- **sec_cred_get_initiator()**

  Returns a credential handle to the privilege attributes of the initiator of a delegation chain

- **sec_cred_get_opt_restrictions()**

  Returns optional restrictions

- **sec_cred_get_pa_data()**

  Returns privilege attributes from a credential handle

- **sec_cred_get_req_restrictions()**

  Returns required restrictions

- **sec_cred_get_tgt_restrictions()**

  Returns target restrictions

- **sec_cred_get_v1_pac()**

  Returns pre-Version 1.1 privilege attributes

- **sec_cred_is_authenticated()**

  Returns TRUE if the caller's privilege attributes are authenticated or FALSE otherwise

The **sec_login_cred_* ()** calls are used by clients that are part of a delegation chain. The calls and the information they return are as follows:

- **sec_login_cred_get_delegate()**

  Returns the privilege attributes of a delegate in a delegation chain.

- **sec_login_cred_get_initiator()**

  Returns the privilege attributes of the initiator of a delegation chain

The **sec_cred_*()** and **sec_login_* ()** calls discussed in this chapter return information about authenticated principals associated with an opaque credential handle supplied to the call. Two credential handles are used:

- **sec_login_handle_t** (returned by a client-side **sec_login_get_current_context()** call)
- **rpc_authz_cred_handle_t** (returned by a server-side **rpc_inq_auth_caller()** call)

These are handles to all the credentials in a delegation chain. The **sec_login_cred_get_initiator()**, **sec_login_cred_get_delegate()**, **sec_cred_get_initiator()**, and **sec_cred_get_delegate()** calls return a handle

of type **sec_cred_pa_handle_t**, which is a handle to the extended privilege attributes of a particular identity in a delegation chain. The other **sec_cred_* ()** and **sec_login_* ()** calls discussed in this chapter take the **sec_cred_pa_handle_t** handle and return the requested information for the particular identity.

27.4 Disabling Delegation

The login API **sec_login_disable_delegation()** call disables delegation for a specified login context. It returns a new login context of type **sec_login_handle_t** without any delegation information and prevents any further delegation.

27.5 Setting Extended Attributes

The login API **sec_login_set_extended_attrs()** call adds extended registry attributes to a login context. The extended registry attributes must have been established and attached to the object by using the ERA API. (For more information on ERAs, see Chapter 29.)

Chapter 28

The Registry API

This chapter describes the registry API. Like the other security APIs, this one provides a simpler binding mechanism than the standard RPC handle structure. It includes facilities for creating and maintaining the registry database. Applications that run in the default DCE registry environment (that is, those that assume the presence of the default registry tools and servers) have no reason to call this API.

28.1 Binding to a Registry Site

Although it is often convenient to speak of the registry database in a way that implies that it is a single physical database, the registry database is replicated in all but the very smallest cells. Replication reduces network traffic and increases the availability of registry data to clients.

A cell's registry database usually consists of an update site (also known as the *master* site), and a number of query sites (also known as read-only, or *slave* sites). Changes to data at the master site are propagated to its slaves by messages sent by the master. Query sites can only satisfy requests for data (for example, **sec_rgy_acct_lookup()**,

which returns account information). Requests for database changes (for example, **sec_rgy_acct_passwd()**, which changes the password for an account) must be directed to the master site; a query site that receives such a request returns an error.

To submit requests to the registry server, a client must first select a site and bind to it. The client may select a site by name, ask the DCE Directory Service to bind to the master site, or select an arbitrary site. In addition, a client may select a cell and bind to a registry site in that cell.

The registry API enables a client to communicate with the registry server via a specified authentication protocol, at a specified protection level, and using a specified authorization protocol. For instance, a developer may decide that the protection level for communicating with an update site should be higher (that is, more secure) than that for a query site; that is, the developer may feel that, on the one hand, the relatively infrequent changes to registry data should be done in a highly secure manner and that, on the other hand, authentication overhead should be reduced for the more frequent requests for registry data. The registry API accommodates these varying needs.

The following calls bind a client to a registry server in preparation for registry operations. The argument list of these calls enables an application to specify the authentication protocol, the protection level, and the authorization protocol to be used:

- **sec_rgy_site_bind()**

  Binds to a specified site

- **sec_rgy_site_bind_update()**

  Binds to the update site

- **sec_rgy_site_bind_query()**

  Binds to any query site

- **sec_rgy_cell_bind()**

  Binds to any registry site in a specified cell

- **sec_rgy_site_binding_get_info()**

  Extracts the registry site name and security information from the binding handle

The following calls are similar to the binding calls just described, except that an application cannot specify security information. By default, however, the following

calls use DCE shared-secret authentication, the packet-integrity level of protection, and DCE authorization.

- **sec_rgy_site_open()**

  Binds to the specified site

- **sec_rgy_site_open_update()**

  Binds to any update site

- **sec_rgy_site_open_query()**

  Binds to any query site

- **sec_rgy_site_get()**

  Gets the registry site name from the binding handle

The following calls provide miscellaneous binding management functionality:

- **sec_rgy_site_close()**

  Terminates binding to a registry site and frees resources associated with this binding

- **sec_rgy_site_is_readonly()**

  Tests whether a bound site is an update or query site

28.2 The Registry Database

The registry database comprises three container objects:

- **principal**

  Contains principal names; each name is associated with account information that is also specified here (for example, the name of the primary group)

- **group**

  Contains groups and the names of their member principals

- **organization**

  Contains organizations and the names of their member principals

These three objects are referred to as *name domains*, and each member of a domain is referred to as a *PGO item*. Principal items are contained in the principal domain, groups in the group domain, and organizations in the organization domain. A principal may have a name such as **/rd/writers/tom**, from which you might infer that **tom** is a member of the group **writers** and the organization **rd**. However, this is not the case because the name **/rd/writers/tom** only indicates that **tom** and the data corresponding to the account of this principal (if any) reside in **/rd/writers** in the principal domain. There may also be a group named **/rd/writers** in the group domain, but the principal **tom** is not a member unless he is explicitly named in the group **/rd/writers** in the group domain.

Each PGO item consists of a print string name, a UUID, and a UNIX number (for compatibility with UNIX system security interfaces). For various administrative reasons, it is frequently convenient to be able to refer to a PGO item by more than one name. Consequently, some PGO items are aliases for other items. An alias uses the same UUID and UNIX number as the PGO item to which it refers, but contains only a pointer to that item.

The registry also contains the **rgy** object, which describes registry properties and policies, and organization policies.

28.2.1 Creating and Maintaining PGO Items

The PGO items in the registry database are created and maintained with routines that are prefixed with **sec_rgy_pgo_**. The contents of a PGO item vary with the domain. If the domain is **group** or **organization**, the contents are the membership list of principal names. If the domain is **principal**, the contents are the data corresponding to the registry account using that name.

The **sec_rgy_pgo_*()** interface contains the following calls for maintaining the PGO trees:

- **sec_rgy_pgo_add()**

  Adds a PGO item

- **sec_rgy_pgo_delete()**

  Deletes a PGO item

- **sec_rgy_pgo_rename()**

Changes the name of a PGO item

- **sec_rgy_pgo_replace()**

  Replaces information corresponding to the specified PGO item

The **sec_rgy_pgo_*()** interface contains the following calls for maintaining PGO membership lists:

- **sec_rgy_pgo_add_member()**

  Adds a member to a group or organization membership list

- **sec_rgy_pgo_delete_member()**

  Deletes a member from a group or organization membership list

- **sec_rgy_pgo_get_members()**

  Returns a list of members of a group or organization

- **sec_rgy_pgo_is_member()**

  Tests whether a principal is a member of a specified group or organization

The **sec_rgy_pgo_* ()** interface contains the following calls for retrieving PGO item data:

- **sec_rgy_pgo_get_by_id()**

  Returns the PGO item with the specified UUID

- **sec_rgy_pgo_get_by_eff_unix_num()**

  Returns the PGO item with the specified effective UNIX number

- **sec_rgy_pgo_get_by_name()**

  Returns the PGO item with the specified name

- **sec_rgy_pgo_get_by_unix_num()**

  Returns the PGO item with the specified UNIX number

- **sec_rgy_pgo_get_next()**

  Returns the PGO item that follows the last PGO item returned

The **sec_rgy_pgo_*()** interface also contains routines that convert PGO item specifiers, as follows:

- **sec_rgy_pgo_id_to_name()**
- **sec_rgy_pgo_id_to_unix_num()**
- **sec_rgy_pgo_name_to_id()**
- **sec_rgy_pgo_unix_num_to_id()**
- **sec_rgy_pgo_name_to_unix_num()**
- **sec_rgy_pgo_unix_num_to_name()**

28.2.2 Creating and Maintaining Accounts

The *login-name* field of an account contains a principal name, a primary group name, and an organization name. The account may also contain a project list (also known as a *concurrent group set*) that specifies all the groups to which the principal corresponding to the account belongs, but the *login-name* field itself specifies only one group name.

An account can be added to the registry database only when all of its constituent PGO items are established. For instance, to create an account with the principal name **tom**, the group name **writers**, and the organization name **rd**, all three names must exist as individual PGO items in the database; and the **writers** group and the **rd** organization must specify that **tom** is a member.

When an account is created with **sec_rgy_acct_add()** (and if a project list is enabled for the new account), the call scans the groups in the registry and creates a project list containing all the groups in which the principal name appears. Subsequently, the project list may be modified with the **sec_rgy_pgo_add_member()** and **sec_rgy_pgo_delete_member()** calls.

The following calls create and maintain accounts:

- **sec_rgy_acct_add()**

  Adds an account to an existing principal item

- **sec_rgy_acct_delete()**

  Deletes an account, leaving the principal item

- **sec_rgy_acct_rename()**

Changes an account login name, perhaps moving the account to a different principal item

The following calls return the information in an account:

- **sec_rgy_acct_get_projlist()**

  Returns the project list for an account

- **sec_rgy_acct_lookup()**

  Returns all the account data

The following calls modify the information in an account:

- **sec_rgy_acct_passwd()**

  Changes an account password

- **sec_rgy_acct_replace_all()**

  Replaces all of an account's data

- **sec_rgy_acct_admin_replace()**

  Replaces only the administrative account data

- **sec_rgy_acct_user_replace()**

  Replaces only the account data that is accessible to the user of the account

28.2.3 Registry Properties and Policies

The following subsections outline some registry API parameters that affect the cell as a whole, and the routines that enable an application to retrieve and set values for them.

28.2.3.1 Registry Properties

Several registry parameters and flags affect all accounts in the registry. These registry properties include the following:

- The version number of the registry software used to create and read the registry

- The name and UUID of the cell associated with the registry, and whether the current registry site is an update site or a query site
- Minimum and default lifetimes for certificates of identity issued to principals
- Bounds on the UNIX numbers used for principals, and whether the UUIDs of principals also contain embedded UNIX numbers

The routines associated with this parameter set are

- **sec_rgy_properties_get_info()**
- **sec_rgy_properties_set_info()**

28.2.3.2 The Registry Authentication Policy

Another set of parameters affecting all principals is the registry authentication policy. This set only controls the maximum lifetime of certificates of identity, upon first issue and renewal. Accounts also have authentication policies, and the policy in effect for any principal is the most restrictive combination of the registry policy and the policy for a principal's account. The associated routines are

- **sec_rgy_auth_plcy_get_info()**
- **sec_rgy_auth_plcy_get_effective()**
- **sec_rgy_auth_plcy_set_info()**

28.2.3.3 Organization Policies

Another parameter set controls the set of accounts of principals that are members of an organization. These parameters control the lifetime and length of passwords, as well as the set of characters from which passwords may be composed. This parameter set also specifies the default lifespan of accounts associated with the organization. The routines associated with this parameter set are

- **sec_rgy_plcy_get_info()**
- **sec_rgy_plcy_get_effective()**
- **sec_rgy_plcy_set_info()**

28.2.4 Routines to Return UNIX Structures

The registry API provides calls to obtain registry entries in a UNIX compatible structure. These APIs return account and group entries similar to the **getpwnam**, **getgrnam**, **getpwuid**, and **getgrid** UNIX library routines. These APIs, which can be called by the corresponding UNIX library routines to ensure compatibility with UNIX programs, are

- **sec_rgy_unix_getpwnam()**

  Returns a UNIX compatible password entry for an account specified by name

- **sec_rgy_unix_getgrnam()**

  Returns a UNIX compatible group entry for an account associated with a specified group name

- **sec_rgy_unix_getpwuid()**

  Returns a UNIX compatible password entry for an account specified by UNIX ID

- **sec_rgy_unix_getgrgid()**

  Returns a UNIX compatible group entry for an account associated with a specified group ID

28.2.5 Miscellaneous Registry Routines

The registry API includes a few miscellaneous routines, as follows:

- **sec_rgy_login_get_info()**

  Returns login information for the specified account.

- **sec_rgy_login_get_effective()**

  Applies local overrides (if such data is available) to registry account information and returns information about which account information fields have been overridden

- **sec_rgy_wait_until_consistent()**

  Blocks until all previous database updates have been propagated to all sites. This is useful for applications that first bind and write to an update site, and then bind to an arbitrary query site and depend upon up-to-date information.

Note: The **sec_rgy_wait_until_consistent()** routine is not available in DCE Release 1.0 Version 1.0.2.

- **sec_rgy_cursor_reset()**

  Resets the database cursor to return the first suitable entry

Chapter 29

The Extended Attribute API

This chapter describes the extended attribute APIs. There are two extended attribute APIs: the extended registry attribute (ERA) interface to create attributes in the registry database and the DCE attribute interface to create attributes in a database of your choice.

The ERA interface (consisting of **sec_attr_* ()** calls) provides facilities for extending the registry database by creating, maintaining, and viewing attribute types and instances, and providing information to and receiving it from outside attribute servers known as *attribute triggers* . It is the preferred API for security schema and attribute manipulations. Application servers that manage legacy security attributes or provide third-party processing of attributes stored in the registry database can export and implement the **sec_attr()** interface. Trigger servers are accessed through the **sec_attr_trig()** interface by the security client agent during certain **sec_rgy_attr_ *()** calls. The ERA interface uses the same binding mechanism as the registry API, described in Chapter 28.

The DCE attribute interface (consisting of **dce_attr_sch_ *()** calls) is provided for schema and attribute manipulation of data repositories other than the registry. Although similar to the ERA interface, the functionality of the DCE attribute interface is limited

to creating schema entries (attribute types). The interface does not provide calls to create and manipulate attribute instances or to access trigger servers.

The chapter first describes the ERA interface and then the DCE attribute interface. Finally is describes macros and utilities provided for developers who use either attribute API.

29.1 The ERA API

The registry is a repository for principal, group, organization, and account data. It stores the network privilege attributes used by DCE and account data used by local operating systems. This local account data, however, is appropriate only for UNIX operating systems. The ERA facility provides a mechanism for extending the registry schema to include data (attributes) required by or useful to operating systems other than UNIX operating systems.

The ERA API provides the ability to define attribute types and to attach attribute instances to registry objects. Registry objects are nodes in the registry database, to which access is controlled by an ACL manager type. The registry objects are

- **principal**
- **group**
- **organization**
- **policy**
- **directory**
- **replist**
- **attr_schema**

All registry objects and their accompanying ACL manager type are described in the *DCE 1.2.2 Administration Guide—Core Components*.

The ERA API also provides a trigger interface that application servers use to integrate their attribute services with ERA services.

29.1.1 Attribute Schema

The schema extensions are implemented in a single attribute schema that is essentially a catalog of schema entries, each of which defines the format and function of an attribute type. The schema can be dynamically updated to create, modify, or delete schema entries.

The attribute schema is identified by the name **xattrschema** under the security junction point (usually **/.:/sec**) in the CDS namespace. Access to the attribute schema (hereinafter called simply *schema*) is controlled by an ACL on the schema object. The schema is propagated from the master security server to replicas, like other registry data. Since the attribute schema is local to a cell, it defines the types that can be used within the cell, but not outside the cell (unless the type is also defined in another cell).

29.1.2 Attribute Types and Instances

Each attribute type definition in the schema consists of attribute type identifiers (UUID and name) and semantics that control the instances of attributes of this type. In this manual, *schema entry* refers to the registry entry that defines an attribute type.

An attribute instance is an attribute that is attached to an object and has a value (as opposed to an attribute type, which has no values but simply defines the semantics to which attribute instances of that attribute type must adhere). Attribute instances contain the UUID of their attribute type.

29.1.3 Attribute Type Components

The **sec_attr_schema_entry_t** data type defines an attribute type. This data type contains attribute type identifiers and characteristics.

The identifiers of attribute types are a name and a UUID. Generally, the name is used for interactive access and the UUID for programmatic access.

Attribute type characteristics describe the format and function of the attribute type and thus control the format and function of instances of that type. These characteristics,

all specified in the **sec_attr_schema_entry_t** data type, are described in the following sections.

29.1.3.1 Attribute Encoding

Attribute encoding defines the legal encoding for instances of the attribute type. The encoding controls the format of the attribute instance values, such as whether the attribute value is an integer, string, a UUID, or a vector of UUIDs that define an attribute set.

Attribute encodings are specified in the **sec_attr_encoding_t** data type (fully described in the *DCE 1.2.2 Application Development Reference*).

The possible encodings for attribute types are

- **any**

  The attribute instance value can be of any legal encoding type.

- **void**

  The attribute instance has no value. It is simply a marker that is either present or absent.

- **printstring**

  The attribute value is a printable IDL character string from the DCE Portable Character Set (PCS).

- **printstring_array**

  The attribute value is an array of print strings.

- **integer**

  The attribute value is a signed 32-bit integer.

- **bytes**

  The attribute value is a string of bytes. The byte string is assumed to be a pickle or is otherwise a self-describing type.

- **confidential_bytes**

The attribute value is a string of encrypted bytes. This encrypted data can be passed over the network and is available to user-developed applications.

- **internationalization_data**

  An internationalized string of bytes with a tag identifying the OSF registered codeset used to encode the data.

- **uuid**

  A DCE UUID.

- **attr_set**

  The value is an attribute set, a vector of attribute type UUIDs used to associate multiple related attribute instances (members of the set). The vector contains the UUIDs of each member of the set. Attribute sets provide a flexible way to group related attributes on an object for easier search and retrieval.

  The attribute type UUIDs referenced in an attribute set instance must correspond to existing attribute schema entries. Although the members specified in a set are generally expected to be attached to the object to which the set instance is attached, no checking is done to confirm that they are. Thus, it is possible to create an attribute set instance on an object before creating member attribute instances on that object. A query on such an attribute set returns all instances of member attributes that exist on the object along with a warning that some attribute types were missing.

  Note that attribute sets cannot be nested; a member UUID of an attribute set cannot itself identify an attribute set.

  A query on an attribute set expands to a query per the set's members. In other words, an attribute lookup operation on an attribute set returns all attribute instances that are members of the set, not the set instance itself. (Certain operations, such as **sec_rgy_attr_set_lookup_by_id()** and **sec_rgy_attr_lookup_by_name()**, can retrieve attribute set instances.)

  Updates to an attribute set (**sec_rgy_attr_update()**) do not expand the update to its members but apply only to the attribute set. Since the value carried by a set instance is a vector containing the UUIDs of the member attribute types, an update makes changes only to the set's members, not the values carried by those member attributes. Deletions of attribute sets delete only the set instance, not the member instances.

Since the attributes that are set members exist independently of the attribute set, they can be manipulated directly like any other attribute.

- **binding**

  The attribute value is a **sec_attr_binding_info_t** type containing authentication, authorization, and binding information suitable for communicating with a DCE server.

29.1.3.2 ACL Manager Set

An attribute type's ACL manager set specifies the ACL manager type or types (by UUID) that control access to the object types to which attribute instances of this type can be attached. Attribute instances can be attached only to objects protected by the ACL manager types in the schema entry. For example, suppose an ACL manager set for an attribute type named **MVSname** lists only the ACL manager type for principals. Then, instances of the attribute type named **MVSname** can be attached only to principals and not any other registry objects.

Access to an attribute instance is controlled by the ACL on the object to which the attribute instance is attached and access control is implemented by the object's ACL manager type. For example, access to an attribute named **MVSname** on the principal object named **delores** is controlled by the ACL on the **delores** object.

Do not confuse access to an attribute type definition (a schema entry) with access to an attribute instance. As described previously, access to a schema entry is controlled by the ACL on the **xattrschema** object. Access to an attribute instance is controlled by the ACL on the object to which the attribute instance is attached.

In addition to the ACL manager types, the ACL manager set defines the permission bits needed to query, update, test, and delete instances of the attribute type. These bits are used by the object's ACL manager to determine rights to the object's attributes.

The ACL manager types and permissions defined for the attribute type apply to all instances of the attribute type.

Note that the ACL manager facility supports additional generic attribute type permissions (**O** through **Z** inclusive). Administrators can assign these permissions to attribute types of their choice. All uses of these additional permission bits are

controlled by the cell's administrator. See the *DCE 1.2.2 Administration Guide—Core Components* for more information.

29.1.3.3 Attribute Flags

The attribute type flags set in a schema entry are described in the following paragraphs.

29.1.3.3.1 The Unique Flag

The unique flag specifies whether the value of each instance of an attribute type must be unique within the cell. For example, assume that an instance of attribute type A is attached to 25 principals in the cell. If the unique flag is set on, the value of the A attribute for each of those 25 principals must be different. If it is set off, all 25 principals can share the same value for attribute A.

29.1.3.3.2 The Multivalued Flag

The multivalued flag specifies whether instances of the attribute can be multivalued. If an attribute is multivalued, multiple instances of the same attribute type can be attached to a single registry object. For example, if the multivalued flag is set on, a single principal can have multiple instances of attribute type A. If the flag is set off, a single principal can have only one instance of attribute type A.

All instances' multivalued attributes share the UUID (the UUID of their attribute type), but the values carried by the instances differ. Generally, to access all instances of a multivalued attribute, you supply the attribute UUID. To access a specific instance of a multivalued attribute, you supply the UUID and the value carried by that instance.

29.1.3.3.3 The Reserved Flag

The reserved flag indicates whether the attribute type can be deleted from the schema. Note that, when an attribute type is deleted, all instances of the attribute type are deleted. If the reserved flag is set on, the entry cannot be deleted. If the reserved flag is set off, authorized principals can delete the schema entry.

29.1.3.3.4 The Apply-Defaults Flag

The apply-defaults flag indicates whether or not default attributes should be returned when objects are queried by a client with the **sec_rgy_attr_get_effective()** call. If the apply-defaults flag is set on, defaults are applied. If it is set off, defaults are not supplied.

Defaults are determined in the following manner:

1. If the requested attribute exists on the principal, that attribute is returned. If it does not, the search continues.
2. The next step in the search depends on the type of object:

   For principals with accounts:

   a. The organization named in the principal's account is examined to see if an attribute of the requested type exists. If it does, it is returned and the search ends. If it does not, the search continues to the **policy** object as described in Step 2b.

   b. The registry **policy** object is examined to see if an attribute of the requested type exists. If it does, it is returned. If it does not, a message indicating that no attribute of the type exists for the object is returned.

   For principals without accounts, for groups, and for organizations:

   The registry **policy** object is examined to see if an attribute of the requested type exists. If it does, it is returned. If it does not, a message indicating that no attribute of the type exists for the object is returned.

29.1.3.4 The Intercell Action Field

The intercell action field of the schema entry specifies the action that should be taken by the privilege server when reading attributes from a foreign cell. This field can contain one of three values:

- **sec_attr_intercell_act_accept**

  To accept the foreign attribute instance

- **sec_attr_intercell_act_reject**

To reject the foreign attribute instance

- **sec_attr_intercell_act_evaluate**

  To call a remote trigger server to determine how the attribute instance should be handled

When the privilege server generates a PTGT for a foreign principal, it retrieves the list of attributes from the foreign principal's EPAC.

These attributes instances may be attached to the **principal** object itself or attached to the group or **organization** object associated with the **principal** object.

The privilege server then checks the local attribute schema for attribute types with UUIDs that match the UUIDs of the the attribute instances from the foreign cell that are contained in the EPAC. At this point, the privilege server takes one of the following two actions:

1. If the privilege server cannot find a matching attribute type in the local attribute schema, it checks the **unknown_intercell_action** attribute on the **policy** object. If the **unknown_intercell_action** attribute is set to

   - **sec_attr_intercell_act_accept**, the foreign attribute instance is retained and included in the EPAC generated for the object by the privilege server.
   - **sec_attr_intercell_act_reject**, the foreign attribute is discarded.

   Note: The **unknown_intercell_action** attribute must be created by the system administrator and attached to the **policy** object. The attribute type, which takes the same values as the intercell_action field, has the following characteristics:

   Name: **unknown_intercell_action**

   Attribute UUID:
   171e0ef2c-d12e-11cc-bb7b-080009353559

   Encoding: **sec_attr_encoding_integer**

   ACL manager set: **policy_acl_mgr**

   Unique: false

   Multivalued: false

   Reserved: true

Comment text: Flag indicating whether to accept or reject foreign attributes for which no schema entry exists

2. If the privilege server finds a matching attribute type in the local attribute schema, it retrieves the attribute. The action it now takes depends on the setting of the attribute type's intercell action field and unique flag as follows:

   - If the intercell action field is set to **sec_attr_intercell_act_accept** and

     — The unique flag is not set on, the privilege server includes the foreign attribute instance in the principal's EPAC.

     — The unique flag is set on, the privilege server includes the foreign attribute instance in the principal's EPAC only if the attribute instance value is unique among all instances of the attribute type within the local cell.

     Note: If the unique attribute type flag is set on and a query trigger exists for a given attribute type, the intercell action field cannot be set to **sec_attr_intercell_act_accept** because, in this case, only the query trigger server can reasonably perform a uniqueness check.

   - If the intercell action field is set to **sec_attr_intercell_act_reject**, the privilege server unconditionally discards the foreign attribute instance.

   - If the intercell action field is set to **sec_attr_intercell_act_evaluate**, the privilege server makes a remote **sec_attr_trig_intercell_avail()** call to an attribute trigger by using the binding information in the local attribute type schema entry. The remote attribute trigger decides whether to retain, discard, or map the attribute instance to another value(s). The privilege server includes the values returned by the attribute trigger in the **sec_attr_trig_query()** call output array in the principal's EPAC.

29.1.3.5 Attribute Scope

The scope field controls the objects to which the attribute can be be attached. If scope is defined, the attribute can be attached only to objects defined by the scope. For example, if the scope for a given attribute type is defined as the directory name **/principal/krbgt**, instances of that attribute type can be attached only to objects in the **/principal/krbgt** directory (a directory that by convention contains only cell principals). If the scope is narrowed by fully specifying an object in the **/principal/krbgt** directory

(for example, **/principal/krbgt/dresden.com**) then the attribute can be attached only to the **dresden.com** principal.

29.1.3.6 Trigger Type Flag

The schema entry trigger type flag specifies whether the trigger server associated with the attribute type is invoked for update or query operations. See Section 29.4 for more information on attribute triggers.

29.1.3.7 Trigger Binding

The schema entry trigger binding field contains a binding handle to a remote trigger that will perform processing for the attribute instances. See Section 29.4 for more information on attribute triggers.

29.2 Calls to Manipulate Schema Entries

This section first introduces the **sec_attr_schema_entry_t** data type used by the calls that create and update schema entries that define attribute types. It then describes the calls that create, modify, delete, and read schema entries.

29.2.1 The sec_attr_schema_entry_t Data Type

The **sec_attr_schema_entry_t** data type is used in the calls that create and update schema entries. The data type consists of four values and six other data types. The values used by the **sec_attr_schema_entry_t** are the attribute type name, UUID, scope, and a text field for comments.

The data types used by the **sec_attr_schema_entry_t** are

- **sec_attr_sch_entry_flags_t**

  Specifies the unique, multivalued, reserved, and apply defaults attribute flags.

- **sec_attr_acl_mgr_info_set_t**

  Specifies the attribute type's ACL manager(s). This data type defines the attribute type ACL manager set. This data type contains an array of pointers of type **sec_attr_mgr_info_p_t**, which reference **sec_attr_acl_mgr_info_t** data types. There is one **sec_attr_acl_mgr_info_t** data type for each ACL manager associated with the attribute type. Each **sec_attr_acl_mgr_info_t** defines the ACL manager UUID and the permission bits.

- **sec_attr_encoding_t**

  Specifies the schema entry encoding.

- **sec_attr_trig_type_t**

  Specifies the type of attribute trigger associated with the attribute type (if an attribute trigger is to be associated with the attribute type). See Section 29.4 for more information on attribute triggers.

- **sec_attr_intercell_action_t**

  Specifies the action to be taken attribute instances of this type that come from a foreign cell.

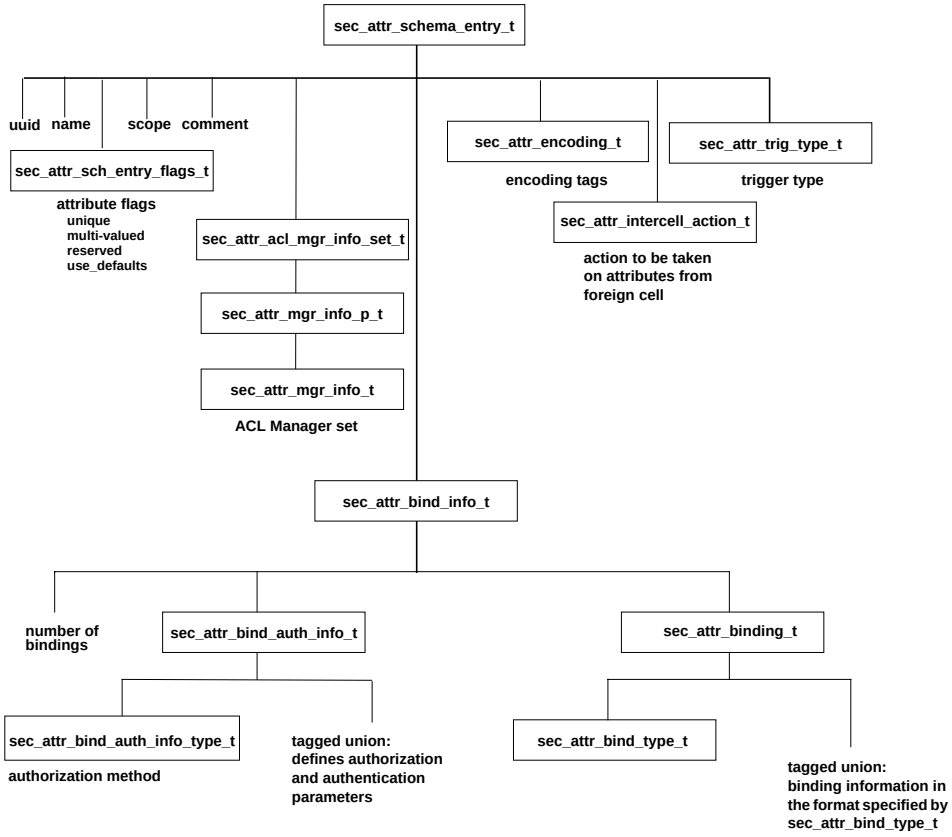
- **sec_attr_bind_info_t**

  Specifies binding information for the trigger server associated with the attribute type (if an attribute trigger is associated with the attribute type).

  The **sec_attr_bind_info_t** data type uses two other data types: **sec_attr_bind_auth_info_t** and **sec_attr_binding_t**. The **sec_attr_bind_info_t** structure for trigger binding is described fully in Section 29.4.

Figure 29-1 illustrates the structure of a **sec_attr_schema_entry_t** data type.

Figure 29–1. The sec_attr_schema_entry_t Data Type



29.2.2 Creating and Managing Schema Entries

This section describes the calls to create, modify, and delete the schema entries that define attribute types.

29.2.2.1 The **sec_rgy_attr_sch_create_entry()** Call

The **sec_rgy_attr_sch_create_entry()** call creates a schema entry that defines an attribute type in the attribute schema.

This call uses the **sec_attr_schema_entry_t** data type that completely defines the schema entry, including the following:

- The attribute type name (generally used for interactive access) and UUID (generally used for programmatic access). Note that attribute instances share the name and UUID of their attribute type.
- The attribute's encoding (described in Section 29.1.3). The encoding is specified as an enumerator of type **sec_attr_encoding_t**. For some kinds of encodings, additional data types are used to further specify the encoding information. These additional data types, the kinds of encodings that require them, and the purpose of the data types are listed in Table 29-1.

Table 29–1. Encodings and Required Data Types

Encoding	Required Data Type	Purpose of Data Type
sec_attr_enc_bytes	**sec_attr_enc_bytes_t**	Defines the length of attribute values
sec_attr_enc_confidential_bytes	**sec_attr_enc_bytes_t**	Defines the length of attribute values
sec_attr_enc_i18n_data	**sec_attr_i18n_data_t**	Defines the internationalization codeset
sec_attr_enc_attr_set	**sec_attr_enc_attr_set_t**	Defines the total number of members in the attribute set and the UUID of each member
sec_attr_enc_printstring	**sec_attr_enc_printstring_t**	Defines a single print string
sec_attr_enc_printstring_array	**sec_attr_enc_str_array_t**	Defines an array of print strings

29.2.2.2 The **sec_rgy_attr_sch_update_entry()** Call

The **sec_rgy_attr_sch_update_entry()** call updates a schema entry that defines an attribute type.

The schema entry components that can be modified are controlled by the ERA API and the *modify_parts* parameter of the **sec_rgy_attr_sch_update_entry()** call.

To ensure that registry and access control data remains consistent, the ERA API allows only the following schema entry components to be modified:

- Attribute name
- Reserved flag
- Apply defaults flag
- Intercell action flag
- Trigger binding
- Comment

Note that ACL managers can be added to a schema entry's ACL manager set, but they cannot be modified or deleted.

To modify any other schema entry fields implies a drastic change to the attribute type. If this change must be made, the schema entry must be deleted (which deletes all attribute instances of that type) and then recreated.

The *modify_parts* parameter of the **sec_rgy_attr_sch_update_entry()** call can also be used to prohibit modification of additional schema entry fields. This parameter, which is actually a **sec_attr_schema_entry_parts_t** data type, lists the fields that can be modified by the call. Only those fields listed in **sec_attr_schema_entry_parts_t** can be modified.

The new values used to update the attribute type are supplied in a **sec_attr_schema_entry_t** data type.

29.2.2.3 The **sec_rgy_attr_sch_delete_entry()** Call

The **sec_rgy_attr_sch_delete_entry()** call deletes attributes types from the attribute schema. The attribute type to be deleted is specified by UUID. When an attribute type is deleted, all instances of that attribute type are invalidated.

29.2.3 Reading Schema Entries

This section describes the calls that read schema entries and the cursor used by the **sec_rgy_attr_sch_scan()** call.

29.2.3.1 Using **sec_attr_cursor_t** with **sec_rgy_attr_sch_scan()**

The **sec_rgy_attr_sch_scan()** call, which reads a specified number of attribute type entries from the attribute schema, uses a cursor of type **sec_attr_cursor_t**. This cursor must be allocated before it can be used as input to the **sec_rgy_attr_sch_scan()** call. In addition, it can also be initialized to the first attribute type entry in the schema, although this is not required. After use, the resources allocated to the **sec_attr_cursor_t** must be released.

The following calls allocate, initialize, and release a **sec_attr_cursor_t** for use with the **sec_rgy_attr_sch_scan()** call:

- **sec_rgy_attr_sch_cursor_init()**

  The **sec_rgy_attr_sch_cursor_init()** call allocates resources to the cursor and initializes the cursor to the first attribute type entry in the attribute schema. This call also supplies the total number of entries in the attribute schema as part of its output. The cursor allocation is a local operation. The cursor initialization is a remote operation and makes a remote call to the registry.

- **sec_rgy_attr_sch_cursor_alloc()**

  The **sec_rgy_attr_sch_cursor_alloc()** call allocates resources to the cursor but does not initialize the cursor. However, since the **sec_rgy_attr_sch_scan()** call will initialize the cursor if it is passed in uninitialized, you may prefer this call to limit the number of remote calls performed by an application. Be aware that the **sec_rgy_attr_sch_cursor_init()** call provides the total number

of entries in the named schema, a piece of information not provided by the **sec_rgy_attr_sch_cursor_alloc()** call.

- **sec_rgy_attr_sch_cursor_release()**

  The **sec_rgy_attr_sch_cursor_release()** call releases all resources allocated to a **sec_attr_cursor_t** cursor used with the **sec_rgy_attr_sch_scan()** call.

- **sec_rgy_attr_sch_cursor_reset()**

  The **sec_rgy_attr_sch_cursor_reset()** call initializes a **sec_attr_cursor_t** cursor used with the **sec_rgy_attr_sch_scan()** call. The reset cursor can then be used without releasing and reallocating.

29.2.3.2 The **sec_rgy_attr_sch_scan()** Call

The **sec_rgy_attr_sch_scan()** call reads a specified number of schema entries from the attribute schema.

The number of entries to read is specified as an unsigned 32-bit integer. The read begins at the entry at which the **sec_attr_cursor_t** cursor is positioned and continues through the number of entries specified. The cursor must be allocated but can be initialized or uninitialized since **sec_rgy_attr_sch_scan()** initializes any uninitialized cursor it receives as input.

The call output includes an array of **sec_attr_schema_entry_t** values and a 32-bit integer that specifies the number of schema entries returned.

To read through all entries in a schema, continue making **sec_rgy_attr_sch_scan()** calls, until the **no_more_entries** message is received. When all calls are complete, release the resources allocated to the **sec_attr_cursor_t** cursor by using the **sec_rgy_attr_sch_cursor_release()** call.

29.2.3.3 The **sec_rgy_attr_sch_lookup_by_id()** and **sec_rgy_attr_sch_lookup_by_name()** Calls

The **sec_rgy_attr_sch_lookup_by_id()** call reads the attribute schema entry identified by UUID. The output of the call is a **sec_attr_schema_entry_t** type that contains the

specified attribute type's name, UUID, and characteristics. Generally, this call is used for programmatic access.

For interactive access, use the **sec_rgy_attr_sch_lookup_by_name()** call. This call returns the same information as the **sec_rgy_attr_sch_lookup_by_id()** call but specifies the schema entry to read by name instead of by UUID.

29.2.4 Reading the ACL Manager Types

Two calls retrieve the ACL manager types that protect objects dominated by a named schema:

- **sec_rgy_attr_sch_get_acl_mgrs()**

  Retrieves the UUIDs of the ACL manager types protecting all objects in a named schema.

- **sec_rgy_attr_sch_aclmgr_strings()**

  Retrieves printable strings for each ACL manager type protecting objects in a named schema. The strings contain the ACL manager type's name, associated help information, and supported permission bits.

29.3 Calls to Manipulate Attribute Instances

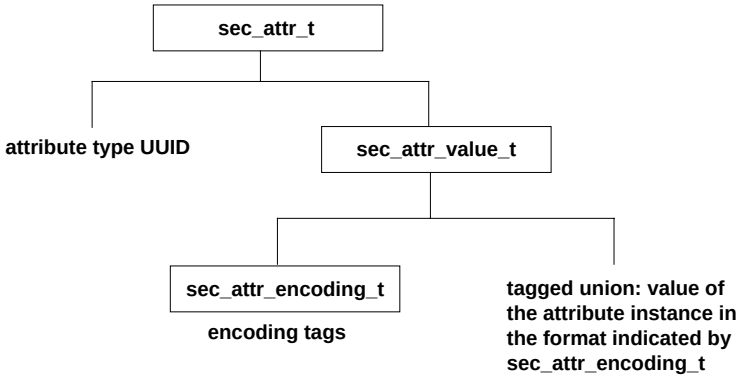
This section introduces the **sec_attr_schema_t** data type used by the calls that create and update attribute instances and then describes the calls that create, modify, delete, and read attribute instances. For all calls, the object whose attributes should be accessed is identified by name and by the domain in which the object exists. (The domain parameter is ignored for the **Policy** and the **Replist** objects.) Registry domains are described in Chapter 28.

29.3.1 The sec_attr_t Data Type

The **sec_attr_t** data type is used in the calls that create and update attribute instances. The data type consists of a value of type **uuid_t** that identifies the attribute to be

accessed by UUID and data type of **sec_attr_value_t**. The **sec_attr_value_t** data type is a tagged union of the actual value assigned (or to be assigned to the attribute instance) and a data type of **sec_attr_encoding_t** that specifies the encoding tags that define the attribute type characteristics. Figure 29-2 illustrates the structure of a **sec_attr_t** data type.

Figure 29–2. The sec_attr_t Data Type



29.3.2 Creating and Managing Attribute Instances

This section describes the calls to create, modify, and delete the attribute instances.

29.3.2.1 The **sec_rgy_attr_update()** Call

The **sec_rgy_attr_update()** call creates new attribute instances and updates existing attribute instances attached to an object specified by name and registry domain. The instances to be created or updated are passed as an array of **sec_attr_t** data types.

Because the new values are passed in as an array, if the update of any attribute instance in the array fails, all fail. However, to help pinpoint the cause of the failure, the call

identifies the first attribute whose update failed in a failure index by array element number.

For existing attribute instances attached to the object, the values passed in the array overwrite the existing values. In other words, if the UUID passed in the input array matches the UUID of an existing instance, the values passed in overwrite the existing values.

If the attribute instance does not exist, it is created. In other words, if the UUID passed in in the array does not match any other attribute type UUID attached to the object, a new attribute instance is created.

For multivalued attributes, because every instance of the multivalued attribute is identified by the same UUID, every instance is overwritten with the supplied value. For example, suppose object **delores** has three attributes of the multivalued type **security_role**. If you pass in one value for **security_role**, the values of all three are changed to the one you enter.

To change only one of the **security_role** values, you must supply the values that should be unchanged as well as the new value. For example, suppose object **delores** has three **security_role** attributes with values of **level1**, **level2**, and **level3**. To change **level1** to **level1O. and retain level2** and **level3**, the input array must contain **level1.5**, **level2**, and **level3** .

To create instances of multivalued attributes, you must create individual **sec_attr_t** data types to define each multivalued attribute instance and then pass all of them in the **sec_rgy_attr_update()** input array.

If an input attribute is associated with an update attribute trigger, the attribute trigger is invoked (by the **sec_attr_trig_update()** call), and the values in the **sec_rgy_attr_update()** input array are used as input to the update attribute trigger. The output values from the update attribute trigger are stored in the registry database and returned in the **sec_rgy_attr_update()** output array.

29.3.2.2 The **sec_rgy_attr_test_and_update()** Call

The **sec_rgy_attr_test_and_update()** call, like the **sec_rgy_attr_update()** call, creates new attribute instances and updates existing attribute instances attached to

an object specified by name and registry domain. However, it performs the update only if a set of specified attribute instances match the attribute instances that already exist for the object. This call is useful to ensure that updates are made only if certain conditions exist.

The attribute instances to be matched are passed in an input array of **sec_attr_t** values. Other than this conditional test, this call functions exactly the same as the **sec_rgy_attr_update()** call.

29.3.2.3 The **sec_rgy_attr_delete()** Call

The **sec_rgy_attr_delete()** call deletes the specified attribute instances from an object identified by name and registry domain. The attribute instances to be deleted are passed in as an array of values of **sec_attr_t**.

To delete attribute instances that are not multivalued and to delete all instances of a multivalued attribute, an attribute UUID is all that is required. For these attribute instances, supply the attribute UUID in the input array and set the attribute encoding (in **sec_attr_encoding_t**) to **sec_attr_enc_void**.

To delete a specific instance of a multivalued attribute, you must supply the UUID and value that uniquely identify the multivalued attribute instance in the input array.

Note that, if the deletion of any attribute instance in the array fails, all fail. However, to help pinpoint the cause of the failure, the call identifies the first attribute whose deletion failed in a failure index by array element number.

29.3.3 Reading Attribute Instances

This section describes the calls that read attribute instances, and it describes the cursor used by the **sec_rgy_attr_lookup_by_id()** call.

29.3.3.1 Using **sec_rgy_attr_cursor_t** with **sec_rgy_attr_lookup_by_id()**

The **sec_rgy_attr_lookup_by_id()** call, which reads attributes for a specified object, uses a cursor of type **sec_attr_cursor_t**. This cursor must be allocated before it can be used as input to the **sec_rgy_attr_lookup_by_id()** call. In addition, it can also be initialized to the first attribute in the specified object's list of attributes, although this is not required. After use, the resources allocated to the **sec_attr_cursor_t** must be released.

The following calls allocate, initialize, and release a **sec_attr_cursor_t** for use with the **sec_rgy_attr_lookup_by_id()** call:

- **sec_rgy_attr_cursor_init()**

  The **sec_rgy_attr_sch_cursor_init()** call allocates resources to and initializes the cursor to the first attribute in the specified object's list of attributes. This call also supplies the total number of attributes attached to the object as part of its output. The cursor allocation is a local operation. The cursor initialization is a remote operation and makes a remote call to the registry.

- **sec_rgy_attr_cursor_alloc()**

  The **sec_rgy_attr_cursor_alloc()** call allocates resources to the cursor but does not initialize the cursor. However, since the **sec_rgy_attr_lookup_by_id()** call will initialize the cursor if it is passed in uninitialized, you may prefer this call to limit the number of remote calls performed by the application. Be aware that the **sec_rgy_attr_cursor_init()** call provides the total number of attributes attached to the specified object, a piece of information not provided by this call.

- **sec_rgy_attr_cursor_release()**

  The **sec_rgy_attr_cursor_release()** call releases all resources allocated to a **sec_attr_cursor_t** cursor used with the **sec_rgy_attr_lookup_by_id()** call.

- **sec_rgy_attr_cursor_reset()**

  The **sec_rgy_attr_cursor_reset()** call reinitializes a **sec_attr_cursor_t** cursor used with the **sec_rgy_attr_lookup_by_id()** call. The reset cursor can then be used without releasing and reallocating.

29.3.3.2 The **sec_rgy_attr_lookup_by_id()** Call

The **sec_rgy_attr_lookup_by_id()** call reads attributes specified by UUID for an object specified by name and domain. Specifically the call returns the following:

- An array of **sec_attr_t** values.
- A count of the total number of attribute instances returned.
- A count of the total number of attribute instances that could not be returned because of size constraints of the **sec_attr_t** array. (Note that the call allows the size of the array to be specified.)

For multivalued attributes, the call returns a **sec_attr_t** for each value as an individual attribute instance. For attribute sets, the call returns a **sec_attr_t** for each member of the set, but not the set instance. This routine is useful for programmatic access.

If the attribute instance to be read is not associated with a query trigger or no additional information is required by the query trigger, an attribute UUID is all that is required. For these attribute instances, supply the attribute UUID in the input array and set the attribute encoding (in **sec_attr_encoding_t**) to **sec_attr_enc_void** .

If the attribute instance to be read is associated with a query attribute trigger that requires additional information before it can process the query request, use a **sec_attr_value_t** to supply the requested information by doing the following:

- Set the **sec_attr_encoding_t** to an encoding type that is compatible with the information required by the query attribute trigger.
- Set the **sec_attr_value_t** to hold the required information.

You can define the number of elements in the input array of **sec_attr_t** values (in the *num_attr_keys* parameter). If you define the number of elements as 0 (zero), the call returns all of the object's attribute instances that the caller is authorized to see. You should be aware, however, that if you define the number of elements as zero and the attribute is associated with a query attribute trigger, you will be unable to pass any information to the query attribute trigger.

29.3.3.3 The **sec_rgy_attr_set_lookup_by_id()** Call

The **sec_rgy_attr_set_lookup_by_id()** call reads attribute sets specified by set instance UUID for an object specified by name and domain. Specifically the call returns the following:

- A **sec_attr_t** for each attribute instance in the attribute set.
- A count of the total number of attribute set instances returned.
- A count of the total number that could not be returned because of size constraints of the **sec_attr_t** array. (Note that the call allows the size and length of the array to be specified.)

Note: Since attribute triggers cannot be associated with an attribute set instance, this call provides no way to supply input data to a query attribute trigger.

29.3.3.4 The **sec_rgy_attr_lookup_by_name()** Call

The **sec_rgy_attr_lookup_by_name**() call reads a single attribute instance specified by name for an object specified by name and domain. The call returns a **sec_attr_t** for the specified attribute instance.

For multivalued attributes, the call returns the first instance of the multivalued attribute. (To retrieve every instance of a multivalued attribute, use the **sec_rgy_attr_lookup_by_id()** call.)

For attribute sets, the call returns the attribute set instance, not the member instances. To retrieve all members of the set, use the **sec_rgy_attr_lookup_by_id()** call.

Note: This call provides no way to supply input data to a query attribute trigger. If the attribute to be read is associated with a query trigger that requires input data, use the **sec_rgy_attr_lookup_by_id()** call.

29.4 The Attribute Trigger Facility

Some attribute types require the support of an outside server either to verify input attribute values or to supply output attribute values when those values are stored in an external database. Such a server could, for example, connect a legacy registry system to the DCE registry. The attribute trigger facility provides for automatic calls to outside DCE servers, known as *attribute triggers*.

Trigger servers, which are written by application developers, export the **sec_attr_trig** interface. They are invoked automatically when an attribute that has been associated with an attribute trigger (during schema entry creation) is queried or updated. The attribute trigger facility consists of three components:

- The attribute schema trigger fields (**trig_types** and **trig_binding**) that associate an attribute trigger and its binding information with an attribute type. These fields are part of the standard creation of a schema entry that defines an attribute type. See Section 29.1.1.
- The **sec_attr_trig** APIs that define the query and update trigger operations. The APIs are provided in the **sec_attr_trig_** ***()** calls.
- The user-written attribute trigger servers are independent from DCE servers. The trigger servers implement the trigger operations for the attribute types that require attribute trigger processing. These servers are not provided as part of DCE, but must be written by application developers.

29.4.1 Defining an Attribute Trigger/Attribute Association

When an attribute is created with the **sec_rgy_attr_update()** call, you define the association between the attribute type and an attribute trigger by specifying the following:

- Trigger Type

  Defines the trigger as a query server (invoked for query operations) or an update server (invoked for updates operations). The trigger type is defined in a **sec_attr_trig_type_t** data type, which is used by a **sec_attr_schema_entry_t** data type.

- Trigger Binding

Defines the server binding handle for the attribute trigger. The details of the trigger binding are defined in a number of data types, which are also used by the **sec_attr_schema_entry_t** data type. Trigger binding is described in detail in Section 29.1.3.7.

Only if both of pieces of information are provided will the association between the attribute type and the attribute trigger be created. You can associate an attribute trigger to any attribute type of any encoding except for attribute sets.

29.4.1.1 Query Triggers

When you execute a call that accesses an attribute associated with a query trigger, the client-side attribute lookup code performs the following tasks:

- Binds to the attribute trigger (using a binding from the attribute type's schema entry)
- Makes the remote **sec_attr_trig_query()** call to the attribute trigger server, passing in the attribute keys and optional information provided by the caller
- If the **sec_attr_trig_query()** call is successful, returns the output attribute(s) to the caller

If you execute a **sec_rgy_attr()** update call with an attribute type that is associated with a query trigger, not an update trigger, the input attribute values are ignored and a ''stub'' attribute instance is created on the named object simply to mark the existence of this attribute on the object. Modifications to the real attribute value must occur at the attribute trigger.

29.4.1.2 Update Triggers

When you execute a call that accesses an attribute associated with an update trigger, the client-side attribute update code performs the following tasks:

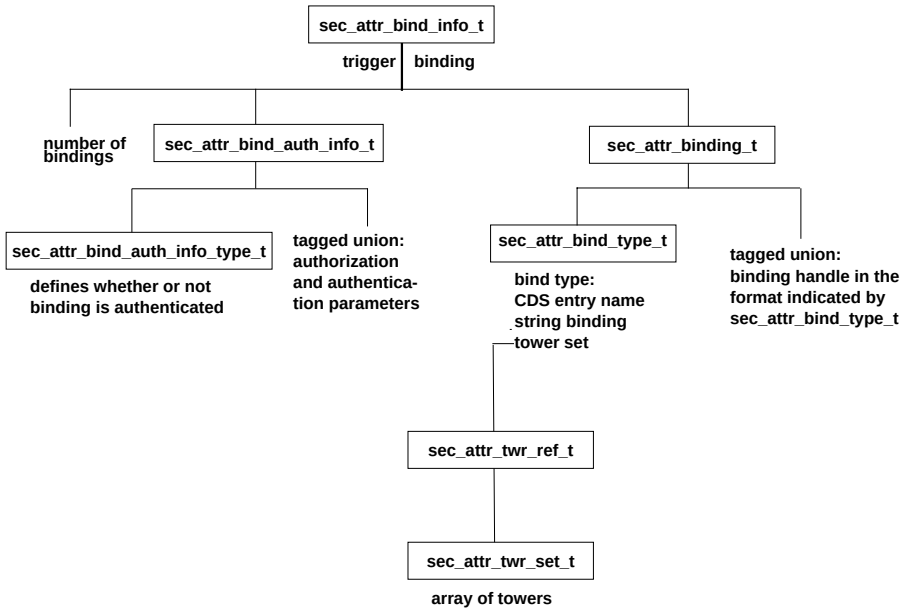
- Binds to the attribute trigger (using a binding from the attribute type's schema entry)
- Makes the remote **sec_attr_trig_update()** call to the attribute trigger server, passing in the attributes provided by the caller

- If the **sec_attr_trig_update()** call is successful, stores the output attribute(s) in the registry database and returns the output attribute(s) to the caller

29.4.2 Trigger Binding

Two data types are used to defined an attribute trigger. The **sec_attr_trig_type_t** type defines the type of attribute trigger. The **sec_attr_bind_info_t** data type, illustrated in Figure 29-3 and described in this section, specifies the attribute trigger's binding.

Figure 29–3. The sec_attr_bind_info_t Data Type



The **sec_attr_bind_info_t** data type uses two data types: **sec_attr_binding_t** , which defines the information used to generate binding handle and **sec_attr_bind_auth_info_t**, which defines the binding authentication and authorization information.

29.4.2.1 The **sec_attr_binding_t** Data Type

To describe the binding handle, the **sec_attr_binding_t** type uses a **sec_attr_bind_type_t** data type that specifies the format to the data used to generate the binding handle and a tagged union that contains the binding handle. The binding handle can be generated from any of the following:

- **A server directory entry name** (used with **rpc_ns_binding_import_ * ()** calls)

  If the binding information is a server name, call **rpc_ns_binding_import_begin()** to establish a context for importing RPC binding handles from the name service database. For the **rpc_ns_binding_import_begin()** call, specify the CDS server directory entry name, an entry name syntax value of **rpc_c_ns_syntax_dce**, and **sec_attr_trig** as the interface handle of the interface to import.

- **A string binding** (used with **rpc_binding_from_string_binding()** calls)

  If the binding information is a string binding, call **rpc_binding_from_string_binding()** to generate an RPC binding handle.

- **An RPC protocol tower set** (used with **rpc_tower_to_binding()** calls)

  If the binding information is a protocol tower, two additional data types are used to pass in an unallocated array of towers, which the server must then allocate. These data types are **sec_attr_twr_ref_t** to point to the tower and **sec_attr_twr_set_t** to define the array of towers.

  Architectural components of DCE can take advantage of the internal **rpc_tower_to_binding** operation in **rpcpvt.idl** to generate a binding handle from the canonical representation of a protocol tower.

Although the server directory entry name, with the actual server address stored in CDS, is the recommended way to specify an attribute trigger binding handle, prototype applications may want to specify a string binding or protocol tower for convenience.

29.4.2.2 The **sec_attr_bind_auth_info_t** Data Type

To describe whether or not RPC calls to the server will be authenticated and, for authenticated calls, to provide authentication and authorization information, the **sec_attr_bind_auth_info_t** type uses the **sec_attr_bind_auth_info_type_t** data type, and a tagged union. The **sec_attr_bind_auth_info_type_t** type defines whether or not

the call is authenticated. The tagged union contains the authentication and authorization parameters.

Once a binding handle is obtained, call **rpc_binding_set_auth_info()** and supply it with the binding handle and authorization and authentication information.

29.4.3 Access Control on Attributes with Triggers

When a query or update call accesses an attribute associated with an attribute trigger, the call checks the ACL of the object with which the attribute is associated to see if the client has the permissions required for the operation. If access is granted, the operation returns a binding handle authenticated with the client's login context. This handle is then used to perform the **sec_attr_trig_query** or **sec_attr_trig_update** operation.

Access to information maintained by an attribute trigger is controlled entirely by that attribute trigger. The attribute trigger can choose to implement any authorization mechanism, including none. For example, the attribute trigger can obtain the client's identity from the RPC runtime to perform name-based authentication and perform ACL checks (or any other type of access control mechanism), and it can query the registry attribute schema for the attribute type's permission set to use for an ACL check. Access control on attribute information stored outside of the registry database is left to the application designer.

29.5 Calls that Access Attribute Triggers

This section describes the calls that send information to and receive it from attribute triggers.

29.5.1 Using sec_attr_trig_cursor_t with sec_attr_trig_query()

The **sec_attr_trig_query()** call, which reads attributes associated with a query attribute trigger, uses a cursor of type **sec_attr_trig_cursor_t**. This cursor must be allocated and initialized before it can be used as input to the **sec_attr_trig_query()** call. After use, the resources allocated to **sec_attr_trig_cursor_t** must be released.

The following calls allocate, initialize, and release a **sec_attr_trig_cursor_t** type for use with the **sec_attr_trig_query()** call:

- **sec_attr_trig_cursor_init()**

  The **sec_attr_trig_cursor_init()** call allocates resources to the cursor and initializes the cursor to the first attribute in the list of attributes for the object whose binding handle is specified. This call makes a remote call.

- **sec_attr_trig_cursor_release()**

  The **sec_rgy_attr_cursor_release()** call releases all resources allocated to a **sec_attr_trig_cursor_t** type by **sec_attr_trig_cursor_init()**.

29.5.2 The sec_rgy_attr_trig_query() and sec_rgy_attr_trig_update() Calls

The **sec_attr_trig_query()** call reads instances of attributes coded with a trigger type of query for a specified object. It passes an array of **sec_attr_t** values to a query attribute trigger and receives the output parameters back from the server. The **sec_attr_trig_update()** routine passes attributes coded with a trigger type of update to an update attribute trigger for evaluation before the updates are made to the registry.

Both calls are called automatically by the DCE attribute lookup or update code for all schema entries that specify a trigger. Although you should not call these calls directly, if you are implementing a trigger server, it will receive input from these calls and the attribute trigger's output should be passed back to them. The data received must be in a form accessible to the call and, if it is the result of an update, a form that can be stored in the registry database.

The object whose attribute instances are to be read or updated is identified by

- The name of the cell in which the object exists
- The name of the object or a UUID in string format that identifies the object

29.5.3 The priv_attr_triq_query() Call

The **priv_attr_trig_query()** call is used by the privilege service to retrieve trigger attributes and add them to a princpal's EPAC. The privilege service executes this call when it receives a request to add a principal and its extended attribute instances to an EPAC and the attributes are associated with a trigger server. The call passes an array of **sec_attr_t** values to the attribute trigger and receives the attribute values back from the trigger server in another array of **sec_attr_t** values. If the principal is being added to a delegation chain, the call also passes the UUIDs of all of the current members of the delegation chain to the trigger server. The trigger server can then evaluate all identities to determine access rights to the requested attributes.

Like the **sec_rgy_attr_trig_update()** calls, you will not call **priv_attr_trig_query()** directly. However, if you are implementing a trigger server, it will receive input from these calls and the attribute trigger's output should be passed back to the call. The data received must be in a form accessible to the call.

29.6 The DCE Attribute API

The DCE attribute calls are not described in detail. This is because, with the exception of the calls that bind to a selected database (**dce_attr_sch_bind()(** and **dce_attr_sch_bind_free()**), the **dce_sec_attr_*()** calls are the same as the **sec_rgy_attr_sch_*()** calls. Refer to Section 29.1 for information on using each call. Note also that the DCE attribute calls are suffixed with **3dce**, not **3sec** (for example, **dce_attr_sch_bind.3dce**).

The DCE attribute API consists of the following calls:

- **dce_attr_sch_bind()**

  Returns an opaque handle of type **dce_attr_sch_handle_t** to a schema object specified by name and sets authentication and authorization parameters for the handle. This is the call used to bind to the schema of your choice.

- **dce_attr_sch_bind_free()**

  Releases an opaque handle of type **dce_attr_sch_handle_t**.

- **dce_attr_sch_create_entry()**

Creates a schema entry in a schema bound to with **dce_attr_sch_bind**. This call is based on **sec_rgy_attr_sch_create_entry()** and is used in the same way.

- **dce_attr_sch_update_entry()**

  Updates a schema entry in a schema bound to with **dce_attr_sch_bind()**. This call is based on **sec_rgy_attr_sch_update_entry()** and is used in the same way.

- **dce_attr_sch_delete_entry()**

  Deletes a schema entry in a schema bound to with **dce_attr_sch_bind()**. This call is based on **sec_rgy_attr_sch_delete_entry()** and is used in the same way.

- **dce_attr_sch_scan()**

  Reads a specified number of schema entries. This call is based on **sec_rgy_attr_sch_scan()** and is used in the same way.

- **dce_attr_sch_cursor_init()**

  Allocates resources to and initializes a cursor used with **dce_attr_sch_scan()**. The **dce_attr_sch_cursor_init()** routine makes a remote call that also returns the current number of schema entries in the schema. The **dce_attr_sch_cursor_init()** call is based on **sec_rgy_attr_sch_cursor_init()** and is used in the same way.

- **dce_attr_sch_cursor_alloc()**

  Allocates resources to a cursor used with **dce_attr_sch_scan()** . The **dce_attr_sch_cursor_alloc()** routine is a local operation. The **dce_attr_sch_cursor_alloc()** call is based on **sec_rgy_attr_sch_cursor_alloc()** and is used in the same way.

- **dce_attr_sch_cursor_release()**

  Releases states associated with a cursor created by **dce_attr_sch_cursor_alloc()** or **dce_attr_sch_cursor_init()**. The **dce_attr_sch_cursor_release()** call is based on **sec_rgy_attr_sch_cursor_release()** and is used in the same way.

- **dce_attr_sch_cursor_reset()**

  Reinitializes a cursor used with **dce_attr_sch_scan()**. The reset cursor can then be reused without releasing and reallocating. This call is based on the **sec_rgy_attr_sch_cursor_reset()** and is used in the same way.

- **dce_attr_sch_lookup_by_id()**

  Reads a schema entry identified by UUID. This call is based on **sec_rgy_attr_lookup_by_id()** and is used in the same way.

- **dce_attr_sch_lookup_by_name()**

  Reads a schema entry identified by name. This call is based on **sec_rgy_attr_sch_lookup_by_name()** and is used in the same way.

- **dce_attr_sch_get_acl_mgrs()**

  Retrieves the UUIDs of ACL manager types protecting objects dominated by a named schema. This call is based on **sec_rgy_attr_sch_get_acl_mgrs()** and is used in the same way.

- **dce_attr_sch_aclmgr_strings()**

  Retrieves the print strings containing information about ACL manager types protecting objects dominated by a named schema. The print strings contain the manager's name, help information, and supported permission bits. This call is based on **sec_rgy_attr_sch_aclmgr_strings()** and is used in the same way.

29.7 Macros to Aid Extended Attribute Programming

The extended attribute APIs includes macros to help programmers using the extended attribute interfaces. The macros perform a variety of functions including

- Accessing fields in data structures
- Calculating the size of data structures
- Performing semantic and flag checks
- Setting flags

The macros are in **dce/rpcbase.h**, which is derived from **dce/rpcbase.idl**.

The following subsections list the definitions of each macro.

29.7.1 Macros to Access Binding Fields

In the following macro definitions, which are used by a **sec_attr_schema_entry_t** and its equivalent **dce_attr_sch** data type, B is a pointer to **sec_attr_bind_info_t**.

```
#define SA_BND_AUTH_INFO(B)  (B)->auth_info
#define SA_BND_AUTH_INFO_TYPE(B) (SA_BND_AUTH_INFO(B)).info_type
#define SA_BND_AUTH_SVR_PNAME_P(B) \
(SA_BND_AUTH_DCE_INFO(B)).svr_princ_name
#define SA_BND_AUTH_PROT_LEVEL(B) \
(SA_BND_AUTH_DCE_INFO(B)).protect_level
#define SA_BND_AUTH_AUTHN_SVC(B) \
(SA_BND_AUTH_DCE_INFO(B)).authn_svc
#define SA_BND_AUTH_AUTHZ_SVC(B) \
(SA_BND_AUTH_DCE_INFO(B)).authz_svc
#define SA_BND_NUM(B)   (B)->num_bindings
#define SA_BND_ARRAY(B,I)  (B)->bindings[I]
#define SA_BND_TYPE(B,I)  (SA_BND_ARRAY(B,I)).bind_type
#define SA_BND_STRING_P(B,I) \
(SA_BND_ARRAY(B,I)).tagged_union.string_binding
#define SA_BND_SVRNAME_P(B,I) \
(SA_BND_ARRAY(B,I)).tagged_union.svrname
#define SA_BND_SVRNAME_SYNTAX(B,I) \
(SA_BND_SVRNAME_P(B,I))->name_syntax
#define SA_BND_SVRNAME_NAME_P(B,I) \
(SA_BND_SVRNAME_P(B,I))->name
#define SA_BND_TWRSET_P(B,I) \
(SA_BND_ARRAY(B,I)).tagged_union.twr_set
#define SA_BND_TWRSET_COUNT(B,I) (SA_BND_TWRSET_P(B,I))->count
#define SA_BND_TWR_P(B,I,J)  (SA_BND_TWRSET_P(B,I))->towers[J]
#define SA_BND_TWR_LEN(B,I,J)  (SA_BND_TWR_P(B,I,J))->tower_length
#define SA_BND_TWR_OCTETS(B,I,J) \
(SA_BND_TWR_P(B,I,J))->tower_octet_string
```

29.7.2 Macros to Access Schema Entry Fields

In the following macro definitions, S is a pointer to **sec_attr_schema_entry_t** (and its equivalent **dce_attr_sch** data type) and I and J are nonnegative integers for array element selection.

```
#define SA_ACL_MGR_SET_P(S)             (S)->acl_mgr_set
#define SA_ACL_MGR_NUM(S)               (SA_ACL_MGR_SET_P(S))->num_acl_mgrs
#define SA_ACL_MGR_INFO_P(S,I)          (SA_ACL_MGR_SET_P(S))->mgr_info[I]
#define SA_ACL_MGR_TYPE(S,I)            (SA_ACL_MGR_INFO_P(S,I))->acl_mgr_type
#define SA_ACL_MGR_PERMS_QUERY(S,I)     (SA_ACL_MGR_INFO_P(S,I))->query_permset
#define SA_ACL_MGR_PERMS_UPDATE(S,I)    (SA_ACL_MGR_INFO_P(S,I))->update_permset
#define SA_ACL_MGR_PERMS_TEST(S,I)      (SA_ACL_MGR_INFO_P(S,I))->test_permset
#define SA_ACL_MGR_PERMS_DELETE(S,I)    (SA_ACL_MGR_INFO_P(S,I))->delete_permset
#define SA_TRG_BND_INFO_P(S)            (S)->trig_binding
#define SA_TRG_BND_AUTH_INFO(S) \
(SA_BND_AUTH_INFO(SA_TRG_BND_INFO_P(S)))
#define SA_TRG_BND_AUTH_INFO_TYPE(S) \
(SA_BND_AUTH_INFO_TYPE(SA_TRG_BND_INFO_P(S)))
#define SA_TRG_BND_AUTH_DCE_INFO(S) \
(SA_BND_AUTH_DCE_INFO(SA_TRG_BND_INFO_P(S)))
#define SA_TRG_BND_AUTH_SVR_PNAME_P(S) \
(SA_BND_AUTH_SVR_PNAME_P(SA_TRG_BND_INFO_P(S)))
#define SA_TRG_BND_AUTH_PROT_LEVEL(S) \
(SA_BND_AUTH_PROT_LEVEL(SA_TRG_BND_INFO_P(S)))
#define SA_TRG_BND_AUTH_AUTHN_SVC(S) \
(SA_BND_AUTH_AUTHN_SVC(SA_TRG_BND_INFO_P(S)))
#define SA_TRG_BND_AUTH_AUTHZ_SVC(S) \
(SA_BND_AUTH_AUTHZ_SVC(SA_TRG_BND_INFO_P(S)))
#define SA_TRG_BND_NUM(S) \
(SA_BND_NUM(SA_TRG_BND_INFO_P(S)))
#define SA_TRG_BND_ARRAY(S,I) \
(SA_BND_ARRAY((SA_TRG_BND_INFO_P(S)),I))
#define SA_TRG_BND_TYPE(S,I) \
(SA_BND_TYPE((SA_TRG_BND_INFO_P(S)),I))
#define SA_TRG_BND_STRING_P(S,I) \
(SA_BND_STRING_P((SA_TRG_BND_INFO_P(S)),I))
#define SA_TRG_BND_SVRNAME_P(S,I) \
(SA_BND_SVRNAME_P((SA_TRG_BND_INFO_P(S)),I))
#define SA_TRG_BND_SVRNAME_SYNTAX(S,I) \
(SA_BND_SVRNAME_SYNTAX((SA_TRG_BND_INFO_P(S)),I))
#define SA_TRG_BND_SVRNAME_NAME_P(S,I) \
(SA_BND_SVRNAME_NAME_P((SA_TRG_BND_INFO_P(S)),I))
#define SA_TRG_BND_TWRSET_P(S,I) \
(SA_BND_TWRSET_P((SA_TRG_BND_INFO_P(S)),I))
```

```
#define SA_TRG_BND_TWRSET_COUNT(S,I) \
(SA_BND_TWRSET_COUNT((SA_TRG_BND_INFO_P(S)),I))
#define SA_TRG_BND_TWR_P(S,I,J) \
(SA_BND_TWR_P((SA_TRG_BND_INFO_P(S)),I,J))
#define SA_TRG_BND_TWR_LEN(S,I,J) \
(SA_BND_TWR_LEN((SA_TRG_BND_INFO_P(S)),I,J))
#define SA_TRG_BND_TWR_OCTETS(S,I,J) \
(SA_BND_TWR_OCTETS((SA_TRG_BND_INFO_P(S)),I,J))
```

29.7.3 Macros to Access Attribute Instance Fields

In the following macro descriptions, S is a pointer to **sec_attr_t**, and I and J are nonnegative integers for array element selection.

```
#define SA_ATTR_ID(S)   (S)->attr_id
#define SA_ATTR_VALUE(S)  (S)->attr_value
#define SA_ATTR_ENCODING(S)  (SA_ATTR_VALUE(S)).attr_encoding
#define SA_ATTR_INTEGER(S) \
(SA_ATTR_VALUE(S)).tagged_union.signed_int
#define SA_ATTR_PRINTSTRING_P(S) \
(SA_ATTR_VALUE(S)).tagged_union.printstring
#define SA_ATTR_STR_ARRAY_P(S) \
(SA_ATTR_VALUE(S)).tagged_union.string_array
#define SA_ATTR_STR_ARRAY_NUM(S) (SA_ATTR_STR_ARRAY_P(S))->num_strings
#define SA_ATTR_STR_ARRAY_ELT_P(S,I) (SA_ATTR_STR_ARRAY_P(S))->strings[I]
#define SA_ATTR_BYTES_P(S) \
(SA_ATTR_VALUE(S)).tagged_union.bytes
#define SA_ATTR_BYTES_LEN(S)  (SA_ATTR_BYTES_P(S))->length
#define SA_ATTR_BYTES_DATA(S,I)  (SA_ATTR_BYTES_P(S))->data[I]
#define SA_ATTR_IDATA_P(S) \
(SA_ATTR_VALUE(S)).tagged_union.idata
#define SA_ATTR_IDATA_CODESET(S) (SA_ATTR_IDATA_P(S))->codeset
#define SA_ATTR_IDATA_LEN(S)  (SA_ATTR_IDATA_P(S))->length
#define SA_ATTR_IDATA_DATA(S,I)  (SA_ATTR_IDATA_P(S))->data[I]
#define SA_ATTR_UUID(S) \
```

```
(SA_ATTR_VALUE(S)).tagged_union.uuid
#define SA_ATTR_SET_P(S) \
(SA_ATTR_VALUE(S)).tagged_union.attr_set
#define SA_ATTR_SET_NUM(S)  (SA_ATTR_SET_P(S))->num_members
#define SA_ATTR_SET_MEMBERS(S,I) (SA_ATTR_SET_P(S))->members[I]
#define SA_ATTR_BND_INFO_P(S) \
(SA_ATTR_VALUE(S)).tagged_union.binding
#define SA_ATTR_BND_AUTH_INFO(S) \
(SA_BND_AUTH_INFO(SA_ATTR_BND_INFO_P(S)))
#define SA_ATTR_BND_AUTH_INFO_TYPE(S) \
(SA_BND_AUTH_INFO_TYPE(SA_ATTR_BND_INFO_P(S)))
#define SA_ATTR_BND_AUTH_DCE_INFO(S) \
(SA_BND_AUTH_DCE_INFO(SA_ATTR_BND_INFO_P(S)))
#define SA_ATTR_BND_AUTH_SVR_PNAME_P(S) \
(SA_BND_AUTH_SVR_PNAME_P(SA_ATTR_BND_INFO_P(S)))
#define SA_ATTR_BND_AUTH_PROT_LEVEL(S) \
(SA_BND_AUTH_PROT_LEVEL(SA_ATTR_BND_INFO_P(S)))
#define SA_ATTR_BND_AUTH_AUTHN_SVC(S) \
(SA_BND_AUTH_AUTHN_SVC(SA_ATTR_BND_INFO_P(S)))
#define SA_ATTR_BND_AUTH_AUTHZ_SVC(S) \
(SA_BND_AUTH_AUTHZ_SVC(SA_ATTR_BND_INFO_P(S)))
#define SA_ATTR_BND_NUM(S) \
(SA_BND_NUM(SA_ATTR_BND_INFO_P(S)))
#define SA_ATTR_BND_ARRAY(S,I) \
(SA_BND_ARRAY((SA_ATTR_BND_INFO_P(S)),I))
#define SA_ATTR_BND_TYPE(S,I) \
(SA_BND_TYPE((SA_ATTR_BND_INFO_P(S)),I))
#define SA_ATTR_BND_STRING_P(S,I) \
(SA_BND_STRING_P((SA_ATTR_BND_INFO_P(S)),I))
#define SA_ATTR_BND_SVRNAME_P(S,I) \
(SA_BND_SVRNAME_P((SA_ATTR_BND_INFO_P(S)),I))
#define SA_ATTR_BND_SVRNAME_SYNTAX(S,I) \
(SA_BND_SVRNAME_SYNTAX((SA_ATTR_BND_INFO_P(S)),I))
#define SA_ATTR_BND_SVRNAME_NAME_P(S,I) \
(SA_BND_SVRNAME_NAME_P((SA_ATTR_BND_INFO_P(S)),I))
#define SA_ATTR_BND_TWRSET_P(S,I) \
(SA_BND_TWRSET_P((SA_ATTR_BND_INFO_P(S)),I))
#define SA_ATTR_BND_TWRSET_COUNT(S,I) \
(SA_BND_TWRSET_COUNT((SA_ATTR_BND_INFO_P(S)),I))
#define SA_ATTR_BND_TWR_P(S,I,J) \
```

```
(SA_BND_TWR_P((SA_ATTR_BND_INFO_P(S)),I,J))
#define SA_ATTR_BND_TWR_LEN(S,I,J) \
(SA_BND_TWR_LEN((SA_ATTR_BND_INFO_P(S)),I,J))
#define SA_ATTR_BND_TWR_OCTETS(S,I,J) \
(SA_BND_TWR_OCTETS((SA_ATTR_BND_INFO_P(S)),I,J))
```

29.7.4 Binding Data Structure Size Calculation Macros

The following macros are supplied to calculate the size of data types that hold binding information. The macros work with the ERA API data types and their equivalent **dce_attr_sch** data types.

```
/*
* SA_BND_INFO_SIZE(N) - calculate the size required
* for a sec_attr_bind_info_t with N bindings.
*/
#define SA_BND_INFO_SIZE(N) ( sizeof(sec_attr_bind_info_t) + \
(((N) - 1) * sizeof(sec_attr_binding_t)) )
/*
* SA_TWR_SET_SIZE(N) - calculate the size required
* for a sec_attr_twr_set_t with N towers.
*/
#define SA_TWR_SET_SIZE(N) ( sizeof(sec_attr_twr_set_t) + \
(((N) - 1) * sizeof(sec_attr_twr_ref_t)) )
/*
* SA_TWR_SIZE(N) - calculate the size required
* for a twr_t with a tower_octet_string of length N.
*/
#define SA_TWR_SIZE(N) ( sizeof(twr_t) + (N) - 1 )
```

29.7.5 Schema Entry Data Structure Size Calculation Macros

The following macro is supplied to calculate the size of a **sec_attr_alc_mgr_info_set_t** data type.

```
/*
* SA_ACL_MGR_SET_SIZE(N) - calculate the size required
* for a sec_attr_acl_mgr_info_set_t with N acl_mgrs.
*/
#define SA_ACL_MGR_SET_SIZE(N) ( sizeof(sec_attr_acl_mgr_info_set_t) + \
(((N) - 1) * sizeof(sec_attr_acl_mgr_info_p_t)) )
```

29.7.6 Attribute Instance Data Structure Size Calculation Macros

The following macros are supplied to calculate the size of data types that hold attribute information.

```
/*
* SA_ATTR_STR_ARRAY_SIZE(N) - calculate the size required
* for a sec_attr_enc_str_array_t with N sec_attr_enc_printstring_p_t-s.
*/
#define SA_ATTR_STR_ARRAY_SIZE(N) ( sizeof(sec_attr_enc_str_array_t) + \
(((N) - 1) * sizeof(sec_attr_enc_printstring_p_t)) )
/*
* SA_ATTR_BYTES_SIZE(N) - calculate the size required
* for a sec_attr_enc_bytes_t with byte string length of N.
*/
#define SA_ATTR_BYTES_SIZE(N) ( sizeof(sec_attr_enc_bytes_t) + (N) - 1 )
/*
* SA_ATTR_IDATA_SIZE(N) - calculate the size required
* for a sec_attr_i18n_data_t with byte string length of N.
*/
#define SA_ATTR_IDATA_SIZE(N) ( sizeof(sec_attr_i18n_data_t) + (N) - 1 )
/*
* SA_ATTR_SET_SIZE(N) - calculate the size required
* for a sec_attr_enc_attr_set_t with N members (uuids).
*/
#define SA_ATTR_SET_SIZE(N) ( sizeof(sec_attr_enc_attr_set_t) + \
(((N) - 1) * sizeof(uuid_t)) )
```

29.7.7 Binding Semantic Check Macros

The following macros are supplied to check the semantics of entries in the binding fields. The macros work with the ERA API data types and their equivalent **dce_attr_sch** data types.

```
/*
* SA_BND_AUTH_INFO_TYPE_VALID(B) - evaluates to TRUE (1)
* if the binding auth_info type is valid; FALSE (0) otherwise.
* B is a pointer to a sec_attr_bind_info_t.
*/
#define SA_BND_AUTH_INFO_TYPE_VALID(B) ( \
(SA_BND_AUTH_INFO_TYPE(B)) == sec_attr_bind_auth_none || \
(SA_BND_AUTH_INFO_TYPE(B)) == sec_attr_bind_auth_dce ? true : false )
/*
* SA_BND_AUTH_PROT_LEV_VALID(B) - evaluates to TRUE (1)
* if the binding auth_info protect_level is valid; FALSE (0) otherwise.
* B is a pointer to a sec_attr_bind_info_t.
*/
#define SA_BND_AUTH_PROT_LEV_VALID(B) ( \
    (SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_default  || \
(SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_none  || \
(SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_connect  || \
(SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_call  || \
(SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_pkt  || \
(SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_pkt_integ || \
(SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_pkt_privacy ? \
true : false )
/*
* SA_BND_AUTH_AUTHN_SVC_VALID(B) - evaluates to TRUE (1)
* if the binding auth_info authentication service is valid;
* FALSE (0) otherwise.
* B is a pointer to a sec_attr_bind_info_t.
*/
```

```
#define SA_BND_AUTH_AUTHN_SVC_VALID(B) ( \
(SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_none  || \
(SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_dce_secret || \
(SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_dce_public || \
(SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_dce_dummy  || \
(SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_dssa_public || \
(SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_default ? \
true : false )
/*
* SA_BND_AUTH_AUTHZ_SVC_VALID(B) - evaluates to TRUE (1)
* if the binding auth_info authorization service is valid;
* FALSE (0) otherwise.
* B is a pointer to a sec_attr_bind_info_t.
*/
#define SA_BND_AUTH_AUTHZ_SVC_VALID(B) ( \
    (SA_BND_AUTH_AUTHZ_SVC(B)) == rpc_c_authz_none || \
(SA_BND_AUTH_AUTHZ_SVC(B)) == rpc_c_authz_name || \
(SA_BND_AUTH_AUTHZ_SVC(B)) == rpc_c_authz_dce ? \
true : false )
```

29.7.8 Schema Entry Semantic Check Macros

The following macros are supplied to check the semantics of schema entry fields. In the macros, S is a pointer to **sec_attr_schema_entry_t** and its equivalent **dce_attr_sch** data type.

```
#define SA_TRG_BND_AUTH_INFO_TYPE_VALID(S) \
(SA_BND_AUTH_INFO_TYPE_VALID(SA_TRG_BND_INFO_P(S)))
#define SA_TRG_BND_AUTH_PROT_LEV_VALID(S) \
(SA_BND_AUTH_PROT_LEV_VALID(SA_TRG_BND_INFO_P(S)))
#define SA_TRG_BND_AUTH_AUTHN_SVC_VALID(S) \
(SA_BND_AUTH_AUTHN_SVC_VALID(SA_TRG_BND_INFO_P(S)))
#define SA_TRG_BND_AUTH_AUTHZ_SVC_VALID(S) \
(SA_BND_AUTH_AUTHZ_SVC_VALID(SA_TRG_BND_INFO_P(S))
```

29.7.9 Attribute Instance Semantic Check Macros

The following macros are supplied to check the semantics of entries in the attribute instance fields. In the following macros, S is a pointer to **sec_attr_t**. F is a **sec_attr_trigs_types_flags_t**.

```
#define SA_ATTR_BND_AUTH_INFO_TYPE_VALID(S) \
(SA_BND_AUTH_INFO_TYPE_VALID(SA_ATTR_BND_INFO_P(S)))
#define SA_ATTR_BND_AUTH_PROT_LEV_VALID(S) \
(SA_BND_AUTH_PROT_LEV_VALID(SA_ATTR_BND_INFO_P(S)))
#define SA_ATTR_BND_AUTH_AUTHN_SVC_VALID(S) \
(SA_BND_AUTH_AUTHN_SVC_VALID(SA_ATTR_BND_INFO_P(S)))
#define SA_ATTR_BND_AUTH_AUTHZ_SVC_VALID(S) \
(SA_BND_AUTH_AUTHZ_SVC_VALID(SA_ATTR_BND_INFO_P(S))
#define SA_SCH_FLAG_IS_SET(S,F) \
(((S)->schema_entry_flags & (F)) == (F))
#define SA_SCH_FLAG_IS_SET_UNIQUE(S) \
(SA_SCH_FLAG_IS_SET((S),sec_attr_sch_entry_unique))
#define SA_SCH_FLAG_IS_SET_MULTI_INST(S) \
(SA_SCH_FLAG_IS_SET((S),sec_attr_sch_entry_multi_inst))
#define SA_SCH_FLAG_IS_SET_RESERVED(S) \
(SA_SCH_FLAG_IS_SET((S),sec_attr_sch_entry_reserved))
#define SA_SCH_FLAG_IS_SET_USE_DEFAULTS(S) \
(SA_SCH_FLAG_IS_SET((S),sec_attr_sch_entry_use_defaults))
```

29.7.10 Schema Entry Flag Set and Unset Macros

The following macros set and unset flag(s) in the schema entry **schema_entry_flags** field. In the following macros, S is a pointer to **sec_attr_schema_entry_t** .

```
/*
* Macros to set the flags.
*/
#define SA_SCH_FLAG_SET(S, FLAG) ((S)->schema_entry_flags |= (FLAG))
#define SA_SCH_FLAG_SET_UNIQUE(S) \
```

```
(SA_SCH_FLAG_SET((S),sec_attr_sch_entry_unique))
#define SA_SCH_FLAG_SET_MULTI_INST(S) \
(SA_SCH_FLAG_SET((S),sec_attr_sch_entry_multi_inst))
#define SA_SCH_FLAG_SET_RESERVED(S) \
(SA_SCH_FLAG_SET((S),sec_attr_sch_entry_reserved))
#define SA_SCH_FLAG_SET_USE_DEFAULTS(S) \
(SA_SCH_FLAG_SET((S),sec_attr_sch_entry_use_defaults))
/*
* Macros to unset the flags.
*/
#define SA_SCH_FLAG_UNSET(S, FLAG) ((S)->schema_entry_flags \
&= ~(FLAG))
#define SA_SCH_FLAG_UNSET_UNIQUE(S) \
(SA_SCH_FLAG_UNSET((S),sec_attr_sch_entry_unique))
#define SA_SCH_FLAG_UNSET_MULTI_INST(S) \
(SA_SCH_FLAG_UNSET((S),sec_attr_sch_entry_multi_inst))
#define SA_SCH_FLAG_UNSET_RESERVED(S) \
(SA_SCH_FLAG_UNSET((S),sec_attr_sch_entry_reserved))
#define SA_SCH_FLAG_UNSET_USE_DEFAULTS(S) \
(SA_SCH_FLAG_UNSET((S),sec_attr_sch_entry_use_defaults))
```

29.7.11 Schema Trigger Entry Flag Check Macros

The following macros evaluate to TRUE if the requested flag(s) is set in the schema entry **trig_types** field. In the following macros, S is a pointer to **sec_attr_schema_entry_t** and F is a **sec_attr_trigs_types_flags_t** type.

```
#define SA_SCH_TRIG_FLAG_IS_SET(S,F) \
(((S)->trig_types & (F)) == (F))
#define SA_SCH_TRIG_FLAG_IS_NONE(S) \
(SA_SCH_TRIG_FLAG_IS_SET((S),sec_attr_trig_type_none))
#define SA_SCH_TRIG_FLAG_IS_QUERY(S) \
(SA_SCH_TRIG_FLAG_IS_SET((S),sec_attr_trig_type_query))
#define SA_SCH_TRIG_FLAG_IS_UPDATE(S) \
(SA_SCH_FLAG_IS_SET((S),sec_attr_trig_type_update))
```

29.8 Utilities to Use with Extended Attribute Calls

The extended attribute APIs includes utilities to help programmers using the extended attribute interfaces. These utilities are

- **sec_attr_util_alloc_copy**—Copies data from one **sec_attr_t** data type to another.
- **sec_attr_util_free**—Frees memory allocated to **sec_attr_t** by the **sec_attr_util_alloc_copy()** function.
- **sec_attr_util_inst_free_ptrs**—Frees nonnull pointers in a **sec_attr_t type.**
- **sec_attr_util_inst_free**—Frees nonnull pointers in a **sec_attr_t** type and the pointer to the **sec_attr_t** itself.
- **sec_attr_util_sch_ent_free_ptrs**—Frees nonnull pointers in a **sec_attr_schema_entry_ttype.**
- **sec_attr_util_sch_ent_free**—Frees nonnull pointers in a **sec_attr_schema_entry_t** type and the pointer to the **sec_attr_schema_entry_t** itself. The utility also works with the equivalent **dce_attr_sch** data type.

Chapter 30

The Login API

The login API communicates with the security server to establish, and possibly change, a principal's login context. A login context contains the information necessary for a principal to qualify for (although not necessarily be granted) access to network services and possibly local resources as well. Login context information normally includes the following:

- Identity information concerning the principal, including its certificate of identity (in shared-secret authentication, this is the TGT), its PAC, and registry policy information such as the maximum lifetime of certificates of identity.
- The context state; that is, whether the authentication service has validated the context or not.
- The source of authentication information. (It may originate from the network authentication service, or locally, if that network service is unavailable.)

30.1 Establishing Login Contexts

This section outlines the basic procedure by which a network login context is established. See Chapter 24 for a detailed description of this process.

The procedure is as follows:

1. The client calls **sec_login_setup_identity()** specifying the name of the principal whose network identity is to be established. Memory is allocated to receive the principal's login context.
2. The client calls **sec_login_valid_and_cert_ident()**, which does the following:
   a. Forwards a TGT request encrypted with the user's secret key and with a random key, to the authentication service, which decrypts the request, authenticates the principal, and returns a TGT for the principal.
   b. The client's security runtime then decrypts the TGT and forwards it to the privilege service, which creates a PAC for the principal and encloses it in a PTGT, which is returned to the client's security runtime.
   c. The runtime decrypts the message containing the PTGT and returns information about the source of the authentication information to the API. (If the authentication information comes from the network security server, then the login context is validated.)
3. Finally, the client invokes **sec_login_set_context()**, which enables child processes spawned from the calling process to inherit the validated context.

In the walkthrough of user authentication in Chapter 24, we mentioned that one of the functions of **sec_login_valid_and_cert_ident()** is to demonstrate that a valid trust path exists between the authentication service and the host computer on which the principal is logging in. After setting up and validating a login context, any application that sets identity information for local processes should check to be sure that the server that provided the certificate of identity is legitimate in order to demonstrate that the trust path between the client and the authentication service is valid.

30.1.1 Validating the Login Context and Certifying the Security Server

Whereas a validated login context is one that is regarded as legitimate by the local security runtime, a validated and certified login context is one that is not only regarded as legitimate but also can be demonstrated to have been (in all likelihood, that is) issued by a legitimate security server. Certifying that the security server is legitimate prevents faked identity information from being propagated to local processes. For example, a spurious server could collaborate with a dishonest user in order to obtain an identity that conferred comprehensive permissions (for example, the **root** identity). With such an identity, the dishonest user could gain access to sensitive local objects, such as key-storage files for server principals that run on the host. (Servers running on other hosts would not trust this principal, however, because it does not know their keys.) Of course, if a spurious server can return to the application a ticket encrypted with the host's secret key, it means the server has access to the host's key; but, if this is the case, network security has already been seriously undermined.

When an application needs to certify the originator of a certificate of identity, it may call **sec_login_certify_identity()**. This routine makes an authenticated remote procedure call to the local security validation service of the **dced** daemon in order to acquire a ticket to the host principal. If **dced** succeeds in decrypting the message containing the ticket, then the server that granted the certificate of identity must know the host principal's secret key; this evidence indicates that it is a legitimate security server. Since **dced** runs with the identity **root** (in order to access the host's key), the process calling **sec_login_certify_identity()** need not.

The **sec_login_valid_and_cert_ident()** is similar to **sec_login_certify_identity()**, except that it combines the validation and certification procedures (and therefore, the password of the principal that is logging in must be known to the process making this call). The **sec_login_valid_and_cert_ident()** routine calls the security server for a ticket to the host and attempts decryption. The process calling **sec_login_valid_and_cert_ident()** must have access to the host's secret key, and so must run as **root**.

Note: Because system login programs should not set local identities derived from an uncertified context, all login API routines that return data from an uncertified context issue a warning.

30.1.2 Validating the Login Context Without Certifying the Security Server

An application that does not use login contexts to set local identity information does not need to certify its login contexts. Since an illegitimate security server is unlikely to know the key of a remote server principal with which the application may communicate, the application will simply be refused the service requested from the remote server principal. If local operating system identity information is assumed to be neither of interest nor of concern to an application, it may call **sec_login_validate_identity()**, which does not attempt to verify the security server's knowledge of the host principal's key.

The **sec_login_validate_identity()** routine does not acquire a PTGT, unlike the **sec_login_certify_identity()** and **sec_login_valid_and_cert_ident()** routines. Instead, the PTGT is acquired when the application first makes an authenticated remote procedure call.

30.1.3 Example of a System Login Program

Following is an example of a system login program that obtains a login context that can be trusted for both network and local operations.

Note: One of the function calls that appears in the following example, **sec_login_purge_context()**, is described in Section 30.6.4.

```
if (sec_login_setup_identity(principal,sec_login_no_flags,
    &login_context,&st))
{
   ...get password...

   if (sec_login_valid_and_cert_ident(login_context, password,
       &reset_passwd, &auth_src,&st))
   {
      if(auth_src==sec_login_auth_src_network)
      {
         if (GOOD_STATUS(&st)
```

```
            sec_login_set_context(login_context);
         }
      }
      if (reset_passwd)
      {
         ...reset the user's password...

         if (passwd_reset_fails)
         {
            sec_login_purge_context(login_context)

            ...application login-failure actions...
         }

         ...application-specific login-valid actions...
      }
   }
```

30.2 Context Inheritance

A process inherits the login context of its parent process unless the child process is associated with a principal that has logged in and so established a separate login context. The following subsections describe two additional aspects of context inheritance:

- How the initial context is established.
- How a process may inhibit context inheritance.

30.2.1 The Initial Context

An application invokes **sec_login_setup_identity()** so that it can then make other authenticated RPC calls. However, **sec_login_setup_identity()** is itself a local interface to an authenticated remote procedure call, and authenticated RPC needs a validated login context in order to execute. For applications like system login, the daemon **dced** supplies the validated context. However, a daemon that is started before **dced** is running on the host needs to be able to assume its host's identity. The initial

context is established at boot time with **sec_login_init_first()**, which establishes the default context inheritance for processes running on the host. The routines **sec_login_setup_first()** and **sec_login_validate_first()** then set up and validate the context in a procedure like that used for user context validation.

30.2.2 Private Contexts

A process may inhibit context inheritance by setting a flag in **sec_login_setup_identity()**. If the flag indicates that the login context is private, then children of the calling process cannot inherit it. A child process can neither set a private context (since it is the function of **sec_login_set_context()** to make the context inheritable) nor export it to any other process.

30.3 Handling Expired Certificates of Identity

For a dishonest principal to make use of an intercepted certificate of identity, it must succeed in decrypting it. In order to make the task of decryption more difficult, a certificate of identity has a limited lifespan; and, once it expires, the associated login context is no longer valid.

Because this security feature may inconvenience users, an application may wish to warn a user when the certificate of identity is about to expire. The **sec_login_get_expiration()** routine returns the expiration date of a certificate of identity. When a certificate of identity is about to expire, the application may call **sec_login_refresh_identity()**, which may be used to refresh any login context.

Similarly, a server principal may need to determine whether a certificate of identity may expire during some long network operation and, if the certificate of identity is likely to expire, refresh it to ensure that the operation is not prevented from completion. Following is an example:

```
sec_login_get_expiration (login_context,&expire_time,&st);

if (expire_time < (current_time + operation_duration))
{
```

```
   if (sec_login_refresh_identity(login_context,&st))
   {
      ...identity has changed and must be validated again...
   }
   else
   {
      ...login context cannot be renewed...

      exit(0);
   }
}

operation();
```

Because **sec_login_refresh_identity()** acquires a certificate of identity, refreshed contexts must be revalidated with **sec_login_validate_identity()** or **sec_login_valid_and_cert_ident()** before they can be used.

The expiration date of a login context has no meaning with respect to local identity information; for the same reason, **sec_login_refresh_identity()** cannot refresh a login context that has been authenticated locally.

30.4 Importing and Exporting Contexts

Under some circumstances, an application may need two processes to run using the same login context. A process may acquire its login context in a form suitable for imparting to another process by calling **sec_login_export_context()**. This call collects the login context from the local context cache and loads it into a buffer. Another process may then call **sec_login_import_context()** to unpack the buffer and create its own login context cache to store the imported context. Since the context has already been validated, the process that imports it may use it immediately. (The CDS clerk is an example of a context importer.)

These operations are strictly local; that is, the exporting and importing processes must be running on the same host. In addition, a process cannot export a private context.

30.5 Changing a Groupset

The **sec_login_newgroups()** routine enables a principal to assume the minimum groupset that is required to accomplish a given task. For example, a user may have privilege attributes that include membership in an administrative group associated with a comprehensive permission set, and membership in a user group associated with a more restricted permission set. Such a user may not want the permissions associated with the administrative group, except when those permissions are essential to an administrative task (so as to avoid inadvertent damage to objects that are accessible to members of the administrative group, but not to members of the user group).

To offer users the capability of removing groups from their groupsets, an application may use the login API as shown in the following example.

Note: Two of the function calls that appear in the following example, **sec_login_get_current_context()** and **sec_login_inquire_net_info()**, are described in the following section.

```
sec_login_get_current_context(&login_context,&st);

sec_login_inquire_net_info(login_context,&net_info,&st);

for (i=0; i < num_groups; i++)
{
   ... query whether user wants to discard any current group
   memberships.  Copy new group set to new_groups array ...
}

if (!sec_login_newgroups(login_context,sec_login_no_flags,
    num_new_groups, new_groups, &restricted_context,&st))
{
   if (st == sec_login_s_groupset_invalid)

      printf("Newgroupsetinvalid\n");

   ...application-specific error handling...
}
```

Note that the **sec_login_newgroups()** call can only return a restricted groupset: it cannot return a groupset larger than the one associated with the login context that is passed to it. This routine also enables the calling process to flag the new login context as private to the calling process.

30.6 Miscellaneous Login API Functions

The following subsections describe a few miscellaneous login API routines, some of which have appeared previously in examples in this chapter.

30.6.1 Getting the Current Context

The **sec_login_get_current_context()** routine returns a handle to the login context for the currently established principal. This routine is useful for several login API functions that take a login context handle as input.

30.6.2 Getting Information from a Login Context

The **sec_login_inquire_net_info()** routine returns a data structure comprising the principal's PAC, account expiration date, password expiration date, and identity expiration date. The **sec_login_free_net_info()** frees the memory allocated to this data structure.

30.6.3 Getting Password and Group Information for Local Process Identities

Two calls, **sec_login_get_pwent()** and **sec_login_get_groups()**, are useful for setting the local identity of a process. These routines return password or group information from the network registry, if that service is available, or from the local files of password and group information, if the network service is unavailable.

30.6.4 Releasing and Purging a Context

When a process is finished using a login context, it may call **sec_login_release_context()** to free storage occupied by the context handle. When a process releases a login context, the context is still available to other processes that use it. If an application needs to destroy a login context, it may call **sec_login_purge_context()**, which also frees storage occupied by the handle. Since a destroyed context is unavailable to all processes that use it, application developers should be careful when using **sec_login_purge_context()**.

Chapter 31

The Key Management API

Every principal has an entry in the registry database that specifies a secret key. In the case of an interactive principal (that is, a user), the secret key is derived from the principal's password. Just as users need to keep their passwords secure by memorizing them (rather than writing them down, for example), a noninteractive principal also needs to be able to store and retrieve its secret key in a secure manner. The key management API provides simple key management functions for noninteractive principals.

While the key management routines themselves are relatively secure, it is up to the application to ensure the security of the file or other device used to store the key. By default, server principals that run on the same computer share a local key file; however, the key management API also allows principals to specify an alternative local file.

When users change their passwords, they are free to forget their old passwords. When a noninteractive principal changes its secret key, however, there may be clients with valid tickets to that principal that are encoded with the old key. To save clients the trouble of having to request new tickets to a noninteractive principal when the principal's key has changed, every key is flagged with a version number, and old key versions are retained until all tickets that could have been encoded with that key have expired.

Finally, if a noninteractive principal's key has been compromised, it may be invalidated (along with all the corresponding tickets held by any clients) by simply deleting it from the local key storage.

Note: The key management API is for use only by applications using the DCE shared-secret authentication protocol and the key-type DES.

31.1 Retrieving a Key

The key management API provides two functions for retrieving a key from the local key storage. The **sec_key_mgmt_get_key()** function returns a specified key version for a specified principal. The meaning of specifying version 0 (zero) in this routine may vary depending on the authentication protocol in effect. (If the protocol is DCE shared-secret, the value 0 for the version identifier means the version that was most recently added to the local storage.) In any case, a principal's login is almost always successful if the principal uses the version 0 key.

When there are valid tickets that are encoded with different key versions, an application may need to retrieve more than one key version. In that case, the application may call **sec_key_mgmt_initialize_cursor()** to set a cursor in the local storage to the first suitable entry corresponding to the named principal and key type, and then call **sec_key_mgmt_get_next_key()** to get all versions of that key in storage. The application may then call **sec_key_mgmt_release_cursor()**, which disposes of information associated with the cursor**.** Neither of the key-retrieval routines can return keys that have been explicitly deleted, or that have been garbage collected after expiring.

The two key-retrieval functions dynamically allocate the memory for the returned key(s). To enable the efficient allocation of memory, an application may call **sec_key_mgmt_free_key()**, which frees the memory occupied by the key and returns it to the allocation pool.

31.2 Changing a Key

The **sec_key_mgmt_change_key()** function communicates with the registry to change the principal's key to a specified string, and also places the new string in the local key

storage. The *keydata* input argument for this call may be a new key that the application specifies or a random key returned by the **sec_key_mgmt_gen_rand_key()** routine. An application may call **sec_key_mgmt_get_next_kvno()** to determine the next key version number that should be assigned to the new key so that it may reference this key version when retrieving a key.

In some circumstances, a principal may need to change its key in the local key storage but not immediately update the registry database. For example, a database application may maintain replicas of a master database that are managed by servers running on different computers. If these servers all provide exactly the same service, it makes sense for them to share the same key (meaning that they share the same principal identity). This way, a user with a ticket to the principal can be directed to whichever server is least busy.

When the registry database obtains a new key for a principal, the authentication service can immediately begin issuing tickets to the principal that are encoded under the new key. However, suppose the master for a single-principal replicated service were to call **sec_key_mgmt_change_key()**, and a client presented a ticket encoded with the latest key to a replica that had not yet learned that key. In this case, the replica would refuse service, even though the ticket was valid. Therefore, if an application employs replicated servers that are also instances of a single principal identity, the application should do the following:

1. Generate a new key by calling **sec_key_mgmt_gen_rand_key()**. This routine simply returns a key to the calling process, without updating the registry or local storage.
2. Disseminate the new key to all replicas.
3. Cause the replicas to call **sec_key_mgmt_set_key()**. This call updates the local storage to the new key but does not update the registry database entry for the principal. (The key version specified in this routine must not be 0 [zero].) The replicas should notify the master when they have completed setting their local stores to the new key.
4. Cause the master to call **sec_key_mgmt_change_key()** (here again, the key version must not be 0) after all replicas have set the new key locally, thereby updating both the master's local storage and the registry database entry.

Of course, if the master and each replica has its own principal identity, each server may call **sec_key_mgmt_change_key()** without coordinating this activity with any others.

31.3 Automatic Key Management

It is sometimes convenient for a principal to be able to change its key on a schedule determined by the password expiration policy for that principal, rather than to rely on a network administrator to decide when this should be done. In this case, the application may call **sec_key_mgmt_manage_key()**. This function invokes **sec_key_mgmt_gen_rand_key()** shortly before the current key is due to expire, updates both the local key storage and the registry database entry with the new key, and then calls **sec_key_mgmt_garbage_collect()** to discard any obsolete keys. This function runs indefinitely; it will never return during normal operation and so should be invoked from a thread dedicated to key management. It is not intended for use by server principals that share the same key.

31.4 Deleting Expired Keys

In order to prevent service interruptions, the key management API does not immediately discard keys that have been replaced; instead, it maintains the keys, with a version number and key-type identifier, in the local key storage. However, after a key has been out of use for longer than the maximum life of a ticket to the principal, it is no longer possible that any client of that principal has a valid ticket encoded with that key. At this time, the key storage may have its garbage collected.

The **sec_key_mgmt_garbage_collect()** routine collects garbage in the local key storage by deleting all keys older than the maximum ticket lifetime for the cell. The *garbage_collect_time* argument, which is returned by **sec_key_mgmt_change_key()**, specifies when key-storage garbage is to be collected.

31.5 Deleting a Compromised Key

When a principal's key has been compromised, it should be deleted as soon as the damage has been discovered in order to prevent another party from masquerading as that principal. Two routines delete a principal's key:

- The **sec_key_mgmt_delete_key()** routine removes all key types having the specified key version identifier from the local key storage, thus invalidating all extant tickets encoded with that key.

- The **sec_key_mgmt_delete_key_type()** routine removes only a specified version of a specified key type.

If the compromised key is the current one, the application should first change the key with **sec_key_mgmt_change_key()**. It is not an error for a process to delete the current key as long as it is done after the login context has been established, but it may inconvenience legitimate clients of a service. The inconvenience may be justified, however, if the application data is sensitive.

Since an application may have no means to discover that its key has been compromised, the **rgy_edit** tool provides interfaces that call **sec_key_mgmt_delete_key()**, **sec_key_mgmt_change_key()**, and **sec_key_mgmt_gen_rand_key()** so that a network administrator, who is more likely to detect that a key has been compromised, may handle a security breach of this kind. As an alternative, the application may provide user interfaces to these routines.

Chapter 32

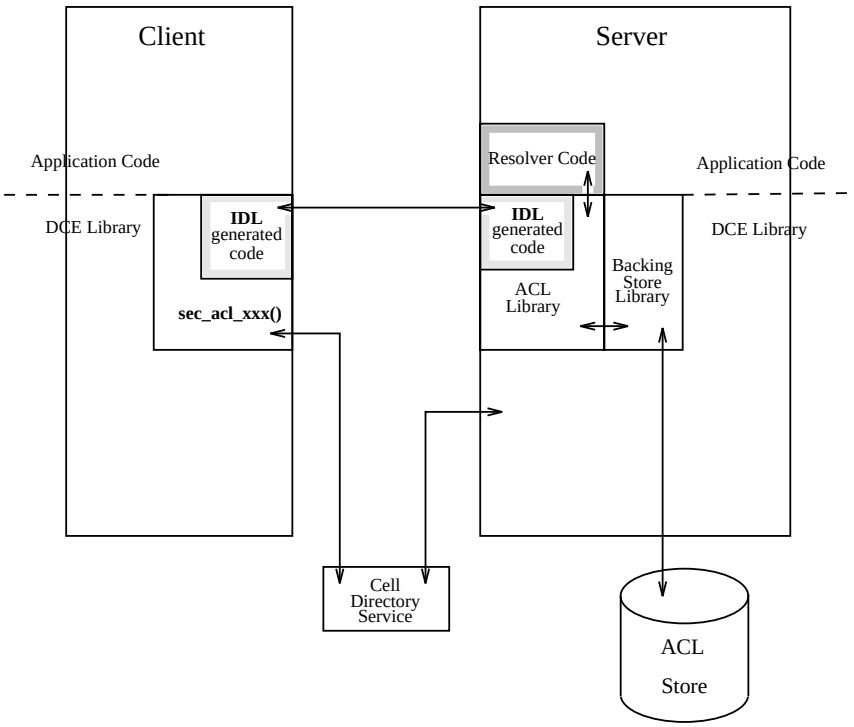
The Access Control List APIs

As a rule, DCE Security Service interfaces are local client-side APIs only. The access control list (ACL) facility includes this kind of interface, and some others as well, as follows:

- The DCE client ACL interface, **sec_acl_* ()**, is a local interface that calls a client-side implementation of the ACL network interface. It enables clients to browse or edit DCE ACLs.
- The DCE server ACL manager library, **dce_acl_* ()**, enables servers to perform DCE-conformant authorization checks at runtime. This ACL library provides an implementation of the ACL manager interface and the ACL network interface. It supports the development of ACL managers for DCE servers.
- The DCE ACL network interface, **rdacl_* ()**, enables servers that manage access control to communicate with **sec_acl**-based clients.

Figure 32-1 provides a schematic view of the relationships and usage of these interfaces, as well as some relevant RPC interfaces. This chapter first discusses the client API, and then the two server program interfaces.

Figure 32–1. ACL Program Interfaces



32.1 The Client-Side API

The client-side API is a local interface consisting of a set of routines that are prefixed **sec_acl**. This is the interface on which the default DCE ACL editor (the DCE control program, or **dcecp**) is built. An application that needs to replace **dcecp** with a DCE ACL editor or browser of its own calls this interface. The following subsections provide specific information on the functionality that this API supports.

32.1.1 Binding to an ACL

Any operation performed on an ACL uses an ACL handle of type **handle_t** to identify the target of the operation. The handle is bound to the server that manages the object protected by the ACL, not to the ACL itself. Since an object may be protected by more than one ACL manager type (see Chapter 25), the ACL itself can only be uniquely identified by the ACL handle in combination with the manager type that manages it. ACL editing calls must also specify the ACL type to be read or otherwise manipulated (the object, default container, or default Object ACL types).

An application calls **sec_acl_bind()** to get an ACL handle. The handle itself is opaque to the calling program, which needs none of the information encoded in it to use the ACL interface. A program can obtain the list of ACL manager types protecting an object and pass this data, along with the ACL type identifier, to another client-side routine. The following two calls perform this function:

- **sec_acl_get_manager_types()** returns a list of UUIDs of the manager types.
- **sec_acl_get_manager_types_semantics()** returns UUIDs of the manager types, and also the POSIX semantics supported by each manager type. The output of this call is used by the **sec_acl_calc_mask()** routine when it calculates a new **mask_obj** mask.

In the absence of CDS, an application may call **sec_acl_bind_to_addr()**; this call binds to a network address rather than a cell namespace entry.

Once an application is finished using an ACL handle, it should call **sec_acl_release_handle()** to dispose of it.

32.1.2 ACL Editors and Browsers

After obtaining a handle to the object in question (and using **sec_acl_get_manager_types()** or **sec_acl_get_manager_types_semantics()** to determine the ACL manager types protecting the object), editors and browsers use the **sec_acl_lookup()** function to return a copy of an object's ACL.

Once an object's ACL is retrieved, the editor can call **sec_acl_get_printstring()** to receive instructions about how to display the permissions of the ACL in a human-readable form. This call returns a symbol or word for each permission (a character

string), and also a bitmask, with a bit (or bits) set to encode the permission. In addition, the print string structure includes a short explanation of each permission.

An ACL cannot be modified in part. To change an ACL, an editor must read the entire ACL (the **sec_acl_t** structure), modify it, and replace it entirely by calling **sec_acl_replace()**. If the ACL manager supports the **mask_obj** mask type, you can use **sec_acl_calc_mask()** to calculate a new **sec_acl_e_type_mask_obj** entry type. This function is supported for POSIX compatibility only, for those applications that use **mask_obj** with its POSIX semantics. Accordingly, **sec_acl_calc_mask()** returns the union of the permissions of all ACL entries *other than***user_obj**, **other_obj**, **unauthenticated** (and the pre-existing **mask_obj**). These correspond approximately to what POSIX calls the ''File Group Class'' of ACL entries, although that designation is not appropriate in the DCE context. In particular, **sec_acl_calc_mask()** works independently of DCE DFS.

Use the **sec_acl_get_manager_types_semantics()** routine to obtain the required POSIX semantics and determine if the manager to which the ACL list will be submitted supports the **sec_acl_e_type_mask_obj** entry type.

An ACL can occupy a substantial amount of memory. The memory management routine, **sec_acl_release()**, frees the memory occupied by an ACL, and returns it to the pool. This is implemented strictly as a local operation.

32.1.3 Errors

Although the ACL API saves errors received from the DCE RPC runtime (or other APIs) in ACL handle data, it returns an error describing the ACL operation that failed as a result of the RPC error. However, if an error occurs and the client needs to know the cause of the ACL operation failure, it may call **sec_acl_get_error_info()**. This routine returns the error code last stored in the handle.

32.2 Guidelines for Constructing ACL Managers

ACL manager names for all of DCE should follow the convention for naming **dcecp** attributes. There is no architectural restriction involved in the guidelines shown here,

merely an attempt at consistency. The DCE control program will accept names outside of this convention, but adherence to it will make usage of ACL managers easier.

The guidelines are as follows:

- Alphabetic characters in names must be lowercase only.
- Names should not contain underscores.
- Names should not contain spaces.
- Names should be no longer than 16 bytes, the defined value of **sec_acl_printstring_len**.
- Names should be similar to object command names supported in **dcecp** whenever possible. For example, the ACL manager name **principal** refers to the object, **/.:/sec/principal** , that contains registry information about principals. Note that **dcecp** allows abbreviations. For example, a user can specify **org** for the ACL manager name **organization**.
- Names must be unique within a component's ACL manager but not necessarily within DCE. For example, the name **xattrschema** can be used for a DCE extended attribute configuration schema ACL object and for a security ERA schema ACL object.
- The help string for an ACL manager must specify the component that owns or manages the objects in question because this information cannot always be derived from the ACL manager name.

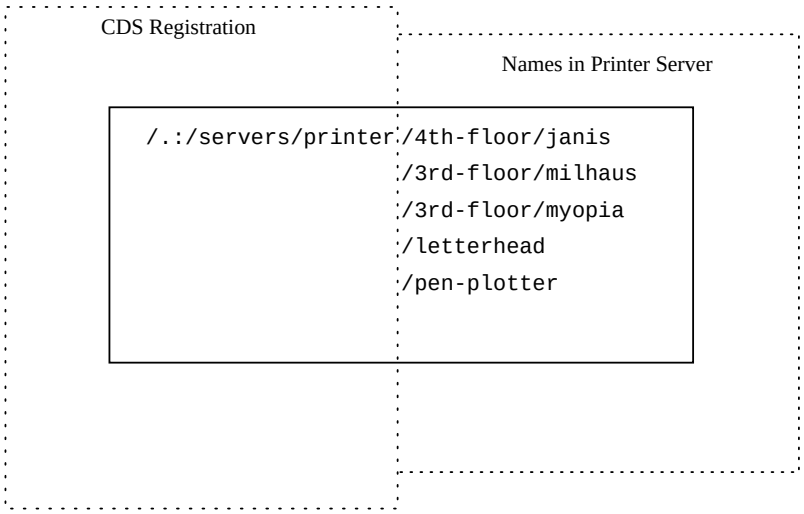
32.3 Extended Naming of Protected Objects

The DCE ACL model supports extended naming so that ACL managers can separately protect objects that are not registered in the cell namespace. This provides an alternative to registering all the server's objects with CDS. The server alone is registered, and it contains code to identify its own objects by name. To achieve ACL protection for these objects, the ACL manager must be able to identify the ACLs in the same way the server identifies the objects. A resolution routine provides this ability.

Figure 32-2 shows the example of a printer server that is registered with CDS, with printers that are not. The ACL manager for the printer server uses the

dce_acl_resolve_by_name() resolution routine to obtain the UUIDs of the several printers that are supported. The administrator in charge of the printers can change the printers, their names, and their ACLs without concern for registering them with CDS.

Figure 32–2. Protection with Extended Naming



When the **dce_acl_register_object_type()** routine registers an object type, it associates a resolution routine with the object type. The ACL library provides two resolution routines: **dce_acl_resolve_by_name()** and **dce_acl_resolve_by_uuid()** . Other resolution routines can be easily written, as required.

To take advantage of extended naming, an ACL manager must register the server name, object UUID, and **rdaclif.idl** interface with the CDS. (Refer to the *DCE 1.2.2 Application Development Guide—Directory Services* for more information). In addition, the ACL manager must register the object UUID and **rdaclif.idl** interface with the RPC endpoint mapper (refer to the chapters concerning RPC in Part 3 of this guide).

32.3.1 The ACL Network Interface

The ACL network interface, **rdacl_*()**, provides a DCE-common interface to ACL managers. It is the interface exported by the default DCE ACL managers to the default DCE ACL client (that is, the **dcecp** tool), and any other client based on the client API.

The client API, **sec_acl_ * ()**, is a local interface that calls a client-side implementation of the ACL network interface. The server side implementation of this interface must conform to the **rdacl_* (3sec)** reference pages. The DCE ACL library provides such an implementation. Following is a summary of the **rdacl_ *()** routines:

- **rdacl_lookup()**

  Retrieves a copy of the object's ACL.

- **rdacl_replace()**

  Replaces the specified ACL.

- **rdacl_get_access()**

  Returns a principal's permissions to an object (useful for implementing operations like the conventional UNIX system access function).

- **rdacl_test_access()**

  Determines whether the calling principal has the requested permission(s).

- **rdacl_test_access_on_behalf()**

  Determines whether the principal represented by the calling principal has the requested permission(s). This function returns TRUE if both the principal and the calling principal acting as its agent have the requested permission(s).

  Note: The **rdacl_test_access_on_behalf()** routine is deprecated and should not be used in new code. Delegation has removed the need for this routine.

- **rdacl_get_manager_types()**

  Returns a list of manager types protecting the object.

- **rdacl_get_printstring()**

  Obtains human-readable representations of permissions.

- **rdacl_get_referral()**

Returns a referral to an ACL update site. This function enables a client that attempts to modify an ACL at a read-only site to recover from the error and rebind to an update site.

32.3.2 The ACL Library

The ACL library provides an implementation of the ACL manager interface and the ACL network interface for the convenience of programmers who are writing ACL managers for DCE servers.

The ACL library meets the following needs:

- It provides stable storage for ACLs.
- It implements the **rdacl_ * ()** interface, including support for multiple object types, initial default Object ACLs, and initial default Container ACLs.
- It implements the full access algorithm, including masks and delegation.
- It provides DCE developers with a set of convenience functions so that servers can easily perform common styles of access control with minimal effort.

32.3.2.1 ACL Library Capabilities

The ACL library provides simple and practical access to the DCE security model.

The library provides a routine that indicates in a single call whether or not a client has the appropriate permissions to perform a particular operation. A server can also easily retrieve the full set of permissions granted to a client by an object's ACL.

The library provides the complete **rdacl_*()** remote interface. Standard routines are provided to map either a UUID attached to a handle or a residual name specified as one of the parameters.

The combination of these capabilities means that most servers will not have any need to use DCE ACL data types directly.

32.3.2.2 The ACL API

The ACL library API, **dce_acl_*()**, is a local interface that provides the server-side implementation of the ACL network interface. The reference pages in *DCE 1.2.2 Application Development Reference* describe the library routines.

The ACL library consists of the following parts:

- Initialization routines, where the server registers each ACL manager type.
- Server queries, where a server can perform various types of access checks.
- ACL object creation, where servers can create ACLs without concern for most low-level data type details.
- The **rdacl_*()** implementation and server callback, where the server maps **rdacl_ * ()** parameters into a specific ACL object. Two sample resolver routines are associated with this part:
  - **dce_acl_resolve_by_name()**

    Finds an ACL's UUID, given an object's name.
  - **dce_acl_resolve_by_uuid()**

    Finds an ACL's UUID, given an object's UUID.

32.3.2.2.1 Initialization Routines

An ACL manager must first define the types of the objects it manages. For example, a simple directory service would have directories and entries, and each type of object would have a different ACL manager. On a practical level, if a server has different types of objects, then the most common difference between the ACL managers is the printed representation of its permission bits. In other words, although the **sec_acl_printstring_t** values differ, the algorithm for evaluating permissions remains the same.

The ACL library provides a global print string that specifies the **read**, **write**, and **control** bits. Application developers are encouraged to use this print string whenever appropriate.

An ACL manager calls the **dce_acl_register_object_type()** routine to register an object type, once for each type of object that the server manages. The manager print string does not define any permission bits; they are set by the library to be the union of all permissions in the ACL print string.

The server must register the **rdacl_*()** interface with the RPC runtime and with the endpoint mapper. See the **dce_server_register(3dce)** reference page.

32.3.2.2.2 Server Queries

The ACL library provides several routines to automate the most common use of DCE ACLs:

- **dce_acl_is_client_authorized()**

  Checks whether a client's credentials are authenticated and, if so, that they grant the desired access.

- **dce_acl_inq_client_permset()**

  Returns the client's permissions, corresponding to an ACL.

- **dce_acl_inq_client_creds()**

  Returns the client's credentials.

- **dce_acl_inq_permset_for_creds()**

  Determines a client's complete extent of access to an object.

- **dce_acl_inq_acl_from_header()**

  Retrieves the UUID of an ACL from the header of an object in the backing store.

- **dce_acl_inq_prin_and_group()**

  Inquires the principal and the group of an RPC caller.

32.3.2.2.3 Creating ACL Objects

The following convenience functions may be used by an application programmer to create ACL objects in other servers or clients.

- **dce_acl_copy_acl()**

Copies an ACL.

- **dce_acl_obj_init()**

  Initializes an ACL for an object.

- **dce_acl_obj_free_entries()**

  Frees space used by an ACL's entries.

- **dce_acl_obj_add_user_entry()**

  Adds permissions for a user ACL entry to the given ACL.

- **dce_acl_obj_add_group_entry()**

  Adds permissions for a group ACL entry to the given ACL.

- **dce_acl_obj_add_id_entry()**

  Adds permissions for an ACL entry to the given ACL.

- **dce_acl_obj_add_unauth_entry()**

  Adds permissions for an **unauthenticated** ACL entry to the given ACL.

- **dce_acl_obj_add_obj_entry()**

  Adds permissions for an **obj** ACL entry to the given ACL.

- **dce_acl_obj_add_foreign_entry()**

  Adds permissions for the ACL entry for a foreign user or group to the given ACL.

- **dce_acl_obj_add_any_other_entry()**

  Adds permissions for the **any_other** ACL entry to a given ACL.

32.3.2.2.4 RDACL Implementation and Server Callback

The ACL library makes a complete implementation of the **rdacl_ *()** interface available to programmers writing servers, in a manner that is mostly transparent to the rest of the server code.

The operations in the **rdacl_ * ()** interface share an initial set of parameters that specify the ACL object being operated upon:

```
handle_t                        h
sec_acl_component_name_t        component_name
uuid_t                          *manager_type
sec_acl_type_t                  sec_acl_type
```

The *sec_acl_type* parameter indicates whether a protection ACL, an initial default Object ACL, or an initial default Container ACL is desired. It does not appear in the **access** operations as it must have the value **sec_acl_type_object**.

In order to implement the **rdacl_*()** interface, the server must provide a **resolution** routine that maps these parameters into the UUID of the desired ACL object; the library includes two such routines: **dce_acl_resolve_by_uuid()** and **dce_acl_resolve_by_name()**.

The resolution routine is required because servers use the namespace in different ways. Here are three examples:

- Servers that export only their binding information and manage a single object, and hence use a single ACL, do not need the resolution parameters. DTS is an example of this case.
- Servers with many objects in the namespace, with a UUID in each entry, will call **rpc_binding_inq_object** on the handle to obtain the object UUID. They then use this same UUID as the index of the ACL object. Many application servers will be of this type. One ACL library resolver function, **dce_acl_resolve_by_uuid()**, matches this paradigm. This paradigm is not appropriate if the number of objects is immense.
- Servers with many objects will use a junction or similar architecture so that the component name (also called the **residual**) specifies the ACL object by name. The DCE security server is essentially of this type. Another ACL library resolver function, **dce_acl_resolve_by_name()**, matches this paradigm.

The following **typedef** specifies the signature for a resolution routine. The first four parameters are the common **rdacl_ *()** parameters mentioned previously.

```
typedef void (*dce_acl_resolve_func_t)(
/* [in] parameters */
    handle_t                    h,
    sec_acl_component_name_t    component_name,
```

```
    sec_acl_type_t              sec_acl_type,
    uuid_t                      *manager_type,
    boolean32                   writing,
    void                        *resolver_arg
/* [out] parameters */
    uuid_t                      *acl_uuid,
    error_status_t              *st
);
```

For situations in which neither of the ACL library resolver functions, **dce_acl_resolve_by_uuid()** or **dce_acl_resolve_by_name()**, is appropriate, application developers must provide their own.

The following two examples illustrate the general structure of the **dce_acl_resolve_by_uuid()** API and **dce_acl_resolve_by_name()** API that are supplied in the ACL library. They may be used as paradigms for creating additional resolver routines.

The first example shows **dce_acl_resolve_by_name()**.

A server has several objects and stores each in a backing store database. Part of the standard header for each object is a structure that contains the UUID of the ACL for that object. (The standard header is not intended to be an abstract type, but rather a common prolog provided to ease server development.) The resolution routine for this server retrieves the object UUID from the handle, uses that as an index into its own backing store, and uses the *sec_acl_type* parameter to retrieve the appropriate ACL UUID from the standard data header.

This routine needs the database handle for the server's object storage, which is specified as the *resolver_arg* parameter in the **dce_acl_register_object_type()** call.

```
#define STAT_CHECK_RET(st) { if (st != error_status_ok) return; }
dce_acl_resolve_func_t
dce_acl_resolve_by_uuid(
  /* in */
    handle_t   h,
    sec_acl_component_name_t component_name,
    sec_acl_type_t  sec_acl_type,
    uuid_t   *manager_type,
```

```
    boolean32   writing,
    void   *resolver_arg,
  /* out */
    uuid_t   *acl_uuid,
    error_status_t  *st
)
{
    dce_db_handle_t             db_h;
    dce_db_header_t             dbh;
    uuid_t                      obj;

    /* Get the object. */
    rpc_binding_inq_object(h, &obj, st);
    STAT_CHECK_RET(*st);

    /* Get object header using the object backing store.
     * The handle was passed in as the resolver_arg in the
     * dce_acl_register_object_type call.
     */
    db_h = (dce_db_handle_t)resolver_arg;
    dce_db_std_header_fetch(db_h, &obj, &dbh, st);
    STAT_CHECK_RET(*st);

    /* Get the appropriate ACL based on the ACL type. */
    dce_acl_inq_acl_from_header(dbh, sec_acl_type, acl_uuid, st);
    STAT_CHECK_RET(*st);
}
```

The next example shows **dce_acl_resolve_by_name()**.

A server uses the residual name to resolve an ACL object by using **dce_acl_resolve_by_name()**. This routine requires a DCE database that maps names into ACL UUIDs. This backing store database must be maintained by the server application so that created objects always get a name, and that name must be a key into a database that stores the UUID identifying the object. The *resolver_arg* parameter given in the **dce_acl_register_object_type()** call must be a handle for that database.

```
#define STAT_CHECK_RET(st) { if (st != error_status_ok) return; }
dce_acl_resolve_func_t
dce_acl_resolve_by_name(
  /* in */
    handle_t   h,
    sec_acl_component_name_t component_name,
    sec_acl_type_t  sec_acl_type,
    uuid_t   *manager_type,
    boolean32   writing,
    void   *resolver_arg,
  /* out */
    uuid_t   *acl_uuid,
    error_status_t  *st
)
{
    dce_db_handle_t             db_h;
    dce_db_header_t             dbh;

    /* Get object header using the object backing store.
     * The handle was passed in as the resolver_arg in the
     * dce_acl_register_object_type call.
     */
    db_h = (dce_db_handle_t)resolver_arg;
    dce_db_std_header_fetch(db_h, component_name, &dbh, st);
    STAT_CHECK_RET(*st);

    /* Get the appropriate ACL based on the ACL type. */
    dce_acl_inq_acl_from_header(dbh, sec_acl_type, acl_uuid, st);
    STAT_CHECK_RET(*st);

}
```

Chapter 33

The ID Map API

In the multicell environment, the global print string representation of a principal identity can be ambiguous, even though every principal and its native cell have unique names in the form of UUIDs to which the print string representations normally resolve. For example, all ACLs maintain UUIDs as the definitive representations of principal and cell names. The **acl_edit** tool, on the other hand, takes as input (and also outputs) this same information as print strings. This string-to-UUID mapping is accomplished easily enough when an ACL entry refers to a local identity; that is, a member of the local cell. However, when a user adds an ACL entry for a foreign principal identity such as **/.../world/dce/rd/writers/tom**, it is not evident to the ACL manager which part of the name identifies the cell, and which identifies the principal within the cell. The name **/.../world/dce** may refer to a cell containing the principal **/rd/writers/tom**, or the cell name may be **/.../world/dce/rd** and the principal name **/writers/tom**.

To parse the fully qualified principal name that the user types into its cell name and local principal-name components, and for these components to be mapped to UUIDs, ACL managers that support entries for foreign identities use the ID map API. For the same reasons, many other kinds of servers in a DCE multicell environment need a facility to parse global names and translate UUIDs into print string names.

The ID map API provides a simple interface to translate a fully qualified name (that is, the global representation of a name) into its components and back again. This API consists of the following calls:

- The **sec_id_parse_name()** call takes as input a registry context handle and a fully qualified principal name, and returns the principal's print string name and UUID, and the print string name and UUID of the principal's native cell.
- The **sec_id_gen_name()** call translates a principal UUID and the UUID of its native cell UUID into a cell-relative principal name, a cell name, and a fully qualified principal name.
- The **sec_id_parse_group()** call is like **sec_id_parse_name()**, except that it operates on group names.
- The **sec_id_gen_group()** call is like **sec_id_gen_name()**, except that it operates on group names.

Chapter 34

DCE Audit Service

Audit plays a critical role in distributed systems. Adequate audit facilities are necessary for detecting and recording critical events in distributed applications.

Audit, a key component of DCE, is provided by the DCE Audit Service.

This chapter provides an introduction to the DCE Audit Service.

34.1 Features of the DCE Audit Service

The DCE Audit Service has the following features:

- An audit daemon performs the logging of audit records based on specified criteria.
- Application programming interfaces (APIs) can be used as part of application server programs to record audit events. These APIs can also be used to create tools that analyze the audit records.
- An administrative command interface to the audit daemon directs the daemon in selecting the events that are going to be recorded based on certain criteria.

- An event classification mechanism is used to logically group a set of audit events for ease of administration.
- Audit records can be directed to logs or to the console.

34.2 Components of the DCE Audit Service

The DCE Audit Service has three basic components:

- application programming interfaces (APIs)

  Provide the functions that are used to detect and record critical events when the application server services a client. The application programmer uses these functions at *code points* in the application server program to actuate the recording of audit events.

  Other APIs are also provided which can be used to create tools that examine and analyze the audit event records.

- audit daemon

  Maintains the filters and the audit logs.

- audit management interface

  Management interface to the audit daemon. Used by the administrator to specify how the audit daemon will filter the recording of audit events. This interface is available from the DCE control program.

34.3 DCE Audit Service Concepts

This section briefly describes the DCE Audit Service concepts that are relevant to DCE application programming.

34.3.1 Audit Clients

All RPC-based servers, such as DCE servers and user-written application servers, are potential audit clients. The DCE Security Service, DTS, and the DCE Audit Service

itself are auditable. That is, code points (discussed in Section 34.3.2) are already in place on these services.

The audit daemon can also audit itself.

34.3.2 Code Point

A *code point* is a location in the application server program where DCE audit APIs are used. Code points generally correspond to operations or functions offered by the application server for which audit is required. For example, if a bank server offers the cash withdrawal function **acct_withdraw()**, this function may be deemed to be an auditable event and be designated as a code point.

As mentioned previously, code points are already in place in the DCE Security Service, DTS, and DCE Audit Service. Code points and their associated events for the DCE Security Service are documented in the **sec_audit_events(5sec)** reference page. Code points and their associated events for the DTS are documented in the **dts_audit_events(5sec)** reference page. Code points and their associated events for the DCE Audit Service are documented in the **aud_audit_events(5sec)** reference page.

34.3.3 Events

An *audit event* is any event that an audit client wishes to record. Generally, audit events involve the integrity of the system. For example, when a client withdraws cash from his bank account, this can be an audit event.

An audit event is associated with a code point in the application server code.

The terms *audit event*, *event*, and *auditable event* are used interchangeably in this book.

34.3.3.1 Event Names and Event Numbers

Each event has a symbolic name as well as a 32-bit number assigned to it. Symbolic names are used only for documentation in identifying audit events. In creating event classes, the administrator uses the event numbers associated with these events.

Event numbers are 32-bit integers. Each event number is a tuple made up of a *set-id* and the *event-id*. The *set-id* corresponds to a set of event numbers and is assigned by OSF to an organization or vendor. The *event-id* identifies an event within the set of events. The organization or vendor manages the issuance of the event ID numbers to generate an event number.

Event numbers must be consecutive. That is, within a range of event numbers, no gaps in the consecutive order of the numbers are allowed.

The structure and administration of event numbers can be likened to the structure and administration of IP addresses. Recall that an IP address is a tuple of a network ID (analogous to the set-id) and a host ID (analogous to the event-id). The format and administration of event numbers are also analogous to IP addresses, as will be discussed in the next sections.

34.3.3.2 Event Number Formats

Events numbers follow one of five formats (A to E), depending on the number of audit events in the organization. The format of an event number can be determined from its four high-order bits.

Format A can be used by large organizations (such as OSF or major DCE vendors) that need more than 16 bits for the event-id. This format allocates 7 bits to the set-id and 24 bits to the event-id. Format A event numbers with zero (0) as its set-id are assigned to OSF. That is, all event numbers used by OSF have a zero in the most significant byte.

Format B can be used by intermediate-sized organizations that need 8 to 16 bits for the event-id.

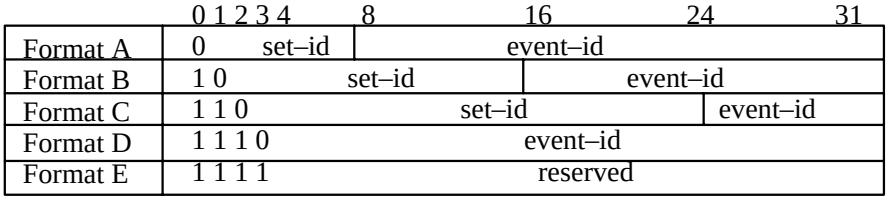
Format C can be used by small organizations that need less than 8 bits for the event-id.

Format D is not administered by OSF and can be used freely within the cell. These event numbers may not be unique across cells and should not be used by application servers that are installed in more than one cell.

Format E is reserved for future use.

The event number formats are illustrated in Figure 34-1.

Figure 34–1. Event Number Formats



34.3.3.3 Sample Event Numbers for DCE Servers

Following are examples of event numbers in the security and time servers, as defined in a header file used 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
...
```

34.3.3.4 Sample Event Numbers for Application Servers

The following is an example of the event numbers in a banking server application, as defined in the application's header file.

```
#define evt_vn_bank_server_acct_open  0x01000000
#define evt_vn_bank_server_acct_close  0x01000001
#define evt_vn_bank_server_acct_withdraw 0x01000002
#define evt_vn_bank_server_acct_deposit  0x01000003
#define evt_vn_bank_server_acct_transfer 0x01000004
```

34.3.3.5 Administration of Event Numbers

Organizations and vendors must administer the event numbers assigned to them (through the set-id) to maintain the unique assignment of event numbers.

34.3.4 Event Class

Audit events can be logically grouped together into an *event class*. Event classes provide an efficient mechanism by which sets of events can be specified by a single value. Generally, an event class consists of audit events with some commonality. For example, in a bank server program, the cash transaction events (deposit, withdrawal, and transfer) may be grouped into an event class.

Typically, the administrator creates and maintains event classes. For more details to event classes, see the *DCE 1.2.2 Administration Guide—Core Components*.

34.3.5 Event Class Number

Each event class is assigned an *event class number*. Like the event number, the event class number is a 32-bit integer and is administered by OSF. Event class numbers are discussed in more detail in the *DCE 1.2.2 Administration Guide—Core Components*.

34.3.6 Filters

Once the code points are identified and placed in the application server, all audit events corresponding to the code points will be logged in the audit trail file, irrespective of the outcome of these audit events. However, recording all audit events under all conditions may neither be practical nor necessary. *Filters* provide a means by which audit records are logged only when certain conditions are satisfied. A filter is composed of *filter guides* that specify these conditions. Filter guides also specify what action to take if the condition (outcome) is met.

A filter answers the following questions:

- Who will be audited?
- What events will be audited?
- What should be the outcome of these events before an audit record is written?
- Will the audit record be logged in the audit trail file or displayed on the system console, or both?

For example, for the bank server program, you can impose the following conditions before an audit record is written:

‘‘Audit all withdrawal transactions (the audit events) that fail because of access denial (outcome of the event) that are performed by all customers in the DCE cell (who to audit).’’

34.3.6.1 Filter Subject Identity

A filter is associated with one *filter subject*, which denotes to what the filter applies. The filter subject is the client of the distributed application who caused the event to happen.

For more information on the filter subject identity, see the *DCE 1.2.2 Administration Guide—Core Components*.

34.3.7 Audit Records

An audit record has a header and a trailer. The header contains the common information of all events; for example, the identities of the client and the server, group privileges used, address, and time. The trailer contains event-specific information; for example, the dollar amount of a fund-transfer event.

Audit records are initialized and filled by calling the audit API functions.

There are four stages in the writing of an audit record:

1. First, the code point registers an audit event. At this point, the audit record does not yet have any form.
2. The audit record descriptor is built. This is a representation of the audit data that is built by the **dce_aud_start()**, **dce_aud_put_ev_info()**, and **dce_aud_commit()** functions. This is stored in a data structure in the client's core memory until the **dce_aud_commit()** function is called. This data is not IDL-encoded until the **dce_aud_commit()** call.
3. The audit record is written to the log. This is stored as IDL-encoded data in the audit log.
4. The audit record is transformed into human-readable form. This is a representation built in a data structure in the core memory by calls to the **dce_aud_next()** and **dce_aud_print()** functions. This is not an IDL-encoded representation.

34.3.8 Audit Trail File

The *audit trail file* contains all the audit records that are written by the audit daemon or the audit APIs. You can specify either a central audit trail file or a local audit trail file. The central audit trail file is maintained by the audit daemon. The local audit trail file is maintained by the audit library. The terms *audit trail file* and *audit trail* are used interchangeably in this book.

34.4 Administration and Programming in DCE Audit

This section gives you an example of how auditing is accomplished using the DCE Audit Service. Both the programmer and the administrator have to perform tasks to enable the writing of audit records in the audit trail. This section looks at the life cycle of an audit trail, from the time that audit events are identified in the server code, to the time that they are filtered and recorded in the audit trail file.

A bank server example illustrates each stage of the life cycle. In this example, the bank server program offers five operations: **acct_open()** , **acct_close()**, **acct_withdraw()** , **acct_deposit()**, and **acct_transfer()** .

34.4.1 Programmer Tasks

The programmer uses the audit APIs to enable auditing in the application server program, as illustrated in the following:

1. The programmer identifies the code points in the bank server program. Because each of the five operations (corresponding to an RPC interface) offered by the bank server is a security-relevant operation, the programmer deems that all these operations are security relevant, and assigns a codepoint to each operation. Each code point corresponds to an audit event.

```
acct_open()         /* first code point */
acct_close()        /* second code point */
acct_withdraw()     /* third code point */
acct_deposit()      /* fourth code point */
```

```
acct_transfer()     /* fifth code point */
```

2. The programmer then assigns an event number to each audit event (corresponding to each code point). For example, the programmer defines these numbers in his header file as follows:

```
/* event number for the 1st code point, acct_open() */
#define evt_vn_bank_server_acct_open      0xC1000000

/* event number for the 2nd code point, acct_close() */
#define evt_vn_bank_server_acct_close     0xC1000001

/* event number for the 3rd code point, acct_withdraw() */
#define evt_vn_bank_server_acct_withdraw  0xC1000002

/* event number for the 4th code point, acct_deposit() */
#define evt_vn_bank_server_acct_deposit    0xC1000003

/* event number for the 5th code point, acct_transfer() */
#define evt_vn_bank_server_acct_transfer   0xC1000004
```

3. The programmer now starts adding audit API functions to the bank server program.

   In the initialization part of the server, the application programmer uses the **dce_aud_open()** API to open an audit trail file for writing the audit records. This function uses the lowest-numbered event as one of its parameters; in this case, **0xC1000000** (**evt_vn_bank_server_acct_open**). Using the lowest-numbered event enhances the performance of the filter search.

```
/* open an audit trail file for writing */
dce_aud_open(aud_c_trl_open_write, description,
               evt_vn_bank_server_acct_open,
               5, &audit_trail, &status);
```

4. The programmer invokes the following DCE audit APIs at each code point:

- The **dce_aud_start()** API, to initialize an audit record. This function assigns the event number to the event represented by the code point. Thus, it uses the event number corresponding to that code point as one of its parameters.
- The **dce_aud_put_ev_info()** API, to add event-specific information to the audit record.
- The **dce_aud_commit()** API, to commit the audit record in the audit trail file.

The use of these three APIs is illustrated in the following example of the bank server program:

```
acct_open()     /* first code point */

/* Uses the event number for acct_open(),
   evt_vn_bank_server_acct_open */
dce_aud_start(evt_vn_bank_server_acct_open,
              binding,options,outcome,&ard, &status);

/* If events need to be logged,
   add trailer info (optional) */
if (ard)
  dce_aud_put_ev_info(ard,info,&status);

/* If events need to be logged,
   add header and trailer info */
if (ard)
  dce_aud_commit(at,ard,options,format,&outcome,&status);

acct_close()   /* second code point */

/* Uses the event number for acct_close(),
 * evt_vn_bank_server_acct_close */
dce_aud_start(evt_vn_bank_server_acct_close,
              binding,options,outcome,&ard, &status);

if (ard) /* If events need to be logged */
  dce_aud_put_ev_info(ard,info,&status);

if (ard) /* If events need to be logged */
```

```
dce_aud_commit(at,ard,options,format,&outcome,&status);
```

5. The programmer uses the **dce_aud_close()** API in the termination routine of the application server. This API closes the audit trail file (and frees up memory) if the applicaton server shuts down.

   The coding of the application program to enable auditing is essentially complete at this point.

34.4.2 Administrator Tasks

The following steps will be performed by the administrator to filter the audit events and control the audit trail file.

1. The administrator obtains the event numbers corresponding to the events represented by the code points in the bank server program from the programmer or from the program's documentation. These events and their assigned event numbers are as follows:

   acct_open() 0xC1000000

   acct_close() 0xC1000001

   acct_withdraw()
   0xC1000002

   acct_deposit()
   0xC1000003

   acct_transfer()
   0xC1000004

1. The administrator decides to create two event classes: the **account_creation_operations** class comprised of **acct_open()** and **acct_close()**, and the **account_balance_operations** class comprised of **acct_withdraw()**, **acct_deposit()**, and **acct_transfer()**.

2. The administrator decides to create two filters: one for all users within the cell (for the cell **/.:/torolabcell**), and the other for all other users.

   The filter for all users within the cell has the following guides:

   a. Audit the events in the event class **account_balance_operations** only, subject to the next condition.

b. Write an audit record only if an operation in that event class failed because of access denial.

c. If the first condition is fulfilled, write the audit record in an audit trail file only.

The filter for all other users has the following filter guides:

a. Audit the events in both event classes, subject to the next condition.

b. Write an audit record if an operation in that event class succeeded or failed.

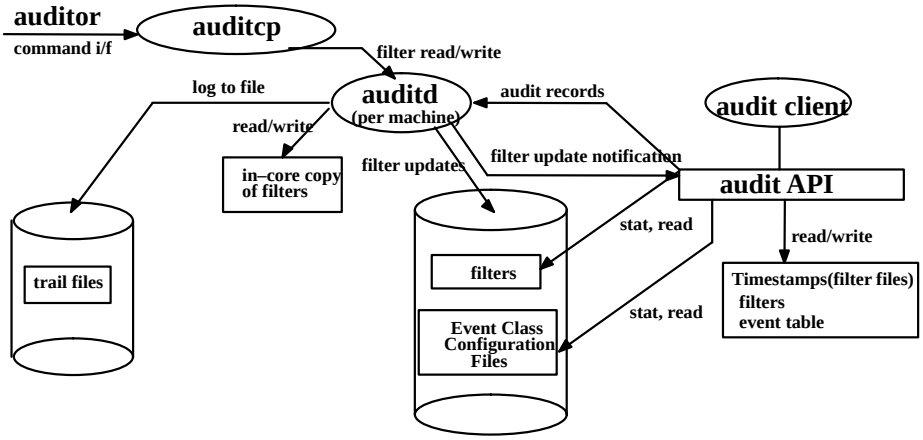
c. Write the audit record both in an audit trail file and the console.

The scenarios described here can be summarized as follows:

- The programmer identifies the code points in the distributed application corresponding to the audit events.
- The programmer uses the audit API functions on those code points to enable auditing.
- The administrator creates event classes that are used to group the audit events.
- The administrator creates filters to narrow down the conditions by which audit records are written for the audit events.

Figure 34-2 illustrates the interactions among the audit client program, the audit API functions (**libaudit**), the audit daemon (**auditd**), and the audit management interface (available from the DCE control program, **dcecp**).

Figure 34–2. Overview of the DCE Audit Service



The audit management interface (accessed through the DCE control program) is used by the systems administrator to specify who, what, when, and how to audit. This is accomplished through the use of the filters. The audit daemon maintains the filter's information in its address space. The filters are also stored in local files so that the filters can be restored when the machine restarts, and so that audit clients can read the filter information from these files.

The audit clients are the users of the filter information. Using the audit APIs, the audit client reads the information on filters and event class configuration. The audit client reads these files only once, unless an update notification is received from the audit daemon (which is triggered by an update initiated by an administrator using the DCE control program).

Chapter 35

Using the Audit API Functions

This chapter describes the use of the audit API functions to add audit capability to distributed applications and to write audit trail analysis and examination tools.

35.1 Adding Audit Capability to Distributed Applications

To record audit events in an audit trail file, the DCE audit API functions must be called in the distributed application to perform the following:

1. Open the audit trail file during the startup of the application.
2. Initialize the audit records at each code point.
3. Add event information to the audit records at each code point. (This is optional.)
4. Commit the audit records at each code point.
5. Close the audit trail file when the application shuts down.

Note that steps 2, 3, and 4 are repeated in sequence at each code point in the distributed applcation.

The use of the audit API functions in each of these steps is illustrated with the bank server example introduced in the previous chapter.

Five code points are identified in the bank server program: **acct_open()**, **acct_close()**, **acct_withdraw()**, **acct_deposit()**, and **acct_transfer()**. Each code point has been assigned an event number and defined in the application server's header file as follows:

```
#define evt_vn_bank_server_acct_open  0x01000000
#define evt_vn_bank_server_acct_close  0x01000001
#define evt_vn_bank_server_acct_withdraw 0x01000002
#define evt_vn_bank_server_acct_deposit  0x01000003
#define evt_vn_bank_server_acct_transfer 0x01000004
```

35.1.1 Opening the Audit Trail

To open the audit trail file, the main routine of the application server uses the **dce_aud_open()** function. With this function call, the audit trail file can be

- opened for reading or for writing.
- directed to the default audit trail file or to a specific file. If **dce_aud_open()** is called without specifying an audit trail file, (by having NULL as the value of the *description* parameter), a default audit trail file is used. This is the *central trail* file that is accessed by RPC calls to the audit daemon.

  If an audit trail file is specified in the **dce_aud_open()** call, (through the *description* parameter), that file is opened directly by the audit library, bypassing RPCs and the audit daemon.

In the bank server application, the function call is as follows:

```
dce_aud_open(aud_c_trl_open_write, &audit_file,
evt_vn_bank_server_acct_open,
5, &audit_trail, &status);
```

In this call, the audit trail file **audit_file** is opened for writing. The third parameter (**evt_vn_bank_server_acct_open**)specifies the lowest event number used in the bank server application. The fourth parameter (**5**) specifies the number of events defined.

The call returns an audit-trail descriptor (**audit_trail**) that will be used to append audit records to the audit trail file.

35.1.2 Initializing the Audit Records

Audit records can be initialized by using the **dce_aud_start_ *()** functions. This function has five variations, and the use of each variation depends on the available information about the server. In general, if you have the RPC binding information about the server, use the **dce_aud_start()** function. If not, use the other four variations of this function, depending on the available information. The five variations are as follows:

- **dce_aud_start()**

  For use by DCE RPC-based server applications.

- **dce_aud_start_with_server_binding()**

  For use by DCE RPC-based client applications.

- **dce_aud_start_with_ pac()**

  For use by applications that do not use DCE RPC, but use the DCE authorization model.

- **dce_aud_start_with_name()**

  For use by applications that use neither DCE RPC nor the DCE authorization model.

- **dce_aud_start_with_uuid()**

  For use by RPC-based applications that know their client's identity in UUID form.

The **dce_aud_start_*()** functions determine if a specified event must be audited based on the subject identity and event outcome that were defined for that event by the filters.

If the event specifics match the event filters (that is, the event has to be audited), these functions return a pointer to an audit record buffer. If it is determined that the event does not need to be audited, a NULL pointer is returned, and the application can then

discontinue any auditing activity. If it cannot be determined whether the event needs to be audited (because the event needs to be audited based on a specific outcome(s) but the outcome is not yet known) these functions return a non-NULL pointer.

When an audit record is initialized, the identification of the audit subject (that is, the client of the distributed application) is recorded.

You can use the **dce_aud_start_*()** functions to specify the amount of header information in the audit record. You can specify any or a combination of the following:

- Information on all groups and addresses
- Information on groups only
- Information on addresses only.

Using these functions, you can bypass the filter altogether and log the event to the audit trail file or display it on the system console. This option is useful for applications whose events require unconditional audit actions.

In our example, each of the bank server routines (**acct_open()** , **acct_close()**, **acct_withdraw()** , **acct_deposit()**, **acct_transfer()**)will make a **dce_aud_start()** function call. In the **acct_transfer()** routine, the function call is made as follows:

```
acct_transfer()
dce_aud_start (evt_vn_bank_server_acct_transfer,
h, aud_c_evt_all_info,
aud_c_esl_cond_success, &ard, &status);
```

where **h** points to the RPC binding of the client making the call. The **aud_c_evt_all_info** option means that all information about the client's groups and addresses are included in the audit record header. The **aud_c_esl_cond_success** event outcome means that the event completed successfully.

35.1.3 Adding Event-Specific Information

If the **dce_aud_start()** function returns an audit record descriptor to the audit record buffer (meaning that the event needs to be audited), the **dce_aud_put_ev_info()**

function call can be used to add event-specific information to the tail of the audit record.

You can opt not to use the **dce_aud_put_ev_info()** function if the information provided by the audit record header is already sufficient for your auditing purposes.

If you elect to use this function, it can be called one or more times, the order of which is preserved in the audit record.

The **dce_aud_put_ev_info()** function has two parameters: the *ard* parameter, which is the pointer to the audit record descriptor, and the *info* parameter, which is a **dce_aud_ev_info_t** type data containing the event-specific information. The programmer can specify the **dce_aud_ev_info_t** data type to include all the audit information that needs to be collected. For more information on the formats of the audit record, see the *DCE 1.2.2 Application Development Reference*.

In the **acct_transfer()** code point of the bank server example, if you want to record the account numbers of the parties involved in the transfer and the amount of each transaction, the data type declarations and the function calls can be made as follows:

```
dce_aud_ev_info_t info;
/* account numbers and transfer amounts are all unsigned
32-bit integers */
info.format = aud_c_evt_info_ulong_int;
info.data = acct_from;
dce_aud_put_ev_info(ard, info, &status);
info.data = acct_to;
dce_aud_put_ev_info(ard, info, &status);
info.data = amount;
dce_aud_put_ev_info(ard, info, &status);
```

35.1.4 Committing an Audit Record

After the header and the optional tail information has been included in the audit record, the **dce_aud_commit()** function call is used to write the audit record in the audit trail file. This function uses the audit trail file previously opened by the **dce_aud_open()** function.

You can specify one of two options in the way the function writes the audit record in the audit trail file:

- Return an error status if the storage or logging service is not available when an attempt is made to write the audit record. This option can be used if the application program can handle write failures in the stable storage.
- If the storage or logging service is not available, keep on trying until the function is able to write to it. This option can be used if the audit record must be written to stable storage before the routine can proceed safely to another task.

In the bank server example, the function call can be made as follows:

dce_aud_commit(audit_trail, ard, *options***,** *format***,** *outcome***, &status);**

The **audit_trail** parameter is the trail descriptor returned in the **dce_aud_open()** call made earlier. The **ard** parameter is the audit record descriptor returned in the **dce_aud_start()** call (and used in the **dce_aud_put_ev_info()** function call). The *format* parameter specifies a format version number of the event-specific information. The initial version number should be zero, and be incremented when the format changes. For example, the data type used for account numbers might change from 32-bit integer to UUID. The event outcome must be provided in this call, even if it has been provided in the **dce_aud_start()** call made earlier. If the event outcome (except **aud_c_esl_cond_unknown**)is provided in both calls, the values must be the same.

35.1.5 Closing an Audit Trail File

The audit trail file must be closed using the **dce_aud_close()** function when the application shuts down (because of the **rpc_mgmt_stop_server_listening()** function call or other exceptional conditions). For example, to close the trail, the bank server's main program can make the following function call:

```
dce_aud_close(audit_trail, &status);
```

This function flushes buffered audit records to stable storage and releases the memory allocated for the trail descriptor.

35.2 Writing Audit Trail Analysis and Examination Tools

The audit APIs can be used to write audit trail analysis and examination tools that selectively review the following:

- Events that are invoked by one or more subjects, for example, principals, groups, and cells
- Events that have a specific outcome
- Events that occurred during a specified time period
- Events that have specific event IDs

In its most basic form, an audit trail analysis and examination tool must perform five functions:

- Open an audit trail file for reading
- Read the audit records into a buffer
- Transform the audit records into human-readable form
- Discard the audit record
- Close the audit trail file

These functions and the APIs that are used for each are discussed in the following sections.

35.2.1 Opening an Audit Trail File for Reading

To open the audit trail file for reading, use the **dce_aud_open()** function and specify **aud_c_trl_open_read** as the value for the *flags* parameter. In this case, the values for the *first_evt_number* and *num_of_evts* does not affect the call. For example:

```
dce_aud_open(aud_c_trl_open_read, AUDIT_TRAIL_FILE,
0, 0, &out_trail, status);
```

35.2.2 Reading the Desired Audit Records into a Buffer

After opening the audit trail file, you can use the **dce_aud_next()** function to retrieve audit records. Audit records are stored in the audit trail file in binary form. The **dce_aud_next()** function does not convert the file into readable form. You must use the **dce_aud_print()** function to translate the audit record into readable form.

The **dce_aud_next()** function allows you to specify a criteria that will be used in selecting the records that will be read from the file. This criteria is known as *predicates* and is expressed by setting the condition on the value of certain attributes. The condition is set by using any of the following operators: = (equal to), > (greater than), and < (less than).

Predicates can be expressed in any of the following forms:

- *attribute= value*
- *attribute> value*
- *attribute< value*

The following list summarizes these attributes and their acceptable values:

SERVER UUID of the principal that generated the record

EVENT Audit event number

OUTCOME
Event outcome of the record

STATUS Authorization status of the application client

CLIENT UUID of the client principal

TIME Time when the record was generated.

CELL The UUID of the application client's cell

GROUP The UUID of the application client's group or groups

ADDR The address (binding handle) of the client

FORMAT The format version number of the audit event record

Details of these attributes, their values, and the allowable operators are discussed in the *DCE 1.2.2 Application Development Reference*.

For example, to have the function retrieve audit records that pertain to the event number 0xC01000001 only, you can set the predicate to the following:

```
EVENT=0xC01000001
```

If the predicate parameter is set to NULL (that is, no criteria), the next audit record is read. For example, to read the next audit record in a previously opened audit trail file, the following call is made:

```
dce_aud_next(out_trail, NULL, &out_ard, status);
```

You can specify multiple predicates, in which case the predicates are treated as a logical AND condition.

The **dce_aud_next()** function returns a pointer to the record that was read. This pointer is used by the **dce_aud_print()**, **dce_aud_get_ev_info()** , and **dce_aud_get_header()** functions in transforming the audit records into ASCII format.

35.2.3 Transforming the Audit Record into Readable Text

After reading in the desired audit record by using the **dce_aud_next()** function, these binary audit records must be transformed into human-readable form.

You can use any of the following three functions to transform the audit record information to human readable form:

- **dce_aud_print()**

  Formats the entire audit record (header and tail) into ASCII format.

- **dce_aud_get_header()**

  Obtains the header information of the audit record and formats it into human readable form.

- **dce_aud_get_ev_info()**

Obtains the event-specific information in the tail of the audit record and formats it into human readable form.

The **dce_aud_next()** function returns the address of the audit record to these functions. These functions then allocate memory for the ASCII-format buffer (using **malloc()**) and fills it with the ASCII representation of the audit record. The user must explicitly release this memory (using **free()**) when all audit record retrieving and transforming tasks have been accomplished.

35.2.4 Discarding the Audit Record

The **dce_aud_discard()** function frees the memory allocated to the binary version of the audit record, that is, the structure returned by the **dce_aud_next()** function. The **dce_aud_discard()** function does not free the structures allocated by **dce_aud_print()**, **dce_aud_get_header()**, or **dce_aud_get_ev_info()**.

35.2.5 Closing the Audit Trail File

Finally, the audit trail file from which the audit records were read must be closed using the **dce_aud_close()** function.

Chapter 36

The Password Management API

User passwords are the weakest link in the chain of DCE security. Users, unless their choices are restricted, typically choose passwords that are easy for them to remember; unfortunately, these memorable passwords are also easy for attackers to ‘‘crack.’’

The password management facility is intended to reduce this risk by providing the tools necessary to develop customized password management servers, and to call them from client password change programs. This facility enables cell administrators to

- Enforce stricter constraints on users’ password choices than those in DCE standard policy
- Offer, or force, automatic generation of user passwords

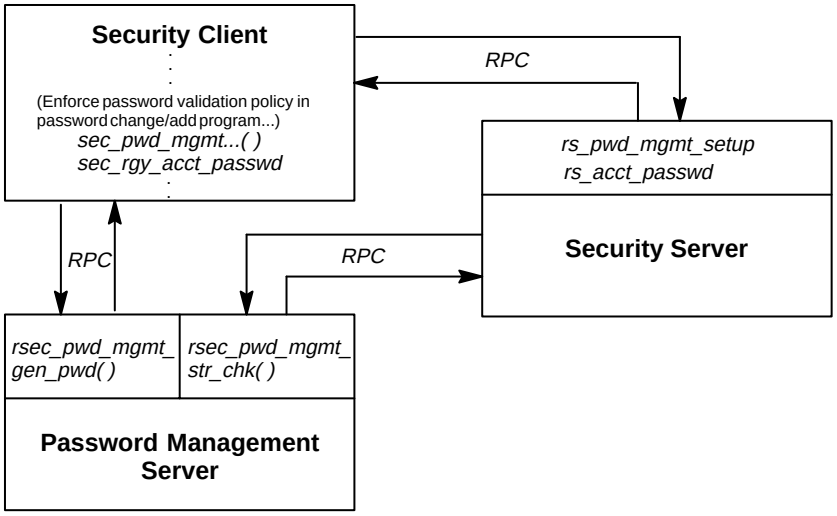
The password management facility includes the following APIs:

- The password management interface, **sec_pwd_mgmt_ *()**, which enables clients to retrieve a principal’s password management ERA values and to request strength-checking and generation of passwords.

- The password management network interface, **rsec_pwd_mgmt_ *()**, which enables a password management server to accept and process password strength checking and generation requests.

Figure 36-1 provides a schematic view of the relationships and usages of these interfaces, as well as some relevant security registry APIs. This chapter first discusses the client API and then the network API.

Figure 36–1. Use of Password Management Facility APIs



For information on how to administer password generation and strength-checking, see the *DCE 1.2.2 Administration Guide—Core Components*.

36.1 The Client-Side API

The DCE control program, **dcecp**, and **rgy_edit** provide support for password generation based on a principal's password validation type ERA. However, if you

want to enhance your own password change program (such as the UNIX **passwd** program), you will need to use the client-side **sec_pwd_mgmt_*()** API.

This API provides functions that retrieve a principal's password management ERA values and request password strength checking and generation from a password management server.

The **sec_pwd_mgmt_*()** API is defined in the **sec_pwd_mgmt.idl** file.

The general procedure for using the client-side password management API in a password change program is as follows. Refer to Figure 36-1 as you read the following steps:

1. The client calls **sec_pwd_mgmt_setup()**, specifying the login name of the principal whose password is being changed. The registry service returns the **pwd_val_type** and **pwd_mgmt_binding** ERAs as well as the registry standard (password) policy for the principal to the client's security runtime, which is stored in a password management handle (an opaque data type).
2. The client calls **sec_pwd_mgmt_get_val_type()**, specifying the handle returned by **sec_pwd_mgmt_setup()** in step 1. The value of the principal's **pwd_val_type** ERA is extracted from the handle and returned to the client.
3. The client analyzes the principal's **pwd_val_type** ERA to determine whether a generated password is required. If so, it calls **sec_pwd_mgmt_gen_pwd()**, specifying the number of passwords needed, and the handle returned by **sec_pwd_mgmt_setup** . The client security runtime makes an RPC call to the password management server, which generates passwords that adhere to the principal's password policy.
4. The client calls **sec_rgy_acct_passwd()** (or some other form), specifying the new password (either input by the user or generated by **sec_pwd_mgmt_gen_pwd()**). If the principal's **pwd_val_type** ERA mandates it, the registry service makes an RPC call to the password management server, specifying the name of the principal and the password to be strength checked. The password management server checks the format of the password according to the user's password policy and accepts or rejects it.
5. The client calls **sec_pwd_mgmt_free_handle()** to free the memory associated with the password management handle.

Following is an example of a password change program that calls the **sec_pwd_mgmt_*()** API as previously described.

```
sec_pwd_mgmt_setup(&pwd_mgmt_h, context, login_name,
    login_context, NULL, &st);
if (GOOD_STATUS(&st)) {
    sec_pwd_mgmt_get_val_type(pwd_mgmt_h, &pwd_val_type, &st);
}
if (GOOD_STATUS(&st)) {
    switch (pwd_val_type) {
        case 0: /* NONE */
        case 1: /* USER_SELECT */
            ... get password ...
            break;
        case 2: /* USER_CAN_SELECT */
            ... if user does not want generated password ... {
                ... get password ...
                break;
            }
        case 3: /* GENERATION_REQUIRED */
            sec_pwd_mgmt_gen_pwd(pwd_mgmt_h, 1, &num_returned,
                &passwd, &st);
            ... display generated password to user - possibly
                prompting for confirmation ...
            break;
    }
}
if (GOOD_STATUS(&st)) {
    sec_rgy_acct_passwd(context, &login_name, &caller_key,
        &passwd, new_keytype, &new_key_version, &st);
}

sec_pwd_mgmt_free_handle(&pwd_mgmt_h, &st);
```

36.2 The Password Management Network Interface

The password management interface, **rsec_pwd_mgmt_* ()**, provides a DCE-common interface to password management servers. It is the interface exported by the sample password management server provided with DCE Version 1.1 (**pwd_strengthd**), and it is the interface that application developers should use to write their own password management servers. Developers should use the sample code provided as a base for enhancements.

The API is defined in the **rsec_pwd_mgmt.idl** file.

Implementations must conform to the **rsec_pwd_mgmt_gen_pwd(3sec)** and **rsec_pwd_mgmt_str_chk(3sec)** reference pages.

The **rsec_pwd_mgmt_*()** routines are

- **rsec_pwd_mgmt_gen_pwd()**

  Generates one or more passwords for a given principal.

- **rsec_pwd_mgmt_str_chk()**

  Strength checks a principal's password according to policy.

Chapter 37

The DCE Certification Service

The DCE certification service provides for the secure retrieval of public keys, stored (through the DCE directory service) under the names of the principals with which the keys are associated. It is a name-to-public key translation service intended to be used both by DCE components and DCE applications. The keys are stored in data structures called ‘‘certificates’’.

Rules that define which entities are trusted to create certificates for which principals are embodied in policy modules, which have the job of retrieving, upon request, the public keys from the certificates (and verifying the certificates themselves when doing so).

DCE certification is a ‘‘secondary’’ facility, in that the service it provides is useful only in the context of some other application activity. Essentially, it does nothing but return public keys when presented with principal names (provided that the public keys have been properly stored under the names in the first place). It is then up to the application to do something useful with the keys.

This chapter is not intended to provide detailed guidance on how DCE applications should use public keys, although some discussion of public key usage is included. It

is mainly concerned with explaining how DCE applications can use the certification service to store and retrieve the keys.

37.1 Who Needs to Use the Certification API?

The DCE certification service is intended to form one part of an implementation of a public key based authentication (and data protection) service in DCE. Thus the first-level users of the certification API will be various components of DCE itself; for example, RPC. However, the certification service can also be (and is intended to be) used by distributed applications that wish to use public keys for their own authentication or data protection purposes. The high-level public key retrieval routines are designed for this kind of use.

The low-level certification routines, on the other hand, are intended for applications that wish to implement and add new policies and/or cryptographic modules. For example, adding a new policy will involve the following development task(s):

- Implementing and registering a policy module (see below)

  For example, a mail application that wished to institute its own model for authenticating users by public key would need to have its own policy module.

- (Optional) Implementing and registering a cryptographic module (see below)

  Cryptographic modules implement the various signature algorithms required to allow policy modules to verify retrieved certificates. Policy modules are generally concerned only with signature verification, and (due to licensing constraints) signature generation functions are not supplied with the cryptographic modules provided with the DCE reference implementation. Applications that wish to use the public keys returned by the DCE certification facility will typically augment the supplied verification functions with signature generation routines.

Other possible users of the DCE certification API might be developers who wish to implement their own signature algorithms (cryptographic modules). (Signature algorithms are specified in a field in the certificate; they are selected at the time a certificate is created.) Only developers who wish to add to the available signature algorithms, or who wish to add signature generation capability to a supplied algorithm, will need to implement new cryptographic modules.

The low-level certification API is not intended to be accessed directly by run-of-the-mill DCE applications.

37.2 Overview of DCE Certification

In the discussion that follows, note that the term ‘‘principal’’ does not necessarily mean or imply ‘‘DCE principal’’. In a general sense, a principal is any name that can be authenticated—that is, any name that has one or more associated key(s). A DCE principal (one that is registered in the DCE registry) has DES key(s) maintained within the registry, while a public key (PK) principal has one or more public keys (generally stored within certificates). The only situation in which a PK principal has to be a DCE principal is where an application is using the ‘‘registry retrieval’’ policy (see ‘‘Direct secd Lookup: DCE Registry Lookup Policy Model’’ below), since this policy retrieves the principal’s public keys from a its registry entry.

The DCE certification service provides for the secure storage and retrieval (by principal name) of public keys. The keys are stored in the DCE directory service, under the principal names with which they are to be associated.

Principals’ public keys are thus easily accessible through the namespace. However, in order to be regarded as valid (certified), the public key information must be properly ‘‘signed’’ by the certifying authority (CA) authorized to deposit public key information for the principal in question. The public key, with the signature of the CA that issued it, is stored (together with various other data) in a format defined by the ISO 9594-8/X.509 standard and called a ‘‘certificate’’. Just who the authorized certifying authority for a given certificate is is defined by the trust policy model applicable to the subject in whose name the certificate is issued.

The CA’s signature is in the form of a checksum on the public key encrypted with the CA’s own private key, and verifiable by decrypting with the CA’s public key. The certificates are thus secure from tampering by any entity but the authorized (according to the defined policy model) CA, which alone possesses the private key required to sign the data.

37.2.1 Use of Public Keys

The DCE certification service stores and retrieves ‘‘public’’ keys. The important characteristic of such keys is that they exist and operate as pairs. Messages encrypted under one of the keys can be decrypted by means of the other (and vice versa); but messages cannot be encrypted and decrypted by means of the same key.

Figure 37–1. How Public Keys Work: Part 1

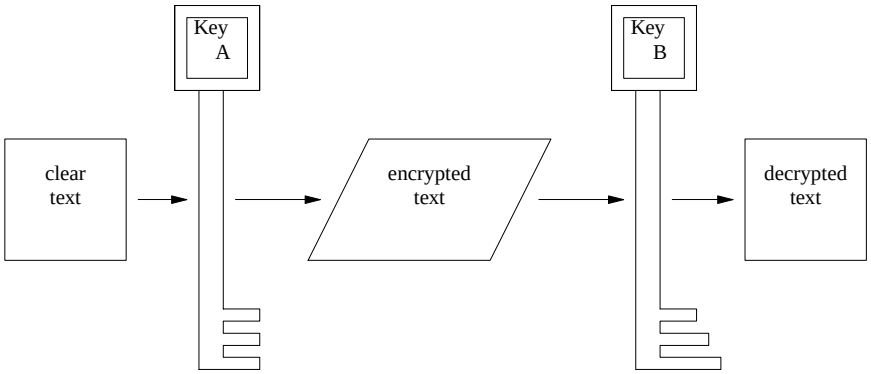
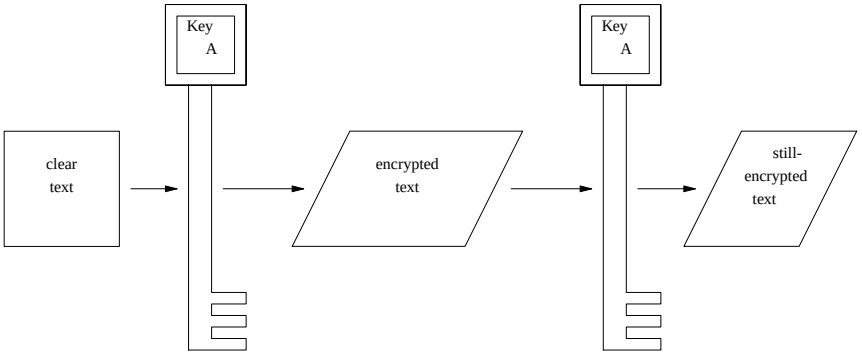


Figure 37–2. How Public Keys Work: Part 2



This asymmetric behavior of the public key-pair makes it ideal for network authentication purposes. One of the keys can be freely publicized in the network, and the other kept secret with, say, a server principal who desires to use it to authenticate itself. The server does this, whenever it is contacted by a prospective client, by simply

encrypting a message under its secret key and sending it back to the client. The client then attempts to decrypt the message using the public key it knows belongs to the principal it wishes to contact. If it can decrypt the message, it regards the server as having authenticated itself. The same procedure can be used to authenticate the client (using a different key-pair).

An important detail in the above scenario is that the prospective client ''knows'' that the public key it uses to decrypt the server's message really is that server's public key. The other (unmentioned) detail is that the client has to get the public key from somewhere. The secure distribution of the public keys is the job of the DCE certification service.

It is not desirable to have all the public keys indiscriminately accessible to everybody, because then no one will have a reliable criterion for believing whose key is whose. The public keys must be deposited in such a way that users can always be sure that a given public key really ''belongs'' to the principal it is supposed to belong to; otherwise entities will be able to impersonate each other simply by switching public keys in the database they are retrieved from. Thus there needs to be some way to make sure that only authorized entities have access to principals' public keys.

The certification service ''certifies'' public keys by storing them with ''signatures'' generated by the distributors of the keys. The authenticating signatures on certificates are themselves implemented by public/private key pairs. A signature is simply the data of which the certificate consists, encrypted under the issuer's private key. A potential user of the certificate must possess the certificate issuer's public key. The issuer's public key can then be passed (along with the certificate contents, including the signature) to a library routine that will check whether the signature can be successfully decrypted to produce the information in the rest of the certificate. If the signature thus tested is found to be authentic, the user of the certificate can be certain that it was issued by the entity whose public key it checked against the certificate signature — namely, the principal that is supposed to have issued the certificate. The public key signature thus ensures the authenticity of data that can be distributed (and thus easily accessed) via the namespace.

A principal's public key can also be used by entities to protect data being sent to the principal. Data encrypted under the public key can be decrypted only by the possessor of the private key.

37.2.2 Contents of Certificates

The primary information that any certificate contains is the public key that is to be associated with some principal name. ‘‘Issuance’’ of a certificate means that the certificate is deposited into the name service, and attached (as a directory attribute) to the principal name it is to be associated with. Certificates are issued by certifying authorities (CAs); the CA’s signature on the certificate is what certifies the public key information that the certificate contains.

A certificate contains the following information:

subject name
: The name of the principal for whom the certificate was issued. This is the name under which the certificate contents will be read by users.

issuer name
: The principal name of the issuer of the certificate, a CA (certifying authority) authorized to issue certificates for the subject.

version number
: Identifies the X.509 format version of the certificate.

serial number
: The certificate serial number, used to identify certificates in certificate revocation lists (CRLs).

start time
: The time from which the certificate’s contents are considered to be valid.

end time
: The time until which the certificate’s contents are valid.

signature algorithm
: An OID (object identifier) that identifies the algorithm used to encrypt the certificate signature.

parameters
: Any parameters necessary to pass to the signature verification algorithm.

signature
: A checksum of the certificate data, encrypted under the certificate issuer’s private key, successful verification of which, by means of the issuer’s public key, constitutes authentication of the certificate.

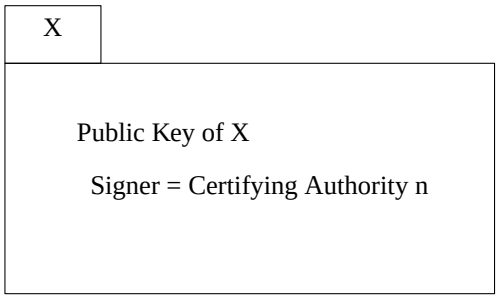
subject key
: The public key that is to be associated with the subject of the certificate (named by ‘‘subject name’’).

subject UUID
: (Optional) A UUID that identifies the certificate subject.

issuer UUID
: (Optional) A UUID that identifies the issuer of the certificate.

The most important ingredients of a certificate are: the principal name which it is stored under; the public key which it contains; and the signature of the CA that issued it. These can be illustrated as shown in the following diagram:

Figure 37–3. The Essential Parts of a Certificate



37.2.3 Component Parts of the DCE Certification API

The DCE certification API is organized into four groups of routines:

- Routines for implementing and registering cryptographic modules

  Cryptographic modules embody the signature algorithms that are used to sign and verify certificates. Certificates are signed by certifying authorities (which are usually invoked by system administrators or some other specially privileged authority to create certificates), and are retrieved (and verified) by policy modules (which are called by various applications seeking principals’ public keys).

- Low-level certificate access and manipulation routines

These routines represent the primitive certificate access operations which are used in the implementation of policy modules.

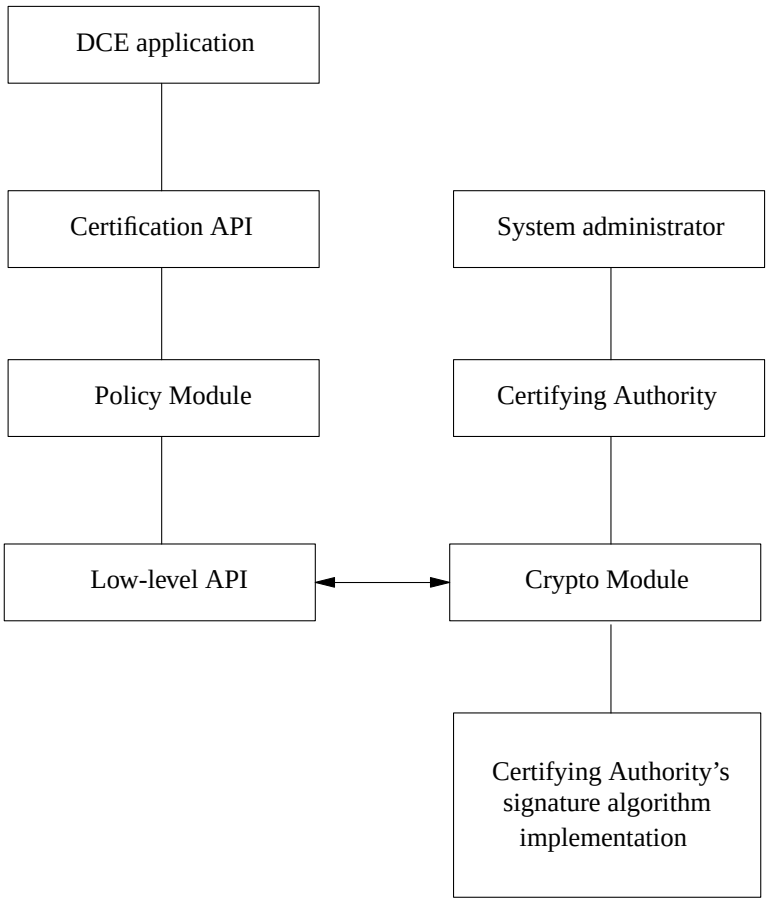
- Routines for implementing and registering policy modules

  Policy modules embody the rules and mechanisms for finding the public keys that are associated with some specific set of principals.

  High-level routines for use by applications that wish to access the certification service

The following diagram shows how these four groups of functionality are related to each other and to their two main groups of user: namely, system administrators and DCE applications.

Figure 37–4. Certification API Organization



Note that certifying authorities merely create the certificates and deposit them in a place from which they can be retrieved (the namespace); they play no part in the retrieval process itself. In fact, this could be said to be the main reason for certificates in the first place: they allow a facility such as the directory service to be used as the distribution point for public keys (that is, they allow an application to not have to arrange for getting its keys to prospective clients by some private mechanism), and at the same time they assure users that the key information that they contain has not been tampered with.

37.2.4 High Level Certification API

The following certification API routines are intended for general DCE application use:

- **pkc_get_registered_policies(3sec)**
- **pkc_init_trustlist(3sec)**
- **pkc_append_to_trustlist(3sec)**
- **pkc_init_trustbase(3sec)**
- **pkc_retrieve_keyinfo(3sec)**
- **pkc_get_key_count(3sec)**
- **pkc_get_key_data(3sec)**
- **pkc_get_key_trust_info(3sec)**
- **pkc_get_key_certifier_count(3sec)**
- **pkc_get_key_certifier_info(3sec)**
- **pkc_free_trustlist(3sec)**
- **pkc_free_trustbase(3sec)**
- **pkc_free_keyinfo(3sec)**
- **pkc_free(3sec)**

Key retrieval consists basically of two operations:

1. Generating an ‘‘initial trust base’’—a starting point for future certification paths, consisting of a list of principals and their keys. An application would normally generate its initial trust base on startup.
2. Using the trust base to retrieve key(s) for a specified principal.

In outline, a typical pattern for an application’s use of the high-level API might proceed according to the following series of calls:

1. **pkc_get_registered_policies(3sec)**

   Called once for the lifetime of the application. It returns a set of OIDs, which point to all currently installed policies.

2. **pkc_init_trustlist(3sec)**

The caller creates an empty ‘‘trust list’’ to hold the set of certificates it initially trusts.

3. **pkc_append_to_trustlist(3sec)**

   Called one or more times, to add certificates or keys which the caller trusts to its list of trusted keys.

   Steps 2 and 3 together build up the initial trust list.

4. **pkc_init_trustbase(3sec)**

   Computes a trust base, given the initial trust list. The caller uses one of the OIDs returned in Step 1, together with the list of trust items constructed in Steps 2 and 3, to access a policy and initialize a ‘‘trust base’’ containing all the certificates initially trusted under the specified policy, given the initial list of trusted keys.

5. **pkc_retrieve_keylist(3sec)**

   Called one or more times, for each individual’s public key that needs to be looked up.

6. **pkc_free_trustlist(3sec)**

   Frees storage allocated for the trust list.

7. **pkc_free_trustbase(3sec)**

   Frees storage allocated for the trust base.

37.2.5 Policy Models

A policy model (or trust policy model) is simply some scheme or set of rules that dictates which certifying authorities are authorized to issue certificates for which principals. In other words, the policy model will prescribe whose signature is to be regarded as a valid certifier for any given principal’s certificates. The policy module which embodies these rules will use them in verifying the certificates it reads from the namespace.

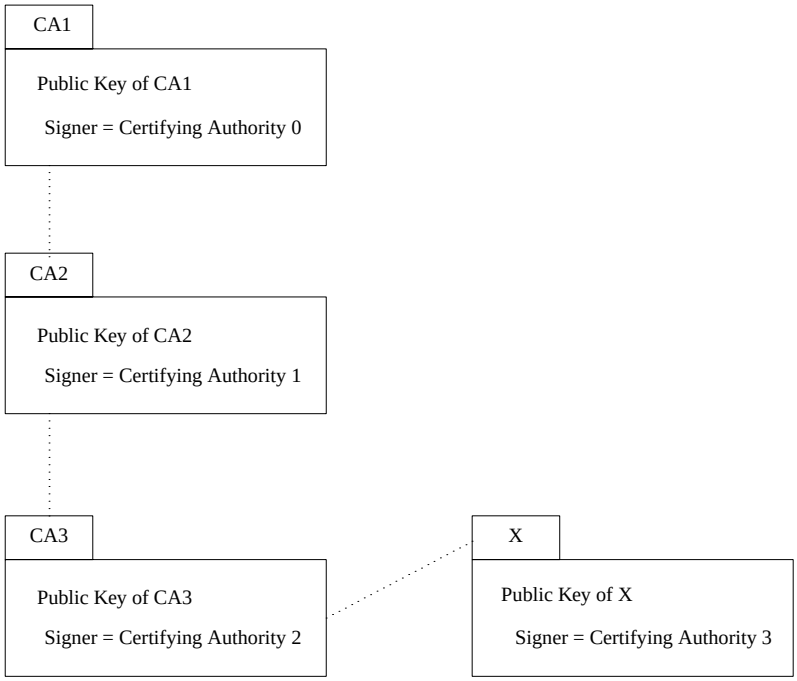
Since the certificates themselves are stored and accessed through the DCE directory service (either GDS or CDS), one obvious policy model will be to organize the certifying authorities’ reponsibilities according to the same hierarchies. However, models that employ other certifying hierarchies, or no hierarchy at all, are also possible.

37.2.5.1 Certification Paths

The mechanism that the certification service uses to embody more complex models is certification paths. A certification path is implemented by a sequence of certificates. Rather than immediately accessing a given principal's certificate to determine its public key, the user must access the beginning of a chain of certificates in order to get to the final certificate that contains the desired principal's public key. The intervening certificates consist of a series of public keys of CAs, each certified by the next CA in the chain.

The following diagram shows how a certificate chain might be used to find the public key of a principal, *X*:

Figure 37–5. A Certificate Chain



In a policy model that uses certification paths, a given principal's public key is found by beginning with a certificate signed by a CA that is trusted by the entity requesting

the public key (in the above diagram, the trusted CA is CA0). This certificate contains the public key of the next CA in the path, namely CA1. The policy module reads this certificate, learns the key of the next CA, CA2, and so on, until the certificate for X, the original target, is found.

The idea of certificate chains is to propagate authenticity via certifying authorities while not propagating the authorities' responsibility, thus reducing the effects of the compromise of single authorities, wherever they may exist in the hierarchy of authorities.

37.3 Implementing and Registering a Cryptographic Module

The routines in an application's cryptographic module make up the lowest level of functionality in the certification mechanism. Each module consists of a set of (at most) five routines, the most important of which are its **sign()** and **verify()** routines. (Note, however, that the **sign()** routine is not mandatory.)

37.3.1 Contents of a Cryptographic Module

Cryptographic modules are registered in the form of **pkc_signature_algorithm_t** structures, which contain the entry points for the following developer-supplied routines:

open() Opens the module

close() Closes the module

verify() Verifies a certificate signature

sign() Affixes a signature to a certificate

verify() and **sign()** are the routines that will actually call the encryption/decryption functions appropriate to the algorithm.

name() Returns the algorithm name, a character string that can be used in auditing or diagnostic messages.

The **pkc_signature_algorithm_t** structure also contains the following data fields:

- a version number

  Note that the version field of a cryptographic module is not the same thing as the version number of a certificate. A crypto module's version number is the version of the certification API that it is designed for (which in particular specifies the format of the **pkc_signature_algorithm_t** structure used to register the crypto module).

- an object identifier (OID) identifying the signature algorithm

37.3.2 Accessing a Registered Cryptographic Module

Signature algorithms are identified by object identifiers (the character string returned by **name()** is intended for use in diagnostic or auditing messages). Certificates contain a field which identifies by OID the algorithm used to sign that certificate.

Policy implementors are recommended to access cryptographic modules mainly through the following routines, which perform all locking necessary to make the calls thread safe, and also transparently handle any context information that a given cryptographic implementation may need.

- **pkc_crypto_get_registered_algorithms(3sec)**

  Call this routine to get an OID set describing the currently registered algorithm implementations.

- **pkc_crypto_sign(3sec)**

  Call this routine to get data signed.

- **pkc_crypto_verify_signature(3sec)**

  Call this routine to verify signed data.

- **pkc_crypto_generate_keypair(3sec)**

  Call this routine to generate a pair of public/private keys.

Information about a cryptographic module may be obtained by calling **pkc_crypto_lookup_algorithm(3sec)**.

Data can also be signed and verified by looking up the desired algorithm (with **pkc_crypto_lookup_algorithm(3sec)**) and then explicitly calling the module's

(sign)() or **verify()** routine, although in this case the calling application must take care to avoid multi-threading issues, and is also responsible for opening the crypto module prior to use, and closing it afterwards.

A list of the OIDs of all currently registered cryptographic modules can be obtained by calling **pkc_crypto_get_registered_algorithms()**. You can then access information about a specific module by calling the **pkc_crypto_lookup_algorithm()** routine. To sign data with a private key or to verify signed data with a public key, either **pkc_crypto_verify_signature()** or **pkc_crypto_sign()** can be called.

In the low-level certificate interrogation API, the **verify()** routine is automatically called by the **pkc_crypto_verify_signature(3sec)** routine.

37.3.3 Signature Algorithms Provided by DCE Certification

The signature algorithms provided with DCE 1.2.2 are **md2WithRSA** and **md5WithRSA**.

37.3.4 Registering a Cryptographic Module

Perform the following steps to register a cryptographic module:

1. Implement the **name()** and **verify()** functions. These two routines must be implemented.
2. If your module needs to perform any initialization or finalization tasks, implement **open()** and/or **close()** routines for them. (These two routines are optional.)
3. Implement **sign()** and **generate_keypair()** functions if necessary. (These two routines are optional.)
4. Create a **pkc_signature_algorithm_t** structure containing the entrypoints of the routines implemented in Steps 1 to 3 (use **NULL** for the entrypoint of any unimplemented routines), and use this structure to register the algorithm implementation.

37.4 Implementing and Registering a Policy Module

A certification trust model simply prescribes which certifying authorities (CAs) can legitimately issue (create and sign) certificates for which principals. A trust model is implemented by a policy module. The ultimate purpose of the certification service is to return public keys, and it is the job of the routines in a policy module to do this. Looked at from this point of view, policy modules are mainly distinguished from each other by the two following things:

- Which principals a policy module is willing to return keys for.
- Which CAs a module is willing to trust the signatures of on the certificates from which it retrieves keys.

A principal's certificates will be retrieved from a directory service entry (exactly what entry depends on the policy used), and the policy module will only look for certain signatures known to it on the certificates it retrieves. However, direct retrieval via the subject's directory entry name is only one trust model. There can be many others. For example, see the discussion of certification paths above.

37.4.1 Policy Modules Provided with DCE Certification

Several certification policy modules are provided by DCE. These policies are described in the following sections.

37.4.1.1 Direct secd Lookup: DCE Registry Lookup Policy Model

The registry lookup policy module simply looks up principals' public keys in the DCE registry, and returns them. These keys are not held in certificates, but are stored as extended registry attributes.

If a caller of the high-level certificate retreival API has DCE credentials, then the registry retrieval policy will authenticate the registry as part of the retrieval operation. If no credentials are available, no authentication is possible. In this case, keys will still be returned, but the certificate API will indicate to the caller that the keys are untrusted.

37.4.1.2 DCE Hierarchical Trust Policy

The DCE hierarchical trust policy supplied with DCE 1.2.2 supports hierarchical cells/ DASS style trust paths. A trust path between principals A and B consists of zero or more up links, followed by zero or one cross link, followed by zero or more down links. The DCE hierarchical trust policy extension uses the DCE namespace certificate store extension (NCSE) for certificate retrieval.

Each cell is assumed to operate as a certification authority (CA) for top level principals registered within that cell. Thus, the CA for the cell **over_cell** is assumed to create a certificate for, say, the principal **felix** within **over_cell**, where the certificate signatory is ''over_cell'' and the certificate target is ''over_cell/felix''. If a cell employs structured names for principals, each level is considered to act as a CA for its subordinate. For example, if cell ''over_cell'' contains a principal **admin/JohnSmith**, then **over_cell/ admin/JohnSmith** is certified by two certificates, the first signed by **over_cell** and certifying **over_cell/admin**, and the second signed by **over_cell/admin** and certifying **over_cell/admin/JohnSmith**.

To avoid requiring that the cell's root CDS directory be used for storing certificates for the cell's principals, the DCE NCSE provided in DCE 1.2.2 allows any directory (in CDS or GDS) to be given an attribute (with OID **1.3.24.9.15**) that names a subdirectory (of the directory to which the attribute is given) within which certificates are to be stored. The value of this attribute is a string which will be appended to the name of the directory to give the name of a new directory within which certificates will be stored.

For example, if the root directory in the cell **over_cell** is the directory to which this ''certificate directory'' attribute is attached, and the attribute contains the value **principals**, the DCE NCSE will attempt to retrieve the certificate for **over_cell/P** from the CDS directory **over_cell/principals/P**. If the ''certificate directory'' attribute is missing or empty, the cell root directory will be searched for principal certificates. (Note that the insertion of the ''certificate directory'' attribute value applies only to locating certificates within the directory service; the above certificate would contain the name **over_cell/P** as its actual subject.)

At most one ''certificate directory'' attribute is considered when looking for certificates for a given name, according to the following algorithm:

- Starting with the name for which certificates are desired, RDNs are removed from the right of the name until either a directory is found that contains the ‘‘certificate directory’’ attribute, or the namespace root is reached.

In general, CDS administrators should define the ‘‘certificate directory’’ attribute within each CDS root directory, rather than storing certificates within the root directory. As well as reducing clutter in the cell root directory, doing this has efficiency benefits (terminating the search for the certificate directory at each CDS root), and also prevents the definition of this attribute at a higher level within the DCE global namespace from influencing the placement of certificates within a cell’s namespace.

37.4.1.3 PEM-like Policy

No explicit PEM-like policy is provided with DCE 1.2.2; however, the DCE hierarchical policy may be used in a PEM-like fashion by specifying root CA keys as the initial trust list, rather than keys belonging to the caller’s immediate CA.

37.5 The Low Level Certificate Manipulation API

The certificate manipulation API is a C++ interface. C++ must be used to retrieve the certificates into trust lists and manilupulate them there.

The contents of the

/usr/include/dce/asn.h

and

/usr/include/dce/x509.h

header files define some of the basic types used by the low-level certificate manipulation routines, including the actual structure of certificates. Following is a list of the low-level certificate routines defined in the

/usr/include/dce/pkc_certs.h

file:

- **pkc_add_trusted_key(3sec)**
- **pkc_lookup_keys_in_trustlist(3sec)**
- **pkc_lookup_key_in_trustlist(3sec)**
- **pkc_lookup_element_in_trustlist(3sec)**
- **pkc_check_cert_against_trustlist(3sec)**
- **pkc_revoke_certificate(3sec)**
- **pkc_revoke_certificates(3sec)**
- **pkc_delete_trustlist(3sec)**
- **pkc_copy_trustlist(3sec)**
- **pkc_display_trustlist(3sec)**

37.5.1 Policy Module Implementation

Implementation of a policy module consists essentially of writing **establish_trustbase()**, **delete_trustbase()**, **retrieve_keyinfo()** and **delete_keyinfo()** routines, and associated interrogation routines. The module will find certificates for principal names according to the rules set out for that module, verify their signatures, and return the public keys found in them to the original callers.

37.5.1.1 Certificate Revocation Lists (CRLs)

Certificate revocation lists are lists of certificates whose contents are no longer to be believed. Use of CRLs is policy-specific. **pkc_certs** provides objects for parsing and manipulating CRLs, and for using them to invalidate portions of a trust list.

37.5.2 Accessing a Registered Policy Module

Policy modules are registered in the form of **pkc_policy_t** structures, which contain the entry points for the following developer-written routines:

open() opens the module

close() closes the module

retrieve_keyinfo()
returns the public key for a specified principal name

name() Returns the name of the policy.

establish_trustbase()
Creates a trust base.

delete_trustbase()
Deletes a trust base.

delete_keyinfo()
Deletes a **keyinfo** handle.

get_key_count()
Returns the number of keys a **keyinfo** handle contains.

get_key_data()
Retrieves an individual key.

get_key_trust()
Returns the type of trust established for a specific key.

get_key_certifier_count()
Returns the number of certifiers in the trust path that certified a key.

get_key_certifier_info()
Returns information about a specific certifier of a key.

The **pkc_policy_t** structure also contains the following data fields:

- a certificate version number
- an object identifier (OID) identifying the policy module

Policy modules, similarly to signature algorithms (cryptographic modules), are identified by object identifiers (the character string returned by **name()** is intended for use in diagnostic or auditing messages).

Also similarly to cryptographic modules, there are two ways in which cryptographic modules can be accessed: either by a single call to which the identifying OID is passed (this is the recommended method); or by calling **pkc_plcy_lookup_policy(3sec)** and then (for example) the module's **(*retrieve_key)()** routine to obtain the public key (a list of the OIDs of all currently registered policy modules can be obtained by calling **pkc_plcy_get_registered_policies()**).

37.5.3 Registering a Policy Module

You must implement the following routines in any policy module:

name() Returns the name of the policy.

establish_trustbase()
Creates a trust base, which is a policy-specific data structure based on the initial set of trusted keys.

retrieve_keyinfo()
Given a trust base, returns a handle to keys for a specific principal.

delete_trustbase()
Deletes a trust base.

delete_keyinfo()
Deletes a **keyinfo** handle.

get_key_count()
Given a **keyinfo** handle, returns the number of keys it contains.

get_key_data()
Retrieves an individual key from a **keyinfo** handle

get_key_trust()
Returns the type of trust established for a specific key, and the purpose(s) for which that trust applies.

The following policy routines are optional:

open()

close() These routines perform any initialization and/or finalization tasks required by the module.

get_key_certifier_count()

This routine is required only for policies that return **CERTIFIED_TRUST** keys; it returns the number of certifiers in the trust path that certified a key.

get_key_certifier_info()

This routine is required if the module implements **get_key_certifier_count()**. It returns information about a specific certifier with the certification path of a specific key. Certifier 0 is the immediate certifier of the key; certifier 1 is the CA that certified certifier 0, and so on.

Once you have implemented all necessary routines for you module, you must create a **pkc_policy_t** structure containing their entrypoints. Unimplemented routines' entrypoints should be specified as **NULL**.

37.5.4 Registering the module

The module is registered by calling the registration function and passing it a **pkc_policy_t** structure, which contains the entry points for the module routines described above:

pkc_plcy_register_policy()

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X