Oracle® Database

Security Guide

10*g* Release 2 (10.2)

B14266-01

June 2005

ORACLE®

Oracle Database Security Guide 10*g* Release 2 (10.2)

B14266-01

Primary Author: Sumit Jeloka

Contributing Authors: Gopal Mulagund, Nina Lewis, Janaki Narasinghanallur, Srividya Tata, Narendra Manappa, Richard Smith, Don Gosselin, Daniel M. Wong

Contents

6 Access Control on Tables, Views, Synonyms, or Rows

7 Security Policies

8 Database Auditing: Security Considerations

12 Configuring and Administering Auditing

13 Introducing Database Security for Application Developers

14 Using Virtual Private Database to Implement Application Security Policies

15 Implementing Application Context and Fine-Grained Access Control

16 Preserving User Identity in Multitiered Environments

17 Developing Applications Using Data Encryption

Part IV Appendixes

List of Tables

List of Figures

Preface

This document provides a comprehensive overview of security for Oracle Database. It includes conceptual information about security requirements and threats, descriptions of Oracle Database security features, and procedural information that explains how to use those features to secure your database.

This preface contains these topics:

Audience

The Oracle Database Security Guide is intended for database administrators (DBAs), security administrators, application developers, and others tasked with performing the following operations securely and efficiently:

- Designing and implementing security policies to protect the organization's data, users, and applications from accidental, inappropriate, or unauthorized actions
- Creating and enforcing policies and practices of auditing and accountability for any such inappropriate or unauthorized actions
- Creating, maintaining, and terminating user accounts, passwords, roles, and privileges
- Developing applications that provide desired services securely in a variety of computational models, leveraging database and directory services to maximize both efficiency and client ease of use

To use this document, you need a basic understanding of how and why a database is used, as well as at least basic familiarity with SQL queries or programming.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to

evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

http://www.oracle.com/accessibility/

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Organization

This document contains:

Part I, "Overview of Security Considerations and Requirements"

Part I presents fundamental concepts of data security, and offers checklists and policies to aid in securing your site's data, operations, and users.

Chapter 1, "Security Requirements, Threats, and Concepts"

This chapter presents fundamental concepts of data security requirements and threats.

Chapter 2, "Security Checklists and Recommendations"

This chapter presents checklists, with brief explanations, for policies and practices that reduce your installation's vulnerabilities.

Chapter 3, "Security Policies and Tips"

This chapter presents basic general security policies, with specific chapter references, that apply to every site. These you must understand and apply to the unique considerations of your own site. The chapter also introduces general application design practices regarding roles and privileges.

Part II, "Security Features, Concepts, and Alternatives"

Part II presents methods and features that address the security requirements, threats, and concepts described in Part I.

Chapter 4, "Authentication Methods"

This chapter deals with verifying the identity of anyone who wants to use data, resources, or applications. Authentication establishes a trust relationship for further interactions as well as accountability linking access and actions to a specific identity.

Chapter 5, "Authorization: Privileges, Roles, Profiles, and Resource Limitations"

This chapter describes standard authorization processes that allow an entity to have certain levels of access and action, but which also limit the access, actions, and resources permitted to that entity.

Chapter 6, "Access Control on Tables, Views, Synonyms, or Rows"

This chapter discusses protecting objects by using object-level privileges and views, as well as by designing and using policies to restrict access to specific tables, views, synonyms, or rows. Such policies invoke functions that you design to specify dynamic predicates establishing the restrictions.

Chapter 7, "Security Policies"

This chapter discusses security policies in separate sections dealing with system security, data security, user security, password management, and auditing. It concludes with a more detailed version of the checklist first presented in Chapter 2.

Chapter 8, "Database Auditing: Security Considerations"

This chapter presents auditing as the monitoring and recording of selected user database actions. Auditing can be based either on individual actions, such as the type of SQL statement executed, or on combinations of factors that can include user name, application, time, and so on. Security policies can trigger auditing when specified elements in an Oracle database are accessed or altered, including the contents within a specified object.

Part III, "Security Implementation, Configuration, and Administration"

Part III presents the details of setting up, configuring, and administering Oracle Database security features.

Chapter 9, "Secure External Password Store"

This chapter discusses the secure external password store which allows you to store password credentials in a client side Oracle Wallet. It discusses client configuration for using the external password store. It also discusses managing external password store credentials.

Chapter 10, "Administering Authentication"

This chapter describes the methods for creating and administering authentication by defining users and how they are to be identified and verified before access is granted. Chapter 10 discusses the four primary methods as database, external, global, and proxy authentication.

Chapter 11, "Administering User Privileges, Roles, and Profiles"

This chapter presents the interwoven tasks and considerations involved in granting, viewing, and revoking database user privileges and roles, and the profiles that contain them.

Chapter 12, "Configuring and Administering Auditing"

This chapter describes auditing and accountability to protect and preserve privacy for the information stored in databases, detect suspicious activities, and enable finely-tuned security responses.

Chapter 13, "Introducing Database Security for Application Developers"

This chapter provides an introduction to the security challenges that face application developers and includes an overview of Oracle Database features they can use to develop secure applications.

Chapter 14, "Using Virtual Private Database to Implement Application Security Policies"

This chapter discusses developing secure applications by using application context, fine-grained access control, or virtual private database to implement security policies.

Chapter 15, "Implementing Application Context and Fine-Grained Access Control"

This chapter provides several examples of applications developed using application context, fine-grained access control, and virtual private database. It includes code examples and their corresponding explanations.

Chapter 16, "Preserving User Identity in Multitiered Environments"

This chapter discusses developing secure multiple tier applications.

Chapter 17, "Developing Applications Using Data Encryption"

This chapter discusses how you can use data encryption to develop secure applications, and the strengths and weaknesses of using this feature.

Part IV, "Appendixes"

Part IV contains two appendixes. The first appendix discusses new changes to the CONNECT role. The second appendix discusses the DBMS_SQLHASH package, which is used to verify data integrity.

Appendix A, "Addressing The CONNECT Role Change"

This appendix discusses the consequences of the fact that all privileges have been removed from the CONNECT role except the CREATE SESSION privilege.

Appendix B, "Verifying Data Integrity with DBMS_SQLHASH"

This appendix discusses the `DBMS_SQLHASH` package that can be used to verify data integrity.

Glossary

Related Documentation

For more information, see these Oracle resources:

- *Oracle Database Concepts*
- *Oracle Database Administrator's Guide*
- *Oracle Data Warehousing Guide*
- *Oracle Streams Advanced Queuing Java API Reference*
- *Oracle Streams Advanced Queuing User's Guide and Reference*

Many of the examples in this book use the sample schemas of the seed database, which is installed by default when you install Oracle. Refer to *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them yourself.

Printed documentation is available for sale in the Oracle Store at

http://oraclestore.oracle.com/

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

http://www.oracle.com/technology/membership/index.html

If you already have a user name and password for OTN, then you can go directly to the documentation section of the OTN Web site at

http://www.oracle.com/technology/documentation/index.html

To access the database documentation search engine directly, please visit

http://tahiti.oracle.com/

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples
- Conventions for Windows Operating Systems

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width) font`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, data types, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, user names, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.

Convention	Meaning	Example
`lowercase monospace (fixed-width) font`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, user names and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the data files and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`. Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase italic monospace (fixed-width) font`	Lowercase italic monospace font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
`[ ]`	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
`{ }`	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`
`|`	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`
`...`	Horizontal ellipsis points indicate either: ■ That we have omitted parts of the code that are not directly related to the example ■ That you can repeat a portion of the code	`CREATE TABLE ... AS `*`subquery`*`;` `SELECT `*`col1`*`, `*`col2`*`, ... , `*`coln`*` FROM employees;`

Convention	Meaning	Example
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	SQL> SELECT NAME FROM V$DATAFILE; NAME ---------- /fsl/dbs/tbs_01.dbf /fs1/dbs/tbs_02.dbf . . . /fsl/dbs/tbs_09.dbf 9 rows selected.
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	acctbal NUMBER(11,2); acct CONSTANT NUMBER(4) := 3;
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	CONNECT SYSTEM/*system_password* DB_NAME = *database_name*
UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	SELECT last_name, employee_id FROM employees; SELECT * FROM USER_TABLES; DROP TABLE hr.employees;
lowercase	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	SELECT last_name, employee_id FROM employees; sqlplus hr/hr CREATE USER mjones IDENTIFIED BY ty3MU9;

Conventions for Windows Operating Systems

The following table describes conventions for Windows operating systems and provides examples of their use.

Convention	Meaning	Example
Choose Start >	How to start a program.	To start the Database Configuration Assistant, choose Start > Programs > Oracle - *HOME_NAME* > Configuration and Migration Tools > Database Configuration Assistant.
File and directory names	File and directory names are not case sensitive. The following special characters are not allowed: left angle bracket (<), right angle bracket (>), colon (:), double quotation marks ("), slash (/), pipe (|), and dash (-). The special character backslash (\) is treated as an element separator, even when it appears in quotes. If the file name begins with \\, then Windows assumes it uses the Universal Naming Convention.	c:\winnt"\"system32 is the same as C:\WINNT\SYSTEM32

Convention	Meaning	Example
`C:\>`	Represents the Windows command prompt of the current hard disk drive. The escape character in a command prompt is the caret (^). Your prompt reflects the subdirectory in which you are working. Referred to as the *command prompt* in this manual.	`C:\oracle\oradata>`
Special characters	The backslash (\) special character is sometimes required as an escape character for the double quotation mark (") special character at the Windows command prompt. Parentheses and the single quotation mark (') do not require an escape character. Refer to your Windows operating system documentation for more information on escape and special characters.	`C:\>exp scott/tiger TABLES=emp QUERY=\"WHERE job='SALESMAN' and sal<1600\"` `C:\>imp SYSTEM/`*`password`*` FROMUSER=scott TABLES=(emp, dept)`
`HOME_NAME`	Represents the Oracle home name. The home name can be up to 16 alphanumeric characters. The only special character allowed in the home name is the underscore.	`C:\> net start Oracle`*`HOME_NAME`*`TNSListener`
`ORACLE_HOME` and *`ORACLE_BASE`*	In releases prior to Oracle8i release 8.1.3, when you installed Oracle components, all subdirectories were located under a top level *`ORACLE_HOME`* directory that by default used one of the following names: ▪ `C:\orant` for Windows NT ▪ `C:\orawin98` for Windows 98 This release complies with Optimal Flexible Architecture (OFA) guidelines. All subdirectories are not under a top level *`ORACLE_HOME`* directory. There is a top level directory called *`ORACLE_BASE`* that by default is `C:\oracle`. If you install the latest Oracle release on a computer with no other Oracle software installed, then the default setting for the first Oracle home directory is `C:\oracle\ora`*`nn`*, where *`nn`* is the latest release number. The Oracle home directory is located directly under *`ORACLE_BASE`*. All directory path examples in this guide follow OFA conventions. Refer to *Oracle Database Platform Guide for Windows* for additional information about OFA compliances and for information about installing Oracle products in non-OFA compliant directories.	Go to the *`ORACLE_BASE`*`\`*`ORACLE_HOME`*`\rdbms\admin` directory.

What's New in Oracle Database Security?

The Oracle Database 10*g* Release 2 (10.2) security features and enhancements described in this section comprise the overall effort to provide superior access control, privacy, and accountability with this release of the database.

The following sections describe new security features of Oracle Database 10*g* Release 2 (10.2) and provide pointers to additional information:

- New Features in Virtual Private Database
- New Features in Auditing
- New PL/SQL Encryption Package: DBMS_CRYPTO

New Features in Virtual Private Database

To provide enhanced access control, privacy, and performance, the following enhancements have been added to Virtual Private Database (VPD), a feature of the Enterprise Edition:

- Column-level VPD and Column-masking

  Column-level VPD policies provides more fine-grained access controls on data. With column-level VPD, security policies can be applied only where a particular column or columns are accessed in the user's query. This means that when a user has rights to access the object itself, VPD can limit the individual rows returned only if the columns the user accesses contain sensitive information, such as salaries or national identity numbers.

  The default behavior of column-level VPD restricts the number of rows returned when a query addresses columns containing sensitive data. In contrast, column-masking behavior allows all rows to be returned for a query against data protected by column-level VPD, but the columns that contain sensitive information are returned as `NULL` values. With column-masking, users see all the data they are supposed to see, but privacy is not compromised.

  See Also:

  - "Column-Level VPD" on page 14-3 for conceptual information about these new features
  - "Adding Policies for Column-Level VPD" on page 15-32 for information about how to add column-level VPD to your applications

- Static, Context-Sensitive, and Shared VPD Policy Types

Static and context-sensitive policy types optimize VPD for significant performance improvements because the policy function does not execute for every SQL query. *Static policies* maintain the same predicate for queries, updates, inserts, and deletes throughout a session. (However application context or attributes such as `SYSDATE` can change the value returned by the predicate.) They are particularly useful for hosting environments where you always need to apply the same predicate. With *context-sensitive policies*, the predicate can change after statement parse time, but VPD reexecutes the policy function only if the application context changes. This ensures that any changes to the predicate since the initial parsing are captured. Context-sensitive policies are useful when VPD policies must enforce two or more different predicates for different users or groups.

Both static and context-sensitive policies can be shared across multiple database objects, so that queries on another database object can use the same cached predicate. Shared policies enable you to further decrease the overhead of reexecuting policy functions for every query, reducing any performance impact.

> **See Also:** "DBMS_RLS.ADD_POLICY Procedure Policy Types" on page 15-29 for more information about these new policy types and how to use them in applications.

- Application context support for parallel queries

  In this release, if you use `SYS_CONTEXT` inside a SQL function that is embedded in a parallel query, then the function picks up the application context.

  > **See Also:** "Using SYS_CONTEXT in a Parallel Query" on page 15-5 for information about using this enhancement.

New Features in Auditing

Oracle Database 10*g* Release 2 (10.2) includes the following new auditing features:

- Improved fine-grained auditing

  This release enhances the security of fine-grained auditing. In addition, fine-grained audit records can now be redirected to the same operating system file that receives standard audit records in XML format.

  > **See Also:** "Fine-Grained Auditing" on page 12-24 for more information about these new features

- XML audit records

  Audit records can now be written to the operating system as XML files. A new dynamic view, `V$XML_AUDIT_TRAIL`, provides enhanced usability by making XML audit records available to DBAs through a SQL query.

  > **See Also:** "Operating System Audit Trail" on page 8-4 for more information about this new view

- Syslog audit records

  Audit records can now be written to the operating system using a syslog audit trail. A potential security vulnerability to an operating system audit trail is that a privileged user such as a DBA can modify or delete audit records. In order to minimize this risk, you can use syslog, which is a standard protocol on

UNIX-based systems for logging information from different components of a network.

> **See Also:** "Syslog Audit Trail" on page 8-5 for more information about this new view

- Uniform Audit Trail

  In this release, the `DBA_COMMON_AUDIT_TRAIL` view has been added, which presents both the standard and the fine-grained audit log records in a single view.

  > **See Also:** "Database Audit Trail Contents" on page 12-7 for more information about this new view

New PL/SQL Encryption Package: DBMS_CRYPTO

In this release, a new flexible interface, `DBMS_CRYPTO`, is provided to encrypt especially sensitive stored data, or it can also be used in conjunction with PL/SQL programs running network communications. This new interface provides support for the following features:

- Encryption algorithms as follows:
  - AES
  - Triple DES (112- and 168-bits)
  - DES
  - RC4
- Cryptographic hash algorithms (such as SHA-1)
- Keyed hash (MAC, or Message Authentication Code) algorithms (such as SHA-1)
- Padding forms (PKCS #5, zeroes)
- Block cipher chaining mode modifiers (CBC, CFB, ECB, OFB)

`DBMS_CRYPTO` is intended to replace the `DBMS_OBFUSCATION_TOOLKIT`, providing greater ease of use and support for a range of algorithms to accommodate new and existing systems.

> **See Also:** Chapter 17, "Developing Applications Using Data Encryption" for information about how to use this package

Part I

Overview of Security Considerations and Requirements

Part I presents fundamental concepts of data security requirements and threats that pertain to connecting to a database, accessing and altering tables, and using applications. In addition, security checklists are provided for DBAs and application developers, which cover installation preparation, database administration best practices, and recommendations for developing secure applications.

This part contains the following chapters:

This part also contains high-level security checklists for DBAs and application developers, covering preparations for installation, best practices for administration, and recommended practices for developing secure applications. References are included to the explanations and alternatives presented in Part II and the examples described in Part III.

1

Security Requirements, Threats, and Concepts

Database security requirements arise from the need to protect data: first, from accidental loss and corruption, and second, from deliberate unauthorized attempts to access or alter that data. Secondary concerns include protecting against undue delays in accessing or using data, or even against interference to the point of denial of service. The global costs of such security breaches run up to billions of dollars annually, and the cost to individual companies can be severe, sometimes catastrophic.

These requirements are dynamic. New technologies and practices continually provide new arenas for unauthorized exploitation, as well as new ways for accidental or deliberate misuse to affect even stable products and environments. Today evolution involves a globally changing technological and cultural environment, in which security concerns necessarily affect both the use of existing solutions and the development of new ones.

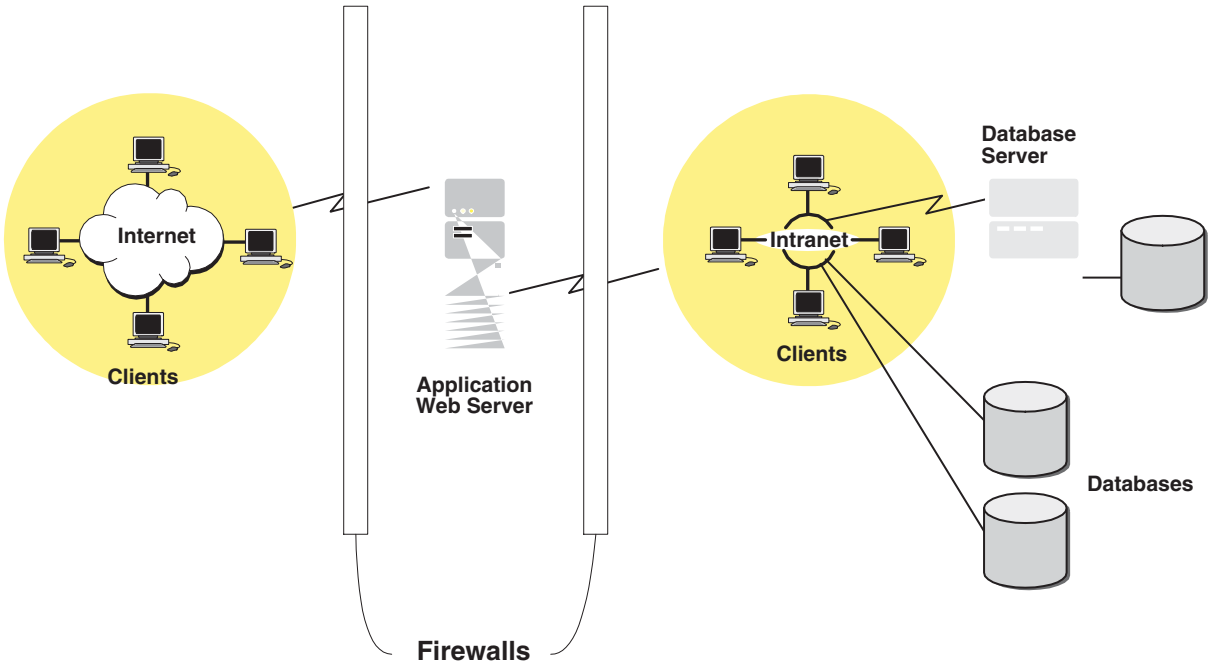
As security requirements are understood with increasing clarity, certain general principles can be developed for satisfying them and for disabling the threats against them deriving from Internet vulnerabilities. Of course, principles can vary in effectiveness. Their implementations usually vary in cost: hardware and software acquisition and maintenance, administrative and programming personnel, and the impact of security measures on processing time and response time. Total cost also includes the costs of managing hardware, software, personnel, efficiency, responsiveness, and so on. These management costs can escalate with increasing volumes of users, transactions, and data types.

This book will address most of those issues, conceptually at first, and with increasing detail as it moves toward discussing specific choices and implementations.

The basic elements and operations of the database environment include connection to a server or a schema, table access and alteration, and application usage. Securing these against accidental or deliberate misuse is the responsibility of security officers, administrators, and application programmers. They must also administer and protect the rights of internal database users, and guarantee electronic commerce confidentiality as customers access databases from anywhere on the Internet.

In the Internet age, the full spectrum of risks to valuable and sensitive data and to user access and confidentiality, is broader than ever before. Figure 1–1 shows the complex computing environment that security plans must protect.

Figure 1–1 Realms Needing Security in an Internet World



The diagram shows several important parts of the security picture, illustrating client communities, connections, databases, and servers, all of which must be secured against inappropriate access or use. These different areas can require different techniques to achieve good security, and they must integrate so as to preclude or minimize security gaps or vulnerabilities.

Table 1–1 provides separate categories that are usable in focusing efforts to create secure operations in a secure environment. When these necessary components are consistent in their security focus, coherent in the ways they work together, and made complete by closing all known channels of attack and misuse, your security is as good as it gets.

These categories reappear in later chapters in discussing checklists and best practices.

Table 1–1 Security Issues by Category

Dimension	Security Issues
Physical	Computers must be made physically inaccessible to unauthorized users by keeping them in a secure physical environment.
Personnel	The people responsible for the physical security, system administration, and data security of the site must be reliable. Performing background checks on DBAs before making hiring decisions is a wise protective measure.

Table 1–1 (Cont.) Security Issues by Category

Dimension	Security Issues
Procedural	The procedures and policies used in the operation of your system must assure reliable data. It is often wise to separate out users' functional roles in data management. For example, one person can be responsible for database backups. Her only role is to be sure the database is up and running. Another person can be responsible for generating application reports involving payroll or sales data. His role is to examine the data and verify its integrity. Further, you can establish policies that protect tables and schemas against unauthorized, accidental, or malicious usage.
Technical	Storage, access, manipulation, and transmission of data must be safeguarded by technology that enforces your particular information control policies.

When you think carefully about security risks, the solutions you adopt will apply well to the actual situation you are addressing. All security problems do not necessarily have a technical fix. For example, employees must occasionally leave their desks unattended. Depending on the sensitivity of their work and on your required level of security, your security procedures could require them to do any of the following:

- Have another person cover for them while they're away
- Clear the desk surface, locking all sensitive materials away, before leaving
- Lock their doors, if they have private offices
- Explicitly lock their computer screens before leaving the desk

No technical solution can fix a physically insecure work environment or a corrupt or disaffected employee. It is true, though, that procedural and technical protections might be able to limit the damage that a physical breach or a disgruntled employee (or an ex-employee) can inflict.

Identity Management: Security in Complex, High-Volume Environments

In addition to the general issues and categories of security, the sheer numbers of people and activities requiring a security focus add the dimension of complexity.

As the number of users, databases, applications, and networks grows from a few to dozens, hundreds, or thousands, the complexity of their interactions rises exponentially. So, too, do the risks, and the management tasks required to maintain security and efficiency.

For example, the number of interactions for ten users accessing any of five databases is potentially 50. Add 90 users and 45 databases, and you have 100 users accessing any of 50 databases, with potential interactions at 5000. When you add to this example some number of applications and of networks, you begin to see extreme complexity, with directly proportional security risks. A security breach anywhere in a network can threaten the security of its databases and users, and that of other connected networks, databases, and users.

This type of complex environment demands speed and flexibility in granting or revoking access rights for any user and any resource. Delays in administrative processes, or their implementation on the corresponding databases, translate either to legitimate access delayed or to access granted when it should have been denied.

For example, when an employee with access to multiple applications and accounts on multiple databases leaves the company, his access to those applications and accounts should stop instantly. Achieving this can be difficult when administrative control and responsibility is distributed across nodes in the network and among different administrators and groups.

But what if an intelligent central repository could efficiently control data regarding identity, accounts, authentication, and authorization and rapidly communicate any needed information to any node or application? Then one change (or few) in one place could alter all access rights and privileges of an employee who is leaving the company. The assumption is that the intelligence imputed to this central repository and its software takes care of all such considerations and connections. Of course, all related system, database, and application routines would need to adjust to relying upon that central repository as the "single source of truth" regarding such information.

These considerations are the basis for an integrated Identity Management solution. Its components and integration are described in subsequent chapters, as the need for them becomes explicit in the context of general security functionality and dimensions.

> **See Also:** For a full discussion of Oracle Identity Management Infrastructure, see the *Oracle Identity Management Concepts and Deployment Planning Guide*

As an overview, however, the following subsections provide an introduction:

- Desired Benefits of Identity Management
- Components of Oracle Identity Management Infrastructure

Desired Benefits of Identity Management

The goal of identity management is to create

- Greater security, because a single point of control is inherently easier to secure and more responsive than multiple such points
- Greater efficiency, because a single point of control automatically eliminates the duplication and delays inherent in a system where dispersed administrative responsibilities require multiple actions for the same accounts

However, cost and complexity remain relevant measures of the viability of any such solution, which ideally should provide the following reductions in resource allocations:

- **One-time cost:** Planning and implementing the identity management infrastructure should be a one-time cost, rather than a necessary part of each enterprise application deployment. In this way, new applications can be rapidly deployed, automatically benefiting from the infrastructure, but without having to re-create it. Examples could include portals, J2EE applications, and e-business applications.
- **Centralized management with distributable tools:** Provisioning and managing identities should be done centrally, even if administered in multiple places using tools that can handle any account alteration needed.
- **Seamless timely distribution:** Changes to user accounts, profiles, or privileges should be instantly available to all enterprise applications and rapidly communicated to distributed databases.
- **User single sign-on:** The centralized security infrastructure should make single sign-on possible across enterprise applications. Single sign-on makes it

unnecessary for users to remember multiple passwords and for security administrators to protect multiple password repositories and provisioning mechanisms.

- **Single point of integration:** The centralized identity management infrastructure should provide a single point of integration between the enterprise environment and other identity management systems. It should eliminate any need for multiple custom point-to-point integration solutions.

An identity management solution meeting all these criteria at a high level would provide an enterprise with high availability, information localization, and delegated component administration. Further, each additional application deployed in that enterprise would then leverage the shared infrastructure for identity management services.

As a real-world example, the Oracle Identity Management infrastructure uses integrated components to provide those benefits as described in the next section.

Components of Oracle Identity Management Infrastructure

The Oracle Identity Management infrastructure includes the following components and capabilities:

- **Oracle Internet Directory** is a scalable, robust, Lightweight Directory Access Protocol (LDAP) version 3 compliant directory service implemented on the Oracle9*i* Database.
- **Oracle Directory Integration and Provisioning** is a part of Oracle Internet Directory, which enables synchronization between Oracle Internet Directory and other directories and user repositories. This service also provides automatic provisioning services for Oracle components and applications and, through standard interfaces, for third-party applications.
- **Oracle Delegated Administration Service** is a part of Oracle Internet Directory, which provides trusted proxy-based administration of directory information by users and application administrators.
- **OracleApplication Server Single Sign-On** provides single sign-on access to Oracle and third party web applications.
- **OracleApplication Server Certificate Authority** generates and publishes X.509 version 3 Public Key Infrastructure (PKI) certificates to support strong authentication methods, secure messaging, and so on.

  In addition to its use of Secure Socket Layer (SSL), Oracle Application Server Containers for J2EE, and Oracle HTTP Server, Oracle's Identity Management infrastructure has a built-in reliance on OracleAS Single Sign-On and Oracle Internet Directory. When OracleAS Certificate Authority is in use, it publishes each valid certificate in a directory entry for the distinguished name in use. The single sign-on server and other components can rely on these directory entries because the certificate authority removes revoked and expired certificates from the directory on a regular basis. Users authenticated by the single sign-on server, and who lack a certificate can rapidly acquire one directly from OracleAS Certificate Authority, enabling them to authenticate to any Oracle component or application that is configured to authenticate users with the single sign-on server.

In a typical enterprise application deployment, a single instance of Oracle Identity Management infrastructure is deployed, consisting of multiple server and component instances. Such a configuration provides the high availability, information localization, and delegated component administration benefits mentioned earlier.

2

Security Checklists and Recommendations

This chapter gives you a broad overview of the many types of tasks you must perform in order to build good security. Understanding the diverse categories of tasks improves your likelihood of preventing security vulnerabilities. Such vulnerabilities, whether exploited accidentally or intentionally, can undermine or overwhelm otherwise tight security that you have created in other areas.

Chapter 1 introduced the requirements for good security, the threats against it, and concepts that have proven useful in creating practical methods for developing and sustaining it.

The overview presented in this chapter identifies categories of tasks useful in meeting those requirements and threats. This chapter presents brief descriptions of these categories and tasks, with cross-references to Parts 2 and 3, where important details necessary to their implementation are described.

Good security requires physical access control, reliable personnel, trustworthy installation and configuration procedures, secure communications, and control of database operations such as selecting, viewing, updating, or deleting database records. Since some of these requirements involve applications or stored procedures as well as human action, security procedures must also account for how these programs are developed and dealt with.

The following practical concerns must also be met:

- Minimizing the costs of equipment, personnel, and training
- Minimizing delays and errors
- Maximizing rapid and thorough accountability

Scalability, too, is an important and independent practical criterion that should be assessed for each proposed solution.

These, then, are the categories with which this overview is concerned. They are discussed in the following sections:

Physical Access Control Checklist

It should not be easy to walk into a facility without a key or badge, or without being required to show identity or authorization. Controlling physical access is your first

line of defense, by protecting your data (and your staff) against the simplest of inadvertent or malicious intrusions and interferences.

Lack of such control can make it easier to observe, copy, or steal your other security controls, including internal keys, key codes, badge numbers or badges, and so on. The security of these measures also depends on how alert and security conscious each of your staff is, but physical access control stops a variety of potential problems.

Each organization must evaluate its own risks and budget. Elaborate measures may not be needed depending on many factors: company size, risk of loss, internal access controls, quantity and frequency of outside visitors, and so on. Preparing for accountability and recovery are additional considerations, possibly prompting alarms or video surveillance of entryways. The visibility of these preparations can also act as deterrence.

Improving physical access control to your facility can add to your security. Make it difficult to get in, difficult to remain or leave unobserved or unidentified, difficult to get at sensitive or secure areas inside, and difficult not to leave a trace.

Personnel Checklist

Your staff makes your organization work well, depending on who they are and how they are managed. Your security is critically dependent on them: first, on how honest and trustworthy they are, and second, on how aware and alert they are to security concerns and considerations. The first issue is a matter of selection, interviewing, observation, and reference checking. Done well, these skills can prevent your hiring people who are (or are likely to become) inappropriate for tasks or environments that depend on establishing and maintaining security. To a very large degree, security depends on individuals. When personnel get careless, resentful, or larcenous, tight security loosens or disappears. Your other measures will not matter if they are carelessly or deliberately undermined or sabotaged.

The second issue is how aware and alert your staff is to security concerns and considerations. Such consciousness is only partly a matter of background, and the environment and training you provide are the most significant influences, given basic honesty and intent to cooperate. When an organization both shows and says that security is important, by establishing and enforcing security procedures and by providing training and bulletins about it, people learn and adapt. The result is better security and safety for them as well as for the data and products of an organization.

Secure Installation and Configuration Checklist

Information security, privacy, and protection of corporate assets and data are of critical importance to every business. For databases, establishing a secure configuration is a very strong first line of defense, using industry-standard best security practices for operational database deployments. The following list of such practices is deliberately general to remain brief. Additional details for each recommendation as it applies to Oracle Database appear in Chapter 7, "Security Policies".

Implementing the following recommendations provides the basis for a secure configuration:

- Install only what is required.

  Do a custom installation. Avoid installing options and products you do not need. Choose to install only those additional products and options, in addition to the database server, that you require. Or, if you choose to do a *typical* installation

instead, then improve your security after the installation processes finish, by removing the options and products you do not need.

- Lock and expire default user accounts.

  Oracle Database installs with many default (preset) database server user accounts. Upon the successful creation of a database server instance, the Database Configuration Assistant automatically locks and expires most default database user accounts.

  > **Note:** If you use Oracle Universal Installer or Database Configuration Assistant, then they will prompt for new `SYS` and `SYSTEM` passwords, and will not accept the defaults "change_on_install" or "manager", respectively.

  Once the database is installed, lock `SYS` and `SYSTEM` as well, and use `AS SYSDBA` for administrator access. Specify administrative passwords individually.

  This account (`AS SYSDBA`) tracks the operating system user name, maintaining accountability. If you only need access for database startup and shutdown, then use `AS SYSOPER` instead. `SYSOPER` has fewer administrative privileges than `SYS`, but enough to perform basic operations such as startup, shutdown, mount, backup, archive, and recover.

  Database Configuration Assistant is not used during a manual installation, so all default database users remain unlocked and are able to gain unauthorized access to data or to disrupt database operations. Therefore, after a manual installation, use SQL to *lock* and *expire* all default database user accounts except `SYS`, `SYSTEM`, `SCOTT`, and `DBSNMP`. Do it to `SCOTT`, too, unless it is being actively used. Also lock `SYS` and `SYSTEM` as described earlier. If a locked account is later needed, then a database administrator can simply unlock and activate that account with a new password.

- Change default user passwords.

  Security is most easily broken when a default database server user account still has a default password *even after installation*. The following steps fix this:

  – Change the default passwords of administrative users immediately after installing the database server.

    In any Oracle environment (production or test), assign strong, secure passwords to the `SYS` and `SYSTEM` user accounts immediately upon successful installation of the database server. Under no circumstances should the passwords for `SYS` and `SYSTEM` retain their default values. Similarly, for production environments, do not use default passwords for any administrative accounts, including `SYSMAN` and `DBSNMP`.

  – Change the default passwords of all users immediately after installation.

    Lock and expire all default accounts after installation. If any such account is later activated, then change its default password to a new secure password.

  – Enforce password management.

    Apply basic password management rules, such as password length, history, and complexity, to all user passwords.

    Mandate that all users change their passwords regularly, such as every eight weeks.

If possible, use Oracle Advanced Security (an option to the Enterprise Edition of Oracle Database) with network authentication services (such as Kerberos), token cards, smart cards, or X.509 certificates. These services provide strong user authentication and enable better protection against unauthorized access.

- Enable data dictionary protection.

  Implement data dictionary protection to prevent users who have the `ANY` system privilege from using it on the data dictionary. Oracle Database sets the `O7_DICTIONARY_ACCESSIBILITY` to `FALSE`. This setting prevents using the `ANY` system privilege on the data dictionary, except for authorized users making DBA-privileged connections (for example `CONNECT/AS SYSDBA`).

- Practice the principle of least privilege.

  The following practices implement this principle:

  – Grant necessary privileges only.

    Do not provide database users more privileges than necessary. Enable only those privileges actually required to perform necessary jobs efficiently:

    1) Restrict the number of system and object privileges granted to database users.

    2) Restrict the number of `SYS`-privileged connections to the database as much as possible. For example, there is generally no need to grant `CREATE ANY TABLE` to any non-DBA-privileged user.

  – Revoke unnecessary privileges and roles from the database server user group `PUBLIC`.

    This default role, granted to every user in an Oracle database, enables unrestricted use of its privileges, such as `EXECUTE` on various PL/SQL packages. If unnecessary privileges and roles are not revoked from `PUBLIC`, then a minimally privileged user could access and execute packages otherwise inaccessible to him. The important packages that may potentially be misused are listed in Chapter 7, "Security Policies".

  – Restrict permissions on run-time facilities.

    Do not assign *all permissions* to any database server run-time facility, such as the Oracle Java Virtual Machine (OJVM).

    Instead, grant specific permissions to the explicit document root file paths for such facilities that may execute files and packages outside the database server. Examples are listed in Chapter 7, "Security Policies".

- Enforce access controls effectively.

  Authenticate clients properly. Although remote authentication can be turned on (`TRUE`), your installation is more secure with it turned off (`FALSE`, which is the default). With remote authentication turned on, the database implicitly trusts every client, because it assumes every client was authenticated by the remote authenticating system. However, clients in general (such as remote PCs) cannot be trusted to perform proper operating system authentication, so turning on this feature is a very poor security practice. To enforce proper server-based authentication of clients connecting to an Oracle database, leave or turn this feature off (`remote_os_authentication=FALSE`, which is the default).

- Restrict operating system access.

  The following practices implement appropriate restrictions on operating system access:

  - Limit the number of operating system users.
  - Limit the privileges of the operating system accounts (administrative, root-privileged, or DBA) on the Oracle Database host (computer) to the fewest and least powerful privileges required for each user.
  - Disallow modifying the default permissions for the Oracle Database home (installation) directory or its contents, even by privileged operating system users or the Oracle owner.
  - Restrict symbolic links. Ensure that when any path or file to the database is provided, neither that file nor any part of that path is modifiable by an untrusted user. The file and all components of the path should be owned by the DBA or some trusted account, such as root. This recommendation applies to all types of files, such as data files, log files, trace files, external tables, bfiles, and so on.

- Restrict network access.

  Details about restricting network access are provided in the Networking Security Checklists section.

- Apply all security patches and workarounds.

  Plug every security hole or flaw as soon as corrective action is identified. Always apply all relevant and current security patches for both the host operating system and Oracle Database, and for all installed Oracle Database options and components.

  Periodically, check the security site on Oracle Technology Network for details on security alerts released by Oracle Corporation at

  `http://www.oracle.com/technology/deploy/security/alerts.htm`

  Also check the Oracle Worldwide Support Service site, Metalink, for details on available and upcoming security-related patches at

  `http://metalink.oracle.com`

- Contact Oracle security products.

  If you believe that you have found a security vulnerability in Oracle Database, then submit an iTAR to Oracle Worldwide Support Services using Metalink, or e-mail a complete description of the problem, including product version and platform, together with any exploit scripts and examples, to the following address:

  `secalert_us@oracle.com`

Networking Security Checklists

Security for network communications is improved by using client, listener, and network checklists to ensure thorough protection. Using SSL is an essential element in these lists, enabling top security for authentication and communications.

SSL Checklist

SSL is the Internet standard protocol for secure communication, providing mechanisms for data integrity and data encryption. These mechanisms can protect the messages sent and received by you or by applications and servers, supporting secure authentication, authorization, and messaging by means of certificates and, if necessary, encryption. Good security practices maximize protection and minimize

gaps or disclosures that threaten security. While the primary documentation for Oracle SSL configuration and practices is *Oracle Database Advanced Security Administrator's Guide*, the following list illustrates the cautious attention to detail necessary for the successful use of SSL:

1. Ensure that configuration files (such as for clients and listeners) use the correct port for SSL, which is the port configured upon installation. You can run HTTPS on any port, but the standards specify port 443, where any HTTPS-compliant browser looks by default. Or the port can be specified in the URL, for example,

   ```
   https://secure.server.com:4445/
   ```

   If a firewall is in use, then it too must use the same port(s) for secure (SSL) communication.

2. Ensure that *tcps* is specified as the `PROTOCOL` in the `ADDRESS` parameter in the `tnsnames.ora` file (typically on the client or in the LDAP directory). An identical specification must appear in the `listener.ora` file (typically in the `$ORACLE_HOME/network/admin` directory).

3. Ensure that the SSL mode is consistent for both ends of every communication. For example, between the database on one side and the user or application on the other. The mode can specify either client or server authentication (one-way), both client and server authentication (two-way), or no authentication.

4. Ensure that the server supports the client cipher suites and the certificate key algorithm in use.

5. Do not remove the encryption from your RSA private key inside your `server.key` file, which requires that you enter your pass phrase to read and parse this file.

   Note: A non-SSL-aware server does not require such a pass phrase.

   However, if you were to decide your server is secure enough, you could remove the encryption from the RSA private key while preserving the original file. This would enable system boot scripts to start the server, because no pass phrase would be needed. However, be very sure that permissions on the `server.key` file allow only root or the Web server user to read it. Ideally, restrict permissions to root alone, and have the Web server start as root but run as another user. Otherwise, anyone who gets this key can impersonate you on the net.

   See Also:

   - For general SSL information, including configuration, see the *Oracle Database Advanced Security Administrator's Guide*
   - For tcps information, see *Oracle Database Net Services Administrator's Guide* and *Oracle Database Net Services Reference*

Client Checklist

Because authenticating client computers is problematic over the Internet, typically, user authentication is performed instead. This approach avoids client system issues that include falsified IP addresses, hacked operating systems or applications, and falsified or stolen client system identities. Nevertheless, the following steps improve the security of client connections:

- Configure the connection to use SSL.

Using SSL communication makes eavesdropping difficult and enables the use of certificates for user and server authentication.

- Set up certificate authentication for clients and servers.

Listener Checklist

Because the listener acts as the database gateway to the network, it is important to limit the consequences of malicious interference:

- Restrict the privileges of the listener, so that it cannot read or write files in the database or the Oracle server address space.

  This restriction prevents external procedure agents spawned by the listener (or procedures executed by such an agent) from inheriting the ability to do such reads or writes. The owner of this separate listener process should not be the owner that installed Oracle or executes the Oracle instance (such as `ORACLE`, the default owner).

- Secure administration by doing the following:
  - Protect the listener with a password.
  - Prevent online administration.
  - Use SSL when administering the listener.
  - Remove the external procedure configuration from the `listener.ora` file if you do not intend to use such procedures.

    See Also: See the section A Security Checklist in Chapter 7, "Security Policies", for more specific details.

- Monitor listener activity.

Network Checklist

Protecting the network and its traffic from inappropriate access or modification is the essence of network security. The following practices improve network security:

1. Restrict physical access to the network. Make it difficult to attach devices for listening to, interfering with, or creating communications.
2. Protect the network access points from unauthorized access. This goal includes protecting the network-related software on the computers, bridges, and routers used in communication.
3. Because you cannot protect physical addresses when transferring data over the Internet, use encryption when this data needs to be secure.
4. Use firewalls.

   Appropriately placed and configured firewalls can prevent outsider access to your organization intranet when you allow internal users to have Internet access.

   - Keep the database server behind a firewall. The Oracle Database network infrastructure supports a variety of firewalls from various vendors. Examples are listed in Chapter 7, "Security Policies".
   - Ensure that the firewall is placed outside the network to be protected.
   - Configure the firewall to accept only those protocols, applications, or client/server sources that you know are safe.

- Use a product like Oracle Connection Manager to multiplex multiple client network sessions through a single network connection to the database. It can filter on source, destination, and host name. This functionality enables you to ensure that connections are accepted only from physically secure terminals or from application Web servers with known IP addresses. (Filtering on IP address alone is not enough for authentication, because it can be faked.)

5. Never poke a hole through a firewall.

   For example, do not leave the Oracle Listener port 1521 open, allowing the database to connect to the Internet or the Internet to connect with the database. This could introduce significant security vulnerabilities that hackers are likely to exploit. Hackers could enable even more port openings through the firewall, create multithreaded operating system server problems, and enable access to crucial information on databases behind the firewall. If the Listener is running without a password, then hackers can probe for critical details about the databases on which it is listening. These details include trace and logging information, banner information, and database descriptors and service names, enabling malicious and damaging attacks on the target databases.

6. Prevent unauthorized administration of the Oracle Listener.

   Always establish a meaningful, well-formed password for the Oracle Listener, to prevent remote configuration of the Oracle Listener. Further, prevent unauthorized administration of the Oracle Listener, as described in Chapter 7, "Security Policies".

7. Check network IP addresses.

   Use the Oracle Net *valid node checking* security feature to allow or deny access to Oracle server processes from network clients with specified IP addresses. Set parameters in the `protocol.ora` file (Oracle Net configuration file) to specify client IP addresses that are denied or allowed connections to the Oracle Listener. This action prevents potential Denial of Service attacks.

8. Encrypt network traffic.

   If possible, utilize Oracle Advanced Security to encrypt network traffic between clients, databases, and application servers.

   Note: Oracle Advanced Security is available only with the Enterprise Edition of Oracle Database.

9. Harden the host operating system (the system on which Oracle Database resides).

   Disable all unnecessary operating system services. Many UNIX and Windows services are not necessary for most deployments. Such services include FTP, TFTP, TELNET, and so forth.

   For each disabled service, be sure to close both the UDP and TCP ports. Leaving either type of port enabled leaves the operating system vulnerable.

In summary, consider all paths the data travels and assess the threats that impinge on each path and node. Then, take steps to lessen or eliminate those threats and the consequences of a breach of security. In addition, monitor and audit to detect either increased threat levels or successful penetration.

3

Security Policies and Tips

The idea of security policies includes many dimensions. Broad considerations include regular backups and storing them off-site. Narrow table or data considerations include ensuring that unauthorized access to confidential data, such as employee salaries, is precluded by built-in restrictions on every type of access to the table that contains such data.

This chapter introduces ideas about security policies and offers tips about recommended practices that can tighten security in the following sections:

Introduction to Database Security Policies

This section briefly introduces security policies. It covers:

Security Threats and Countermeasures

An organization should create a written security policy to enumerate the security threats it is trying to guard against and the specific measures the organization must take. Security threats can be addressed with different types of measures:

- Procedural, such as requiring data center employees to display security badges
- Personnel-related, such as performing background checks or "vetting" key personnel
- Physical, such as securing computers in restricted-access facilities
- Technical, such as implementing strong authentication requirements for critical business systems

Consider whether the appropriate response to a threat is procedural, physical, technical, personnel-related, or a combination of many such measures.

For example, one possible security threat is the disruption of critical business systems caused by a malicious person damaging a computer. A physical response to this threat is to secure key business computers in a locked facility. A procedural response is to create system backups at regular intervals. Personnel measures could include background checks on employees who access or manage key business systems.

Oracle Database offers many mechanisms you can use to implement the technical measures of a good security policy. These are discussed in Chapter 7, "Security Policies" and Chapter 14, "Using Virtual Private Database to Implement Application Security Policies".

What Information Security Policies Can Cover

In addition to addressing requirements unique to your environment, you should also design and implement technical measures in your information security policies to address important generic issues, such as the concerns described in Table 3–1.

Table 3–1 Issues and Actions that Security Policies Must Address

Security Concern/Practice	Recommended Actions	References
Establish and maintain application-level security	Attach privileges and roles to each application. Ensure that users cannot misuse those roles and privileges when they are not using the application. Base use of roles on user-defined criteria, such as a user connecting only from a particular IP address, or only through a particular middle tier.	See Ref [1]
Manage privileges and attributes (system/object/user)	Permit only certain users to access, process, or alter data, including the rights to execute a particular type of SQL statement or to access another user's object. Apply varying limitations on users' access to or actions on objects, such as schemas, tables, or rows, or resources, such as time (CPU, connect, or idle times).	See Ref [2]
Create, manage, and control roles (database, enterprise)	Create named groups of privileges to facilitate granting them to users, including previously named groups (roles).	See Ref [3]
Establish the granularity of access control desired	Set up session-based attributes securely. For example, store user attributes (user name, employee number, and so on) to be retrieved later in the session, enabling fine-grained access control. Create security policy functions and attach them to critical or sensitive tables, views, or synonyms used by an application. DML statements on such objects are then modified dynamically, and transparently to the user, to preclude inappropriate access. Enforce fine-grained or label-based access control automatically with policy functions or data and user labels, quickly limiting access to sensitive data, often without additional programming	See Ref [4]
Establish and manage the use of encryption	Use SSL connections, well-established encryption suites, or PKI certificates for critical or sensitive transmissions and applications.	See Ref [5]
Establish and maintain security in 3-tier applications	Preserve user identity through a middle tier to the database. Avoid the overhead of separate database connections by proxying user identities (and credentials like a password or certificate) through the middle tier to the database.	See Ref [6]
Control query access, data misuse, and intrusions	Monitor query access based on specific content or row to detect data misuse or intrusions. Use proxy authentication to support auditing of proxied user connections. Use regular auditing and fine-grained auditing to detect unauthorized or inappropriate access or actions.	See Ref [7]

[1] Discussed under "Creating Secure Application Roles" on page 13-4

[2] Discussed under Introduction to Privileges on page 5-1, Understanding User Privileges and Roles on page 11-11, and Granting User Privileges and Roles on page 11-18

[3] Discussed under "Introduction to Roles" on page 5-14 and Managing User Roles on page 11-15

[4] Discussed under Chapter 7, "Security Policies", Chapter 14, "Using Virtual Private Database to Implement Application Security Policies" & Chapter 15, "Implementing Application Context and Fine-Grained Access Control"

[5] Discussed under Chapter 4, "Authentication Methods"

[6] Discussed under Chapter 15, "Implementing Application Context and Fine-Grained Access Control" & Chapter 16, "Preserving User Identity in Multitiered Environments"

[7] Discussed under Chapter 15, "Implementing Application Context and Fine-Grained Access Control" & Chapter 16, "Preserving User Identity in Multitiered Environments"; for auditing, Chapter 8, "Database Auditing: Security Considerations"and Chapter 12, "Configuring and Administering Auditing"

The security practices and recommended actions of Table 3–1 are readily implemented using the Oracle features, facilities, and products listed in Table 3–2. Discussions of these terms and products appear in the corresponding chapters (or book) listed in Table 3–2.

Table 3–2 Reference Terms and Chapters for Oracle Features and Products

Reference Terms	Reference Chapters
Application Context	Chapter 15, "Implementing Application Context and Fine-Grained Access Control"
Data Encryption	Chapter 17, "Developing Applications Using Data Encryption"
Fine-Grained Access Control	Chapter 15, "Implementing Application Context and Fine-Grained Access Control"
Fine-Grained Auditing	Chapter 8, "Database Auditing: Security Considerations" Chapter 12, "Configuring and Administering Auditing"
Oracle Label Security	*Oracle Label Security Administrator's Guide*
Proxy Authentication	Chapter 16, "Preserving User Identity in Multitiered Environments"
End-User Identity Propagation	Chapter 16, "Preserving User Identity in Multitiered Environments"
Secure Application Roles	Chapter 5, "Authorization: Privileges, Roles, Profiles, and Resource Limitations" Chapter 13, "Introducing Database Security for Application Developers"

Recommended Application Design Practices to Reduce Risk

To avoid or minimize potential problems, use the following recommended practices for database roles and privileges. Each practice is explained in the following sections in detail:

- Tip 1: Enable and Disable Roles Promptly
- Tip 2: Encapsulate Privileges in Stored Procedures
- Tip 3: Use Role Passwords Unknown to the User
- Tip 4: Use Proxy Authentication and a Secure Application Role
- Tip 5: Use Secure Application Roles to Verify IP Address
- Tip 6: Use Application Context and Fine-Grained Access Control

Tip 1: Enable and Disable Roles Promptly

Enable a role only when the application starts, and disable it as soon as the application terminates. To do this, you must take the following approach:

- Give each application distinct roles.
- For each application, establish one role that contains *all* the privileges necessary to use the application successfully.
- If needed, establish several additional roles that contain only some of these privileges, to provide tighter or less restrictive security to different users or uses of the application
- Protect each database role by a password (or by operating system authentication) to prevent unauthorized use.

  One role should contain only non destructive privileges associated with the application (`SELECT` privileges for specific tables or views associated with the application). This read-only role allows an application user to generate custom reports using ad hoc tools, such as SQL*Plus, but disallows modifying table data. A role designed for an ad hoc query tool may or may not be protected by a password (or by operating system authentication).
- Use the `SET ROLE` statement at application startup to enable one of the database roles associated with that application. For a role authorized by a password, the `SET ROLE` statement within the application must include that password, preferably encrypted by the application. If a role is authorized by the operating system, then the system administrator must set up accounts in advance to provide application users with appropriate operating system privileges.
- Disable a previously enabled database role upon application termination.
- Grant application users database roles as needed.

> **Note:** Database roles granted to users can be enabled by users *outside the application*. Such use is not controlled by application-based security, but it can be controlled by virtual private database. In three-tier systems, using a secure application role prevents users from acquiring the role outside the application.

Additionally, you can use the `PRODUCT_USER_PROFILE` table to do the following:

- Specify what roles to enable when a user starts SQL*Plus. This functionality is similar to that of a precompiler or Oracle Call Interface (OCI) application that issues a `SET ROLE` statement to enable specific roles upon application startup.
- Disable the use of the `SET ROLE` statement for SQL*Plus users, thereby restricting such users to only the privileges associated with the roles enabled when SQL*Plus starts.
- Enable other ad hoc query and reporting tools to restrict the roles and commands that each user can use while running that product.

See Also:

- "Ways to Use Application Context with Fine-Grained Access Control" on page 14-13
- "Creating Secure Application Roles" on page 13-4
- The appropriate tool manual, such as the *SQL*Plus User's Guide and Reference*, which contains information about how to create and how to use the `PRODUCT_USER_PROFILE` table

Tip 2: Encapsulate Privileges in Stored Procedures

Restrict ad hoc query tools from exercising application privileges, by encapsulating these privileges into stored procedures. Grant users execute privileges on these procedures rather than issuing direct privilege grants to the users, so that the privileges cannot be used outside the appropriate procedure.

Users can then exercise privileges only in the context of well-formed business applications. For example, consider authorizing users to update a table only by executing a stored procedure, rather than by updating the table directly. By doing this, you avoid the problem of the user having the `SELECT` privilege and using it outside the application.

See Also: "Example 3: Event Triggers, Secure Application Context, Fine-Grained Access Control, and Encapsulation of Privileges" on page 15-11

Tip 3: Use Role Passwords Unknown to the User

Grant privileges through roles that require a password unknown to the user. For privileges that the user should exercise only within an application, enable the role by a password *known only by the creator of the role*. Use the application to issue a `SET ROLE` statement. Since the user does not have the role password, it must either be embedded in the application or retrievable from a database table by a stored procedure.

Hiding the password discourages users from trying to use the privileges without using the application. This improves security, but it is not foolproof.

Security by obscurity is not a good security practice. It protects against lazy users who merely want to bypass the application, even though they could, with access to the application code, potentially find the password. It does *not* protect against users who want to deliberately misuse privileges without using the application code (malicious users). Because malicious users can decompile client code and recover embedded passwords, you should only use the *embedded password* method to protect against lazy users.

Retrieving the role password from a database table is a bit more secure. It requires that the user uncover what stored procedure to use, gain `EXECUTE` permission on that procedure, execute it, and retrieve the password. Only then could the user use the role outside of the application.

Tip 4: Use Proxy Authentication and a Secure Application Role

To enable a role in three-tier systems, the user must access the database through a middle-tier application that requires proxy authentication and a secure application role.

Proxy authentication distinguishes between a middle tier creating a session on behalf of a user and the user connecting directly. Both the proxy user (the middle tier) and the *real* user information are captured in the user session.

A secure application role is implemented by a package, performing desired validation before allowing a user to assume the privileges in the role. When an application uses proxy authentication, the secure application role package validates that the user session was created by proxy. If the user is connecting to the database through an application, the role can be set, but if the user is connecting directly, it cannot.

Consider a situation in which you want to restrict use of an HR administration role to users accessing the database (by proxy) through the middle tier `HRSERVER`. You could create the following secure access role:

```
CREATE ROLE admin_role IDENTIFIED USING hr.admin;
```

Here, the `hr.admin` package performs the desired validation, permitting the role to be set only if it determines that the user is connected by proxy. The `hr.admin` package can use `SYS_CONTEXT ('userenv', 'proxy_userid')`, or `SYS_CONTEXT (userenv', 'proxy_user')`. Both return the ID and name of the proxy user (`HRSERVER`, in this case). If the user attempts to connect directly to the database, the `hr.admin` package will not allow the role to be set.

See Also:

- Chapter 16, "Preserving User Identity in Multitiered Environments"
- "Creating Secure Application Roles" on page 13-4
- "Application Context" on page 6-5.
- Chapter 14, "Using Virtual Private Database to Implement Application Security Policies"
- Chapter 15, "Implementing Application Context and Fine-Grained Access Control"
- *Oracle Database SQL Reference* for information about `SYS_CONTEXT ('userenv', 'proxy_user')` and `SYS_CONTEXT ('userenv', proxy_userid')`

Tip 5: Use Secure Application Roles to Verify IP Address

The secure application role package can use additional information in the user session to restrict access, such as the original IP address of the user. You should never use IP addresses to make primary access control decisions, because IP addresses can be spoofed. You can, however, use an IP address to increase access restrictions after using other criteria for the primary access control decision.

For example, you may want to ensure that a user session was created by proxy for a middle-tier user connecting from a particular IP address. Of course, the middle tier must authenticate itself to the database before creating a lightweight session, and the database ensures that the middle tier has privilege to create a session on behalf of the user.

Your secure application role package could validate the IP address of the incoming connection. Before allowing `SET ROLE` to succeed, you can to ensure that the `HRSERVER` connection (or the lightweight user session) is coming from the appropriate IP address by using `SYS_CONTEXT (userenv',' 'ip_address')`. Doing so provides an additional layer of security.

Tip 6: Use Application Context and Fine-Grained Access Control

In this scenario, you combine server-enforced fine-grained access control and, through application context, session-based attributes.

See Also:

- Chapter 14, "Using Virtual Private Database to Implement Application Security Policies"
- Chapter 15, "Implementing Application Context and Fine-Grained Access Control"

Part II

Security Features, Concepts, and Alternatives

Part II presents methods and Oracle Database features that address the security requirements, threats, and concepts presented in Part I.

This part contains the following chapters:

4

Authentication Methods

Authentication means verifying the identity of someone (a user, device, or an entity) who wants to access data, resources, or applications. Validating that identity establishes a trust relationship for further interactions. Authentication also enables accountability by making it possible to link access and actions to specific identities. After authentication, authorization processes can allow or limit the levels of access and action permitted to that entity as described in Chapter 5, "Authorization: Privileges, Roles, Profiles, and Resource Limitations".

Oracle allows a single database instance to use any or all methods. Oracle requires special authentication procedures for database administrators, because they perform special database operations. Oracle also encrypts passwords during transmission to ensure the security of network authentication.

To validate the identity of database users and prevent unauthorized use of a database user name, you can authenticate users by using any combination of the methods described in the following sections:

Authentication Considerations About ...	Links to Topics
Operating Systems	Authentication by the Operating System
Networks and LDAP Directories	Authentication by the Network
Databases	Authentication by Oracle Database
Multitier Systems	Multitier Authentication and Authorization
Secure Socket Layer Usage	Authentication of Database Administrators
Database Administrators	Authentication of Database Administrators

See Also:

- Chapter 10, "Administering Authentication", discusses how to configure and administer these authentication methods.
- Chapter 16, "Preserving User Identity in Multitiered Environments", discusses proxy authentication.

Authentication by the Operating System

Some operating systems permit Oracle to use information they maintain to authenticate users. This has the following benefits:

- Once authenticated by the operating system, users can connect to Oracle more conveniently, without specifying a user name or password. For example, an

operating-system-authenticated user can invoke SQL*Plus and skip the user name and password prompts by entering the following:

```
SQLPLUS /
```

- With control over user authentication centralized in the operating system, Oracle need not store or manage user passwords, though it still maintains user names in the database.
- Audit trails in the database and operating system can use the same user names.

When an operating system is used to authenticate database users, managing distributed database environments and database links requires special care.

See Also:

- *Oracle Database Administrator's Guide* sections on (and index entries for) authentication, operating systems, distributed database concepts, and distributed data management
- Operating system-specific documentation by Oracle for more information about authenticating by using your operating system

Authentication by the Network

Authentication over a network is handled by the SSL protocol or by third-party services as described in the following subsections:

- Authentication Using SSL
- Authentication Using Third-Party Services
  - Kerberos Authentication
  - PKI-Based Authentication
  - Authentication with RADIUS
  - Directory-Based Services

Authentication Using SSL

The Secure Socket Layer (SSL) protocol is an application layer protocol. It can be used for user authentication to a database, and it is independent of global user management in Oracle Internet Directory. That is, users can use SSL to authenticate to the database even without a directory server in place.

Authentication Using Third-Party Services

Authentication over a network makes use of third-party network authentication services. Prominent examples include Kerberos, Public Key Infrastructure (PKI), the Remote Authentication Dial-In User Service (RADIUS), and directory-based services, as described in the following subsections.

If network authentication services are available to you, then Oracle can accept authentication from the network service. If you use a network authentication service, then some special considerations arise for network roles and database links.

Note: To use a network authentication service with Oracle, you need Oracle Enterprise Edition with the Oracle Advanced Security option.

See Also:

- *Oracle Database Administrator's Guide* for more information about network authentication.
- *Oracle Database Advanced Security Administrator's Guide* for information about Oracle Enterprise Edition with the Oracle Advanced Security option

Kerberos Authentication

Kerberos is a trusted third-party authentication system that relies on shared secrets. It presumes that the third party is secure, and provides single sign-on capabilities, centralized password storage, database link authentication, and enhanced PC security. It does this through a Kerberos authentication server, or through Cybersafe Active Trust, a commercial Kerberos-based authentication server.

See Also: *Oracle Database Advanced Security Administrator's Guide* for more information about Kerberos.

PKI-Based Authentication

Authentication systems based on PKI issue digital certificates to user clients, which use them to authenticate directly to servers in the enterprise without directly involving an authentication server. Oracle provides a PKI for using public keys and certificates, consisting of the following components:

- Authentication and secure session key management using SSL.
- Oracle Call Interface (OCI) and PL/SQL functions

  These are used to sign user-specified data using a private key and certificate. The verification of the signature on data is done by using a trusted certificate.

- **Trusted certificates**

  These are used to identify third-party entities that are trusted as signers of user certificates when an identity is being validated. When the user certificate is being validated, the signer is checked by using trust points or a trusted certificate chain of certificate authorities stored in the validating system. If there are several levels of trusted certificates in this chain, then a trusted certificate at a lower level is simply trusted without needing to have all its higher-level certificates reverified.

- **Oracle wallets**

  These are data structures that contain the private key of a user, a user certificate, and the set of trust points of a user (trusted certificate authorities).

- **OracleAS Certificate Authority**

  This is a component of the Oracle Identity Management infrastructure, which provides an integrated solution for provisioning X.509 version 3 certificates for individuals, applications, and servers that require certificates for PKI-based operations such as authentication, SSL, S/MIME, and so on.

- **Oracle Wallet Manager**

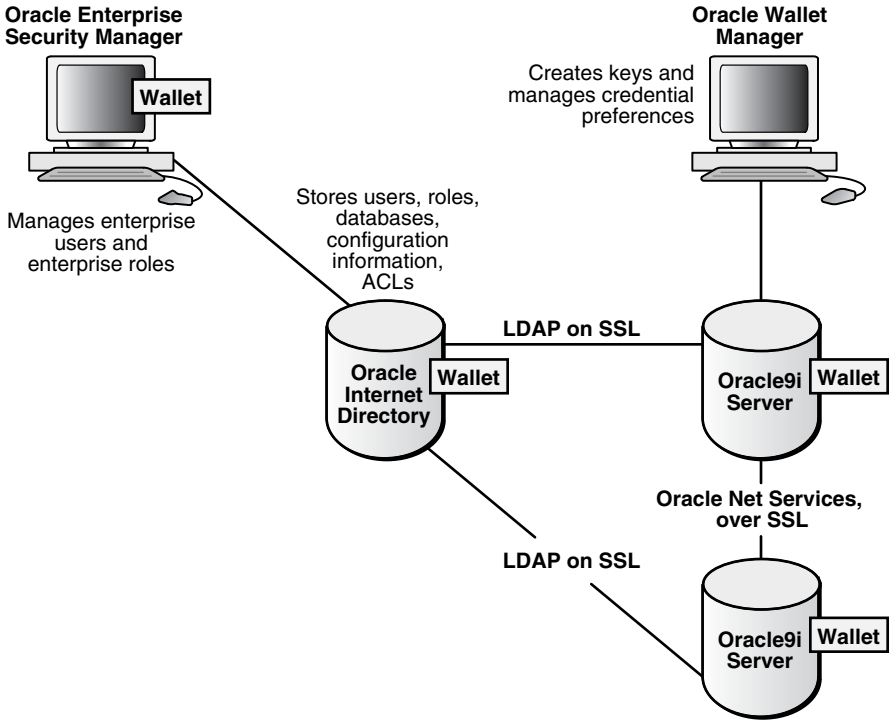
This is a standalone Java application used to manage and edit the security credentials in Oracle wallets. It performs the following operations:

- Protects user keys
- Manages X.509 version 3 certificates on Oracle clients and servers
- Generates a public-private key pair and creates a certificate request for submission to a certificate authority
- Installs a certificate for the entity
- Configures trusted certificates for the entity
- Creates wallets
- Opens a wallet to enable access to PKI-based services

- X.509 version 3 certificates obtained from (and signed by) a trusted entity, a certificate authority. Because the certificate authority is trusted, these certificates certify that the requesting entity's information is correct and that the public key on the certificate belongs to the identified entity. The certificate is loaded into an Oracle wallet to enable future authentication.

Oracle public key infrastructure is illustrated in Figure 4–1.

Figure 4–1 Oracle Public Key Infrastructure



Authentication with RADIUS

Oracle supports remote authentication of users through the Remote Authentication Dial-In User Service (RADIUS), a standard lightweight protocol used for user authentication, authorization, and accounting.

See Also: *Oracle Database Advanced Security Administrator's Guide* for information about Oracle Advanced Security

Directory-Based Services

Using a central directory can make authentication and its administration extremely efficient. Directory-based sevices include the following:

- Oracle Internet Directory, which uses the Lightweight Directory Access Protocol (LDAP), enables information about users (called enterprise users) to be stored and managed centrally. Although database users must be created (with passwords) in each database that they need to access, enterprise user information is accessible centrally in the Oracle Internet Directory. You can also integrate this directory with Active Directory and iPlanet.

  Oracle Internet Directory lets you manage the security attributes and privileges for users, including users authenticated by X.509 certificates. Oracle Internet Directory also enforces attribute-level access control. This feature enables read, write, or update privileges on specific attributes to be restricted to specific users, such as an enterprise security administrator. Directory queries and responses can use SSL encryption for enhanced protection during authentication and other interactions.

- **Oracle Enterprise Security Manager**, which provides centralized privilege management to make administration easier and increase security levels. Oracle Enterprise Security Manager lets you store and retrieve roles from Oracle Internet Directory.

  See Also:

  - *Oracle Internet Directory Administrator's Guide*
  - *Oracle Database 2 Day DBA*
  - *Oracle Enterprise Manager Concepts*
  - *Oracle Enterprise Manager Advanced Configuration*

Authentication by Oracle Database

Oracle Database can authenticate users attempting to connect to a database, by using information stored in that database itself. To set up Oracle Database to use database authentication, you must create each user with an associated password. The user must provide this user name and password when attempting to establish a connection. This process prevents unauthorized use of the database, because the connection will be denied if the user provides an incorrect password. Oracle Database stores user passwords in the data dictionary in an encrypted format to prevent unauthorized alteration. Users can change their passwords at any time.

To identify the authentication protocols that are allowed by a client or a database, a DBA can explicitly set the `SQLNET.ALLOWED_LOGON_VERSION` parameter in the server `sqlnet.ora` file. Then each connection attempt is tested, and if the client or server does not meet the minimum version specified by its partner, authentication fails with an ORA-28040 error. The parameter can take the values 10, 9, or 8, which is the default. These values represent database server versions. Oracle recommends the value 10.

Database authentication includes the following features:

- Password Encryption While Connecting.
  This protection is always in force, by default.

- Account Locking
- Password Lifetime and Expiration
- Password History
- Password Complexity Verification

Password Encryption While Connecting

Passwords are always automatically and transparently encrypted during network (client/server and server/server) connections, using AES (Advanced Encryption Standard) before sending them across the network.

> **See Also:** For more information about encrypting passwords in network systems, refer to:
>
> - Chapter 7, "Security Policies"
> - *Oracle Database Administrator's Guide*
> - *Oracle Database Advanced Security Administrator's Guide*

Account Locking

Oracle can lock a user's account after a specified number of consecutive failed login attempts. You can configure the account to unlock automatically after a specified time interval or to require database administrator intervention to be unlocked.

Use the `CREATE PROFILE` statement to establish how many failed login attempts a user can attempt before the account locks, and how long it remains locked before it unlocks automatically.

The database administrator can also lock accounts manually, so that they cannot unlock automatically but must be unlocked explicitly by the database administrator.

> **See Also:** "Profiles" on page 5-24

> **Note:** Failed login attempts will be limited by default in future Oracle Database releases.

Password Lifetime and Expiration

The database administrator can specify a lifetime for passwords, after which they expire and must be changed before account login is again permitted. A grace period can be established, during which each attempt to login to the database account receives a warning message to change the password. If it is not changed by the end of that period, then the account is locked. No further logins to that account are allowed without assistance by the database administrator.

The database administrator can also set the password state to expired, causing the user account status to change to expired. The user or the database administrator must then change the password before the user can log in to the database.

> **See Also:** Password Management Policy in Chapter 7, "Security Policies"

Password History

The password history option checks each newly specified password to ensure that a password is not reused for a specified amount of time or for a specified number of

password changes. The database administrator can configure the rules for password reuse with `CREATE PROFILE` statements.

See Also:

- For the complete syntax of the CREATE PROFILE command, see the *Oracle Database SQL Reference* .
- For a more complete discussion of password history policies, see the Password History section in Chapter 7, "Security Policies"

Password Complexity Verification

Complexity verification checks that each password is complex enough to provide reasonable protection against intruders who try to break into the system by guessing passwords.

The sample Oracle password complexity verification routine (the PL/SQL script `UTLPWDMG.SQL`, which sets the default profile parameters) checks that each password meet the following requirements:

- Be a minimum of four characters in length
- Not equal the userid
- Include at least one alphabet character, one numeric character, and one punctuation mark
- Not match any word on an internal list of simple words like welcome, account, database, user, and so on
- Differ from the previous password by at least three characters

See Also: Discussion about Password Complexity Verification on page 7-12.

Multitier Authentication and Authorization

In a multitier environment, Oracle controls the security of middle-tier applications by limiting their privileges, preserving client identities through all tiers, and auditing actions taken on behalf of clients. In applications that use a heavy middle tier, such as a transaction processing monitor, the identity of the clients connecting to the middle tier must be preserved. One advantage of using a middle tier is **connection pooling**, which allows multiple users to access a data server without each of them needing a separate connection. In such environments, you need to be able to set up and break down connections very quickly.

For these environments, Oracle database administrators can use the Oracle Call Interface to create **lightweight sessions**, which allow database password authentication for each user. This method preserves the identity of the real user through the middle tier without the overhead of a separate database connection for each user.

You can create lightweight sessions with or without passwords. However, if a middle tier is outside or on a firewall, then security is better when each lightweight session has its own password. For an internal application server, lightweight sessions without passwords might be appropriate.

Issues of administration and security in multitier environments are discussed in the following sections:

Clients, Application Servers, and Database Servers

In a multitier environment, an application server provides data for clients and serves as an interface from them to one or more database servers. The application server can validate the credentials of a client, such as a web browser, and the database server can audit operations performed by the application server. These auditable operations include actions performed by the application server on behalf of clients, such as requests that information be displayed on the client. A request to connect to the database server is an example of an application server operation not related to a specific client.

Note: While client-side authentication is possible, Oracle strongly recommends disabling it by setting the `remote_os_authentication` parameter to `FALSE`.

Authentication in a multitier environment is based on trust regions. Client authentication is the domain of the application server. The application server itself is authenticated by the database server. The following operations are performed:

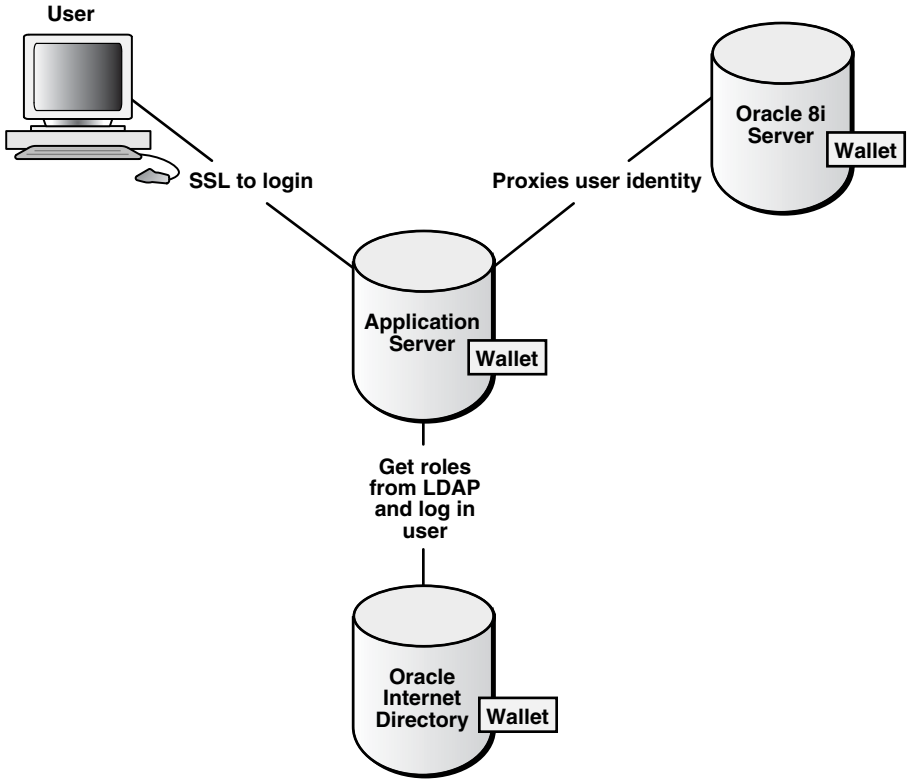
- The client provides proof of authenticity to the application server, typically, by using a password or an X.509 certificate.
- The application server authenticates the client and then authenticates itself to the database server.
- The database server authenticates the application server, verifies that the client exists, and verifies that the application server has the privilege to connect for this client.

Application servers can also enable roles for a client on whose behalf they connect. The application server can obtain these roles from a directory, which thus serves as an authorization repository. The application server can only request that these roles be enabled. The database verifies the following requirements:

- That the client has these roles by checking its internal role repository
- That the application server has the privilege to connect on behalf of the user, and thus to use these roles as the user could

Figure 4–2 shows an example of multitier authentication.

Figure 4–2 Multitier Authentication



Security Issues for Middle-Tier Applications

Security for middle-tier applications must address the following key issues:

- Accountability: The database server must be able to distinguish between the actions of a client and the actions an application takes on behalf of a client. It must be possible to audit both kinds of actions.
- Differentiation: The database server must be able to distinguish between a client accessing the database directly and an application server acting either for itself or on behalf of a browser client.
- Least privilege: Users and middle tiers should be given the fewest privileges necessary to perform their actions, to minimize the danger of inadvertent or malicious unauthorized activities.

Identity Issues in a Multitier Environment

Multitier authentication maintains the identity of the client through all tiers of the connection in order to maintain useful audit records. If the identity of the originating client is lost, then specific accountability of that client is lost. It becomes impossible to distinguish operations performed by the application server on behalf of the client from those done by the application server by itself.

Restricted Privileges in a Multitier Environment

Privileges in a multitier environment must be limited to those necessary to perform the requested operation.

Client Privileges

Client privileges must be as limited as possible in a multitier environment. Operations are performed on behalf of the client by the application server.

Application Server Privileges

Application server privileges in a multitier environment must also be limited, so that the application server cannot perform unwanted or unneeded operations while performing a client operation.

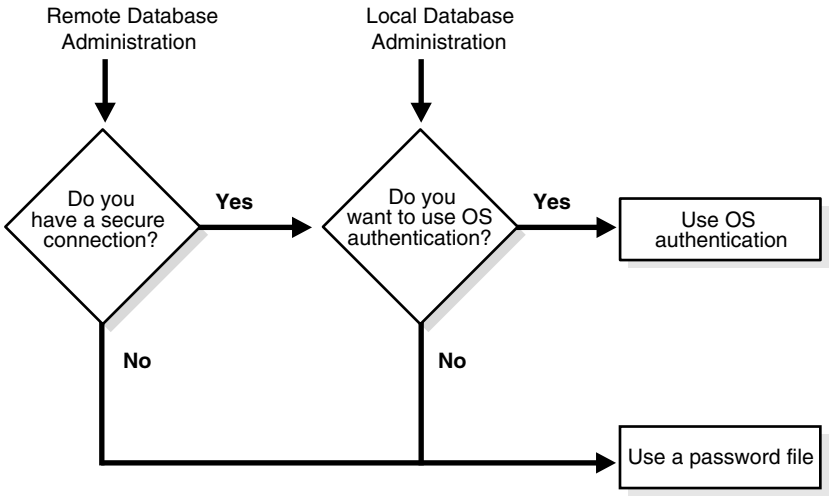
> **See Also:** Chapter 10, "Administering Authentication", and the *Oracle Database Administrator's Guide*, for more information about multitier authentication

Authentication of Database Administrators

Database administrators perform special operations (such as shutting down or starting up a database) that should not be performed by normal database users. Oracle provides for secure authentication of database administrator user names, for which you can choose either operating system authentication or password files.

Figure 4–3 illustrates the choices you have for database administrator authentication schemes. Different choices apply to administering your database locally (on the machine where the database resides) and to administering many different database machines from a single remote client.

Figure 4–3 Database Administrator Authentication Methods



Operating system authentication for a database administrator typically involves establishing a group on the operating system, assigning DBA privileges to that group, and then adding the names of persons who should have those privileges to that group.

> **Note:** On UNIX systems, the special group is called the **dba** group.

The database uses password files to keep track of those database user names that have been granted the `SYSDBA` and `SYSOPER` privileges. These privileges enable the following operations and capabilities:

- SYSOPER lets database administrators perform STARTUP, SHUTDOWN, ALTER DATABASE OPEN/MOUNT, ALTER DATABASE BACKUP, ARCHIVE LOG, and RECOVER. SYSOPER also includes the RESTRICTED SESSION privilege.
- SYSDBA has all system privileges with ADMIN OPTION, including the SYSOPER system privilege, and permits CREATE DATABASE and time-based recovery.

Note: Connections requested AS SYSDBA or AS SYSOPER must use these phrases; without them, the connection fails. The Oracle parameter 07_DICTIONARY_ACCESSIBILITY is set to FALSE by default, to limit sensitive data dictionary access only to those authorized. The parameter also enforces the required AS SYSDBA or AS SYSOPER syntax.

See Also:

- Administrator Security in Chapter 7, "Security Policies"
- Your Oracle operating system-specific documentation for information about configuring operating system authentication of database administrators
- *Oracle Database Administrator's Guide* for information on creation and maintenance of password files

Oracle Database 10*g* Release 2 (10.2) enhances password file based authentication by making it easier to use. The following enhancements have been made:

- Password file based authentication is enabled by default. This means that the database is ready to use a password file for authenticating users that have SYSDBA or SYSOPER system privileges. Password file based authentication is activated as soon as you create a password file using the ORAPWD utility.
- A password file containing users with SYSDBA or SYSOPER privileges can be shared between different databases. You can have a shared password file that contains users in addition to the SYS user.

Note: In order to share a password file between different databases, the REMOTE_LOGIN_PASSWORDFILE parameter needs to be changed to SHARED in the init.ora file. The default value of this parameter is EXCLUSIVE. This is the recommended setting.

5

Authorization: Privileges, Roles, Profiles, and Resource Limitations

Authorization includes primarily two processes:

- Permitting only certain users to access, process, or alter data
- Applying varying limitations on user access or actions. The limitations placed on (or removed from) users can apply to objects, such as schemas, tables, or rows; or to resources, such as time (CPU, connect, or idle times).

This chapter introduces the basic concepts and mechanisms for placing or removing such limitations on users, individually or in groups, in the following sections:

Topic Category	Links to Topics
How Privileges Are Acquired and Used	Introduction to Privileges, including system, schema, object, table, procedure, and other privileges
How Roles Are Acquired, Used, and Restricted	Introduction to Roles
How and Why Resource Limits Are Applied to Users	User Resource Limits
How Profiles Are Determined and Used	Profiles

> **See Also:** Chapter 11, "Administering User Privileges, Roles, and Profiles", discusses how to configure and administer privileges, roles, and profiles for users, including DBAs and application programmers.

Introduction to Privileges

A **privilege** is a right to execute a particular type of SQL statement or to access another user's object. Some examples of privileges include the right to:

- Connect to the database (create a session)
- Create a table
- Select rows from another user's table
- Execute another user's stored procedure

You grant privileges to users so these users can accomplish tasks required for their jobs. You should grant a privilege only to a user who requires that privilege to accomplish the necessary work. Excessive granting of unnecessary privileges can compromise security. A user can receive a privilege in two different ways:

- You can grant privileges to users explicitly. For example, you can explicitly grant to user `SCOTT` the privilege to insert records into the `employees` table.
- You can also grant privileges to a role (a named group of privileges), and then grant the role to one or more users. For example, you can grant the privileges to select, insert, update, and delete records from the `employees` table to the role named `clerk`, which in turn you can grant to users `scott` and `brian`.

Because roles allow for easier and better management of privileges, you should normally grant privileges to roles and not to specific users.

See Also:

- Chapter 11, "Administering User Privileges, Roles, and Profiles"
- *Oracle Database Administrator's Guide* for discussions of managing and using system and schema object privileges
- *Oracle Database SQL Reference* for the complete list of system privileges and their descriptions

There are six major categories of privileges, some with significant subcategories:

- System Privileges
- Schema Object Privileges
- Table Privileges
- View Privileges
- Procedure Privileges
- Type Privileges

System Privileges

A **system privilege** is the right to perform a particular action, or to perform an action on any schema objects of a particular type. For example, the privileges to create tablespaces and to delete the rows of any table in a database are system privileges. There are over 100 distinct system privileges to manage as described in the following subsections:

- Granting and Revoking System Privileges
- Who Can Grant or Revoke System Privileges?

Granting and Revoking System Privileges

You can grant or revoke system privileges to users and roles. If you grant system privileges to roles, then you can use the roles to manage system privileges. For example, roles permit privileges to be made selectively available.

Note: In general, you grant system privileges only to administrative personnel and application developers. End users normally do not require and should not have the associated capabilities.

Use either of the following to grant or revoke system privileges to users and roles:

- The Oracle Enterprise Manager 10g Database Control
- The `GRANT` and `REVOKE` SQL statements

See Also:

- For more information about Database Control, see *Oracle Database 2 Day DBA*.
- For information about modifying users with Database Control, see the topic "Creating, Editing, and Deleting Users" in the Enterprise Manager online help.

Who Can Grant or Revoke System Privileges?

Only two types of users can grant system privileges to other users or revoke such privileges from them:

- Users who have been granted a specific system privilege with the `ADMIN OPTION`
- Users with the system privilege `GRANT ANY PRIVILEGE`

Schema Object Privileges

A **schema object privilege** is the permission to perform a particular action on a specific schema object.

Different object privileges are available for different types of schema objects. The privilege to delete rows from the `departments` table is an example of an object privilege.

Some schema objects, such as clusters, indexes, triggers, and database links, do not have associated object privileges. Their use is controlled with system privileges. For example, to alter a cluster, a user must own the cluster or have the `ALTER ANY CLUSTER` system privilege.

The following subsections discuss granting and revoking such privileges:

- Granting and Revoking Schema Object Privileges
- Who Can Grant Schema Object Privileges?
- Using Privileges with Synonyms

Object privileges that apply to specific schema objects are discussed in the following sections:

- Table Privileges
- View Privileges
- Sequences (see Managing Sequences in *Oracle Database Administrator's Guide*
- Procedure Privileges
- Functions and Packages (Managing Object Dependencies in *Oracle Database Administrator's Guide*)
- Type Privileges

Granting and Revoking Schema Object Privileges

Schema object privileges can be granted to and revoked from users and roles. If you grant object privileges to roles, then you can make the privileges selectively available.

Object privileges for users and roles can be granted or revoked using the following:

- The SQL statements GRANT and REVOKE
- The Oracle Enterprise Manager 10g Database Control

See Also:

- For more information about Database Control, see *Oracle Database 2 Day DBA*.
- For information about modifying privileges with Database Control, see the Enterprise Manager online help.

Who Can Grant Schema Object Privileges?

A user automatically has all object privileges for schema objects contained in his or her schema. A user can grant any object privilege on any schema object he or she owns to any other user or role. A user with the GRANT ANY OBJECT PRIVILEGE can grant or revoke any specified object privilege to another user with or without the GRANT OPTION of the GRANT statement. Otherwise, the grantee can use the privilege, but cannot grant it to other users.

For example, assume user SCOTT owns a table named t2:

```
SQL>GRANT GRANT ANY OBJECT PRIVILEGE TO u1;
SQL> CONNECT u1/u1
Connected.
SQL> GRANT SELECT ON scott.t2 TO u2;
SQL> SELECT GRANTEE, OWNER, GRANTOR, PRIVILEGE, GRANTABLE FROM DBA_TAB_PRIVS
 WHERE TABLE_NAME = 'employees';

GRANTEE                        OWNER
------------------------------ ------------------------------
GRANTOR                        PRIVILEGE                                GRA
------------------------------ ---------------------------------------- ---
U2                             SCOTT
SCOTT                          SELECT                                   NO
```

See Also: *Oracle Database SQL Reference*

Using Privileges with Synonyms

A schema object and its synonym are equivalent with respect to privileges. That is, the object privileges granted for a table, view, sequence, procedure, function, or package apply whether referencing the base object by name or by using a synonym.

For example, assume there is a table jward.emp with a synonym named jward.employee, and the user jward issues the following statement:

```
GRANT SELECT ON emp TO swilliams;
```

The user swilliams can query jward.emp by referencing the table by name or by using the synonym jward.employee:

```
SELECT * FROM jward.emp;
SELECT * FROM jward.employee;
```

If you grant object privileges on a table, view, sequence, procedure, function, or package to a synonym for the object, then the effect is the same as if no synonym were used. For example, if jward wanted to grant the SELECT privilege for the emp table to swilliams, then jward could issue either of the following statements:

```
GRANT SELECT ON emp TO swilliams;
```

```
GRANT SELECT ON employee TO swilliams;
```

If a synonym is dropped, then all grants for the underlying schema object remain in effect, even if the privileges were granted by specifying the dropped synonym.

Table Privileges

Schema object privileges for tables enable table security at the Data Manipulation Language (DML) or Data Definition Language (DDL) level of operation, as discussed in the following subsections:

- DML Operations
- DDL Operations

DML Operations

You can grant privileges to use the `DELETE`, `INSERT`, `SELECT`, and `UPDATE` DML operations on a table or view. Grant these privileges only to users and roles that need to query or manipulate data in a table.

You can restrict `INSERT` and `UPDATE` privileges for a table to specific columns of the table. With selective `INSERT`, a privileged user can insert a row with values for the selected columns. All other columns receive `NULL` or the default value of the column. With selective `UPDATE`, a user can update only specific column values of a row. Selective `INSERT` and `UPDATE` privileges are used to restrict user access to sensitive data.

For example, if you do not want data entry users to alter the `salary` column of the `employees` table, then selective `INSERT` or `UPDATE` privileges can be granted that exclude the `salary` column. Alternatively, a view that excludes the `salary` column could satisfy this need for additional security.

> **See Also:** *Oracle Database SQL Reference* for more information about DML operations

DDL Operations

The `ALTER`, `INDEX`, and `REFERENCES` privileges allow DDL operations to be performed on a table. Because these privileges allow other users to alter or create dependencies on a table, you should grant privileges conservatively.

A user attempting to perform a DDL operation on a table may need additional system or object privileges. For example, to create a trigger on a table, the user requires both the `ALTER TABLE` object privilege for the table and the `CREATE TRIGGER` system privilege.

As with the `INSERT` and `UPDATE` privileges, the `REFERENCES` privilege can be granted on specific columns of a table. The `REFERENCES` privilege enables the grantee to use the table on which the grant is made as a parent key to any foreign keys that the grantee wishes to create in his or her own tables. This action is controlled with a special privilege because the presence of foreign keys restricts the data manipulation and table alterations that can be done to the parent key. A column-specific `REFERENCES` privilege restricts the grantee to using the named columns (which, of course, must include at least one primary or unique key of the parent table).

> **See Also:** Data Integrity in *Oracle Database Concepts* for more information about primary keys, unique keys, and integrity constraints

View Privileges

A view is a presentation of data selected from one or more tables (possibly including other views). A view shows the structure of the underlying tables as well as the selected data, and can be thought of as the result of a stored query. The view contains no actual data but rather derives what it shows from the tables and views on which it is based. A view can be queried, and the data it represents can be changed. Data in a view can be updated or deleted, and new data inserted. These operations directly alter the tables on which the view is based and are subject to the integrity constraints and triggers of the base tables.

DML object privileges for tables can be applied similarly to views. Schema object privileges for a view allow various DML operations, which as noted affect the base tables from which the view is derived. These privileges are discussed in the following subsections:

- Privileges Required to Create Views
- Increasing Table Security with Views

Privileges Required to Create Views

To create a view, you must meet the following requirements:

- You must have been granted one of the following system privileges, either explicitly or through a role:
  - The `CREATE VIEW` system privilege (to create a view in your schema)
  - The `CREATE ANY VIEW` system privilege (to create a view in another user's schema)
- You must have been explicitly granted one of the following privileges:
  - The `SELECT`, `INSERT`, `UPDATE`, or `DELETE` object privileges on all base objects underlying the view
  - The `SELECT ANY TABLE`, `INSERT ANY TABLE`, `UPDATE ANY TABLE`, or `DELETE ANY TABLE` system privileges
- In addition, in order to grant other users access to your view, you must have received object privileges to the base objects with the `GRANT OPTION` clause or appropriate system privileges with the `ADMIN OPTION` clause. If you have not, then grantees cannot access your view.

See Also: *Oracle Database SQL Reference*

Increasing Table Security with Views

To use a view, you require appropriate privileges only for the view itself. You do not require privileges on base objects underlying the view.

Views add two more levels of security for tables, column-level security and value-based security:

- A view can provide access to selected columns of base tables. For example, you can define a view on the `employees` table to show only the `employee_id`, `last_name`, and `manager_id` columns:

```
CREATE VIEW employees_manager AS
    SELECT last_name, employee_id, manager_id FROM employees;
```

- A view can provide value-based security for the information in a table. A `WHERE` clause in the definition of a view displays only selected rows of base tables. Consider the following two examples:

```
CREATE VIEW lowsal AS
    SELECT * FROM employees
    WHERE salary < 10000;
```

The `LOWSAL` view allows access to all rows of the `employees` table that have a salary value less than 10000. Notice that all columns of the `employees` table are accessible in the `LOWSAL` view.

```
CREATE VIEW own_salary AS
    SELECT last_name, salary
    FROM employees
    WHERE last_name = USER;
```

In the `own_salary` view, only the rows with an `last_name` that matches the current user of the view are accessible. The `own_salary` view uses the `user` pseudocolumn, whose values always refer to the current user. This view combines both column-level security and value-based security.

Procedure Privileges

`EXECUTE` is the only **schema object privilege** for procedures, including standalone procedures and functions as well as packages. Grant this privilege only to users who need to execute a procedure or to compile another procedure that calls a desired procedure. To create and manage secure and effective use of procedure privileges, you need to understand the following subsections:

- Procedure Execution and Security Domains
- System Privileges Needed to Create or Alter a Procedure
- Packages and Package Objects

Procedure Execution and Security Domains

A user with the `EXECUTE` object privilege for a specific procedure can execute the procedure or compile a program unit that references the procedure. No run-time privilege check is made when the procedure is called. A user with the `EXECUTE ANY PROCEDURE` system privilege can execute any procedure in the database. Privileges to execute procedures can be granted to a user through roles.

The owner of a procedure, called the *definer*, must have all the necessary object privileges for referenced objects. If the owner grants to another user the right to use that procedure, then the owner object privileges for the objects referenced by the procedure apply to that user's exercise of the procedure. These are termed "definer's rights."

The user of a procedure who is not its owner is called the "invoker." Additional privileges on referenced objects are required for invoker's rights procedures, but not for definer's rights procedures.

See Also: "PL/SQL Blocks and Roles" on page 5-18

Definer's Rights

A user of a definer's rights procedure requires only the privilege to execute the procedure and no privileges on the underlying objects that the procedure accesses, because a definer's rights procedure operates under the security domain of the user

who owns the procedure, regardless of who is executing it. The owner of the procedure must have all the necessary object privileges for referenced objects. Fewer privileges have to be granted to users of a definer's rights procedure, resulting in tighter control of database access.

You can use definer's rights procedures to control access to private database objects and add a level of database security. By writing a definer's rights procedure and granting only `EXECUTE` privilege to a user, the user can be forced to access the referenced objects only through the procedure.

At run-time, the privileges of the owner of a definer's rights stored procedure are always checked before the procedure is executed. If a necessary privilege on a referenced object has been revoked from the owner of a definer's rights procedure, then the procedure cannot be executed by the owner or any other user.

Note: Trigger execution follows the same patterns as definer's rights procedures. The user executes a SQL statement, which that user is privileged to execute. As a result of the SQL statement, a trigger is fired. The statements within the triggered action temporarily execute under the security domain of the user that owns the trigger.

See Also: "Triggers" in *Oracle Database Concepts*

Invoker's Rights

An invoker's rights procedure executes with all of the invoker's privileges. Roles are enabled unless the invoker's rights procedure was called directly or indirectly by a definer's rights procedure. A user of an invoker's rights procedure needs privileges (either directly or through a role) on objects that the procedure accesses through external references that are resolved in the invoker's schema.

The invoker needs privileges at run-time to access program references embedded in DML statements or dynamic SQL statements, because they are effectively recompiled at run-time.

For all other external references, such as direct PL/SQL function calls, the owner's privileges are checked at compile time, and no run-time check is made. Therefore, the user of an invoker's rights procedure needs no privileges on external references outside DML or dynamic SQL statements. Alternatively, the developer of an invoker's rights procedure only needs to grant privileges on the procedure itself, not on all objects directly referenced by the invoker's rights procedure.

You can create a software bundle that consists of multiple program units, some with definer's rights and others with invoker's rights, and restrict the program entry points *(controlled step-in)*. A user who has the privilege to execute an entry-point procedure can also execute internal program units indirectly, but cannot directly call the internal programs.

See Also:

- "Introduction to Fine-Grained Access Control" on page 14-4
- *PL/SQL Packages and Types Reference* for detailed documentation of the Oracle supplied packages

System Privileges Needed to Create or Alter a Procedure

To create a procedure, a user must have the CREATE PROCEDURE or CREATE ANY PROCEDURE system privilege. To alter a procedure, that is, to manually recompile a procedure, a user must own the procedure or have the ALTER ANY PROCEDURE system privilege.

The user who owns the procedure also must have privileges for schema objects referenced in the procedure body. To create a procedure, you must have been explicitly granted the necessary privileges (system or object) on all objects referenced by the procedure. You cannot have obtained the required privileges through roles. This includes the EXECUTE privilege for any procedures that are called inside the procedure being created.

> **Note:** Triggers also require that privileges to referenced objects be granted explicitly to the trigger owner. Anonymous PL/SQL blocks can use any privilege, whether the privilege is granted explicitly or through a role.

Packages and Package Objects

A user with the EXECUTE object privilege for a package can execute any public procedure or function in the package and access or modify the value of any public package variable. Specific EXECUTE privileges cannot be granted for individual constructs in a package. Therefore, you may find it useful to consider two alternatives for establishing security when developing procedures, functions, and packages for a database application. These alternatives are described in the following examples.

Packages and Package Objects: Example 1

This example shows four procedures created in the bodies of two packages.

```
CREATE PACKAGE BODY hire_fire AS
  PROCEDURE hire(...) IS
    BEGIN
      INSERT INTO employees . . .
    END hire;
  PROCEDURE fire(...) IS
    BEGIN
      DELETE FROM employees . . .
    END fire;
END hire_fire;

CREATE PACKAGE BODY raise_bonus AS
  PROCEDURE give_raise(...) IS
    BEGIN
      UPDATE employees SET salary = . . .
    END give_raise;
  PROCEDURE give_bonus(...) IS
    BEGIN
      UPDATE employees SET bonus = . . .
    END give_bonus;
END raise_bonus;
```

Access to execute the procedures is given by granting the EXECUTE privilege for the package by using the following statements:

```
GRANT EXECUTE ON hire_fire TO big_bosses;
GRANT EXECUTE ON raise_bonus TO little_bosses;
```

Note: Granting EXECUTE privilege granted for a package provides uniform access to all package objects.

Packages and Package Objects: Example 2

This example shows four procedure definitions within the body of a single package. Two additional standalone procedures and a package are created specifically to provide access to the procedures defined in the main package.

```
CREATE PACKAGE BODY employee_changes AS
  PROCEDURE change_salary(...) IS BEGIN ... END;
  PROCEDURE change_bonus(...) IS BEGIN ... END;
  PROCEDURE insert_employee(...) IS BEGIN ... END;
  PROCEDURE delete_employee(...) IS BEGIN ... END;
END employee_changes;

CREATE PROCEDURE hire
  BEGIN
    employee_changes.insert_employee(...)
  END hire;

CREATE PROCEDURE fire
  BEGIN
    employee_changes.delete_employee(...)
  END fire;

PACKAGE raise_bonus IS
  PROCEDURE give_raise(...) AS
    BEGIN
      employee_changes.change_salary(...)
    END give_raise;

  PROCEDURE give_bonus(...)
    BEGIN
      employee_changes.change_bonus(...)
    END give_bonus;
```

Using this method, the procedures that actually do the work (the procedures in the employee_changes package) are defined in a single package and can share declared global variables, cursors, on so on. By declaring top-level procedures, hire and fire, and an additional package, raise_bonus, you can grant selective EXECUTE privileges on procedures in the main package:

```
GRANT EXECUTE ON hire, fire TO big_bosses;
GRANT EXECUTE ON raise_bonus TO little_bosses;
```

Type Privileges

The following subsections describe the use of privileges for types, methods, and objects:

- System Privileges for Named Types
- Object Privileges
- Method Execution Model
- Privileges Required to Create Types and Tables Using Types
- Example of Privileges for Creating Types and Tables Using Types

- Privileges on Type Access and Object Access
- Type Dependencies

System Privileges for Named Types

Oracle defines system privileges shown in Table 5–1 for named types (object types, VARRAYs, and nested tables):

Table 5–1 System Privileges for Named Types

Privilege	Allows you to...
CREATE TYPE	Create named types in your own schemas
CREATE ANY TYPE	Create a named type in any schema
ALTER ANY TYPE	Alter a named type in any schema
DROP ANY TYPE	Drop a named type in any schema
EXECUTE ANY TYPE	Use and reference a named type in any schema

The RESOURCE role includes the CREATE TYPE system privilege. The DBA role includes all of these privileges.

Object Privileges

The only object privilege that applies to named types is EXECUTE. If the EXECUTE privilege exists on a named type, then a user can use the named type to:

- Define a table
- Define a column in a relational table
- Declare a variable or parameter of the named type

The EXECUTE privilege permits a user to invoke the methods in the type, including the type constructor. This is similar to EXECUTE privilege on a stored PL/SQL procedure.

Method Execution Model

Method execution is the same as any other stored PL/SQL procedure.

See Also: "Procedure Privileges" on page 5-7

Privileges Required to Create Types and Tables Using Types

To create a type, you must meet the following requirements:

- You must have the CREATE TYPE system privilege to create a type in your schema or the CREATE ANY TYPE system privilege to create a type in the schema of another user. These privileges can be acquired explicitly or through a role.
- The owner of the type must be explicitly granted the EXECUTE object privileges to access all other types referenced within the definition of the type, or have been granted the EXECUTE ANY TYPE system privilege. The owner cannot have obtained the required privileges through roles.
- If the type owner intends to grant access to the type to other users, then the owner must have received the EXECUTE privileges to the referenced types with the GRANT OPTION or the EXECUTE ANY TYPE system privilege with the ADMIN OPTION. If not, then the type owner has insufficient privileges to grant access on the type to other users.

To create a table using types, you must meet the requirements for creating a table and the following additional requirements:

- The owner of the table must have been explicitly granted the EXECUTE object privileges to access all types referenced by the table, or have been granted the EXECUTE ANY TYPE system privilege. The owner cannot have obtained the required privileges through roles.
- If the table owner intends to grant access to the table to other users, then the owner must have received the EXECUTE privileges to the referenced types with the GRANT OPTION or the EXECUTE ANY TYPE system privilege with the ADMIN OPTION. If not, then the table owner has insufficient privileges to grant access on the type to other users.

> **See Also:** "Table Privileges" on page 5-5 for the requirements for creating a table

Example of Privileges for Creating Types and Tables Using Types

Assume that three users exist with the CONNECT and RESOURCE roles:

- user1
- user2
- user3

User1 performs the following DDL in his schema:

```
CREATE TYPE type1 AS OBJECT (
  attr1 NUMBER);

CREATE TYPE type2 AS OBJECT (
  attr2 NUMBER);

GRANT EXECUTE ON type1 TO user2;
GRANT EXECUTE ON type2 TO user2 WITH GRANT OPTION;
```

User2 performs the following DDL in his schema:

```
CREATE TABLE tab1 OF user1.type1;
CREATE TYPE type3 AS OBJECT (
  attr3 user1.type2);
CREATE TABLE tab2 (
  col1 user1.type2);
```

The following statements succeed because user2 has EXECUTE privilege on user1.type2 with the GRANT OPTION:

```
GRANT EXECUTE ON type3 TO user3;
GRANT SELECT on tab2 TO user3;
```

However, the following grant fails because user2 does not have EXECUTE privilege on user1.type1 with the GRANT OPTION:

```
GRANT SELECT ON tab1 TO user3;
```

User3 can successfully perform the following statements:

```
CREATE TYPE type4 AS OBJECT (
  attr4 user2.type3);
CREATE TABLE tab3 OF type4;
```

Note: Customers should discontinue using the CONNECT and RESOURCE roles, as they will be deprecated in future Oracle Database releases. The CONNECT role presently retains only the CREATE SESSION privilege.

Privileges on Type Access and Object Access

Existing column-level and table-level privileges for DML statements apply to both column objects and row objects. Oracle defines the privileges shown in Table 5–2 for object tables.

Table 5–2 Privileges for Object Tables

Privilege	Allows you to...
SELECT	Access an object and its attributes from the table
UPDATE	Modify the attributes of the objects that make up the rows in the table
INSERT	Create new objects in the table
DELETE	Delete rows

Similar table privileges and column privileges apply to column objects. Retrieving instances does not in itself reveal type information. However, clients must access named type information in order to interpret the type instance images. When a client requests such type information, Oracle Database checks for EXECUTE privilege on the type.

Consider the following schema:

```
CREATE TYPE emp_type (
    eno NUMBER, ename CHAR(31), eaddr addr_t);
CREATE TABLE emp OF emp_t;
```

In addition, consider the following two queries:

```
SELECT VALUE(emp) FROM emp;
SELECT eno, ename FROM emp;
```

For either query, Oracle Database checks the SELECT privilege of the user for the emp table. For the first query, the user needs to obtain the emp_type type information to interpret the data. When the query accesses the emp_type type, Oracle checks the EXECUTE privilege of the user.

Execution of the second query, however, does not involve named types, so Oracle does not check type privileges.

In addition, by using the schema from the previous section, user3 can perform the following queries:

```
SELECT tab1.col1.attr2 FROM user2.tab1 tab1;
SELECT attr4.attr3.attr2 FROM tab3;
```

Note that in both SELECT statements, user3 does not have explicit privileges on the underlying types, but the statement succeeds because the type and table owners have the necessary privileges with the GRANT OPTION.

Oracle Database checks privileges on the following events and returns an error if the client does not have the privilege for the action:

- Pinning an object in the object cache using its REF value causes Oracle Database to check for the SELECT privilege on the containing object table.
- Modifying an existing object or flushing an object from the object cache causes Oracle Database to check for the UPDATE privilege on the destination object table.
- Flushing a new object causes Oracle Database to check for the INSERT privilege on the destination object table.
- Deleting an object causes Oracle Database to check for the DELETE privilege on the destination table.
- Pinning an object of a named type causes Oracle to check EXECUTE privilege on the object.

Modifying the attributes of an object in a client third-generation language application causes Oracle Database to update the entire object. Therefore, the user needs UPDATE privilege on the object table. Having the UPDATE privilege on only certain columns of the object table is not sufficient, even if the application only modifies attributes corresponding to those columns. Therefore, Oracle Database does not support column-level privileges for object tables.

Type Dependencies

As with stored objects, such as procedures and tables, types being referenced by other objects are called dependencies. There are some special issues for types that tables depend on. Because a table contains data that relies on the type definition for access, any change to the type causes all stored data to become inaccessible. Changes that can cause this are when necessary privileges required by the type are revoked or the type or dependent types are dropped. If either of these actions occur, then the table becomes invalid and cannot be accessed.

A table that is invalid because of missing privileges can automatically become valid and accessible if the required privileges are granted again. A table that is invalid because a dependent type has been dropped can never be accessed again, and the only permissible action is to drop the table.

Because of the severe effects that revoking a privilege on a type or dropping a type can cause, the SQL statements REVOKE and DROP TYPE, by default, implement a restrict semantics. This means that if the named type in either statement has table or type dependents, then an error is received and the statement aborts. However, if the FORCE clause for either statement is used, then the statement always succeeds. If there are depended-upon tables, then they are invalidated.

> **See Also:** *Oracle Database Reference* for details about using the REVOKE, DROP TYPE, and FORCE clauses

Introduction to Roles

Managing and controlling privileges is made easier by using **roles**, which are named groups of related privileges that you grant as a group to users or other roles. Within a database, each role name must be unique, different from all user names and all other role names. Unlike schema objects, roles are not contained in any schema. Therefore, a user who creates a role can be dropped with no effect on the role.

Roles are designed to ease the administration of an end-user system and schema object privileges and are often maintained in Oracle Internet Directory. However, roles are not meant to be used by application developers, because the privileges to access schema objects within stored programmatic constructs need to be granted directly.

The effective management of roles is discussed in the following subsections:

Authentication Considerations in These Topical Areas	Links to Relevant Subsection
Why Roles Are Advantageous	Properties of Roles
How Roles are Typically Used	Common Uses of Roles
How Users Get Roles (or Role Restrictions	Granting and Revoking Roles
How Roles Affect The Scope of a User's Privileges	Security Domains of Roles and Users
How Roles Work in PL/SQL Blocks	PL/SQL Blocks and Roles
How Roles Aid or Restrict DDL Usage	DDL Statements and Roles
What Roles are Predefined in Oracle	Predefined Roles
How Can Operating Systems Aid Roles	Operating System and Roles
How Roles Work in a Remote Session	Roles in a Distributed Environment
How Secure Application Roles Are Created and Used	Secure Application Roles

Properties of Roles

Table 5–3 discusses the properties of roles that enable easier privilege management within a database:

Table 5–3 Properties of Roles and Their Description

Property	Description
Reduced privilege administration	Rather than granting the same set of privileges explicitly to several users, you can grant the privileges for a group of related users to a role, and then only the role needs to be granted to each member of the group.
Dynamic privilege management	If the privileges of a group must change, then only the privileges of the role need to be modified. The security domains of all users granted the group's role automatically reflect the changes made to the role.
Selective availability of privileges	You can selectively enable or disable the roles granted to a user. This allows specific control of a user's privileges in any given situation.
Application awareness	The data dictionary records which roles exist, so you can design applications to query the dictionary and automatically enable (or disable) selective roles when a user attempts to execute the application by way of a given user name.
Application-specific security	You can protect role use with a password. Applications can be created specifically to enable a role when supplied the correct password. Users cannot enable the role if they do not know the password.

Database administrators often create roles for a database application. The DBA grants a secure application role all privileges necessary to run the application. The DBA then grants the secure application role to other roles or users. An application can have several different roles, each granted a different set of privileges that allow for more or less data access while using the application.

The DBA can create a role with a password to prevent unauthorized use of the privileges granted to the role. Typically, an application is designed so that when it starts, it enables the proper role. As a result, an application user does not need to know the password for an application role.

See Also:

- "DDL Statements and Roles" on page 5-18 for information about restrictions for procedures
- *Oracle Database Application Developer's Guide - Fundamentals* for instructions for enabling roles from an application

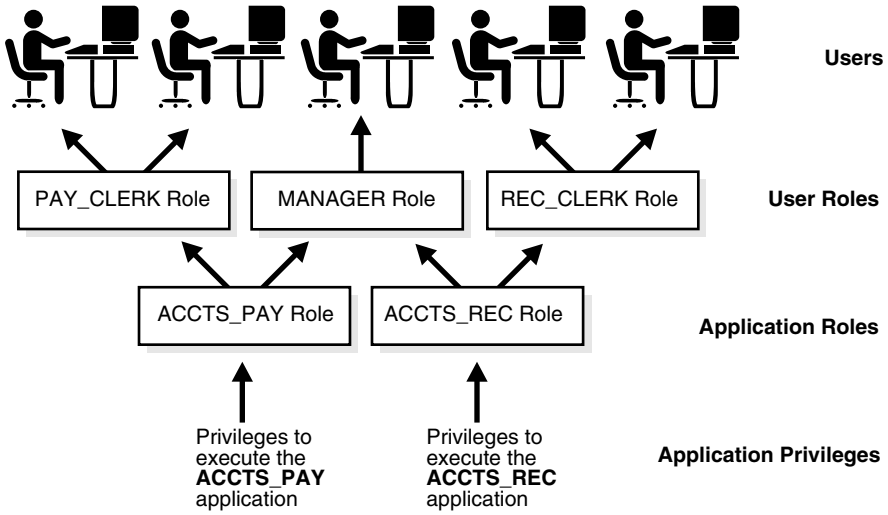
Common Uses of Roles

In general, you create a role to serve one of two purposes:

- To manage the privileges for a database application (Application Roles)
- To manage the privileges for a user group (User Roles)

Figure 5–1 and the sections that follow describe the two uses of roles.

Figure 5–1 Common Uses for Roles



Application Roles

You grant an application role all privileges necessary to run a given database application. Then, you grant the secure application role to other roles or to specific users. An application can have several different roles, with each role assigned a different set of privileges that allow for more or less data access while using the application.

Note: Grants of password protection or application roles to another role will not be allowed in future Oracle Database releases.

User Roles

You create a user role for a group of database users with common privilege requirements. You manage user privileges by granting secure application roles and privileges to the user role and then granting the user role to appropriate users.

Granting and Revoking Roles

System or schema object privileges can be granted to a role, and any role can be granted to any database user or to another role (but not to itself). However, a role cannot be granted circularly, that is, a role `X` cannot be granted to role `Y` if role `Y` has previously been granted to role `X`.

To provide selective availability of privileges, Oracle Database allows applications and users to enable and disable roles. Each role granted to a user is, at any given time, either enabled or disabled. The security domain of a user includes the privileges of all roles currently enabled for the user and excludes the privileges of any roles currently disabled for the user.

A role granted to a role is called an indirectly granted role. It can be explicitly enabled or disabled for a user. However, whenever you enable a role that contains other roles, you implicitly enable all indirectly granted roles of the directly granted role.

You grant roles to (or revoke roles from) users or other roles by using either of the following methods:

- Oracle Enterprise Manager 10g Database Control
- The SQL statements, `GRANT` and `REVOKE`

Privileges are granted to and revoked from roles using the same options. Roles can also be granted to and revoked from users using the operating system that runs Oracle, or through network services.

> **See Also:** For more information about
>
> - Database Control, see *Oracle Database 2 Day DBA*
> - Modifying users, roles, or privileges with the Database Control, see the Enterprise Manager online help

Who Can Grant or Revoke Roles?

Any user with the `GRANT ANY ROLE` system privilege can grant or revoke any role except a global role to or from other users or roles of the database. You should grant this system privilege conservatively because it is very powerful.

Any user granted a role with the `ADMIN OPTION` can grant or revoke that role to or from other users or roles of the database. This option allows administrative powers for roles on a selective basis.

> **See Also:** *Oracle Database Administrator's Guide* for information about global roles

Security Domains of Roles and Users

Each role and user has its own unique security domain. The security domain of a role includes the privileges granted to the role plus those privileges granted to any roles that are granted to the role.

The security domain of a user includes privileges on all schema objects in the corresponding schema, the privileges granted to the user, and the privileges of roles

granted to the user that are **currently enabled**. (A role can be simultaneously enabled for one user and disabled for another.) This domain also includes the privileges and roles granted to the user group `PUBLIC`.

PL/SQL Blocks and Roles

The use of roles in a PL/SQL block depends on whether it is an anonymous block or a named block (stored procedure, function, or trigger), and whether it executes with definer's rights or invoker's rights.

Named Blocks with Definer's Rights

All roles are disabled in any named PL/SQL block (stored procedure, function, or trigger) that executes with definer's rights. Roles are not used for privilege checking and you cannot set roles within a definer's rights procedure.

The `SESSION_ROLES` view shows all roles that are currently enabled. If a named PL/SQL block that executes with definer's rights queries `SESSION_ROLES`, then the query does not return any rows.

See Also: *Oracle Database Reference*

Anonymous Blocks with Invoker's Rights

Named PL/SQL blocks that execute with invoker's rights and anonymous PL/SQL blocks are executed based on privileges granted through enabled roles. Current roles are used for privilege checking within an invoker's rights PL/SQL block, and you can use dynamic SQL to set a role in the session.

See Also:

- *PL/SQL User's Guide and Reference* for an explanation of invoker's and definer's rights
- "Dynamic SQL in PL/SQL" in *Oracle Database Concepts*

DDL Statements and Roles

A user requires one or more privileges to successfully execute a DDL statement, depending on the statement. For example, to create a table, the user must have the `CREATE TABLE` or `CREATE ANY TABLE` system privilege. To create a view of a table that belongs to another user, the creator requires the `CREATE VIEW` or `CREATE ANY VIEW` system privilege and either the `SELECT` *`object`* privilege for the table or the `SELECT ANY TABLE` system privilege.

Oracle Database avoids the dependencies on privileges received by way of roles by restricting the use of specific privileges in certain DDL statements. The following rules outline these privilege restrictions concerning DDL statements:

- All system privileges and schema object privileges that permit a user to perform a DDL operation are usable when received through a role. For example:
  - System Privileges: `CREATE TABLE`, `CREATE VIEW`, and `CREATE PROCEDURE` privileges
  - Schema Object Privileges: `ALTER` and `INDEX` privileges for a table

  Exception: The `REFERENCES` object privilege for a table cannot be used to define the foreign key of a table if the privilege is received through a role.

- All system privileges and object privileges that allow a user to perform a DML operation that is required to issue a DDL statement are *not* usable when received through a role. For example:
  - A user who receives the SELECT ANY TABLE system privilege or the SELECT *object* privilege for a table through a role can use neither privilege to create a view on a table that belongs to another user.

The following example further clarifies the permitted and restricted uses of privileges received through roles.

Assume that a user is:

- Granted a role that has the CREATE VIEW system privilege
- Granted a role that has the SELECT *object* privilege for the employees table, but the user is indirectly granted the SELECT *object* privilege for the employees table
- Directly granted the SELECT *object* privilege for the departments table

Given these directly and indirectly granted privileges:

- The user can issue SELECT statements on both the employees and departments tables.
- Although the user has both the CREATE VIEW and SELECT privilege for the employees table through a role, the user cannot create a usable view on the employees table, because the SELECT *object* privilege for the employees table was granted through a role. Any views created will produce errors when accessed.
- The user can create a view on the departments table, because the user has the CREATE VIEW privilege through a role and the SELECT privilege for the departments table directly.

Predefined Roles

The following roles are defined automatically for Oracle Database:

- CONNECT
- RESOURCE
- DBA
- EXP_FULL_DATABASE
- IMP_FULL_DATABASE

These roles are provided for backward compatibility to earlier versions of Oracle Database and can be modified in the same manner as any other role in an Oracle database.

> **Note:** Each installation should create its own roles and assign only those privileges that are needed, thus retaining detailed control of the privileges in use. This process also removes any need to adjust existing roles, privileges, or procedures whenever Oracle Database changes or removes roles that Oracle Database defines. For example, the CONNECT role now has only one privilege: CREATE SESSION. Both CONNECT and RESOURCE roles will be deprecated in future Oracle versions.

Operating System and Roles

In some environments, you can administer database security using the operating system. The operating system can be used to manage the granting (and revoking) of database roles and to manage their password authentication. This capability is not available on all operating systems.

> **See Also:** Your operating system specific Oracle documentation for details on managing roles through the operating system

Roles in a Distributed Environment

When you use roles in a distributed database environment, you must ensure that all needed roles are set as the default roles for a distributed (remote) session. These roles cannot be enabled when you connect to a remote database from within a local database session. For example, you cannot execute a remote procedure that attempts to enable a role at the remote site.

> **See Also:** *Oracle Database Heterogeneous Connectivity Administrator's Guide*

Secure Application Roles

Oracle Database provides secure application roles, which are roles that can only be enabled by authorized PL/SQL packages. This mechanism restricts the enabling of such roles to the invoking application.

Security is strengthened when passwords are not embedded in application source code or stored in a table. Instead, a secure application role can be created, specifying which PL/SQL package is authorized to enable the role. Package identity is used to determine whether privileges are sufficient to enable the roles. Before enabling the role, the application can perform authentication and customized authorization, such as checking whether the user has connected through a proxy.

> **Note:** Because of the restriction that users cannot change the security domain inside definer's right procedures, secure application roles can only be enabled inside invoker's right procedures.

Creation of Secure Application Roles

Secure application roles are created by using the statement `CREATE ROLE ... IDENTIFIED USING`. Here is an example:

```
CREATE ROLE admin_role IDENTIFIED USING hr.admin;
```

This statement indicates the following:

- The role `admin_role` to be created is a secure application role.
- The role can only be enabled by modules defined inside the PL/SQL package `hr.admin`.

You must have the `CREATE ROLE` system privilege to execute this statement.

When such a role is assigned to a user, it becomes a default role for that user, which is automatically enabled at login without resorting to the package. A user with a default role does not have to be authenticated in any way to use the role. For example, the password for the role is not requested or required.

To restrict the role solely to the use specified by the `IDENTIFIED USING` clause, you can take either of the following actions:

- Immediately after granting such a role to a user, issue an `ALTER USER` statement with the clause `DEFAULT ROLE ALL EXCEPT` *`role`*, substituting the application role for *`role`*. Then *`role`* can only be used by applications executing the authorized package.
- When assigning roles, use GRANT ALL EXCEPT *`role`*.

Roles that are enabled inside an *invoker's right* procedure remain in effect even after the procedure exits. Therefore, you can have a dedicated procedure that deals with enabling the role for the rest of the session to use.

See Also:

- *Oracle Database SQL Reference*
- *PL/SQL User's Guide and Reference*
- *PL/SQL Packages and Types Reference*
- *Oracle Database Application Developer's Guide - Fundamentals*

User Resource Limits

You can set limits on the amount of various system resources available to each user as part of the security domain of that user. By doing so, you can prevent the uncontrolled consumption of valuable system resources such as CPU time.

This resource limit feature is very useful in large, multiuser systems, where system resources are very expensive. Excessive consumption of these resources by one or more users can detrimentally affect the other users of the database. In single-user or small-scale multiuser database systems, the system resource feature is not as important, because user consumption of system resources is less likely to have detrimental impact.

You manage user resource limits by means of Database Resource Manager. You can set password management preferences using profiles, either set individually or using a default profile for many users. Each Oracle database can have an unlimited number of profiles. Oracle allows the security administrator to enable or disable the enforcement of profile resource limits universally.

See Also:

- For resource management, see *Database Resource Manager*
- For passwords, see Password Management Policy on page 7-9

Setting resource limits causes a slight performance degradation when users create sessions, because Oracle loads all resource limit data for each user upon each connection to the database.

See Also: *Oracle Database Administrator's Guide* for information about security administrators

Resource limits and profiles are discussed in the following sections:

- Types of System Resources and Limits
- Profiles

Types of System Resources and Limits

Oracle Database can limit the use of several types of system resources, including CPU time and logical reads. In general, you can control each of these resources at the session level, call level, or both, as discussed in the following subsections:

- Session Level
- Call Level
- CPU Time
- Logical Reads
- Limiting Other Resources

Session Level

Each time a user connects to a database, a session is created. Each session consumes CPU time and memory on the computer that runs Oracle Database. You can set several resource limits at the session level.

If a user exceeds a session-level resource limit, then Oracle terminates (rolls back) the current statement and returns a message indicating that the session limit has been reached. At this point, all previous statements in the current transaction are intact, and the only operations the user can perform are `COMMIT`, `ROLLBACK`, or disconnect (in this case, the current transaction is committed). All other operations produce an error. Even after the transaction is committed or rolled back, the user can accomplish no more work during the current session.

Call Level

Each time a SQL statement is executed, several steps are taken to process the statement. During this processing, several calls are made to the database as a part of the different execution phases. To prevent any one call from using the system excessively, Oracle Database lets you set several resource limits at the call level.

If a user exceeds a call-level resource limit, then Oracle Database halts the processing of the statement, rolls back the statement, and returns an error. However, all previous statements of the current transaction remain intact, and the user session remains connected.

CPU Time

When SQL statements and other types of calls are made to Oracle Database, a certain amount of CPU time is necessary to process the call. Average calls require a small amount of CPU time. However, a SQL statement involving a large amount of data or a runaway query can potentially consume a large amount of CPU time, reducing CPU time available for other processing.

To prevent uncontrolled use of CPU time, you can set fixed or dynamic limits on the CPU time for each call and the total amount of CPU time used for Oracle calls during a session. The limits are set and measured in CPU one-hundredth seconds (0.01 seconds) used by a call or a session.

Logical Reads

I/O is one of the most expensive operations in a database system. SQL statements that are I/O-intensive can monopolize memory and disk use and cause other database operations to compete for these resources.

To prevent single sources of excessive I/O, Oracle Database lets you limit the logical data block reads for each call and for each session. Logical data block reads include data block reads from both memory and disk. The limits are set and measured in number of block reads performed by a call or during a session.

Limiting Other Resources

Oracle Database also provides for limiting several other resources at the session level:

- You can limit the number of concurrent sessions for each user. Each user can create only up to a predefined number of concurrent sessions.
- You can limit the idle time for a session. If the time between calls in a session reaches the idle time limit, then the current transaction is rolled back, the session is aborted, and the resources of the session are returned to the system. The next call receives an error that indicates that the user is no longer connected to the instance. This limit is set as a number of elapsed minutes.

  Note: Shortly after a session is aborted because it has exceeded an idle time limit, the process monitor (PMON) background process cleans up after the aborted session. Until PMON completes this process, the aborted session is still counted in any session or user resource limit.

- You can limit the elapsed connect time for each session. If the duration of a session exceeds the elapsed time limit, then the current transaction is rolled back, the session is dropped, and the resources of the session are returned to the system. This limit is set as a number of elapsed minutes.

  Note: Oracle Database does not constantly monitor the elapsed idle time or elapsed connection time. Doing so would reduce system performance. Instead, it checks every few minutes. Therefore, a session can exceed this limit slightly (for example, by five minutes) before Oracle enforces the limit and aborts the session.

- You can limit the amount of private System Global Area (SGA) space (used for private SQL areas) for a session. This limit is only important in systems that use the shared server configuration. Otherwise, private SQL areas are located in the Program Global Area (PGA). This limit is set as a number of bytes of memory in the SGA of an instance. Use the characters **K** or **M** to specify kilobytes or megabytes.

  See Also: For instructions on enabling or disabling resource limits:

  - Viewing Information About Database Users and Profiles on page 11-7
  - Managing User Roles on page 11-15
  - *Oracle Database Administrator's Guide*

Profiles

In general, the word **profile** refers to a collection of attributes that apply to a user, enabling a single point of reference for any of multiple users that share those exact attributes. User profiles in Oracle Internet Directory contain a wide range of attributes pertinent to directory usage and authentication for each user. Similarly, profiles in Oracle Label Security contain attributes useful in label security user administration and operations management. Profile attributes can include restrictions on system resources, but for that purpose Database Resource Manager is preferred.

See Also:

- For resources, see discussion of the *Database Resource Manager* in the *Oracle Database Administrator's Guide*
- For viewing resource information, see Viewing Information About Database Users and Profiles on page 11-7
- For password policies, see Password Management Policy in Chapter 7, "Security Policies"

Determining Values for Resource Limits

Before creating profiles and setting the resource limits associated with them, you should determine appropriate values for each resource limit. You can base these values on the type of operations a typical user performs. For example, if one class of user does not normally perform a high number of logical data block reads, then set the `LOGICAL_READS_PER_SESSION` and `LOGICAL_READS_PER_CALL` limits conservatively.

Usually, the best way to determine the appropriate resource limit values for a given user profile is to gather historical information about each type of resource usage. For example, the database or security administrator can use the `AUDIT SESSION` clause to gather information about the limits `CONNECT_TIME`, `LOGICAL_READS_PER_SESSION`, and `LOGICAL_READS_PER_CALL`.

You can gather statistics for other limits using the Monitor feature of Oracle Enterprise Manager (or SQL*Plus), specifically the Statistics monitor.

See Also:

- Chapter 8, "Database Auditing: Security Considerations"
- For more information about the Database Control, see *Oracle Database 2 Day DBA*
- For information about the Monitor feature, see the Enterprise Manager online help

6

Access Control on Tables, Views, Synonyms, or Rows

The authentication processes described in Chapter 4 validate the identities of the entities using your networks, databases, and applications. The authorization processes described in Chapter 5 provide limits to their access and actions, limits that are linked to their identities and roles.

This chapter describes restrictions associated not with users, but with the objects they access. This provides protection to objects regardless of the entity who seeks, by whatever means, to access or alter them.

You can provide object protection using object-level privileges and views, as well as by designing and using policies to restrict access to specific tables, views, synonyms, or rows. This level of control, which enables you to use application context with fine-grained access control, is called Virtual Private Database (VPD). Such policies call functions that you design to specify dynamic predicates that establish the restrictions. You can also group established policies by applying a policy group to a particular application.

Having established such protection, you need to be notified when it is threatened or breached. Auditing capabilities enable you to receive notifications of activities you want watched, and to investigate in advance of or in response to being notified. Given notification, you can strengthen your defense and deal with the consequences of inappropriate actions and the entities that caused them. Oracle auditing facilities are introduced in Chapter 8, "Database Auditing: Security Considerations" and described in detail in Chapter 12, "Configuring and Administering Auditing".

This chapter describes Oracle access control capabilities in the following sections:

- Introduction to Views
- Fine-Grained Access Control
- Security Followup: Auditing and Prevention

See Also:

- Regarding policies, see Chapter 7, "Security Policies"
- Regarding application development considerations on policies, see Chapter 13, "Introducing Database Security for Application Developers" and Chapter 15, "Implementing Application Context and Fine-Grained Access Control"
- Regarding fine-grained access control, see Chapter 14, "Using Virtual Private Database to Implement Application Security Policies", about developing applications to configure and administer such controls
- Regarding security conditions for auditing, see Chapter 8, "Database Auditing: Security Considerations"
- Regarding how to configure and administer auditing features and mechanisms for both standard users and DBAs, see Chapter 12, "Configuring and Administering Auditing"

Introduction to Views

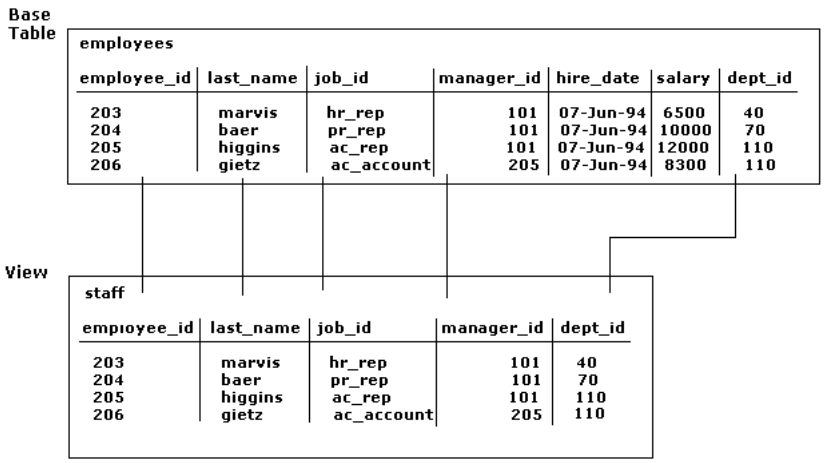
A view is a presentation of data selected from one or more tables (possibly including other views). In addition to showing the selected data, a view also shows the structure of the underlying tables, and can be thought of as the result of a stored query.

The view contains no actual data but rather derives what it shows from the tables and views on which it is based. A view can be queried, and the data it represents can be changed. Data in a view can be updated or deleted, and new data inserted. These operations directly alter the tables on which the view is based and are subject to the integrity constraints and triggers of the base tables.

For example, a base table of all employee data may have several columns and numerous rows of information. If you want a certain set of users to see only specific columns, then you can create a view of that table, containing only the allowable columns. You can then grant other users access to the new view, while disallowing access to the base table.

Figure 6–1 shows an example of a view called staff derived from the base table employees. Notice that the view shows only five of the columns in the base table.

Figure 6–1 An Example of a View



A **schema object privilege** is a privilege or right to perform a particular action on a specific schema object. Different object privileges are available for different types of schema objects. Privileges related to views are discussed in Chapter 5 under the section titled "View Privileges". Some schema objects, such as clusters, indexes, triggers, and database links, do not have associated object privileges. Their use is controlled with system privileges. For example, to alter a cluster, a user must own the cluster or have the `ALTER ANY CLUSTER` system privilege.

See Also:

- All these privileges, including those for tables, views, procedures, types and more, are introduced in Chapter 5 under the section titled "Introduction to Privileges".
- The tools and processes for managing these security facilities are discussed in Chapter 11, "Administering User Privileges, Roles, and Profiles".

In some circumstances, a finer level of access control than provided by views is needed for tables or rows and the possible actions on them, sometimes associated with particular applications. When such controls are needed, Oracle fine-grained access control capabilities can be used as described in the next section.

Fine-Grained Access Control

Fine-grained access control enables you to use functions to implement security policies and to associate those security policies with tables, views, or synonyms. The database server automatically enforces your security policies, no matter how the data is accessed, including, for example, through an application by ad hoc queries.

See Also: Using application context with fine-grained access control is called Virtual Private Database, or VPD. See these references:

- Chapter 14, "Using Virtual Private Database to Implement Application Security Policies"
- Chapter 15, "Implementing Application Context and Fine-Grained Access Control"

Fine-grained access control enables you to use all of the following capabilities:

- Limit access at the row level by using different policies for `SELECT`, `INSERT`, `UPDATE`, and `DELETE`.
- Use security policies only where you need them (for example, on salary information).
- Invoke a policy only if a particular column is referenced.
- Restrict access using a combination of row-level and column-level controls, by applying a VPD policy to a view.
- Have some policies that are *always* applied, called static policies, and others that can change during execution, called dynamic policies (see Application Context on page 6-5).
- Use more than one policy for each table, including building on top of base policies in packaged applications.

- Distinguish policies between different applications by using *policy groups*. Each policy group is a set of policies that belong to an application.
- Distinguish and control the use of `INDEX` in row level security policies.
- Designate an application context, called a *driving context*, to indicate the policy group in effect. When tables, views, or synonyms are accessed, the fine-grained access control engine looks up the driving context to determine the policy group in effect and enforces all the associated policies that belong to that policy group.

The PL/SQL package `DBMS_RLS` let you administer your security policies. Using this package, you can add, drop, enable, disable, and refresh the policies (or policy groups) you create.

See Also:

- The DBMS_RLS chapter in *PL/SQL Packages and Types Reference* for information about package implementation
- *Oracle Database Application Developer's Guide - Fundamentals* for information and examples on establishing security policies

Using application context with fine-grained access control is called VPD. See the following references for more information:

- Application Context on page 6-5
- Chapter 14, "Using Virtual Private Database to Implement Application Security Policies"
- Chapter 15, "Implementing Application Context and Fine-Grained Access Control"

The following subsections describe how fine-grained access control works:

- Dynamic Predicates
- Application Context
- Dynamic Contexts

Dynamic Predicates

A dynamic predicate for a table, view, or synonym is generated by a PL/SQL function, which you write and associate with a security policy through a PL/SQL interface. Dynamic predicates are acquired at statement parse time, when the base table or view is referenced in a query using `SELECT` or a DML statement.

The function or package that implements the security policy you create returns a predicate (a `WHERE` condition). This predicate controls access according to the policy you specify. Rewritten queries are fully optimized and shareable.

Here is an example of such a policy:

```
DBMS_RLS.ADD_POLICY (
   'hr', 'employees', 'emp_policy', 'hr', 'emp_sec', 'select');
```

Whenever the `EMPLOYEES` table, under the HR schema, is referenced in a query or subquery (`SELECT`), the server calls the `EMP_SEC` function (under the `HR` schema). This function returns a predicate (called `P1`) defined in the function, which in this example could be specific to the current user for the `EMP_POLICY` policy. Your policy function can generate the predicates based on the session environment variables available during the function call, that is, from the application context as described in

the next section. The policy can specify any combination of security-relevant columns and any combination of these statement types: SELECT, INSERT, UPDATE, DELETE, or INDEX. You can also specify whether the result of an INSERT or UPDATE should immediately be checked against the policy.

The server then produces a transient view, with the following query:

```
SELECT * FROM hr.employees WHERE P1
```

Here, P1 (for example, WHERE SAL > 10000, or even a subquery) is the predicate returned from the EMP_SEC function. The server treats the EMPLOYEES table as a view and does the view expansion just like the ordinary view, except that the view text is taken from the transient view instead of the data dictionary.

The policy function creates a WHERE clause relevant to the current user by using information from the set of session environment variables called application context.

Note: Oracle Database does not implement fine-grained access control during MERGE statements. You must use equivalent INSERT and UPDATE statements instead of MERGE to avoid error messages and to ensure correct access control.

Application Context

Application context helps you apply fine-grained access control because you can link function-based security policies with applications.

Oracle provides a built-in application context namespace, USERENV, which provides access to predefined attributes. These attributes are session primitives which is information that the database automatically captures about a user session. For example, the IP address from which a user connects, the user name, and the proxy user name (in cases where a user connection is proxied through a middle tier), are all available as predefined attributes through the USERENV application context.

Each application has its own application-specific context, which users cannot arbitrarily change (for example, through SQL*Plus). Context attributes are accessible to the functions implementing security policies.

For example, context attributes you could use from a human resources application could include *position*, *organizational unit*, and *country*. Attributes available from an order-entry control system could include *customer number* and *sales region*.

Application contexts thus permit flexible, parameter-based access control using context attributes relevant to an application and to policies you might want to create for controlling its use.

You can:

- Base predicates on context values
- Use context values within predicates as bind variables
- Set user attributes
- Access user attributes

To define an application context:

1. Create a PL/SQL package with functions that validate and set the context for the application. You may want to use an event trigger on login to set the initial context for logged-in users.

2. Use `CREATE CONTEXT` to specify a unique context name and associate it with the PL/SQL package that you created.
3. Then do either of the following:
   - Reference the application context within the policy function implementing fine-grained access control.
   - Create an event trigger on login to set the initial context for a user. For example, you could query a user employee number and set this as an *employee number* context value.
4. Reference the application context. For example, to limit customers to seeing their own records only, use fine-grained access control to dynamically modify the user's query from `SELECT * FROM Orders_tab` to the following:

```
SELECT * FROM Orders_tab
   WHERE Custno = SYS_CONTEXT ('order_entry', 'cust_num');
```

See Also: Regarding how applications configure, administer, and use application context, see

- Chapter 14, "Using Virtual Private Database to Implement Application Security Policies"
- Chapter 15, "Implementing Application Context and Fine-Grained Access Control"
- *PL/SQL User's Guide and Reference*
- *PL/SQL Packages and Types Reference*
- *Oracle Database Application Developer's Guide - Fundamentals*

The next subsection, Dynamic Contexts, describes run-time efficiencies you can establish by identifying how dynamic each of your policies is, using these categories: static, shared, context-sensitive, or dynamic.

Dynamic Contexts

When you create a policy, you can establish run-time efficiencies by specifying whether the policy is static, shared, context-sensitive, or dynamic. Table 6–1 lists the policy types and run-time efficiencies.

Table 6–1 Policy Types and Run-Time Efficiencies

Policy Type	Predicate and Policy Function	Description and Operational Explanation
Static	Same predicate string for anyone accessing the object	Executed once and cached in SGA. Policies for statements accessing the same object do not reexecute the policy function, but use the cached predicate instead.
Shared-static	Same as static, except the policy can be shared across multiple objects	Ideal for data partitions in hosting environments because almost all objects share the same function and the policy is static. Executed once and cached in SGA, but the server first looks for a cached predicate generated by the same policy function of the same policy type.
Context-sensitive and shared context-sensitive	Policy function executed when statement parsed, but value returned is not cached	The policy function is not reevaluated at statement execution time unless the server detects context changes since the last use of the cursor. (For session pooling where multiple clients share a database session, the middle tier must reset context during client switches.) When a context-sensitive policy is labeled *shared*, the server first looks for a cached predicate generated by the same policy function of the same policy type within the same database session. If the predicate is found in the session memory, then the policy function is not rerun and the cached value is valid until session private application context changes occur.
Dynamic	Policy function always re-executed on each statement parsing or execution	Server assumes the predicate may be affected by any system or session environment at any time. Dynamic is the system default. If no policy type is specified when `DBMS_RLS.ADD_POLICY` is called, then dynamic is assumed.

See also: The section titled How to Add a Policy to a Table, View, or Synonym in Chapter 15, "Implementing Application Context and Fine-Grained Access Control".

Security Followup: Auditing and Prevention

Even after designing and implementing protective measures using privileges, views, and policies, you want to know when these measures are threatened or breached. Auditing can notify you of suspicious or questionable activities. You can then investigate, strengthen your defenses, and deal with inappropriate actions, consequences, and security offenders.

Use auditing to complement your access controls from several perspectives:

- Audit important data with database security mechanisms like access controls.
- Audit as a way of verifying that your access control mechanisms are implemented properly and work as you intended.
- Design audit policies that you expect will never actually *fire* because your other security mechanisms (authentication, authorization, access controls) should be protecting that data. If such an audit policy still fires, then you are alerted that you have a security breach. It may mean, for example, that your security protections are not operating as you expected them to.

See Also: Chapter 8, "Database Auditing: Security Considerations" introduces Oracle auditing facilities, and Chapter 12, "Configuring and Administering Auditing" describes them in detail.

7

Security Policies

Security considerations range from requiring backups to be done regularly and stored off-site to narrow table or data considerations, which include ensuring that unauthorized access to sensitive data, such as employee salaries, is precluded by built-in restrictions on every type of access to the table that contains them.

This chapter discusses security policies in the following sections:

- System Security Policy
- Data Security Policy
- User Security Policy
- Password Management Policy
- Auditing Policy
- A Security Checklist

System Security Policy

This section describes the different aspects of system security policy, and contains the following topics:

- Database User Management
- User Authentication
- Operating System Security

Each database has one or more administrators who are responsible for maintaining all aspects of the security policy: the security administrators. If the database system is small, then the database administrator may have the responsibilities of the security administrator. However, if the database system is large, then a special person or group of people may have responsibilities limited to those of a security administrator.

After deciding who will manage the security of the system, a security policy must be developed for every database. A database security policy should include several sub-policies, as explained in the following sections.

Database User Management

Database users are the access paths to the information in an Oracle database. Therefore, tight security should be maintained for the management of database users. Depending on the size of a database system and the amount of work required to manage database users, the security administrator may be the only user with the privileges required to create, alter, or drop database users. On the other hand, there

may be a number of administrators with privileges to manage database users. In any case, only trusted individuals should have the powerful privileges to administer database users.

See Also: *Oracle Database Administrator's Guide*

User Authentication

Database users can be **authenticated** (verified as the correct person) by Oracle using database passwords, the host operating system, network services, or by Secure Sockets Layer (SSL).

Note: To be authenticated using network authentication services or SSL requires that you have Oracle Advanced Security installed. Refer to the *Oracle Database Advanced Security Administrator's Guide* for information about these types of authentication.

User authentication and how it is specified is discussed in "User Authentication Methods" on page 10-1.

Operating System Security

The following security issues must also be considered for the operating system environment executing Oracle and any database applications:

- Database administrators must have the operating system privileges to create and delete files.
- Typical database users should not have the operating system privileges to create or delete files related to the database.
- If the operating system identifies database roles for users, then the security administrators must have the operating system privileges to modify the security domain of operating system accounts.

See Also: Operating-system-specific Oracle documentation for more information about operating system security issues

Data Security Policy

Data security includes the mechanisms that control the access to and use of the database at the object level. Your data security policy determines which users have access to a specific schema object, and the specific types of actions allowed for each user on the object. For example, the policy could establish that user `scott` can issue `SELECT` and `INSERT` statements but not `DELETE` statements using the `emp` table. Your data security policy should also define the actions, if any, that are audited for each schema object.

Primarily, the level of security you want to establish for the data in your database determines your data security policy. For example, it may be acceptable to have little data security in a database when you want to allow any user to create any schema object, or grant access privileges for their objects to any other user of the system. Alternatively, it might be necessary for data security to be very controlled when you want to allow only a database or security administrator to create objects and grant access privileges for objects to roles and users.

Overall data security should be based on the sensitivity of data. If information is not sensitive, then the data security policy can be more lax. However, if data is sensitive, then a security policy should be developed to maintain tight control over access to objects.

Some means of implementing data security include system and object privileges, and through roles. A role is a set of privileges grouped together that can be granted to users.

> **See Also:** Privileges and roles are discussed in Chapter 11, "Administering User Privileges, Roles, and Profiles"

Views can also implement data security because their definition can restrict access to table data. They can exclude columns containing sensitive data.

Another means of implementing data security is through fine-grained access control and use of an associated application context. Fine-grained access control is a feature of Oracle Database that enables you to implement security policies with functions, and to associate those security policies with tables or views. In effect, the security policy function generates a `WHERE` condition that is appended to relevant SQL statements, thereby restricting user access to rows of data in the table or view. An application context is a secure data cache for storing information used to make access control decisions.

> **See Also:**
>
> - Introduction to Views in Chapter 6
> - Introduction to Fine-Grained Access Control in Chapter 14
> - Introduction to Application Context in Chapter 14

User Security Policy

This section describes aspects of user security policy, and contains the following topics:

- General User Security
- End-User Security
- Administrator Security
- Application Developer Security
- Application Administrator Security

General User Security

For all types of database users, consider the following general user security issues:

- Password Security
- Privilege Management

Password Security

If user authentication is managed by the database, then security administrators should develop a password security policy to maintain database access security. For example, database users should be required to change their passwords at regular intervals, and of course, when their passwords are revealed to others. By forcing a user to modify passwords in such situations, unauthorized database access can be reduced.

Passwords are always automatically and transparently encrypted during network (client/server and server/server) connections, by using a modified Data Encryption Standard (DES) algorithm, before sending them across the network.

Privilege Management

Security administrators should consider issues related to privilege management for all types of users. For example, in a database with many user names, it may be beneficial to use roles (named groups of related privileges that you grant to users or other roles) to manage the privileges available to users. Alternatively, in a database with a handful of user names, it may be easier to grant privileges explicitly to users and avoid the use of roles.

Security administrators managing a database with many users, applications, or objects should take advantage of the benefits offered by roles. Roles greatly simplify the task of privilege management in complicated environments.

End-User Security

Security administrators must define a policy for end-user security. If a database has many users, then the security administrator can decide which groups of users can be categorized into user groups, and then create user roles for these groups. The security administrator can grant the necessary privileges or application roles to each user role, and assign the user roles to the users. To account for exceptions, the security administrator must also decide what privileges must be explicitly granted to individual users.

Using Roles for End-User Privilege Management

Roles are the easiest way to grant and manage the common privileges needed by different groups of database users.

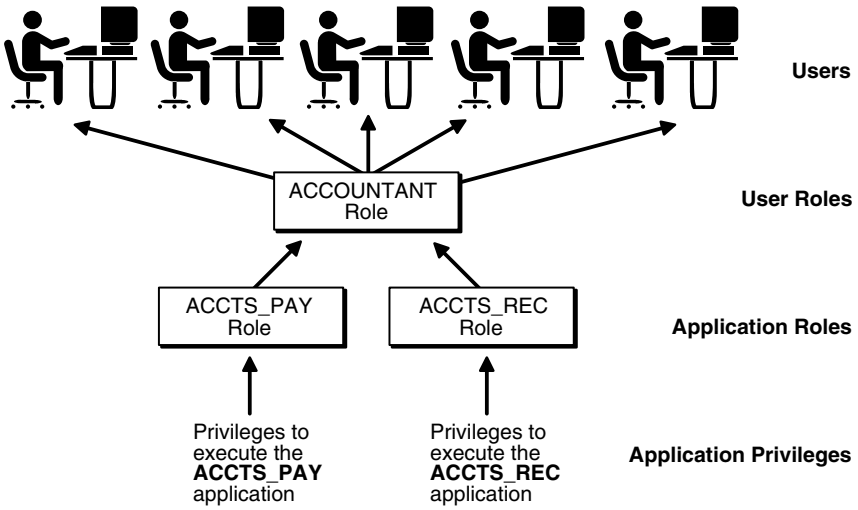
Consider a situation where every user in the accounting department of a company needs the privileges to run the accounts receivable and accounts payable database applications (`ACCTS_REC` and `ACCTS_PAY`). Roles are associated with both applications, and they contain the object privileges necessary to execute those applications.

The following actions, performed by the database or security administrator, address this simple security situation:

1. Create a role named `accountant`.
2. Grant the roles for the `ACCTS_REC` and `ACCTS_PAY` database applications to the `accountant` role.
3. Grant each user of the accounting department the `accountant` role.

This security model is illustrated in Figure 7–1.

Figure 7–1 User Roles



This plan addresses the following potential situations:

- If accountants subsequently need a role for a new database application, then the role for that application can be granted to the `accountant` role, and all users in the accounting department will automatically receive the privileges associated with the new database application. The role for the new application does not need to be granted to individual users requiring use of the application.
- Similarly, if the accounting department no longer requires the need for a specific application, then the role associated with that application can be dropped from the `accountant` role.
- If the privileges required by the `ACCTS_REC` and `ACCTS_PAY` applications change, then the new privileges can be granted to, or revoked from, the role associated with the application. The security domain of the `accountant` role, and all users granted the `accountant` role, automatically reflect the privilege modification.

Use roles in all possible situations to make end-user privilege management efficient and simple.

Using a Directory Service for End-User Privilege Management

You can also manage users and their authorizations centrally in a directory service through the enterprise user and enterprise role features of Oracle Advanced Security. See the *Oracle Database Advanced Security Administrator's Guide* for information about this functionality.

Administrator Security

Security administrators should also have a policy addressing database administrator security. For example, when the database is large and there are several types of database administrators, the security administrator may decide to group related administrative privileges into several administrative roles. The administrative roles can then be granted to appropriate administrator users. Alternatively, when the database is small and has only a few administrators, it may be more convenient to create one administrative role and grant it to all administrators.

Protection for Connections as SYS and SYSTEM

After database creation, if you used the default passwords for SYS and SYSTEM, then change the passwords for the SYS and SYSTEM administrative user names *immediately*. Connecting as SYS or SYSTEM gives a user powerful privileges to modify a database. For example, connecting as SYS allows a user to alter data dictionary tables. The privileges associated with these user names are extremely sensitive, and should only be available to select database administrators.

If you have installed options that have caused other administrative user names to be created, then such user name accounts are initially created locked. To unlock these accounts, use the ALTER USER statement. The ALTER USER statement should also be used to change the associated passwords for these accounts.

The passwords for these accounts can be modified by using the procedures described in "Altering Users" on page 11-5.

Protection for Administrator Connections

Only database administrators should be able to connect to a database with administrative privileges. For example:

```
CONNECT username/password AS SYSDBA/SYSOPER
```

Connecting as SYSOPER gives a user the ability to perform basic operational tasks (such as STARTUP, SHUTDOWN, and recovery operations). Connecting as SYSDBA gives the user these abilities plus unrestricted privileges to perform any actions with a database or the objects within a database (including, CREATE, DROP, and DELETE). Connecting as SYSDBA places a user in the SYS schema, where he can alter data dictionary tables.

Notes:

- Connections requested AS SYSDBA or AS SYSOPER must use these phrases; without them, the connection fails. The Oracle parameter 07_DICTIONARY_ACCESSIBILITY is set to FALSE by default, to limit sensitive data dictionary access only to those authorized.
- Such connections are authorized only after verification with the password file or with the operating system privileges and permissions. If operating system authentication is used, then the database does not use the supplied user name and password. Operating system authentication is used if there is no password file, or if the supplied user name or password is not in that file, or if no user name and password is supplied.
- However, if authentication succeeds by means of the password file, then the connection is logged with the user name. If authentication succeeds by means of the operating system, then it is a CONNECT / connection that does not record the specific user.

Using Roles for Administrator Privilege Management

Roles are the easiest way to restrict the powerful system privileges and roles required by personnel administrating the database.

Consider a scenario where the database administrator responsibilities at a large installation are shared among several database administrators, each responsible for the following specific database management jobs:

- Object creation and maintenance
- Database tuning and performance
- Creation of new users and granting roles and privileges to database users
- Routine database operation (for example: `STARTUP`, `SHUTDOWN`, and backup and recovery operations)
- Emergency situations, such as database recovery

There are also new, inexperienced database administrators needing limited capabilities to experiment with database management

In this scenario, the security administrator should structure the security for administrative personnel as follows:

1. Define six roles to contain the distinct privileges required to accomplish each type of job (for example, `dba_objects`, `dba_tune`, `dba_security`, `dba_maintain`, `dba_recov`, `dba_new`).
2. Grant each role the appropriate privileges.
3. Grant each type of database administrator the corresponding role.

This plan reduces the likelihood of future problems in the following ways:

- If a database administrator job description changes to include more responsibilities, then that database administrator can be granted other administrative roles corresponding to the new responsibilities.
- If a database administrator job description changes to include fewer responsibilities, then that database administrator can have the appropriate administrative roles revoked.
- The data dictionary always stores information about each role and each user, so information is available to disclose the task of each administrator.

Application Developer Security

Security administrators must define a special security policy for the application developers using a database. A security administrator could grant the privileges to create necessary objects to application developers. Alternatively, the privileges to create objects could be granted only to a database administrator, who then receives requests for object creation from developers.

Application Developers and Their Privileges

Database application developers are unique database users who require special groups of privileges to accomplish their jobs. Unlike end users, developers need system privileges, such as `CREATE TABLE`, `CREATE PROCEDURE`, and so on. However, only specific system privileges should be granted to developers to restrict their overall capabilities in the database.

Application Developer Environment: Test and Production Databases

In many cases, application development is restricted to test databases and is not allowed on production databases. This restriction ensures that application developers

do not compete with end users for database resources, and that they cannot detrimentally affect a production database.

After an application has been thoroughly developed and tested, it is permitted access to the production database and made available to the appropriate end users of the production database.

Free Versus Controlled Application Development

The database administrator can define the following options when determining which privileges should be granted to application developers:

- Free development

  An application developer is allowed to create new schema objects, including tables, indexes, procedures, packages, and so on. This option allows the application developer to develop an application independent of other objects.

- Controlled Development

  An application developer is not allowed to create new schema objects. A database administrator creates all required tables, indexes, procedures, and so on, as requested by an application developer. This option allows the database administrator to completely control the space usage of a database and the access paths to information in the database.

Although some database systems use only one of these options, other systems could mix them. For example, application developers can be allowed to create new stored procedures and packages, but not allowed to create tables or indexes. Security administrators should base their decision on the following:

- The control desired over the space usage of a database
- The control desired over the access paths to schema objects
- The database used to develop applications
  If a test database is being used for application development, then a more liberal development policy would be in order.

Roles and Privileges for Application Developers

Security administrators can create roles to manage the privileges required by the typical application developer. For example, a typical role named `APPLICATION_DEVELOPER` might include the `CREATE TABLE`, `CREATE VIEW`, and `CREATE PROCEDURE` system privileges. Consider the following when defining roles for application developers:

- `CREATE` system privileges are usually granted to application developers so that they can create their own objects. However, `CREATE ANY` system privileges, which allow a user to create an object in any user schema, are not usually granted to developers. This restricts the creation of new objects only to the developer user account.
- Object privileges are rarely granted to roles used by application developers, because granting object privileges through roles often restricts their usability in creating other objects (primarily views and stored procedures). It is more practical to allow application developers to create their own objects for development purposes.

Space Restrictions Imposed on Application Developers

While application developers are typically given the privileges to create objects as part of the development process, security administrators must maintain limits on what and how much database space can be used by each application developer. For example, as the security administrator, you should specifically set limits or restrict access to the following for each application developer:

- The tablespaces in which the developer can create tables or indexes
- The quota for each tablespace accessible to the developer

Both limitations can be set by altering a developer's security domain. This is discussed in Altering Users on page 11-5.

Application Administrator Security

In large database systems with many database applications, you might consider assigning application administrators. An application administrator is responsible for the following types of tasks:

- Creating roles for an application and managing the privileges of each application role
- Creating and managing the objects used by a database application
- Maintaining and updating the application code and Oracle procedures and packages as necessary

Often, an application administrator is also the application developer who designed an application. However, an application administrator could be any individual familiar with the database application.

Password Management Policy

Database security systems that are dependent on passwords require that passwords be kept secret at all times. Because passwords are vulnerable to theft, forgery, and misuse, Oracle Database uses a password management policy. DBAs and security officers control this policy through user profiles, enabling greater control over database security.

Use the `CREATE PROFILE` statement to create a user profile. The profile is assigned to a user with the `CREATE USER` or `ALTER USER` statement. Details of creating and altering database users are not discussed in this section. This section is concerned with the password parameters that can be specified using the `CREATE PROFILE` (or `ALTER PROFILE`) statement.

This section contains the following topics relating to Oracle password management:

- Account Locking
- Password Aging and Expiration
- Password History
- Password Complexity Verification

See Also:

- "Managing Resources with Profiles" on page 11-10
- "Managing Oracle Users" on page 11-1
- *Oracle Database SQL Reference* for syntax and specific information about SQL statements discussed in this section

Account Locking

When a particular user exceeds a designated number of failed login attempts, the server automatically locks that user account. You specify the permissible number of failed login attempts using the `CREATE PROFILE` statement. You can also specify the amount of time accounts remain locked.

In the following example, the maximum number of failed login attempts for the user `johndoe` is four, and the amount of time the account will remain locked is 30 days. The account will unlock automatically after the passage of 30 days.

```
CREATE PROFILE prof LIMIT
    FAILED_LOGIN_ATTEMPTS 4
    PASSWORD_LOCK_TIME 30;
ALTER USER johndoe PROFILE prof;
```

If you do not specify a time interval for unlocking the account, then `PASSWORD_LOCK_TIME` assumes the value specified in a default profile. If you specify `PASSWORD_LOCK_TIME` as `UNLIMITED`, then the account must be explicitly unlocked using an `ALTER USER` statement. For example, assuming that `PASSWORD_LOCK_TIME UNLIMITED` is specified for `johndoe`, then the following statement must be used to unlock the account:

```
ALTER USER johndoe ACCOUNT UNLOCK;
```

After a user successfully logs into an account, the unsuccessful login attempt count for the user, if it exists, is reset to 0.

The security officer can also explicitly lock user accounts. When this occurs, the account cannot be unlocked automatically, and only the security officer should unlock the account. The `CREATE USER` or `ALTER USER` statements are used to explicitly lock or unlock user accounts. For example, the following statement locks the user account, `susan`:

```
ALTER USER susan ACCOUNT LOCK;
```

Password Aging and Expiration

Use the `CREATE PROFILE` statement to specify a maximum lifetime for passwords. When the specified amount of time passes and the password expires, the user or DBA must change the password. The following statements create and assign a profile to user `johndoe`, and the `PASSWORD_LIFE_TIME` clause specifies that `johndoe` can use the same password for 90 days before it expires.

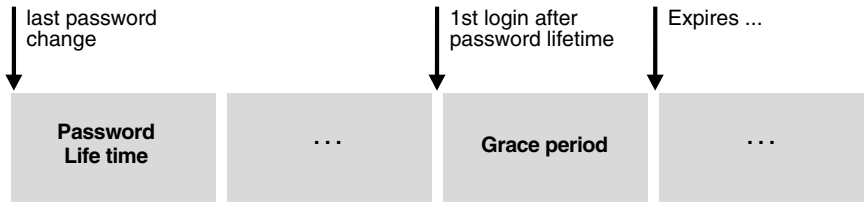
```
CREATE PROFILE prof LIMIT
   FAILED_LOGIN_ATTEMPTS 4
   PASSWORD_LOCK_TIME 30
   PASSWORD_LIFE_TIME 90;
ALTER USER johndoe PROFILE prof;
```

You can also specify a grace period for password expiration. Users enter the grace period upon the first attempt to log in to a database account after their password has

expired. During the grace period, a warning message appears each time users try to log in to their accounts, and continues to appear until the grace period expires. Users must change the password within the grace period. If the password is not changed within the grace period, then users are prompted for a new password each time an attempt is made to access their accounts. Access to an account is denied until a new password is supplied.

Figure 7–2 shows the chronology of the password lifetime and grace period.

Figure 7–2 Chronology of Password Lifetime and Grace Period



In the following example, the profile assigned to `johndoe` includes the specification of a grace period: `PASSWORD_GRACE_TIME = 3`. The first time `johndoe` tries to log in to the database after 90 days (this can be *any* day after the 90th day, that is, the 70th day, 100th day, or another day), he receives a warning message that his password will expire in three days. If three days pass, and he does not change her password, then the password expires. After this, he receives a prompt to change his password on any attempt to log in, and cannot log in until he does so.

```
CREATE PROFILE prof LIMIT
   FAILED_LOGIN_ATTEMPTS 4
   PASSWORD_LOCK_TIME 30
   PASSWORD_LIFE_TIME 90
   PASSWORD_GRACE_TIME 3;
ALTER USER johndoe PROFILE prof;
```

Oracle provides a means of explicitly expiring a password. The `CREATE USER` and `ALTER USER` statements provide this functionality. The following statement creates a user with an expired password. This setting forces the user to change the password before the user can log in to the database.

```
CREATE USER jbrown
     IDENTIFIED BY zX83yT
     ...
     PASSWORD EXPIRE;
```

Password History

The following two parameters control user ability to reuse an old password:

Table 7–1 Parameters Controlling Reuse of an Old Password

Parameter Name	Description and Use
PASSWORD_REUSE_TIME	Requires either of the following: ■ A number specifying how many days (or a fraction of a day) between the earlier use of a password and its next use ■ The word UNLIMITED.
PASSWORD_REUSE_MAX	Requires either of the following: ■ An integer to specify the number of password changes required before a password can be reused ■ The word UNLIMITED

If you specify neither, then the user can reuse passwords at any time, which is not a good security practice.

If neither parameter is UNLIMITED, then password reuse is allowed, but only after meeting both conditions. The user must have changed the password the specified number of times, and the specified number of days must have passed since the old password was last used.

For example, suppose that the user A's profile had PASSWORD_REUSE_MAX set to 10 and PASSWORD_REUSE_TIME set to 30. Then user A cannot reuse a password until the password has been reset 10 times, and until 30 days had passed since the password was last used.

If either parameter is specified as UNLIMITED, then the user can never reuse a password.

If both parameters are set to UNLIMITED, then Oracle ignores both, and the user can reuse any password at any time.

Note: If you specify DEFAULT for either parameter, then Oracle uses the value defined in the DEFAULT profile, which sets all parameters to UNLIMITED. Oracle thus uses UNLIMITED for any parameter specified as DEFAULT, unless you change the setting for that parameter in the DEFAULT profile.

Password Complexity Verification

Oracle sample password complexity verification routine can be specified using a PL/SQL script (UTLPWDMG.SQL), which sets the default profile parameters.

The password complexity verification routine ensures that the password meets the following requirements:

- Is at least four characters long
- Differs from the user name
- Has at least one alpha, one numeric, and one punctuation mark character
- Is not simple or obvious, such as welcome, account, database, or user
- Differs from the previous password by at least 3 characters

Note: The ALTER USER command now has a REPLACE clause by using which users can change their own unexpired passwords by supplying the old password to authenticate themselves.

If the password has expired, then the user cannot log in to SQL to issue the ALTER USER command. Instead, the OCIPasswordChange() function must be used, which also requires the old password.

A DBA with ALTER ANY USER privilege can alter any user password (force a new password) without supplying the old one.

Password Verification Routine Formatting Guidelines

You can enhance the existing password verification complexity routine or create other password verification routines using PL/SQL or third-party tools.

The PL/SQL call must adhere to the following format:

```
routine_name
(
userid_parameter IN VARCHAR(30),
password_parameter IN VARCHAR (30),
old_password_parameter IN VARCHAR (30)
)
RETURN BOOLEAN
```

After a new routine is created, it must be assigned as the password verification routine by using the user profile or the system default profile.

```
CREATE/ALTER PROFILE profile_name LIMIT
PASSWORD_VERIFY_FUNCTION routine_name
```

The password verification routine must be owned by the SYS user.

Sample Password Verification Routine

You can use the following sample password verification routine as a model when developing your own complexity checks for a new password.

The default password complexity function performs the following minimum complexity checks:

- The password satisfies minimum length requirements.
- The password is not the username. You can modify this function based on your requirements.

This function must be created in SYS schema, and you must CONNECT SYS/*password* AS SYSDBA before running the script.

```
CREATE OR REPLACE FUNCTION verify_function
(username varchar2,
   password varchar2,
   old_password varchar2)
   RETURN boolean IS
   n boolean;
   m integer;
   differ integer;
   isdigit boolean;
   ischar  boolean;
   ispunct boolean;
```

```
   digitarray varchar2(20);
   punctarray varchar2(25);
   chararray varchar2(52);

BEGIN
   digitarray:= '0123456789';
   chararray:= 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';
   punctarray:='!"#$%&()''*+,-/:;<=>?_';

--Check if the password is same as the username
IF password = username THEN
   raise_application_error(-20001, 'Password same as user');
END IF;

--Check for the minimum length of the password
IF length(password) < 4 THEN
   raise_application_error(-20002, 'Password length less than 4');
END IF;

--Check if the password is too simple. A dictionary of words may be
--maintained and a check may be made so as not to allow the words
--that are too simple for the password.
IF NLS_LOWER(password) IN ('welcome', 'database', 'account', 'user',
   'password', 'oracle', 'computer', 'abcd')
   THEN raise_application_error(-20002, 'Password too simple');
END IF;

--Check if the password contains at least one letter,
--one digit and one punctuation mark.
--1. Check for the digit
--You may delete 1. and replace with 2. or 3.
isdigit:=FALSE;
m := length(password);
FOR i IN 1..10 LOOP
  FOR j IN 1..m LOOP
    IF substr(password,j,1) = substr(digitarray,i,1) THEN
      isdigit:=TRUE;
        GOTO findchar;
    END IF;
   END LOOP;
END LOOP;
IF isdigit = FALSE THEN
  raise_application_error(-20003, 'Password should contain at least one \
  digit, one character and one punctuation');
END IF;
--2. Check for the character

<<findchar>>
ischar:=FALSE;
FOR i IN 1..length(chararray) LOOP
  FOR j IN 1..m LOOP
    IF substr(password,j,1) = substr(chararray,i,1) THEN
       ischar:=TRUE;
         GOTO findpunct;
       END IF;
    END LOOP;
END LOOP;
IF ischar = FALSE THEN
  raise_application_error(-20003, 'Password should contain at least one digit,\
    one character and one punctuation');
```

```
END IF;
--3. Check for the punctuation

<<findpunct>>
ispunct:=FALSE;
FOR i IN 1..length(punctarray) LOOP
  FOR j IN 1..m LOOP
    IF substr(password,j,1) = substr(punctarray,i,1) THEN
       ispunct:=TRUE;
         GOTO endsearch;
       END IF;
   END LOOP;
END LOOP;
IF ispunct = FALSE THEN raise_application_error(-20003, 'Password should \
 contain at least one digit, one character and one punctuation');
END IF;

<<endsearch>>
--Check if the password differs from the previous password by at least 3 letters
IF old_password = '' THEN
  raise_application_error(-20004, 'Old password is null');
END IF;
--Everything is fine; return TRUE ;
differ := length(old_password) - length(password);
IF abs(differ) < 3 THEN
  IF length(password) < length(old_password) THEN
    m := length(password);
  ELSE
    m:= length(old_password);
  END IF;
  differ := abs(differ);
  FOR i IN 1..m LOOP
    IF substr(password,i,1) != substr(old_password,i,1) THEN
             differ := differ + 1;
    END IF;
  END LOOP;
  IF differ < 3 THEN
    raise_application_error(-20004, 'Password should differ by at \
      least 3 characters');
    END IF;
  END IF;
--Everything is fine; return TRUE ;
  RETURN(TRUE);
END;
```

Auditing Policy

Security administrators should define a policy for the auditing procedures of each database. You may, for example, decide to have database auditing disabled unless questionable activities are suspected. When auditing is required, the security administrator must decide what level of detail to audit the database; usually, general system auditing is followed by more specific types of auditing after the origins of suspicious activity are determined. In addition to standard database auditing, Oracle Database supports fine-grained auditing using policies that can monitor multiple specific objects, columns, and statements, including INDEX.

> **See Also:** Auditing is discussed in Chapter 8, "Database Auditing: Security Considerations" and Chapter 12, "Configuring and Administering Auditing".

A Security Checklist

Information security and privacy and protection of corporate assets and data are of pivotal importance in any business. Oracle Database comprehensively addresses the need for information security by offering cutting-edge security features such as deep data protection, auditing, scalable security, secure hosting and data exchange.

The Oracle Database server leads the industry in security. However, in order to fully maximize the security features offered by Oracle Database in any business environment, it is imperative that the database itself be well-protected. Furthermore, proper use of its security features and adherence to basic security practices will help protect against database-related threats and attacks. Such an approach provides a much more secure operating environment for Oracle Database.

This security checklist provides guidance on configuring Oracle Database in a secure manner by adhering to and recommending industry-standard and advisable "security practices" for operational database deployments.

Before looking at the more detailed checklist: consider all paths the data travels and assess the threats that impinge on each path and node. Then, take steps to lessen or eliminate both the threats and the consequences of a successful breach of security. Monitoring and auditing to detect either increased threat levels or successful penetration increases the likelihood of preventing and minimizing security losses.

Details on specific database-related tasks and actions can be found throughout the Oracle documentation set.

The following security checklist includes guidelines that help secure your database:

1. **Install only what is required.**

   Options and Products

   The Oracle Database CD pack contains a host of options and products in addition to the database server. Install additional products and options only as necessary. Use Custom Installation to avoid installing unnecessary products or, perform a typical installation, and then deinstall unrequired options and products. There is no need to maintain additional products and options if they are not being used. They can always be properly and easily reinstalled as required.

   Sample Schemas

   Oracle Corporation provides Sample Schemas to provide a common platform for examples. If your database will be used in a production environment, then do not install the Sample Schema. If you have installed the Sample Schema on a test database, then before going production, remove or relock the Sample Schema accounts.

2. **Lock and expire default user accounts.**

   Oracle Database installs with a number of default (preset) database server user accounts. Upon successful installation of the database server, the Database Configuration Assistant automatically locks and expires most default database user accounts.

   If a manual (without using Database Configuration Assistant) installation of Oracle Database is performed, then no default database users are locked upon successful installation of the database server. Left open in their default states, these user accounts can be exploited to gain unauthorized access to data or disrupt database operations.

Therefore, after performing any kind of initial installation that does not use the Database Configuration Assistant, you should *lock* and *expire* all default database user accounts. Oracle Database provides SQL statements to perform such operations.

Installing additional products and components later also results in creating more default database server accounts. Database Configuration Assistant automatically locks and expires all additionally created database server user accounts. Unlock only those accounts that need to be accessed on a regular basis and assign a strong, meaningful password to each of these unlocked accounts. Oracle provides SQL and password management to perform such operations.

Table 7–2 shows the database users after a typical Oracle Database installation using Database Configuration Assistant.

Table 7–2 Default Accounts and Their Status (Standard Installation)

Username	Account Status
ANONYMOUS	EXPIRED & LOCKED
CTXSYS	EXPIRED & LOCKED
DBSNMP	EXPIRED & LOCKED
DIP	EXPIRED & LOCKED
DMSYS	EXPIRED & LOCKED
EXFSYS	EXPIRED & LOCKED
HR	EXPIRED & LOCKED
MDDATA	EXPIRED & LOCKED
MDSYS	EXPIRED & LOCKED
MGMT_VIEW	EXPIRED & LOCKED
ODM	EXPIRED & LOCKED
ODM_MTR	EXPIRED & LOCKED
OE	EXPIRED & LOCKED
OLAPSYS	EXPIRED & LOCKED
ORDPLUGINS	EXPIRED & LOCKED
ORDSYS	EXPIRED & LOCKED
OUTLN	EXPIRED & LOCKED
PM	EXPIRED & LOCKED
QS	EXPIRED & LOCKED
QS_ADM	EXPIRED & LOCKED
QS_CB	EXPIRED & LOCKED
QS_CBADM	EXPIRED & LOCKED
QS_CS	EXPIRED & LOCKED
QS_ES	EXPIRED & LOCKED
QS_OS	EXPIRED & LOCKED
QS_WS	EXPIRED & LOCKED
RMAN	EXPIRED & LOCKED

Table 7–2 (Cont.) Default Accounts and Their Status (Standard Installation)

Username	Account Status
SCOTT	EXPIRED & LOCKED
SH	EXPIRED & LOCKED
SI_INFORMTN_SCHEMA	EXPIRED & LOCKED
SYS	OPEN
SYSMAN	EXPIRED & LOCKED
SYSTEM	OPEN
TSMSYS	EXPIRED & LOCKED
WK_TEST	EXPIRED & LOCKED
WKPROXY	EXPIRED & LOCKED
WKSYS	EXPIRED & LOCKED
WMSYS	EXPIRED & LOCKED
XDB	EXPIRED & LOCKED

If any default database server user account other than the ones left open is required for any reason, then a database administrator (DBA) need simply unlock and activate that account with a new, secure password.

Enterprise Manager Accounts

The preceding list of accounts depends on whether you choose to install Enterprise Manager. If so, SYSMAN and DBSNMP are open as well, unless you configure Enterprise Manager for Central Administration. In this case, the SYSMAN account (if present) will be locked as well.

If you do not install Enterprise Manager, then only SYS and SYSTEM are open. Database Configuration Assistant locks and expires all other accounts (including SYSMAN and DBSNMP).

3. **Change default user passwords.**

   The most trivial method by which Oracle Database can be compromised is a default database server user account which still has a default password associated with it *even after installation*. The following guidelines are recommended:

   a. **Change default passwords of administrative users.**

      Oracle Database 10g enables you to use the same or different passwords for the SYS, SYSTEM, SYSMAN and DBSNMP administrative accounts. Use different passwords for each: in any Oracle environment (production or test), assign strong, secure, and distinct passwords to these administrative accounts. If Database Configuration Assistant is used, then it requires you to enter passwords for the SYS and SYSTEM accounts, disallowing the use of the defaults CHANGE_ON_INSTALL and MANAGER.

      Similarly, for production environments, do not use default passwords for any administrative accounts, including SYSMAN and DBSNMP.

      At the end of database creation, Database Configuration Assistant displays a page requiring you to enter and confirm new passwords for the SYS and SYSTEM user accounts.

   b. **Change default passwords of all users.**

In Oracle Database, SCOTT no longer installs with default password TIGER, but instead is locked and expired, as is DBSNMP. Each of the other accounts install with a default password that is exactly the same as that user account (For example, user MDSYS installs with the password MDSYS).

If any of the default user accounts that were locked and expired upon installation need to be activated, then assign a new secure password to each such user account.

Even though Oracle does not explicitly mandate changing the default password for the user SCOTT, Oracle recommends that this user account also be locked in a production environment.

c. Enforce password management.

Oracle recommends that basic password management rules (such as password length, history, complexity, and so forth) as provided by the database be applied to all user passwords and that all users be required to change their passwords periodically.

Oracle also recommends, if possible, using Oracle Advanced Security (an option to the Enterprise Edition of Oracle Database) with network authentication services (such as Kerberos), token cards, smart cards or X.509 certificates. These services enable strong authentication of users to provide better protection against unauthorized access to Oracle Database.

4. Enable data dictionary protection.

Oracle recommends that customers implement data dictionary protection to prevent users having the ANY system privileges from using such privileges on the data dictionary.

To enable dictionary protection, set the following configuration parameter to FALSE, in the init<sid>.ora control file:

```
O7_DICTIONARY_ACCESSIBILITY = FALSE
```

By doing so, only those authorized users making DBA-privileged (for example CONNECT / AS SYSDBA) connections can use the ANY system privilege on the data dictionary. If O7_DICTIONARY_ACCESSIBILITY is not set to FALSE, then any user with a DROP ANY TABLE (for example) system privilege will be able to maliciously drop parts of the data dictionary.

However, if a user *needs* view access to the data dictionary, then it is permissible to grant that user the SELECT ANY DICTIONARY system privilege.

Notes:

- Regarding O7_DICTIONARY_ACCESSIBILITY, note that in Oracle Database, the default is FALSE. However, in Oracle8i, this parameter is set to TRUE by default and must specifically be changed to FALSE to enable this security feature.
- Regarding the SELECT ANY DICTIONARY privilege: this privilege is not included in the GRANT ALL PRIVILEGES statement, but it can be granted through a role.

---

5. Practice the principle of least privilege.

The following guidelines are recommended:

a. Grant necessary privileges only.

Do not provide database users more privileges than are necessary. In other words, the *principle of least privilege* is that users be given only those privileges that are actually required to efficiently perform their jobs.

To implement this principle, restrict the following as much as possible:

1) The number of SYSTEM and OBJECT privileges granted to database users, and

2) The number of people who are allowed to make SYS-privileged connections to the database.

For example, there is generally no need to grant CREATE ANY TABLE to any non-DBA-privileged user.

b. Revoke unnecessary privileges from the PUBLIC user group.

Revoke all unnecessary privileges and roles from the database server user group PUBLIC. PUBLIC acts as a default role granted to every user in an Oracle database. Any database user can exercise privileges that are granted to PUBLIC. Such privileges include EXECUTE on various PL/SQL packages, potentially enabling a minimally-privileged user to access and execute functions that he would not otherwise be permitted to access directly. The more powerful packages that may potentially be misused are listed in the following table:

Package	Description
UTL_SMTP[1]	This package permits arbitrary mail messages to be sent from one arbitrary user to another arbitrary user. Granting this package to PUBLIC may permit unauthorized exchange of mail messages.
UTL_TCP[1]	This package permits outgoing network connections to be established by the database server to any receiving (or waiting) network service. Thus, arbitrary data may be sent between the database server and any waiting network service.
UTL_HTTP[1]	This package allows the database server to request and retrieve data using HTTP. Granting this package to PUBLIC may permit using HTML forms to send data to a malicious Web site.
UTL_FILE[1]	If configured improperly, then this package allows text level access to any file on the host operating system. Even when properly configured, this package may allow unauthorized access to sensitive operating system files, such as trace files, because it does not distinguish between its calling applications. The result can be that one application accessing UTL_FILE may write arbitrary data into the same location that is written to by another application.
DBMS_RANDOM	This package can be used to encrypt stored data. Generally, most users should not have the privilege to encrypt data since encrypted data may be non-recoverable if the keys are not securely generated, stored, and managed.

[1] These packages should be revoked from PUBLIC and made executable for an application only when absolutely necessary.

These packages are extremely useful to the applications that need them. They require proper configuration and usage for safe and secure operation, and may not be suitable for most applications.

For applications that need these packages, create roles with EXECUTE privilege on the particular packages needed and assign those roles only to applications

that specifically need to use them. Oracle intends to revoke such privileges from PUBLIC in subsequent releases.

c. Grant a role to users only if they need all privileges of the role.

Roles (groups of privileges) are useful for quickly and easily granting permissions to users. Although you can use Oracle-defined roles, you have more control and continuity if you create your own roles containing only the privileges pertaining to your requirements. Oracle may change or remove the privileges in an Oracle-defined role, as it has with CONNECT, which now has only the CREATE SESSION privilege. Formerly this role had eight other privileges. Both CONNECT and RESOURCE roles will be deprecated in future Oracle versions.

Ensure that the roles you define contain only the privileges that reflect job responsibility. If your application users do not need all the privileges encompassed by an existing role, then apply a different set of roles that supply just the right privileges. Alternatively, create and assign a more restricted role.

For example, it is imperative to strictly limit the privileges of SCOTT. Drop the CREATE DBLINK privilege for SCOTT. Then drop the entire role for the user, because privileges acquired by means of a role cannot be dropped individually. Recreate your own role with only the privileges needed, and grant that new role to that user. Similarly, for even better security, drop the CREATE DBLINK privilege from all users who do not require it.

d. Restrict permissions on run-time facilities.

Do not assign all permissions to any database server run-time facility such as the Oracle Java Virtual Machine (OJVM). Grant specific permissions to the explicit document root file paths for such facilities that may execute files and packages outside the database server.

Here is an example of a vulnerable run-time call:

```
call dbms_java.grant_permission('SCOTT',
'SYS:java.io.FilePermission','<<ALL FILES>>','read');
```

Here is an example of a better (more secure) run-time call:

```
call dbms_java.grant_permission('SCOTT',
'SYS:java.io.FilePermission','<<actual directory path>>','read');
```

6. Enforce access controls effectively and authenticate clients stringently.

Authenticate clients properly.

By default, Oracle allows operating-system-authenticated logins only over secure connections, which precludes using Oracle Net and a shared server configuration. This default restriction prevents a remote user from impersonating another operating system user over a network connection.

Setting the initialization parameter REMOTE_OS_AUTHENT to TRUE forces the RDBMS to accept the client operating system user name received over a nonsecure connection and use it for account access. Since clients, such as PCs, are not trusted to perform operating system authentication properly, it is very poor security practice to turn on this feature.

The default setting, REMOTE_OS_AUTHENT = FALSE, creates a more secure configuration that enforces proper, server-based authentication of clients connecting to an Oracle database.

You should not alter the default setting of the `REMOTE_OS_AUTHENT` initialization parameter, which is `FALSE`.

Setting this parameter to `FALSE` does not mean that users cannot connect remotely. It simply means that the database will not trust that the client has already authenticated, and will therefore apply its standard authentication processes.

7. **Restrict operating system access.**

   Limit the number of operating system users. Limit the privileges of the operating system accounts (administrative, root-privileged or DBA) on the Oracle Database host (physical machine) to the least privileges needed for the user's tasks.

   Oracle also recommends:

   - Restricting the ability to modify the default file and directory permissions for the Oracle Database home (installation) directory or its contents. Even privileged operating system users and the Oracle owner should not modify these permissions, unless instructed otherwise by Oracle.
   - Restricting symbolic links. Ensure that when providing a path or file to the database, neither the file nor any part of the path is modifiable by an untrusted user. The file and all components of the path should be owned by the DBA or some trusted account, such as *root*.

     This recommendation applies to all types of files: data files, log files, trace files, external tables, bfiles, and so on.

8. **Restrict network access.**

   The following guidelines are recommended:

   a. **Use a firewall.**

      Keep the database server behind a firewall. Oracle Database network infrastructure, Oracle Net (formerly known as Net8 and SQL*Net), offers support for a variety of firewalls from various vendors. Supported proxy-enabled firewalls include' Gauntlet from Network Associates and Raptor from Axent . Supported packet-filtering firewalls include PIX Firewall from Cisco, and supported stateful inspection firewalls (more sophisticated packet-filtered firewalls) include Firewall-1 from CheckPoint .

   b. **Never poke a hole through a firewall.**

      If Oracle Database is behind a firewall, then do not, under any circumstances, poke a hole through the firewall. For example, do not leave open port 1521 for Oracle Listener to make a connection to the Internet or vice versa.

      Doing this will introduce a number of significant security vulnerabilities including more port openings through the firewall, multi-threaded operating system server issues, and revelation of crucial information on databases behind the firewall. Furthermore, an Oracle Listener running without an established password may be probed for critical details about the databases on which it is listening such as trace and logging information, banner information and database descriptors and service names.

      All this information and the availability of an ill-configured firewall will provide an attacker ample opportunity to launch malicious attacks on the target databases.

   c. **Protect the Oracle listener.**

Because the listener acts as the database gateway to the network, it is important to limit the consequences of malicious interference:

* Restrict the privileges of the listener, so that it cannot read or write files in the database or the Oracle server address space.

  This restriction prevents external procedure agents spawned by the listener (or procedures executed by such an agent) from inheriting the ability to do such reads or writes. The owner of this separate listener process should not be the owner that installed Oracle or executes the Oracle instance (such as `ORACLE`, the default owner).

  Sample configuration:

```
EXTPROC_LISTENER=
  (DESCRIPTION=
    (ADDRESS=
      (PROTOCOL=ipc)(KEY=extproc)))
SID_LIST_EXTPROC_LISTENER=
  (SID_LIST=
    (SID_DESC=
      (SID_NAME=plsextproc)
      (ORACLE_HOME=/u1/app/oracle/9.0)
      (PROGRAM=extproc)))
```

* Secure administration of the database by doing the following:

  i. Prevent online administration by requiring the administrator to have write privileges on the `LISTENER.ORA` file and the listener password:

  Add or alter this line in the LISTENER.ORA file

```
ADMIN_RESTRICTIONS_LISTENER=ON
```

  Then RELOAD the configuration.

  ii. Use SSL when administering the listener, by making the TCPS protocol the first entry in the address list as follows:

```
LISTENER=
  (DESCRIPTION=
    (ADDRESS_LIST=
      (ADDRESS=
        (PROTOCOL=tcps)
        (HOST = ed-pdsun1.us.oracle.com)
        (PORT = 8281)))
```

  To administer the listener remotely, you need to define the listener in the `listener.ora` file on the client computer. For example, to access listener USER281 remotely, use the following configuration:

```
user281 =
  (DESCRIPTION =
    (ADDRESS =
      (PROTOCOL = tcps)
      (HOST = ed-pdsun1.us.oracle.com)
      (PORT = 8281))
    )
  )
```

  iii. Always establish a secure, well-formed password for the Oracle listener to prevent remote configuration of the Oracle listener. Password protect the listener as follows:

```
LSNRCTL> CHANGE_PASSWORD
Old password: lsnrc80
New password: lsnrc90
Reenter new password: lsnrc90
LSNRCTL> SET PASSWORD
Password:
The command completed successfully
LSNRCTL> SAVE_CONFIG
The command completed successfully
```

* Remove the external procedure configuration from the `listener.ora` file if you do not intend to use such procedures.
* Monitor listener activity.

d. Monitor who accesses your systems.

Authenticating client computers over the Internet is problematic. Do user authentication instead, which avoids client system issues that include falsified IP addresses, hacked operating systems or applications, and falsified or stolen client system identities. The following steps improve client computer security:

* Configure the connection to use SSL. Using SSL (Secure Sockets Layer) communication makes eavesdropping unfruitful and enables the use of certificates for user and server authentication.
* Set up certificate authentication for clients and servers such that:

  i. The organization is identified by unit and certificate issuer and the user is identified by distinguished name and certificate issuer.

  ii. Applications test for expired certificates.

  iii. Certificate revocation lists are audited.

e. Check network IP addresses.

Use the Oracle Net *valid node checking* security feature to allow or deny access to Oracle server processes from network clients with specified IP addresses. To use this feature, set the following `protocol.ora` (Oracle Net configuration file) parameters:

```
tcp.validnode_checking = YES

tcp.excluded_nodes = {list of IP addresses}

tcp.invited_nodes = {list of IP addresses}
```

The first parameter turns on the feature whereas the latter parameters respectively deny and allow specific client IP addresses from making connections to the Oracle listener (This helps in preventing potential Denial of Service attacks).

f. Encrypt network traffic.

If possible, use Oracle Advanced Security to encrypt network traffic between clients, databases, and application servers.

Note: Oracle Advanced Security is available only with the Enterprise Edition of the Oracle database. It installs in Typical Installation mode and can be configured, after licensing, with the Oracle Net Manager tool or by manually setting six `sqlnet.ora` parameters to enable network encryption.

---

g. Harden the operating system.

Harden the host operating system by disabling all unnecessary operating system services. Both UNIX and Windows platforms provide a variety of operating system services, most of which are not necessary for most deployments. Such services include FTP, TFTP, TELNET, and so forth. Be sure to close both the UDP and TCP ports for each service that is being disabled. Disabling one type of port and not the other does not make the operating system more secure.

9. Apply all security patches and workarounds.

Always apply all relevant and current security patches for both the operating system on which Oracle Database resides and Oracle Database itself, and for all installed Oracle Database options and components.

Periodically check the security site on Oracle Technology Network for details on security alerts released by Oracle Corporation at

`http://www.oracle.com/technology/deploy/security/alerts.htm`

Also check Oracle Worldwide Support Service site, Metalink, for details on available and upcoming security-related patches at

`http://metalink.oracle.com`

10. Contact Oracle Security Products if you come across a vulnerability in Oracle Database.

If you believe that you have found a security vulnerability in Oracle Database, then submit an iTAR to Oracle Worldwide Support Services using Metalink, or e-mail a complete description of the problem, including product version and platform, together with any exploit scripts and examples to the following address:

`secalert_us@oracle.com`

8

Database Auditing: Security Considerations

Auditing is the monitoring and recording of selected user database actions. It can be based on individual actions, such as the type of SQL statement executed, or on combinations of factors that can include user name, application, time, and so on. Security policies can trigger auditing when specified elements in an Oracle database are accessed or altered, including the contents within a specified object.

An overview of database auditing appears in Chapter 6. Chapter 12 provides detailed information and guidelines on configuring auditing parameters and administering auditing actions and results.

The present chapter describes the different types of auditing, what it involves, and the resulting audit trails and records.

Auditing is typically used to:

- Enable future accountability for current actions taken in a particular schema, table, or row, or affecting specific content
- Deter users (or others) from inappropriate actions based on that accountability
- Investigate suspicious activity

  For example, if some user is deleting data from tables, then the security administrator might decide to audit all connections to the database and all successful and unsuccessful deletions of rows from all tables in the database.
- Notify an auditor that an unauthorized user is manipulating or deleting data and that the user has more privileges than expected which can lead to reassessing user authorizations
- Monitor and gather data about specific database activities

  For example, the database administrator can gather statistics about which tables are being updated, how many logical I/Os are performed, or how many concurrent users connect at peak times.
- Detect problems with an authorization or access control implementation

  For example, you can create audit policies that you expect will never generate an audit record because the data is protected in other ways. However, if these policies do generate audit records, then you will know the other security controls are not properly implemented.

This chapter describes the types of auditing available in Oracle systems, in the following sections:

- Privilege Auditing
- Schema Object Auditing
- Fine-Grained Auditing
- Focusing Statement, Privilege, and Schema Object Auditing
- Auditing in a Multitier Environment

See Also:

- Chapter 12, "Configuring and Administering Auditing"
- *Oracle Database Administrator's Guide*

Auditing Types and Records

Oracle allows audit options to be focused or broad, enabling you to audit the following:

- Successful statement executions, unsuccessful statement executions, or both
- Statement executions once in each user session or once every time the statement is executed
- Activities of all users or of a specific user

Table 8–1 describes the different Oracle auditing mechanisms. Each entry in the first column is a link to a more extensive discussion of that particular method.

Table 8–1 Auditing Types and Descriptions

Type of Auditing (link to discussion)	Meaning/Description
Statement Auditing	Enables you to audit SQL statements by type of statement, not by the specific schema objects on which they operate. Typically broad, statement auditing audits the use of several types of related actions for each option. For example, `AUDIT TABLE` tracks several DDL statements regardless of the table on which they are issued. You can also set statement auditing to audit selected users or every user in the database.
Privilege Auditing	Enables you to audit the use of powerful system privileges that enable corresponding actions, such as `AUDIT CREATE TABLE`. Privilege auditing is more focused than statement auditing, which audits only a particular type of action. You can set privilege auditing to audit a selected user or every user in the database.
Schema Object Auditing	Enables you to audit specific statements on a particular schema object, such as `AUDIT SELECT ON employees`. Schema object auditing is very focused, auditing only a single specified type of statement (such as SELECT) on a specified schema object. Schema object auditing always applies to all users of the database.
Fine-Grained Auditing	Enables you to audit at the most granular level, data access and actions based on content, using any Boolean measure, such as `value > 1,000,000`. Enables auditing based on access to or changes in a column.

The following subsections explain the records and timing of the different audit trails:

- Audit Records and Audit Trails

- When Are Audit Records Created?

Audit Records and Audit Trails

Audit records include information about the operation that was audited, the user performing the operation, and the date and time of the operation. Audit records can be stored in either a data dictionary table, called the **database audit trail**, or in operating system files, called an **operating system audit trail**.

> **See Also:** The complete contents of these audit trails is described in Chapter 12, "Configuring and Administering Auditing", in the section entitled What Information Is Contained in the Audit Trail?

The two general types of auditing are standard auditing, which is based on privileges, schemas, objects, and statements, and fine-grained auditing. Standard audit records can be written either to `DBA_AUDIT_TRAIL` (the `sys.aud$` table) or to the operating system. Fine-grained audit records are written to `DBA_FGA_AUDIT_TRAIL` (the `sys.fga_log$` table) and the `DBA_COMMON_AUDIT_TRAIL` view, which combines standard and fine-grained audit log records.

The following subsections describe these trails and records:

- Database Audit Trail (DBA_AUDIT_TRAIL)
- Operating System Audit Trail
- Syslog Audit Trail
- Operating System and Syslog Audit Records
- Records Always in the Operating System and Syslog Audit Trail

Database Audit Trail (DBA_AUDIT_TRAIL)

The database audit trail consists of a single table named `SYS.AUD$` in the `SYS` schema of the data dictionary of each Oracle database. Several predefined views are provided to help you use the information in this table, such as `DBA_AUDIT_TRAIL`.

Audit trail records can contain different types of information, depending on the events audited and the auditing options set. The partial list in Table 8–2 shows columns that *always* appear in the audit trail. If the data they represent is available, then that data populates the corresponding column. (For certain columns, this list shows the column name displayed in the audit record in parentheses.)

Table 8–2 Columns Shown in the Database Audit Trail (DBA_AUDIT_TRAIL)

Column Description (Name)	Also Appears in the Operating System Audit Trail?
Operating system login user name (`CLIENT USER`)	Yes.
Database user name (`DATABASE USER`)	Yes.
Session identifier	Yes.
Terminal identifier	Yes.
Name of the schema object accessed	Yes.
Operation performed or attempted (`ACTION`)	Yes.
Completion code of the operation	Yes.
Date and time stamp in UTC (Coordinated Universal Time) format	Date and time yes, but not in UTC format.

Table 8–2 (Cont.) Columns Shown in the Database Audit Trail (DBA_AUDIT_TRAIL)

Column Description (Name)	Also Appears in the Operating System Audit Trail?
System privileges used (PRIVILEGE)	Yes.
Proxy Session audit ID	No.
Global User unique ID	No.
Distinguished name	Yes.
Instance number	No.
Process number	No.
Transaction ID	No.
SCN (system change number) for the SQL statement	No.
SQL text that triggered the auditing (SQLTEXT)	No.
Bind values used for the SQL statement, if any (SQLBIND)	No.

Notes: SQLBIND and SQLTEXT are not populated unless you specify AUDIT_TRAIL=DB, EXTENDED in the database initialization file, init.ora. This is because CLOBs are comparatively expensive to populate.

If the database destination for audit records becomes full or is unavailable, and is therefore unable to accept new records, then an audited action cannot complete. Instead, it causes an error message and is not done. In some cases, an operating system log allows such an action to complete.

Operating System Audit Trail

Oracle Database allows audit trail records to be directed to an operating system audit trail if the operating system makes such an audit trail available to Oracle Database. If not, then audit records are written to a file outside the database. The target directory varies by platform: on the Solaris platform, it is $ORACLE_HOME/rdbms/audit, but for other platforms you must check the platform documentation to learn the correct target directory. In Windows, the information is accessed through Event Viewer.

If init.ora specifies AUDIT_TRAIL=XML, then audit records are written to the operating system as XML files. A new dynamic view, V$XML_AUDIT_TRAIL, makes such XML audit records available to DBAs through a SQL query, providing enhanced usability. Querying this view causes all XML files (all files with a .xml extension) in the AUDIT_FILE_DEST directory to be parsed and presented in relational table format. Because XML is a standard document format, many utilities are available to parse and analyze such XML data.

See Also: Operating-system-specific Oracle documentation, to see if this feature has been implemented on your operating system

However, an operating system audit trail or file system can become full and therefore unable to accept new records, including audit records directed to the operating system. In this circumstance, Oracle Database still allows certain actions that are *always* audited to continue, even though the audit record cannot be stored because the operating system destination is full. Using a database audit trail prevents audited actions from completing if their audit records cannot be stored.

System administrators configuring operating system auditing should ensure that the operating system audit trail or the file system does not fill completely. Most operating systems provide administrators with sufficient information and warning to enable them to ensure this does not occur.

Note, however, that configuring auditing to use the database audit trail removes this potential loss of audit information. The Oracle Database server prevents audited events from occurring if the audit trail is unable to accept the database audit record for the statement.

Syslog Audit Trail

One potential security vulnerability for an operating system audit trail is that a privileged user, such as a DBA, can modify or delete audit records. In order to minimize this risk, you can use a syslog audit trail. Syslog is a standard protocol on UNIX-based systems for logging information from different components of a network. Applications call the `syslog()` function to log information to the syslog daemon, which then determines where to log the information. You can configure syslog to log information to a file name `syslog.conf`, to the console, or to a remote, dedicated log host. You can also configure syslog to alert a specified set of users when information is logged.

Because applications, such as an Oracle process, use the `syslog()` function to log information to the `syslog` daemon, a privileged user does not need to have permissions to the file system where messages are logged. For this reason, audit records stored using a syslog audit trail can be more secure than audit records stored using an operating system audit trail. In addition to restricting permissions to a file system for a privileged user, for a syslog audit trail to be secure, neither privileged users nor the Oracle process should have `root` access to the system where the audit records are written.

Note: The security vulnerability that is exposed with an operating system audit trail is not an issue on Windows operating systems, because audit records cannot by modified directly. Instead, audit records on Windows operating systems are stored and monitored through Event Viewer.

Operating System and Syslog Audit Records

The operating system and syslog audit trails are encoded, but are decoded in data dictionary files and error messages. The following fields are included:

- **Action code** describes the operation performed or attempted. The `AUDIT_ACTIONS` data dictionary table contains a list of these codes and their descriptions.
- **Privileges used** describes any system privileges used to perform the operation. The `SYSTEM_PRIVILEGE_MAP` table lists all of these codes and their descriptions.
- **Completion code** describes the result of the attempted operation. Successful operations return a value of zero, and unsuccessful operations return the Oracle error code describing why the operation was unsuccessful.

See Also:

- Table 8–2, " Columns Shown in the Database Audit Trail (DBA_AUDIT_TRAIL)", which also indicates the columns that appear in the operating system audit trail
- *Oracle Database Administrator's Guide* for instructions for creating and using predefined views
- *Oracle Database Error Messages* for a list of completion codes

Records Always in the Operating System and Syslog Audit Trail

Some database-related actions are always recorded into the operating system and syslog audit trails regardless of whether database auditing is enabled. The fact that these records are always created is sometimes referred to as mandatory auditing. The following actions are recorded:

- At instance startup, an audit record is generated that includes the operating system user starting the instance, the terminal identifier of the user, and the date and time stamp. This information is recorded into the operating system or syslog audit trails, because the database audit trail is not available until after startup has successfully completed.
- At instance shutdown, an audit record is generated that details the operating system user shutting down the instance, the terminal identifier of the user, and the date and time stamp.
- During connections made with administrator privileges, an audit record is generated that details the operating system user connecting to Oracle Database with administrator privileges. This record provides accountability regarding users connected with administrator privileges.

On operating systems that do not make an audit trail accessible to Oracle Database, these audit trail records are placed in an Oracle audit trail file in the same directory as background process trace files, and in a similar format.

See Also: Operating-system-specific Oracle documentation for more information about the operating system and syslog audit trail

When Are Audit Records Created?

Standard auditing for the entire database is either enabled or disabled by the security administrator. If it is disabled, then no audit records are created.

Note: Fine-grained auditing uses audit policies applied to individual objects. Therefore, standard audit settings that are on or off for the entire database do not affect fine-grained auditing.

If database auditing is enabled by the security administrator, then individual audit options become effective. These audit options can be set by any authorized database user for database objects he owns.

When auditing is enabled in the database and an action set to be audited occurs, an audit record is generated during the execute phase of the statement.

SQL statements inside PL/SQL program units are individually audited, as necessary, when the program unit is executed.

The generation and insertion of an audit trail record is independent of a user transaction being committed. That is, even if a user transaction is rolled back, the audit trail record remains committed.

Statement and privilege audit options in effect at the time a database user connects to the database remain in effect for the duration of the session. Setting or changing statement or privilege audit options in a session does not take effect in that session. The modified statement or privilege audit options take effect only when the current session ends and a new session is created.

In contrast, changes to schema object audit options become effective for current sessions immediately.

Note: Operations by the SYS user and by users connected through SYSDBA or SYSOPER can be fully audited with the AUDIT_SYS_OPERATIONS initialization parameter. Every successful SQL statement from SYS is audited. This specialized form of auditing audits all actions performed by every user with the SYSDBA privilege and writes only to an operating system location. It is not dependent on the standard auditing parameter, AUDIT_TRAIL. Storing these records in a location separate from the usual database audit trail in the SYS schema provides for greater auditing security.

See Also:

- *Oracle Database Administrator's Guide* for instructions on enabling and disabling auditing
- "SQL, PL/SQL, and Java" in *Oracle Database Concepts* for information about the different phases of SQL statement processing and shared SQL

Statement Auditing

Statement auditing is the selective auditing of related groups of statements regarding a particular type of database structure or schema object, but not a specifically named structure or schema object. These statements fall into the following categories:

- DDL statements: As an example, AUDIT TABLE audits all CREATE and DROP TABLE statements
- DML statements: As an example, AUDIT SELECT TABLE audits all SELECT ... FROM TABLE/VIEW statements, regardless of the table or view

Statement auditing can be broad or focused, for example, by auditing the activities of all database users or of only a select list.

Privilege Auditing

Privilege auditing audits statements that use a system privilege, such as SELECT ANY TABLE. For example, when AUDIT SELECT ANY TABLE is in force, all statements issued by users with the SELECT ANY TABLE privilege are audited.

You can audit the use of any system privilege. Like statement auditing, privilege auditing can audit the activities of all database users or of only a specified list.

If similar statement and privilege audit options are both set, then only a single audit record is generated. For example, if the statement clause TABLE and the system

privilege CREATE TABLE are both audited, then only a single audit record is generated each time a table is created.

Thus privilege auditing does not occur if the action is already permitted by the existing owner and schema object privileges. Privilege auditing is triggered only if they are insufficient, that is, only if what makes the action possible is a system privilege.

Privilege auditing is more focused than statement auditing, because each privilege auditing option audits only specific types of statements, not a related list of statements. For example, the statement auditing clause, TABLE, audits CREATE TABLE, ALTER TABLE, and DROP TABLE statements. However, the privilege auditing option, CREATE TABLE, audits only CREATE TABLE statements, because only the CREATE TABLE statement requires the CREATE TABLE privilege.

Schema Object Auditing

Schema object auditing can audit all SELECT and DML statements permitted by schema object privileges, such as SELECT or DELETE statements on a given table. The GRANT and REVOKE statements that control those privileges are also audited.

You can audit statements that reference tables, views, sequences, standalone stored procedures or functions, and packages, but not individual procedures within packages. Further discussion appears in the next section, entitled Schema Object Audit Options for Views, Procedures, and Other Elements.

Statements that reference clusters, database links, indexes, or synonyms are not audited directly. However, you can indirectly audit access to these schema objects, by auditing the operations that affect the base table.

Schema object audit options are always set for all users of the database. These options cannot be set for a specific list of users. You can set default schema object audit options for all auditable schema objects.

> **See Also:** *Oracle Database SQL Reference* for information about auditable schema objects

Schema Object Audit Options for Views, Procedures, and Other Elements

The definitions for views and procedures (including stored functions, packages, and triggers) reference underlying schema objects. Because of this dependency, some unique characteristics apply to auditing views and procedures, such as the likelihood of generating multiple audit records.

Views and procedures are subject to the enabled audit options on the base schema objects, including the default audit options. These options apply to the resulting SQL statements as well.

Consider the following series of SQL statements:

```
AUDIT SELECT ON employees;

CREATE VIEW employees_departments AS
  SELECT employee_id, last_name, department_id
    FROM employees, departments
    WHERE employees.department_id = departments.department_id;

AUDIT SELECT ON employees_departments;

SELECT * FROM employees_departments;
```

As a result of the query on employees_departments, two audit records are generated: one for the query on the employees_departments view and one for the query on the base table employees (indirectly through the employees_departments view). The query on the base table departments does not generate an audit record because the SELECT audit option for this table is not enabled. All audit records pertain to the user that queried the employees_departments view.

The audit options for a view or procedure are determined when the view or procedure is first used and placed in the shared pool. These audit options remain set until the view or procedure is flushed from, and subsequently replaced in, the shared pool. Auditing a schema object invalidates that schema object in the cache and causes it to be reloaded. Any changes to the audit options of base schema objects are not observed by views and procedures in the shared pool.

In the given example, if the "AUDIT SELECT ON employees;" statement is omitted, then using the employees_departments view will not generate an audit record for the employees table.

Table 8–3 lists auditing actions that were not available before Oracle Database.

Table 8–3 Auditing Actions Newly Enabled by Oracle Database 10g

Object or Element	Newly Auditable Actions
Materialized Views	DELETE, INSERT, SELECT, UPDATE, and FLASHBACK
Tables & views	FLASHBACK
Library	EXECUTE
Java source	EXECUTE
Queue	ENQUEUE and DEQUEUE

Focusing Statement, Privilege, and Schema Object Auditing

Oracle Database lets you focus statement, privilege, and schema object auditing in three areas, as discussed in the following subsections:

- Auditing Statement Executions: Successful, Unsuccessful, or Both
- Number of Audit Records from Multiple Executions of a Statement
- Audit by User

Auditing Statement Executions: Successful, Unsuccessful, or Both

For statement, privilege, and schema object auditing, Oracle Database allows the selective auditing of successful executions of statements, unsuccessful attempts to execute statements, or both. Therefore, you can monitor actions even if the audited statements do not complete successfully. Monitoring unsuccessful SQL can expose users who are snooping or acting maliciously, though of course most unsuccessful SQL is neither.

Auditing an unsuccessful statement execution provides a report only if a valid SQL statement is issued but fails, because it lacks proper authorization or references a nonexistent schema object. Statements that fail to execute because they were not valid cannot be audited.

For example, an enabled privilege auditing option set to audit unsuccessful statement executions audits statements that use the target system privilege but have failed for

other reasons. One example is when a CREATE TABLE auditing condition is set, but some CREATE TABLE statements fail due to lack of quota for the specified tablespace.

When your audit statement includes the WHENEVER SUCCESSFUL clause, you will be able to audit only successful executions of the audited statement.

When your audit statement includes the WHENEVER NOT SUCCESSFUL clause, you will be auditing only unsuccessful executions of the audited statement.

When your audit statement includes neither of the preceding two clauses, you will be able to audit both successful and unsuccessful executions of the audited statement.

Number of Audit Records from Multiple Executions of a Statement

If an audited statement is issued multiple times in a single user session, then your audit trail can have one or more related records. The controlling clause BY ACCESS causes each execution of an auditable operation within a cursor to generate a separate audit record. If you use the BY SESSION clause instead, then your audit trail will contain a single audit record for each session, for each user and schema object. Only one audit record results, no matter how often the statement occurs in that session.

However, several audit options can be set only BY ACCESS:

- All statement audit options that audit DDL statements
- All privilege audit options that audit DDL statements

For all other audit options, BY SESSION is used by default.

This section provides detailed examples of using each clause, in the following subsections:

- BY SESSION
- BY ACCESS

See Also: *Oracle Database SQL Reference*

BY SESSION

For any type of audit (schema object, statement, or privilege), BY SESSION inserts only one audit record in the audit trail, for each user and schema object, during a session that includes an audited action.

A **session** is the time between when a user connects to and disconnects from Oracle Database.

BY SESSION Example 1

Assume the following:

- The SELECT TABLE statement auditing option is set BY SESSION.
- JWARD connects to the database and issues five SELECT statements against the table named departments and then disconnects from the database.
- SWILLIAMS connects to the database and issues three SELECT statements against the table employees and then disconnects from the database.

In this case, the audit trail contains two audit records for the eight SELECT statements, one for each session that issued a SELECT statement.

BY SESSION Example 2

Alternatively, assume the following:

- The SELECT TABLE statement auditing option is set BY SESSION.
- JWARD connects to the database and issues five SELECT statements against the table named departments, and three SELECT statements against the table employees, and then disconnects from the database.

In this case, the audit trail contains two records, one for each schema object against which the user issued a SELECT statement in a session.

> **Note:** If you use the BY SESSION clause when directing audit records to the operating system audit trail, then Oracle Database generates and stores an audit record each time an access is made. Therefore, in this auditing configuration, BY SESSION is equivalent to BY ACCESS.

BY ACCESS

Setting audit BY ACCESS inserts one audit record into the audit trail for each execution of an auditable operation within a cursor. Events that cause cursors to be reused include the following:

- An application, such as Oracle Forms, holding a cursor open for reuse
- Subsequent execution of a cursor using new bind variables
- Statements executed within PL/SQL loops where the PL/SQL engine optimizes the statements to reuse a single cursor

Note that auditing is *not* affected by whether a cursor is shared. Each user creates her or his own audit trail records on first execution of the cursor.

For example, assume that:

- The SELECT TABLE statement auditing option is set BY ACCESS.
- JWARD connects to the database and issues five SELECT statements against the table named departments and then disconnects from the database.
- SWILLIAMS connects to the database and issues three SELECT statements against the departments table and then disconnects from the database.

The single audit trail contains eight records for the eight SELECT statements.

Audit by User

Statement and privilege audit options can audit statements issued by any user or statements issued by a specific list of users. By focusing on specific users, you can minimize the number of audit records generated.

Audit by User Example To audit statements by the users SCOTT and BLAKE that query or update a table or view, use the following statements:

```
AUDIT SELECT TABLE, UPDATE TABLE
     BY scott, blake;
```

> **See Also:** *Oracle Database SQL Reference* for more information about auditing by user

Auditing in a Multitier Environment

In a multitier environment, Oracle can preserve the identity of a client through all tiers. Thus, you can audit actions taken on behalf of the client by a middle-tier application. To do so, use the `BY proxy` clause in your `AUDIT` statement.

This clause allows you a few options. You can:

- Audit SQL statements issued by the specific proxy on its own behalf
- Audit statements executed on behalf of a specified user or users
- Audit all statements executed on behalf of any user

The middle tier can also set the user client identity in a database session, enabling audit of end-user actions through the mid-tier application. The end-user client identity then shows up in the audit trail.

See Also:

- *Oracle Database Application Developer's Guide - Fundamentals*
- *Oracle Call Interface Programmer's Guide*
- *PL/SQL User's Guide and Reference*

Fine-Grained Auditing

Fine-Grained Auditing (FGA) enables you to monitor data access based on content. A built-in audit mechanism in the database prevents users from bypassing the audit.

While Oracle Database triggers can potentially monitor DML actions such as `INSERT`, `UPDATE`, and `DELETE`, monitoring `SELECT` statements can be costly. In some cases, a trigger may audit too much, and in others, its effectiveness or completeness may be uncertain. Triggers also do not enable users to define their own alert action in response to a triggered audit, beyond simply inserting an audit record into the audit trail.

FGA provides an extensible interface for creating policies to audit SELECT and DML statements on tables and views. The `DBMS_FGA` package administers these value-based audit policies. Using `DBMS_FGA`, the security administrator creates an audit policy on the target object. If any rows returned from a query match the audit condition, then an audit event entry is inserted into the fine-grained audit trail. This entry includes all the information reported in the regular audit trail. See the Audit Records and Audit Trails section on page 8-3. Only one row of audit information is inserted into the audit trail for every FGA policy that evaluates to true. The extensibility framework in FGA also enables administrators optionally to define an appropriate audit event handler to process the event, for example by sending an alert page to the administrator.

The administrator uses the `DBMS_FGA.ADD_POLICY` interface to define each FGA policy for a table or view, identifying any combination of `SELECT`, `UPDATE`, `DELETE`, or `INSERT` statements.

FGA policies associated with a table or view may also specify *relevant columns*, so that any specified statement type affecting a relevant column is audited. If no *relevant column* is specified, then auditing applies to all columns, that is, auditing occurs whenever any specified statement type affects any column, independent of whether any rows are returned or not.

The relevant-column capability enables you to hone in on particularly important types of data to audit. Examples include privacy-relevant columns, such as those containing social security numbers, salaries, patient diagnoses, and so on. You could combine the

fine-grained audit records to surface queries that had addressed both name and social security number, a potential violation of privacy security laws.

One added benefit is that the audit records created are relevant, because they relate to specific data of interest or concern. Another benefit is that fewer total audit records need be generated, because each is now more specific and useful than what could be tracked in earlier releases.

See Also:

- Chapter 12, "Configuring and Administering Auditing"
- *Oracle Database Application Developer's Guide - Fundamentals*
- The DBMS_FGA chapter in *PL/SQL Packages and Types Reference*

Part III

Security Implementation, Configuration, and Administration

Part III presents the details of configuring and administering Oracle Database security features.

This part contains the following chapter:

9

Secure External Password Store

Password credentials for connecting to databases can now be stored in a client-side Oracle wallet, a secure software container used to store authentication and signing credentials.

This wallet usage can simplify large-scale deployments that rely on password credentials for connecting to databases. When this feature is configured, application code, batch jobs, and scripts no longer need embedded user names and passwords. Risk is reduced because such passwords are no longer exposed in the clear, and password management policies are more easily enforced without changing application code whenever user names or passwords change.

This feature is explained in the following sections:

- How Does the External Password Store Work?
- Configuring Clients to Use the External Password Store
- Managing External Password Store Credentials

> **See Also:** *Oracle Database Advanced Security Administrator's Guide* for general information about Oracle wallets

---

> **Note:** The external password store of the wallet is separate from the area where public key infrastructure (PKI) credentials are stored. Consequently, you cannot use Oracle Wallet Manager to manage credentials in external password store of the wallet. Instead, the command-line utility, `mkstore`, is provided to manage these credentials.

How Does the External Password Store Work?

Typically, users (including applications, batch jobs, and scripts) connect to databases by using a standard `CONNECT` statement that specifies a *`database_connect_string`*. This string can include a user name and password, and an Oracle Net service name identifying the database on an Oracle network. For example, the service name could be the URL that uniquely identifies that database, or a TNS alias you entered in the `tnsnames.ora` file in the database. Another possibility is a *`host:port:sid`* string.

The following examples are standard `CONNECT` statements that could be used for a client that is not configured to use the external password store:

```
connect salesapp/2Ip6Cg8@sales_db.us.acme.com
or
```

```
connect salesapp/2Ip6Cg8@ORASALES
or
connect salesapp/2Ip6Cg8@ourhost37:1527:DB17
```

In these examples, `salesapp` is the user name and `2Ip6Cg8` is the password, with the unique connect string for the database shown as specified in three different ways. You could use its URL `sales_db.us.acme.com`, or its TNS alias `ORASALES` from the `tnsnames.ora` file, or its *`host:port:sid`* string.

However, when clients are configured to use the secure external password store, applications can connect to a database with the following `CONNECT` statement syntax, without specifying database login credentials:

```
connect /@db_connect_string

where db_connect_string is a valid connect string to access the intended database,
such as the service name, URL, or alias as illustrated in the earlier examples.
```

In this case, the database credentials, username and password, are securely stored in an Oracle wallet created for this purpose. The autologin feature of this wallet is turned on so the system does not need a password to open the wallet. From the wallet, it gets the credentials to access the database for the user they represent.

> **See Also:** Refer to *Oracle Database Advanced Security Administrator's Guide* for information about autologin wallets.

Configuring Clients to Use the External Password Store

If your client is already configured to use external authentication, such as Windows native authentication or Secure Sockets Layer (SSL), then that authentication method will be used. The same credentials used for such authentication are typically also used to log in to the database.

For clients not using such authentication methods or wanting to override them for database authentication, a new parameter (`SQLNET.WALLET_OVERRIDE`) in `sqlnet.ora` can be set to `TRUE`. The default value for `SQLNET.WALLET_OVERRIDE` is `FALSE`, allowing standard use of authentication credentials as before.

If you want a client to use the secure external password store feature, then perform the following configuration tasks.

To enable clients to use the external password store:

1. Create a wallet on the client by using the following syntax at the command line:

   ```
   mkstore -wrl <wallet_location> -create
   ```

   `wallet_location` is the path to the directory where you want to create and store the wallet. This command creates an Oracle wallet with the autologin feature enabled at the location you specify. The autologin feature enables the client to access the wallet contents without supplying a password. Refer to *Oracle Database Advanced Security Administrator's Guide* for information about autologin wallets.

2. Create database connection credentials in the wallet by using the following syntax at the command line:

   ```
   mkstore -wrl <wallet_location> -createCredential <db_connect_string> <username>
   <password>
   ```

   `wallet_location` is the path to the directory where you created the wallet in Step 1. The *`db_connect_string`* can be the TNS alias you use to specify the

database in the tnsnames.ora file or any service name you use to identify the database on an Oracle network. The *username* and *password* are the database login credentials.

Repeat this step for each database you want accessible using the CONNECT /@*db_connect_string* syntax.

> **Note:** The *db_connect_string* used in the CONNECT /@*db_connect_string* statement must be identical to the *db_connect_string* specified in the -createCredential command.

3. In the client sqlnet.ora file, enter the WALLET_LOCATION parameter and set it to the directory location of the wallet you created in Step 1.

   For example, if you created the wallet in $ORACLE_HOME/network/admin and your Oracle home is set to /private/ora102, then you need to enter the following into your client sqlnet.ora file:

```
WALLET_LOCATION =
  (SOURCE =
    (METHOD = FILE)
    (METHOD_DATA =
      (DIRECTORY = /private/ora102/network/admin)
    )
  )
```

4. In the client sqlnet.ora file, enter the SQLNET.WALLET_OVERRIDE parameter and set it to TRUE as follows:

```
SQLNET.WALLET_OVERRIDE = TRUE
```

   This setting causes all CONNECT /@*db_connect_string* statements to use the information in the wallet at the specified location to authenticate to databases.

   When external authentication is in use, an authenticated user with such a wallet can use the CONNECT /@*db_connect_string* syntax to access the previously specified databases without providing a user name and password. However, if a user fails that external authentication, then these connect statements also fail.

> **Note:** If an application uses SSL for encryption, then the sqlnet.ora parameter, SQLNET.AUTHENTICATION_SERVICES, specifies SSL and an SSL wallet is created. If this application wants to use secret store credentials to authenticate to databases (instead of the SSL certificate), then those credentials must be stored in the SSL wallet. After SSL authentication, if SQLNET.WALLET_OVERRIDE = TRUE, then the user names and passwords from the wallet are used to authenticate to databases. If SQLNET.WALLET_OVERRIDE = FALSE, then the SSL certificate is used.

Example 9–1 shows a sample sqlnet.ora file with the WALLET_LOCATION and the SQLNET.WALLET_OVERRIDE parameters set as described in Steps 3 and 4.

Example 9–1 Sample SQLNET.ORA File with Wallet Parameters Set

```
WALLET_LOCATION =
   (SOURCE =
```

```
    (METHOD = FILE)
    (METHOD_DATA =
      (DIRECTORY = /private/ora102/network/admin)
    )
   )

SQLNET.WALLET_OVERRIDE = TRUE
SSL_CLIENT_AUTHENTICATION = FALSE
SSL_VERSION = 0
```

Managing External Password Store Credentials

This section summarizes the following tasks you can perform to manage credentials in the external password store by using the `mkstore` command-line utility:

- Listing External Password Store Contents
- Adding Credentials to an External Password Store
- Modifying Credentials in an External Password Store
- Deleting Credentials from an External Password Store

Listing External Password Store Contents

Periodically, you may want to view all contents of a client wallet external password store, or you may need to check specific credentials by viewing them. Listing the external password store contents provides information you can use to decide whether to add or delete credentials from the store.

To list the contents of the external password store, enter the following command at the command line:

```
mkstore -wrl <wallet_location> -listCredential
```

wallet_location specifies the path to the directory where the wallet, whose external password store contents you want to view, is located. This command lists all of the credential database service names (aliases) and the corresponding user name (schema) for that database. Passwords are not listed.

Adding Credentials to an External Password Store

You can store multiple credentials in one client wallet. For example, if a client batch job connects to `hr_database` and a script connects to `sales_database`, then you can store the login credentials in the same client wallet. You cannot, however, store multiple credentials (for logging in to multiple schemas) for the same database in the same wallet. If you have multiple login credentials for the same database, then they must be stored in separate wallets.

To add database login credentials to an existing client wallet, enter the following command at the command line:

```
mkstore -wrl <wallet_location> -createCredential <db_alias> <username> <password>
```

wallet_location is the path to the directory where the client wallet to which you want to add credentials is stored. *db_alias* can be the TNS alias you use to specify the database in the `tnsnames.ora` file or any service name you use to identify the database on an Oracle network. The *username* and *password* are the database login credentials for the schema to which your application connects.

Modifying Credentials in an External Password Store

If database connect strings change, then you can modify the database login credentials that are stored in the wallet.

To modify database login credentials in a wallet, enter the following command at the command line:

```
mkstore -wrl <wallet_location> -modifyCredential <dbase_alias> <username>
<password>
```

wallet_location is the path to the directory where the wallet is located. The *db_alias* is a new or different alias you want to use to identify the database. It can be a TNS alias you use to specify the database in the `tnsnames.ora` file or any service name you use to identify the database on an Oracle network. The *username* and *password* are the new or different database login credentials.

Deleting Credentials from an External Password Store

If a database no longer exists or if you want to disable connections to a specific database, then you can delete all login credentials for that database from the wallet.

To delete database login credentials from a wallet, enter the following command at the command line:

```
mkstore -wrl <wallet_location> -deleteCredential <db_alias>
```

wallet_location is the path to the directory where the wallet is located. The *db_alias* can be the TNS alias you use to specify the database in the `tnsnames.ora` file, or any service name you use to identify the database on an Oracle network.

10

Administering Authentication

Authentication is the process of verifying the identity of a user, device, or other entity in a computer system, often as a prerequisite to granting access to resources in a system.

User Authentication Methods

Oracle provides several means for users to be authenticated before they are allowed to create a database session, as discussed in the following sections:

Topics: You can define users who are	Links to Topics
Identified and authenticated by the database, which is called **database authentication**.	Database Authentication
Authenticated by the operating system or network service, which is called **external authentication**.	External Authentication
Authenticated **globally** by Secure Sockets Layer (SSL), called **global users**, whose database access is through **global roles**, authorized by an enterprise directory.	Global Authentication and Authorization
Allowed to connect through a middle-tier server that authenticates the user, assumes that identity, and can enable specific roles for the user. This combination of actions and abilities is called **proxy authentication** and authorization.	Proxy Authentication and Authorization

Database Authentication

If you choose database authentication for a user, then administration of the user account including authentication of that user is performed entirely by Oracle Database. To have Oracle Database authenticate a user, specify a password for the user when you create or alter the user. Users can change their password at any time. Passwords are stored in an encrypted format. While user names can be multibyte, each password must be made up of single-byte characters, even if your database uses a multibyte character set.

Note: Oracle recommends that you encode user names and passwords in ASCII or EBCDIC characters only, depending on your platform. Doing so will maintain compatibility for supporting future changes to your database character set.

By using user names or passwords on characters that expand in size when migrated to a new target character set can cause login difficulties. Authentication can fail after such a migration because the encrypted user names and passwords in the data dictionary are not updated during a migration to a new database character set.

For example, assuming the current database character set is WE8MSWIN1252 and the target database character set is UTF8, the user name `scött` (o with an umlaut) will change from 5 bytes to 6 bytes in UTF8. the user `scött` will no longer be able to log in.

If user names and passwords are not based on ASCII or EBCDIC characters and a migration to a new character set occurs, then the affected user names and passwords will need to be reset.

To enhance security when using database authentication, Oracle recommends the use of password management, including account locking, password aging and expiration, password history, and password complexity verification.

See Also: "Password Management Policy" on page 7-9

Creating a User Who Is Authenticated by the Database

The following statement creates a user who is identified and authenticated by Oracle Database. User `scott` must specify the password `tiger` whenever connecting to Oracle Database.

```
CREATE USER scott IDENTIFIED BY tiger;
```

See Also: *Oracle Database SQL Reference* for more information about valid passwords, and how to specify the `IDENTIFIED BY` clause in the `CREATE USER` and `ALTER USER` statements

Advantages of Database Authentication

Following are the advantages of database authentication:

- User accounts and all authentication are controlled by the database. There is no reliance on anything outside of the database.
- Oracle provides strong password management features to enhance security when using database authentication.
- It is easier to administer when there are small user communities.

External Authentication

When you choose external authentication for a user, the user account is maintained by Oracle Database, but password administration and user authentication is performed by an external service. This external service can be the operating system or a network service, such as Oracle Net.

With external authentication, your database relies on the underlying operating system or network authentication service to restrict access to database accounts. A database

password is not used for this type of login. If your operating system or network service permits, then you can have it authenticate users. Set the initialization parameter OS_AUTHENT_PREFIX, and use this prefix in Oracle Database user names. The OS_AUTHENT_PREFIX parameter defines a prefix that Oracle Database adds to the beginning of every user's operating system account name. Oracle compares the prefixed user name with the Oracle user names in the database when a user attempts to connect.

For example, assume that OS_AUTHENT_PREFIX is set as follows:

```
OS_AUTHENT_PREFIX=OPS$
```

> **Note:** The text of the OS_AUTHENT_PREFIX initialization parameter is case-sensitive on some operating systems. Refer to your operating system specific Oracle documentation for more information about this initialization parameter.

If a user with an operating system account named tsmith is to connect to an Oracle database installation and be authenticated by the operating system, then Oracle Database checks that there is a corresponding database user OPS$tsmith and, if so, allows the user to connect. All references to a user authenticated by the operating system must include the prefix, OPS$, as seen in OPS$tsmith.

The default value of this parameter is OPS$ for backward compatibility with previous versions of Oracle Database. However, you might prefer to set the prefix value to some other string or a null string (an empty set of double quotes: ""). Using a null string eliminates the addition of any prefix to operating system account names, so that Oracle user names exactly match operating system user names.

After you set OS_AUTHENT_PREFIX, it should remain the same for the life of a database. If you change the prefix, then any database user name that includes the old prefix cannot be used to establish a connection, unless you alter the user name to have it use password authentication.

Creating a User Who Is Authenticated Externally

The following statement creates a user who is identified by Oracle and authenticated by the operating system or a network service. This example assumes that OS_AUTHENT_PREFIX = "".

```
CREATE USER scott IDENTIFIED EXTERNALLY;
```

Using CREATE USER ... IDENTIFIED EXTERNALLY, you create database accounts that must be authenticated by the operating system or network service. Oracle will then rely on this external login authentication when it provides that specific operating system user with access to the database resources of a specific user.

> **See Also:** *Oracle Database Advanced Security Administrator's Guide* for more information about external authentication

Operating System Authentication

By default, Oracle allows operating-system-authenticated logins only over secure connections, which precludes using Oracle Net and a shared server configuration. This default restriction prevents a remote user from impersonating another operating system user over a network connection.

Setting REMOTE_OS_AUTHENT to TRUE in the database initialization parameter file forces the RDBMS to accept the client operating system user name received over a nonsecure connection and use it for account access. Because clients, in general, such as PCs, are not trusted to perform operating system authentication properly, it is very poor security practice to turn on this feature.

The default setting, REMOTE_OS_AUTHENT = FALSE, creates a more secure configuration that enforces proper, server-based authentication of clients connecting to an Oracle database.

Any change to this parameter takes effect the next time you start the instance and mount the database. Generally, user authentication through the host operating system offers faster and more convenient connection to Oracle without specifying a separate database user name or password. Also, user entries correspond in the database and operating system audit trails.

Network Authentication

Network authentication is performed using Oracle Advanced Security, which can be configured to use a third-party service such as Kerberos. If you are using Oracle Advanced Security as your only external authentication service, then the REMOTE_OS_AUTHENT parameter setting is irrelevant, because Oracle Advanced Security only allows secure connections.

Advantages of External Authentication

Following are the advantages of external authentication:

- More choices of authentication mechanism are available, such as smart cards, fingerprints, Kerberos, or the operating system.
- Many network authentication services, such as Kerberos support single sign-on, enabling users to have fewer passwords to remember.
- If you are already using some external mechanism for authentication, such as one of those listed earlier, then there may be less administrative overhead to use that mechanism with the database as well.

Global Authentication and Authorization

Oracle Advanced Security enables you to centralize management of user-related information, including authorizations, in an LDAP-based directory service. Users can be identified in the database as global users, meaning that they are authenticated by SSL and that the management of these users is done outside of the database by the centralized directory service. Global roles are defined in a database and are known only to that database, but authorizations for such roles is done by the directory service.

> **Note:** You can also have users authenticated by SSL, whose authorizations are not managed in a directory, that is, they have local database roles only. Refer to the *Oracle Database Advanced Security Administrator's Guide* for details.

This centralized management enables the creation of **enterprise users** and **enterprise roles**. Enterprise users are defined and managed in the directory. They have unique identities across the enterprise and can be assigned enterprise roles that determine their access privileges across multiple databases. An enterprise role consists of one or more global roles, and might be thought of as a container for global roles.

Creating a User Who Is Authorized by a Directory Service

You have a couple of options as to how you specify users who are authorized by a directory service.

Private Schemas The following statement illustrates the creation of a global user with a private schema, authenticated by SSL, and authorized by the enterprise directory service:

```
CREATE USER scott
    IDENTIFIED GLOBALLY AS 'CN=scott,OU=division1,O=oracle,C=US';
```

The string provided in the `AS` clause provides an identifier (**distinguished name,** or **DN**) meaningful to the enterprise directory.

In this case, `scott` is truly a global user. But, the disadvantage here is that user `scott` must then be created in every database that he must access, plus the directory.

Shared Schemas Multiple enterprise users can share a single schema in the database. These users are authorized by the enterprise directory service but do not own individual private schemas in the database. These users are not individually created in the database. They connect to a shared schema in the database.

The process of creating a schema-independent user is as follows:

1. Create a shared schema in the database as follows.

   ```
   CREATE USER appschema INDENTIFIED GLOBALLY AS '';
   ```

2. In the directory, you now create multiple enterprise users, and a mapping object.

   The mapping object tells the database how you want to map users' DNs to the shared schema. You can either do a full DN mapping (one directory entry for each unique DN), or you can map, for example, every user containing the following DN components to the `appschema`:

   ```
   OU=division,O=Oracle,C=US
   ```

> **See Also:** Refer to the *Oracle Database Enterprise User Administrator's Guide* for an explanation of these mappings.

Most users do not need their own schemas, and implementing schema-independent users divorces users from databases. You create multiple users who share the same schema in a database, and as enterprise users, they can access shared schemas in other databases as well.

Advantages of Global Authentication and Global Authorization

Some of the advantages of global user authentication and authorization are the following:

- Provides strong authentication using SSL, Kerberos, or Windows NT native authentication
- Enables centralized management of users and privileges across the enterprise
- Is easy to administer: You do not have to create a schema for every user in every database in the enterprise
- Facilitates single sign-on: Users only need to sign on once to access multiple databases and services. Further, users using passwords can have a single

password to access multiple databases accepting password-authenticated enterprise users.

- Because global user authentication and authorization provide password-based access, previously defined password-authenticated database users can be migrated to the directory (using the User Migration Utility) to be centrally administered. This makes global authentication and authorization available for prior Oracle release clients that are still supported.
- CURRENT_USER database links connect as a global user. A local user can connect as a global user in the context of a stored procedure, that is, without storing the global user password in a link definition.

See Also: The following books contain additional information about global authentication and authorization and enterprise users and roles:

- *Oracle Database Advanced Security Administrator's Guide*
- *Oracle Database Enterprise User Administrator's Guide*

Proxy Authentication and Authorization

It is possible to design a middle-tier server to proxy clients in a secure fashion.

Oracle provides three forms of proxy authentication:

- The middle-tier server authenticates itself with the database server and a client, in this case an application user or another application, authenticates itself with the middle-tier server. Client identities can be maintained all the way through to the database.
- The client, in this case a database user, is not authenticated by the middle-tier server. The clients identity and database password are passed through the middle-tier server to the database server for authentication.
- The client, in this case a global user, is authenticated by the middle-tier server, and passes one of the following through the middle tier for retrieving the client's user name.
  - Distinguished name (DN)
  - Certificate

Note: The use of certificates for proxy authentication may not be supported in future Oracle Database releases.

In all cases, the middle-tier server must be authorized to act on behalf of the client by the administrator.

To authorize a middle-tier server to proxy a client, use the GRANT CONNECT THROUGH clause of the ALTER USER statement. You can also specify roles that the middle tier is permitted to activate when connecting as the client. Operations done on behalf of a client by a middle-tier server can be audited.

The PROXY_USERS data dictionary view can be queried to see which users are currently authorized to connect through a middle tier.

Use the REVOKE CONNECT THROUGH clause of ALTER USER to disallow a proxy connection.

See Also:

- *Oracle Call Interface Programmer's Guide* and *Oracle Database Application Developer's Guide - Fundamentals* for details about designing a middle-tier server to proxy users
- *Oracle Database SQL Reference* for a description and syntax of the proxy clause for `ALTER USER`
- "Auditing in a Multitier Environment" on page 8-12 for details about auditing operations done on behalf of a user by a middle tier

Authorizing a Middle Tier to Proxy and Authenticate a User

The following statement authorizes the middle-tier server `appserve` to connect as user `bill`. It uses the `WITH ROLE` clause to specify that `appserve` activate all roles associated with `bill`, except `payroll`.

```
ALTER USER bill
    GRANT CONNECT THROUGH appserve
    WITH ROLE ALL EXCEPT payroll;
```

To revoke the middle-tier server (`appserve`) authorization to connect as user `bill`, the following statement is used:

```
ALTER USER bill REVOKE CONNECT THROUGH appserve;
```

Authorizing a Middle Tier to Proxy a User Authenticated by Other Means

Use the `AUTHENTICATED USING` clause of the `ALTER USER ... GRANT CONNECT THROUGH` statement to authorize a user to be proxied, but not authenticated, by a middle tier. Currently, `PASSWORD` is the only means supported.

The following statement illustrates this form of authentication:

```
ALTER USER mary
    GRANT CONNECT THROUGH midtier
    AUTHENTICATED USING PASSWORD;
```

In the preceding statement, middle-tier server `midtier` is authorized to connect as user `mary`, and `midtier` must also pass the user password to the database server for authorization.

11

Administering User Privileges, Roles, and Profiles

Many tasks, with many interwoven considerations, are involved in administering user privileges, roles, and profiles. These necessary operations and principles are discussed in the following sections:

Managing Oracle Users

Each Oracle database has a list of valid database users. To access a database, a user must run a database application and connect to the database instance using a valid user name defined in the database. This section explains how to manage users for a database, and contains the following topics:

> **See Also:** *Oracle Database SQL Reference* for more information about SQL statements used for managing users

Creating Users

You create a database user with the `CREATE USER` statement.To create a user, you must have the `CREATE USER` system privilege. Because it is a powerful privilege, a

DBA or security administrator is normally the only user who has the CREATE USER system privilege.

Example 11–1 creates a user and specifies the user password, default tablespace, temporary tablespace where temporary segments are created, tablespace quotas, and profile.

Example 11–1 Create a User and Grant the Create Session System Privilege

```
CREATE USER jward
    IDENTIFIED BY AZ7BC2
    DEFAULT TABLESPACE data_ts
    QUOTA 100M ON test_ts
    QUOTA 500K ON data_ts
    TEMPORARY TABLESPACE temp_ts
    PROFILE clerk;
GRANT create session TO jward;
```

A newly created user cannot connect to the database until granted the CREATE SESSION system privilege.

Note: As administrator, you should create your own roles and assign only those privileges that are needed. For example, many users formerly granted the CONNECT privilege did not need the additional privileges CONNECT used to provide. Instead, only CREATE SESSION was actually needed, and in fact that is the only privilege CONNECT presently retains.

Creating its own roles gives an organization detailed control of the privileges it assigns, and protects it in case Oracle changes the roles that it defines. For example, both CONNECT and RESOURCE roles will be deprecated in future Oracle versions.

This section refers to the preceding example as it discusses the following aspects of creating a user:

- Specifying a Name
- Setting Up User Authentication
- Assigning a Default Tablespace
- Assigning Tablespace Quotas
- Assigning a Temporary Tablespace
- Specifying a Profile
- Setting Default Roles

See Also: Granting System Privileges and Roles on page 25-11

Specifying a Name

Within each database, a user name must be unique with respect to other user names and roles. A user and role cannot have the same name. Furthermore, each user has an associated schema. Within a schema, each schema object must have a unique name.

Setting Up User Authentication

In Example 11–1, the new user is to be authenticated using the database. In this case, the connecting user must supply the correct password to the database to connect successfully.

Selecting and specifying the method of user authentication is discussed in "User Authentication Methods" on page 10-1.

Assigning a Default Tablespace

Each user should have a default tablespace. When a user creates a schema object and specifies no tablespace to contain it, Oracle Database stores the object in the default user tablespace.

The default setting for the default tablespaces of all users is the `SYSTEM` tablespace. If a user does not create objects, and has no privileges to do so, then this default setting is fine. However, if a user is likely to create any type of object, then you should specifically assign the user a default tablespace. Using a tablespace other than `SYSTEM` reduces contention between data dictionary objects and user objects for the same data files. In general, it is not advisable for user data to be stored in the `SYSTEM` tablespace.

You can create a permanent default tablespace other than `SYSTEM` at the time of database creation, to be used as the database default for permanent objects. By separating the user data from the system data, you reduce the likelihood of problems with the `SYSTEM` tablespace, which can in some circumstances cause the entire database to become nonfunctional. This default permanent tablespace is not used by system users, that is, `SYS`, `SYSTEM`, and `OUTLN`, whose default permanent tablespace remains `SYSTEM`. A tablespace designated as the default permanent tablespace cannot be dropped. To accomplish this goal, another tablespace must first be designated as the default permanent tablespace. It is possible to `ALTER` the default permanent tablespace to another tablespace, affecting all users or objects created after the `ALTER DDL` commits.

You can also set a user default tablespace during user creation, and change it later with the `ALTER USER` statement. Changing the user default tablespace affects only objects created after the setting is changed.

When you specify the default tablespace for a user, also specify a quota on that tablespace.

In Example 11–1, the default tablespace for user `jward` is `data_ts`, and his quota on that tablespace is `500K`.

Assigning Tablespace Quotas

You can assign each user a tablespace quota for any tablespace (except a temporary tablespace). Assigning a quota does the following things:

- Users with privileges to create certain types of objects can create those objects in the specified tablespace.
- Oracle Database limits the amount of space that can be allocated for storage of a user's objects within the specified tablespace to the amount of the quota.

By default, a user has no quota on any tablespace in the database. If the user has the privilege to create a schema object, then you must assign a quota to allow the user to create objects. Minimally, assign users a quota for the default tablespace, and additional quotas for other tablespaces in which they can create objects.

You can assign a user either individual quotas for a specific amount of disk space in each tablespace or an unlimited amount of disk space in all tablespaces. Specific quotas prevent a user's objects from consuming too much space in the database.

You can assign quotas to a user tablespace when you create the user, or add or change quotas later. If a new quota is less than the old one, then the following conditions hold true:

- If a user has already exceeded a new tablespace quota, then the user's objects in the tablespace cannot be allocated more space until the combined space of these objects falls below the new quota.
- If a user has not exceeded a new tablespace quota, or if the space used by the user's objects in the tablespace falls under a new tablespace quota, then the user's objects can be allocated space up to the new quota.

Revoking User Ability to Create Objects in a Tablespace You can revoke the ability of a user to create objects in a tablespace by changing the current quota of the user to zero. After a quota of zero is assigned, the user's objects in the tablespace remain, but new objects cannot be created and existing objects cannot be allocated any new space.

UNLIMITED TABLESPACE System Privilege To permit a user to use an unlimited amount of any tablespace in the database, grant the user the `UNLIMITED TABLESPACE` system privilege. This overrides all explicit tablespace quotas for the user. If you later revoke the privilege, then explicit quotas again take effect. You can grant this privilege only to users, not to roles.

Before granting the `UNLIMITED TABLESPACE` system privilege, you must consider the consequences of doing so.

Advantage:

You can grant a user unlimited access to all tablespaces of a database with one statement.

Disadvantages:

- The privilege overrides all explicit tablespace quotas for the user.
- You cannot selectively revoke tablespace access from a user with the `UNLIMITED TABLESPACE` privilege. You can grant selective or restricted access only after revoking the privilege.

Assigning a Temporary Tablespace

Each user also should be assigned a temporary tablespace. When a user executes a SQL statement that requires a temporary segment, Oracle stores the segment in the user's temporary tablespace. These temporary segments are created by the system when doing sorts or joins and are owned by `SYS`, which has resource privileges in all tablespaces.

In the previous `CREATE USER` statement, the temporary tablespace of jward is `temp_ts`, a tablespace created explicitly to contain only temporary segments. Such a tablespace is created using the `CREATE TEMPORARY TABLESPACE` statement.

If the temporary tablespace of a user is not explicitly set, then the user is assigned the default temporary tablespace that was specified at database creation, or by an `ALTER DATABASE` statement at a later time. If there is no default temporary tablespace explicitly assigned, then the default is the `SYSTEM` tablespace or another permanent default established by the system administrator. It is not advisable for user data to be stored in the `SYSTEM` tablespace. Also, assigning a tablespace to be used specifically as

a temporary tablespace eliminates file contention among temporary segments and other types of segments.

> **Note:** If your SYSTEM tablespace is locally managed, then users must be assigned a specific default (locally managed) temporary tablespace. They may not be allowed to default to using the SYSTEM tablespace because temporary objects cannot be placed in permanent locally managed tablespaces.

You can set the temporary tablespace for a user at user creation, and change it later using the ALTER USER statement. Do not set a quota for temporary tablespaces. You can also establish tablespace groups instead of assigning individual temporary tablespaces.

> **See Also:** The following sections in Chapter 8, Managing Tablespaces in the *Oracle Database Administrator's Guide*::
>
> - Temporary Tablespaces
> - Multiple Temporary Tablespaces: Using Tablespace Groups

Specifying a Profile

You also specify a profile when you create a user. A profile is a set of limits on database resources and password access to the database. If no profile is specified, then the user is assigned a default profile.

> **See Also:** "Managing Resources with Profiles" on page 11-10

Setting Default Roles

You cannot set default roles for a user in the CREATE USER statement. When you first create a user, the default role setting for the user is ALL, which causes all roles subsequently granted to the user to be default roles. Use the ALTER USER statement to change the default roles for the user.

Altering Users

Users can change their own passwords. However, to change any other option of a user security domain, you must have the ALTER USER system privilege. Security administrators are typically the only users that have this system privilege, as it allows a modification of *any* user security domain. This privilege includes the ability to set tablespace quotas for a user on any tablespace in the database, even if the user performing the modification does not have a quota for a specified tablespace.

You can alter user security settings with the ALTER USER statement. Changing user security settings affects the future user sessions, not current sessions.

The following statement alters the security settings for the user, avyrros:

```
ALTER USER avyrros
    IDENTIFIED EXTERNALLY
    DEFAULT TABLESPACE data_ts
    TEMPORARY TABLESPACE temp_ts
    QUOTA 100M ON data_ts
    QUOTA 0 ON test_ts
    PROFILE clerk;
```

The ALTER USER statement here changes the security settings for the user avyrros as follows:

- Authentication is changed to use the operating system account of the user avyrros.
- The default and temporary tablespaces are explicitly set for user avyrros.
- avyrros is given a 100M quota for the data_ts tablespace.
- The quota on the test_ts is revoked for the user avyrros.
- avyrros is assigned the clerk profile.

Changing User Authentication Mechanism

Most non-DBA users can still change their own passwords with the ALTER USER statement, as follows:

```
ALTER USER andy
   IDENTIFIED BY swordfish;
```

No special privileges (other than those to connect to the database) are required for a user to change passwords. Users should be encouraged to change their passwords frequently.

Users must have the ALTER USER privilege to switch between methods of authentication. Usually, only an administrator has this privilege.

> **See Also:** "User Authentication Methods" on page 10-1 for information about authentication methods that are available for Oracle Database users

Changing User Default Roles

A default role is one that is automatically enabled for a user when the user creates a session. You can assign a user zero or more default roles.

> **See Also:** "Managing User Roles" on page 11-15

Dropping Users

When a user is dropped, the user and associated schema are removed from the data dictionary and all schema objects contained in the user schema, if any, are immediately dropped.

> **Notes:**
>
> - If a user schema and associated objects must remain but the user must be denied access to the database, then revoke the CREATE SESSION privilege from the user.
> - Do not attempt to drop the SYS or SYSTEM user. Doing so will corrupt your database.

A user that is currently connected to a database cannot be dropped. To drop a connected user, you must first terminate the user sessions using the SQL statement ALTER SYSTEM with the KILL SESSION clause.

You can drop a user from a database using the DROP USER statement. To drop a user and all the user schema objects (if any), you must have the DROP USER system privilege. Because the DROP USER system privilege is powerful, a security administrator is typically the only type of user that has this privilege.

If the user's schema contains any dependent schema objects, then use the CASCADE option to drop the user and all associated objects and foreign keys that depend on the tables of the user successfully. If you do not specify CASCADE and the user schema contains dependent objects, then an error message is returned and the user is not dropped. Before dropping a user whose schema contains objects, thoroughly investigate which objects the user's schema contains and the implications of dropping them. Pay attention to any unknown cascading effects. For example, if you intend to drop a user who owns a table, then check whether any views or procedures depend on that particular table.

The following statement drops the user, jones and all associated objects and foreign keys that depend on the tables owned by jones.

```
DROP USER jones CASCADE;
```

> **See Also:** *Oracle Database Administrator's Guide* for more information about terminating sessions

Viewing Information About Database Users and Profiles

The wide variety of options that are available for viewing such information are discussed in the following subsections:

- User and Profile Information in Data Dictionary Views
- Listing All Users and Associated Information
- Listing All Tablespace Quotas
- Listing All Profiles and Assigned Limits
- Viewing Memory Use for Each User Session

User and Profile Information in Data Dictionary Views

The following data dictionary views contain information about database users and profiles:

View	Description
DBA_USERS	Describes all users of the database
ALL_USERS	Lists users visible to the current user, but does not describe them
USER_USERS	Describes only the current user
DBA_TS_QUOTAS USER_TS_QUOTAS	Describes tablespace quotas for users
USER_PASSWORD_LIMITS	Describes the password profile parameters that are assigned to the user
USER_RESOURCE_LIMITS	Displays the resource limits for the current user
DBA_PROFILES	Displays all profiles and their limits
RESOURCE_COST	Lists the cost for each resource

View	Description
V$SESSION	Lists session information for each current session, includes user name
V$SESSTAT	Lists user session statistics
V$STATNAME	Displays decoded statistic names for the statistics shown in the V$SESSTAT view
PROXY_USERS	Describes users who can assume the identity of other users

The following sections present some examples of using these views, and assume a database in which the following statements have been executed:

```
CREATE PROFILE clerk LIMIT
    SESSIONS_PER_USER 1
    IDLE_TIME 30
    CONNECT_TIME 600;

CREATE USER jfee
    IDENTIFIED BY wildcat
    DEFAULT TABLESPACE users
    TEMPORARY TABLESPACE temp_ts
    QUOTA 500K ON users
    PROFILE clerk;

CREATE USER dcranney
    IDENTIFIED BY bedrock
    DEFAULT TABLESPACE users
    TEMPORARY TABLESPACE temp_ts
    QUOTA unlimited ON users;

CREATE USER userscott
     IDENTIFIED BY scott1;
```

See Also: *Oracle Database SQL Reference* for complete descriptions of the preceding data dictionary and dynamic performance views

Listing All Users and Associated Information

The following query lists all users and their associated information as defined in the database:

```
SELECT USERNAME, PROFILE, ACCOUNT_STATUS FROM DBA_USERS;

USERNAME        PROFILE         ACCOUNT_STATUS
--------------- --------------- ---------------
SYS             DEFAULT         OPEN
SYSTEM          DEFAULT         OPEN
USERSCOTT       DEFAULT         OPEN
JFEE            CLERK           OPEN
DCRANNEY        DEFAULT         OPEN
```

All passwords are encrypted to preserve security. If a user queries the PASSWORD column, then that user is not able to determine the password of another user.

Listing All Tablespace Quotas

The following query lists all tablespace quotas specifically assigned to each user:

```
SELECT * FROM DBA_TS_QUOTAS;

TABLESPACE    USERNAME    BYTES     MAX_BYTES    BLOCKS    MAX_BLOCKS
----------    ---------  --------   ----------   -------   ----------
USERS         JFEE              0       512000         0          250
USERS         DCRANNEY          0           -1         0           -1
```

When specific quotas are assigned, the exact number is indicated in the `MAX_BYTES` column. Note that this number is always a multiple of the database block size, so if you specify a tablespace quota that is not a multiple of the database block size, then it is rounded up accordingly. Unlimited quotas are indicated by `-1`.

Listing All Profiles and Assigned Limits

The following query lists all profiles in the database and associated settings for each limit in each profile:

```
SELECT * FROM DBA_PROFILES
   ORDER BY PROFILE;

PROFILE             RESOURCE_NAME              RESOURCE   LIMIT
-----------------   ---------------            ---------- --------------
CLERK               COMPOSITE_LIMIT            KERNEL     DEFAULT
CLERK               FAILED_LOGIN_ATTEMPTS      PASSWORD   DEFAULT
CLERK               PASSWORD_LIFE_TIME         PASSWORD   DEFAULT
CLERK               PASSWORD_REUSE_TIME        PASSWORD   DEFAULT
CLERK               PASSWORD_REUSE_MAX         PASSWORD   DEFAULT
CLERK               PASSWORD_VERIFY_FUNCTION   PASSWORD   DEFAULT
CLERK               PASSWORD_LOCK_TIME         PASSWORD   DEFAULT
CLERK               PASSWORD_GRACE_TIME        PASSWORD   DEFAULT
CLERK               PRIVATE_SGA                KERNEL     DEFAULT
CLERK               CONNECT_TIME               KERNEL     600
CLERK               IDLE_TIME                  KERNEL     30
CLERK               LOGICAL_READS_PER_CALL     KERNEL     DEFAULT
CLERK               LOGICAL_READS_PER_SESSION  KERNEL     DEFAULT
CLERK               CPU_PER_CALL               KERNEL     DEFAULT
CLERK               CPU_PER_SESSION            KERNEL     DEFAULT
CLERK               SESSIONS_PER_USER          KERNEL     1
DEFAULT             COMPOSITE_LIMIT            KERNEL     UNLIMITED
DEFAULT             PRIVATE_SGA                KERNEL     UNLIMITED
DEFAULT             SESSIONS_PER_USER          KERNEL     UNLIMITED
DEFAULT             CPU_PER_CALL               KERNEL     UNLIMITED
DEFAULT             LOGICAL_READS_PER_CALL     KERNEL     UNLIMITED
DEFAULT             CONNECT_TIME               KERNEL     UNLIMITED
DEFAULT             IDLE_TIME                  KERNEL     UNLIMITED
DEFAULT             LOGICAL_READS_PER_SESSION  KERNEL     UNLIMITED
DEFAULT             CPU_PER_SESSION            KERNEL     UNLIMITED
DEFAULT             FAILED_LOGIN_ATTEMPTS      PASSWORD   UNLIMITED
DEFAULT             PASSWORD_LIFE_TIME         PASSWORD   UNLIMITED
DEFAULT             PASSWORD_REUSE_MAX         PASSWORD   UNLIMITED
DEFAULT             PASSWORD_LOCK_TIME         PASSWORD   UNLIMITED
DEFAULT             PASSWORD_GRACE_TIME        PASSWORD   UNLIMITED
DEFAULT             PASSWORD_VERIFY_FUNCTION   PASSWORD   UNLIMITED
DEFAULT             PASSWORD_REUSE_TIME        PASSWORD   UNLIMITED
32 rows selected.
```

Viewing Memory Use for Each User Session

The following query lists all current sessions, showing the Oracle user and current User Global Area (UGA) memory use for each session:

```
SELECT USERNAME, VALUE || 'bytes' "Current UGA memory"
   FROM V$SESSION sess, V$SESSTAT stat, V$STATNAME name
WHERE sess.SID = stat.SID
   AND stat.STATISTIC# = name.STATISTIC#
   AND name.NAME = 'session uga memory';

USERNAME                       Current UGA memory
------------------------------ ----------------------------------------------
                               18636bytes
                               17464bytes
                               19180bytes
                               18364bytes
                               39384bytes
                               35292bytes
                               17696bytes
                               15868bytes
USERSCOTT                      42244bytes
SYS                            98196bytes
SYSTEM                         30648bytes

11 rows selected.
```

To see the maximum UGA memory ever allocated to each session since the instance started, replace `'session uga memory'` in the preceding query with `'session uga memory max'`.

Managing Resources with Profiles

A profile is a named set of resource limits and password parameters that restrict database usage and instance resources for a user. You can assign a profile to each user, and a default profile to all others. Each user can have only one profile, and creating a new one supersedes any earlier one.

Profile resource limits are enforced only when you enable resource limitation for the associated database. Enabling such limitation can occur either before starting up the database (the `RESOURCE_LIMIT` initialization parameter) or while it is open (using an `ALTER SYSTEM` statement).

While password parameters reside in profiles, they are unaffected by `RESOURCE_LIMIT` or `ALTER SYSTEM` and password management is always enabled. In Oracle Database 10*g* Release 2 (10.2), resource allocations and restrictions are primarily handled through the Database Resource Manager.

See Also:

- For resource allocation, see the *Database Resource Manager*, as described in the *Oracle Database Administrator's Guide*
- For password policies, see Password Management Policy on page 7-9
- For `ALTER SYSTEM` or `RESOURCE_LIMIT`, see *Oracle Database SQL Reference*

A profile can be created, assigned to users, altered, and dropped at any time (using CREATE USER or ALTER USER) by any authorized database user. Profiles can be assigned only to users and not to roles or other profiles. Profile assignments do not affect current sessions, instead, they take effect only in subsequent sessions.

See Also:

- *Oracle Database SQL Reference* for more information about the SQL statements used for managing profiles, such as `CREATE PROFILE`, and for information on how to calculate composite limits.
- *Database Resource Manager* in the Oracle Database Administrator's Guide
- "Creating Users" on page 11-1
- "Altering Users" on page 11-5

Dropping Profiles

To drop a profile, you must have the `DROP PROFILE` system privilege. You can drop a profile (other than the default profile) using the SQL statement `DROP PROFILE`. To successfully drop a profile currently assigned to a user, use the `CASCADE` option.

The following statement drops the profile `clerk`, even though it is assigned to a user:

```
DROP PROFILE clerk CASCADE;
```

Any user currently assigned to a profile that is dropped is automatically assigned to the `DEFAULT` profile. The `DEFAULT` profile cannot be dropped. When a profile is dropped, the drop does not affect currently active sessions. Only sessions created after a profile is dropped abide by any modified profile assignments.

Understanding User Privileges and Roles

A user **privilege** is the right to run a particular type of SQL statement, or the right to access an object belonging to another user, run a PL/SQL package, and so on. The types of privileges are defined by Oracle Database.

Roles are created by users (usually administrators) to group together privileges or other roles. They are a means of facilitating the granting of multiple privileges or roles to users.

This section describes Oracle user privileges, and contains the following topics:

- System Privileges
- Object Privileges
- User Roles

See Also: *Oracle Database Concepts* for additional information about privileges and roles

System Privileges

There are over 100 distinct system privileges. Each system privilege allows a user to perform a particular database operation or class of database operations.

Caution: System privileges can be very powerful, and should be granted only when necessary to roles and trusted users of the database.

See Also: *Oracle Database SQL Reference.* for the complete list of system privileges and their descriptions

Restricting System Privileges

Because system privileges are so powerful, Oracle recommends that you configure your database to prevent regular (non-DBA) users exercising ANY system privileges (such as UPDATE ANY TABLE) on the data dictionary. In order to secure the data dictionary, ensure that the O7_DICTIONARY_ACCESSIBILITY initialization parameter is set to FALSE, the default value. This feature is called the dictionary protection mechanism.

Note: The O7_DICTIONARY_ACCESSIBILITY initialization parameter controls restrictions on system privileges when you upgrade from Oracle Database version 7 to Oracle8*i* and higher releases. If the parameter is set to TRUE, then access to objects in the SYS schema is allowed (Oracle database version 7 behavior). If this parameter is set to FALSE, then system privileges that allow access to objects in any schema do not allow access to objects in the SYS schema. The default for O7_DICTIONARY_ACCESSIBILITY is FALSE.

When this parameter is not set to FALSE, the ANY privilege applies to the data dictionary, and a malicious user with ANY privilege could access or alter data dictionary tables.

See the *Oracle Database Reference* for more information on the O7_DICTIONARY_ACCESSIBILITY initialization parameter to understand its usage.

If you enable dictionary protection (O7_DICTIONARY_ACCESSIBILITY is set to FALSE), then access to objects in the SYS schema (dictionary objects) is restricted to users with the SYS schema. These users are SYS and those who connect as SYSDBA. System privileges providing access to objects in other schemas do *not* give other users access to objects in the SYS schema. For example, the SELECT ANY TABLE privilege allows users to access views and tables in other schemas, but does not enable them to select dictionary objects (base tables of dynamic performance views, views, packages, and synonyms). These users can, however, be granted explicit object privileges to access objects in the SYS schema.

Accessing Objects in the SYS Schema

Users with explicit object privileges or those who connect with administrative privileges (SYSDBA) can access objects in the SYS schema. Another means of allowing access to objects in the SYS schema is by granting users any of the following roles:

- SELECT_CATALOG_ROLE

  This role can be granted to users to allow SELECT privileges on data dictionary views.

- EXECUTE_CATALOG_ROLE

This role can be granted to users to allow EXECUTE privileges for packages and procedures in the data dictionary.

- DELETE_CATALOG_ROLE

  This role can be granted to users to allow them to delete records from the system audit table (AUD$).

Additionally, the following system privilege can be granted to users who require access to tables created in the SYS schema:

- SELECT ANY DICTIONARY

  This system privilege allows query access to any object in the SYS schema, including tables created in that schema. It must be granted individually to each user requiring the privilege. It is not included in GRANT ALL PRIVILEGES, but it can be granted through a role.

> **Caution:** You should grant these roles and the SELECT ANY DICTIONARY system privilege with extreme care, because the integrity of your system can be compromised by their misuse.

Object Privileges

Each type of object has different privileges associated with it.

You can specify ALL [PRIVILEGES] to grant or revoke all available object privileges for an object. ALL is not a privilege, rather, it is a shortcut, or a way of granting or revoking all object privileges with one GRANT and REVOKE statement. Note that if all object privileges are granted using the ALL shortcut, then individual privileges can still be revoked.

Likewise, all individually granted privileges can be revoked by specifying ALL. However, if you REVOKE ALL, and revoking causes integrity constraints to be deleted (because they depend on a REFERENCES privilege that you are revoking), then you must include the CASCADE CONSTRAINTS option in the REVOKE statement.

> **See Also:** *Oracle Database SQL Reference.* for the complete list of object privileges

User Roles

A **role** groups several privileges and roles, so that they can be granted to and revoked from users simultaneously. A role must be enabled for a user before it can be used by the user.

Oracle Database provides some predefined roles to help in database administration. These roles, listed in Table 11–1, are automatically defined for Oracle databases when you run the standard scripts that are part of database creation. You can grant privileges and roles to, and revoke privileges and roles from, these predefined roles in the same way as you do with any role you define.

Note: Each installation should create its own roles and assign only those privileges that are needed. This principle enables the organization to retain detailed control of its roles and privileges.

It also avoids the necessity to adjust if Oracle changes or removes Oracle-defined roles, as it has with CONNECT, which now has only the CREATE SESSION privilege. Formerly, it also had eight other privileges. Both CONNECT and RESOURCE roles will be deprecated in future Oracle versions.

Table 11–1 Predefined Roles

Role Name	Created By (Script)	Description
CONNECT	SQL.BSQ	Includes only the following system privilege: CREATE SESSION
RESOURCE	SQL.BSQ	Includes the following system privileges: CREATE CLUSTER, CREATE INDEXTYPE, CREATE OPERATOR, CREATE PROCEDURE, CREATE SEQUENCE, CREATE TABLE, CREATE TRIGGER, CREATE TYPE
DBA	SQL.BSQ	All system privileges WITH ADMIN OPTION
EXP_FULL_DATABASE	CATEXP.SQL	Provides the privileges required to perform full and incremental database exports and includes: SELECT ANY TABLE, BACKUP ANY TABLE, EXECUTE ANY PROCEDURE, EXECUTE ANY TYPE, ADMINISTER RESOURCE MANAGER, and INSERT, DELETE, and UPDATE on the tables SYS.INCVID, SYS.INCFIL, and SYS.INCEXP. Also the following roles: EXECUTE_CATALOG_ROLE and SELECT_CATALOG_ROLE.
IMP_FULL_DATABASE	CATEXP.SQL	Provides the privileges required to perform full database imports. Includes an extensive list of system privileges (use view DBA_SYS_PRIVS to view privileges) and the following roles: EXECUTE_CATALOG_ROLE and SELECT_CATALOG_ROLE.
DELETE_CATALOG_ROLE	SQL.BSQ	Provides DELETE privilege on the system audit table (AUD$)
EXECUTE_CATALOG_ROLE	SQL.BSQ	Provides EXECUTE privilege on objects in the data dictionary. Also, HS_ADMIN_ROLE.
SELECT_CATALOG_ROLE	SQL.BSQ	Provides SELECT privilege on objects in the data dictionary. Also, HS_ADMIN_ROLE.
RECOVERY_CATALOG_OWNER	CATALOG.SQL	Provides privileges for owner of the recovery catalog. Includes: CREATE SESSION, ALTER SESSION, CREATE SYNONYM, CREATE VIEW, CREATE DATABASE LINK, CREATE TABLE, CREATE CLUSTER, CREATE SEQUENCE, CREATE TRIGGER, and CREATE PROCEDURE
HS_ADMIN_ROLE	CATHS.SQL	Used to protect access to the Heterogeneous Services (HS) data dictionary tables (grants SELECT) and packages (grants EXECUTE). It is granted to SELECT_CATALOG_ROLE and EXECUTE_CATALOG_ROLE such that users with generic data dictionary access also can access the HS data dictionary.

Table 11–1 (Cont.) Predefined Roles

Role Name	Created By (Script)	Description
AQ_USER_ROLE	CATQUEUE.SQL	Obsolete, but kept mainly for release 8.0 compatibility. Provides execute privilege on DBMS_AQ and DBMS_AQIN.
AQ_ADMINISTRATOR_ROLE	CATQUEUE.SQL	Provides privileges to administer Advance Queuing. Includes ENQUEUE ANY QUEUE, DEQUEUE ANY QUEUE, and MANAGE ANY QUEUE, SELECT privileges on AQ tables and EXECUTE privileges on AQ packages.

Note: The first three roles in Table 11–1 namely, CONNECT, RESOURCE, and DBA, are provided to maintain compatibility with previous versions of Oracle and may not be created automatically in future versions of Oracle. Oracle recommends that you design your own roles for database security, rather than relying on these roles.

If you install other options or products, then other predefined roles may be created.

Managing User Roles

This section describes aspects of managing roles, and contains the following topics:

- Creating a Role
- Specifying the Type of Role Authorization
- Dropping Roles

Creating a Role

You can create a role using the CREATE ROLE statement, but you must have the CREATE ROLE system privilege to do so. Typically, only security administrators have this system privilege.

Note: Immediately after creation, a role has no privileges associated with it. To associate privileges with a new role, you must grant privileges or other roles to the new role.

You must give each role you create a unique name among existing user names and role names of the database. Roles are not contained in the schema of any user. In a database that uses a multibyte character set, Oracle recommends that each role name contain at least one single-byte character. If a role name contains only multibyte characters, then the encrypted role name and password combination is considerably less secure.

The following statement creates the clerk role, which is authorized by the database using the password bicentennial:

```
CREATE ROLE clerk IDENTIFIED BY bicentennial;
```

The IDENTIFIED BY clause specifies how the user must be authorized before the role can be enabled for use by a specific user to which it has been granted. If this clause is not specified, or NOT IDENTIFIED is specified, then no authorization is required when the role is enabled. Roles can be specified to be authorized by:

- The database using a password
- An application using a specified package
- Externally by the operating system, network, or other external source
- Globally by an enterprise directory service

These authorizations are discussed in following sections.

Later, you can set or change the authorization method for a role using the `ALTER ROLE` statement. The following statement alters the `clerk` role to specify that the user must have been authorized by an external source before enabling the role:

```
ALTER ROLE clerk IDENTIFIED EXTERNALLY;
```

To alter the authorization method for a role, you must have the `ALTER ANY ROLE` system privilege or have been granted the role with the `ADMIN OPTION`.

See Also:

- *Oracle Database SQL Reference* for syntax, restrictions, and authorization information about the SQL statements used to manage roles and privileges
- *Oracle Database Advanced Security Administrator's Guide*

Specifying the Type of Role Authorization

The methods of authorizing roles are presented in this section. A role must be enabled for you to use it.

See Also: "When Do Grants and Revokes Take Effect?" on page 11-26 for a discussion about enabling roles

Role Authorization by the Database

The usage of a role authorized by the database can be protected by an associated password. If you are granted a role protected by a password, then you can enable or disable the role by supplying the proper password for the role in a `SET ROLE` statement. However, if the role is made a default role and enabled at connect time, then the user is not required to enter a password.

The following statement creates a role called `manager`. When it is enabled, the password `morework` must be supplied.

```
CREATE ROLE manager IDENTIFIED BY morework;
```

Note: In a database that uses a multibyte character set, passwords for roles must include only singlebyte characters. Multibyte characters are not accepted in passwords. See the *Oracle Database SQL Reference* for information about specifying valid passwords.

Role Authorization by an Application

The `INDENTIFIED USING` *`package_name`* clause lets you create an application role, which is a role that can be enabled only by applications using an authorized package. Application developers do not need to secure a role by embedding passwords inside applications. Instead, they can create an application role and specify which PL/SQL package is authorized to enable the role.

The following example indicates that the role admin_role is an application role and the role can only be enabled by any module defined inside the PL/SQL package hr.admin.

```
CREATE ROLE admin_role IDENTIFIED USING hr.admin;
```

When enabling the default roles of the user at login as specified in the user profile, no checking is performed for application roles.

Role Authorization by an External Source

The following statement creates a role named accts_rec and requires that the user be authorized by an external source before it can be enabled:

```
CREATE ROLE accts_rec IDENTIFIED EXTERNALLY;
```

Role Authorization by the Operating System Role authentication through the operating system is useful only when the operating system is able to dynamically link operating system privileges with applications. When a user starts an application, the operating system grants an operating system privilege to the user. The granted operating system privilege corresponds to the role associated with the application. At this point, the application can enable the application role. When the application is terminated, the previously granted operating system privilege is revoked from the operating system account of the user.

If a role is authorized by the operating system, then you must configure information for each user at the operating system level. This operation is operating system dependent.

If roles are granted by the operating system, then you do not need to have the operating system authorize them also. This is redundant.

> **See Also:** "Granting Roles Using the Operating System or Network" on page 11-27 for more information about roles granted by the operating system

Role Authorization and Network Clients If users connect to the database over Oracle Net, then by default, their roles cannot be authenticated by the operating system. This includes connections through a shared server configuration, as this connection requires Oracle Net. This restriction is the default because a remote user could impersonate another operating system user over a network connection.

If you are not concerned with this security risk and want to use operating system role authentication for network clients, then set the initialization parameter REMOTE_OS_ROLES in the database initialization parameter file to TRUE. The change will take effect the next time you start the instance and mount the database. The parameter is FALSE by default.

Role Authorization by an Enterprise Directory Service

A role can be defined as a global role, where a (global) user can only be authorized to use the role by an enterprise directory service. You define the global role locally in the database by granting privileges and roles to it, but you cannot grant the global role itself to any user or other role in the database. When a global user attempts to connect to the database, the enterprise directory is queried to obtain any global roles associated with the user.

The following statement creates a global role:

```
CREATE ROLE supervisor IDENTIFIED GLOBALLY;
```

Global roles are one component of enterprise user security. A global role only applies to one database, but it can be granted to an enterprise role defined in the enterprise directory. An enterprise role is a directory structure which contains global roles on multiple databases, and which can be granted to enterprise users.

A general discussion of global authentication and authorization of users, and its role in enterprise user management, was presented earlier in "Global Authentication and Authorization" on page 10-4.

> **See Also:** *Oracle Database Advanced Security Administrator's Guide* and *Oracle Internet Directory Administrator's Guide* for information about enterprise user management and how to implement it

Dropping Roles

In some cases, it may be appropriate to drop a role from the database. The security domains of all users and roles granted a dropped role are immediately changed to reflect the absence of the dropped role privileges. All indirectly granted roles of the dropped role are also removed from affected security domains. Dropping a role automatically removes the role from all user default role lists.

Because the creation of objects is not dependent on the privileges received through a role, tables and other objects are not dropped when a role is dropped.

You can drop a role using the SQL statement `DROP ROLE`. To drop a role, you must have the `DROP ANY ROLE` system privilege or have been granted the role with the `ADMIN OPTION`.

The following statement drops the role `CLERK`:

```
DROP ROLE clerk;
```

Granting User Privileges and Roles

This section describes the granting of privileges and roles, and contains the following topics:

- Granting System Privileges and Roles
- Granting Object Privileges
- Granting Privileges on Columns

It is also possible to grant roles to a user connected through a middle tier or proxy. This is discussed in "Proxy Authentication and Authorization" on page 10-6.

Granting System Privileges and Roles

You can grant system privileges and roles to other users and roles using the `GRANT` statement. The following privileges are required:

- To grant a system privilege, you must have been granted the system privilege with the `ADMIN OPTION` or have been granted the `GRANT ANY PRIVILEGE` system privilege.
- To grant a role, you must have been granted the role with the `ADMIN OPTION` or have been granted the `GRANT ANY ROLE` system privilege.

The following statement grants the system privilege `CREATE SESSION` and the `accts_pay` role to the user `jward`:

```
GRANT CREATE SESSION, accts_pay TO jward;
```

> **Note:** Object privileges *cannot* be granted along with system privileges and roles in the same GRANT statement.

Granting the ADMIN OPTION

A user or role that is granted a privilege or role, which specifies the WITH ADMIN OPTION clause, has several expanded capabilities:

- The grantee can grant or revoke the system privilege or role to or from any user or other role in the database. Users cannot revoke a role from themselves.
- The grantee can further grant the system privilege or role with the ADMIN OPTION.
- The grantee of a role can alter or drop the role.

In the following statement, the security administrator grants the new_dba role to michael:

```
GRANT new_dba TO michael WITH ADMIN OPTION;
```

The user michael cannot only use all of the privileges implicit in the new_dba role, but can also grant, revoke, and drop the new_dba role as deemed necessary. Because of these powerful capabilities, exercise caution when granting system privileges or roles with the ADMIN OPTION. Such privileges are usually reserved for a security administrator and rarely granted to other administrators or users of the system.

> **Note:** When a user creates a role, the role is automatically granted to the creator with the ADMIN OPTION

Creating a New User with the GRANT Statement

Oracle enables you to create a new user with the GRANT statement. If you specify a password using the IDENTIFIED BY clause, and the user name/password does not exist in the database, then a new user with that user name and password is created. The following example creates ssmith as a new user while granting ssmith the CONNECT system privilege:

```
GRANT CONNECT TO ssmith IDENTIFIED BY p1q2r3;
```

See Also: "Creating Users" on page 11-1

Granting Object Privileges

You also use the GRANT statement to grant object privileges to roles and users. To grant an object privilege, you must fulfill one of the following conditions:

- You own the object specified.
- You possess the GRANT ANY OBJECT PRIVILEGE system privilege that enables you to grant and revoke privileges on behalf of the object owner.
- The WITH GRANT OPTION clause was specified when you were granted the object privilege by its owner.

> **Note:** System privileges and roles cannot be granted along with object privileges in the same GRANT statement.

The following statement grants the SELECT, INSERT, and DELETE object privileges for all columns of the emp table to the users, jfee and tsmith:

```
GRANT SELECT, INSERT, DELETE ON emp TO jfee, tsmith;
```

To grant all object privileges on the salary view to the user jfee, use the ALL keyword as shown in the following example:

```
GRANT ALL ON salary TO jfee;
```

> **Note:** A grantee cannot regrant access to objects unless the original grant included the GRANT OPTION. Thus in the example just given, jfee cannot use the GRANT statement to grant object privileges to anyone else.

Specifying the GRANT OPTION

Specify WITH GRANT OPTION to enable the grantee to grant the object privileges to other users and roles. The user whose schema contains an object is automatically granted all associated object privileges with the GRANT OPTION. This special privilege allows the grantee several expanded privileges:

- The grantee can grant the object privilege to any users in the database, with or without the GRANT OPTION, and to any role in the database.
- If both of the following conditions are true, then the grantee can create views on the table and grant the corresponding privileges on the views to any user or role in the database.
  - The grantee receives object privileges for the table with the GRANT OPTION.
  - The grantee has the CREATE VIEW or CREATE ANY VIEW system privilege.

> **Note:** The GRANT OPTION is not valid when granting an object privilege to a role. Oracle Database prevents the propagation of object privileges through roles so that grantees of a role cannot propagate object privileges received by means of roles.

Granting Object Privileges on Behalf of the Object Owner

The GRANT ANY OBJECT PRIVILEGE system privilege allows users to grant and revoke any object privilege on behalf of the object owner. This privilege provides a convenient means for database and application administrators to grant access to objects in any schema without requiring that they connect to the schema. Login credentials need not be maintained for schema owners who have this privilege, which reduces the number of connections required during configuration.

This system privilege is part of the Oracle supplied DBA role and is thus granted (with the ADMIN OPTION) to any user connecting AS SYSDBA (user SYS). As with other system privileges, the GRANT ANY OBJECT PRIVILEGE system privilege can only be granted by a user who possesses the ADMIN OPTION.

The *recorded* grantor of access rights to an object is either the object owner or the person exercising the GRANT ANY OBJECT PRIVILEGE system privilege. If the

grantor with GRANT ANY OBJECT PRIVILEGE does *not* have the object privilege with the GRANT OPTION, then the object owner is shown as the grantor. Otherwise, when that grantor has the object privilege with the GRANT OPTION, then that grantor is recorded as the grantor of the grant.

> **Note:** The audit record generated by the GRANT statement always shows the real user who performed the grant.

For example, consider the following. User adams possesses the GRANT ANY OBJECT PRIVILEGE system privilege. He does not possess any other grant privileges. He issues the following statement:

```
GRANT SELECT ON hr.employees TO blake WITH GRANT OPTION;
```

If you examine the DBA_TAB_PRIVS view, then you will see that hr is shown as the grantor of the privilege:

```
SQL> SELECT GRANTEE, OWNER, GRANTOR, PRIVILEGE, GRANTABLE
  2>     FROM DBA_TAB_PRIVS
  3>     WHERE TABLE_NAME = 'EMPLOYEES' and OWNER = 'HR';

GRANTEE  OWNER GRANTOR PRIVILEGE    GRANTABLE
-------- ----- ------- -----------  ----------
BLAKE    HR    HR       SELECT      YES
```

Now assume that the user blake also has the GRANT ANY OBJECT PRIVILEGE system. He, issues the following statement:

```
GRANT SELECT ON hr.employees TO clark;
```

In this case, when you query the DBA_TAB_PRIVS view again, you see that blake is shown as being the grantor of the privilege:

```
GRANTEE  OWNER GRANTOR  PRIVILEGE    GRANTABLE
-------- ----- -------- ---------    ----------
BLAKE    HR    HR       SELECT       YES
CLARK    HR    BLAKE    SELECT       NO
```

This occurs because blake already possesses the SELECT privilege on hr.employees with the GRANT OPTION.

> **See Also:** "Revoking Object Privileges on Behalf of the Object Owner" on page 11-23

Granting Privileges on Columns

You can grant INSERT, UPDATE, or REFERENCES privileges on individual columns in a table.

> **Caution:** Before granting a column-specific INSERT privilege, determine if the table contains any columns on which NOT NULL constraints are defined. Granting selective insert capability without including the NOT NULL columns prevents the user from inserting any rows into the table. To avoid this situation, make sure that each NOT NULL column can either be inserted into or has a non-NULL default value. Otherwise, the grantee will not be able to insert rows into the table and will receive an error.

The following statement grants INSERT privilege on the acct_no column of the accounts table to scott:

```
GRANT INSERT (acct_no) ON accounts TO scott;
```

In another example, object privilege for the ename and job columns of the emp table are granted to the users jfee and tsmith:

```
GRANT INSERT(ename, job) ON emp TO jfee, tsmith;
```

Row-Level Access Control

You can also provide access control at the row level, that is, within objects, using Virtual Private Database (VPD) or Oracle Label Security.

See Also:

- Chapter 14, "Using Virtual Private Database to Implement Application Security Policies"
- Adding Policies for Column-Level VPD, in Chapter 15, "Implementing Application Context and Fine-Grained Access Control"

Revoking User Privileges and Roles

This section describes aspects of revoking user privileges and roles, and contains the following topics:

- Revoking System Privileges and Roles
- Revoking Object Privileges
- Cascading Effects of Revoking Privileges

Revoking System Privileges and Roles

You can revoke system privileges and roles using the SQL statement REVOKE. Any user with the ADMIN OPTION for a system privilege or role can revoke the privilege or role from any other database user or role. The revoker does not have to be the user that originally granted the privilege or role. Users with GRANT ANY ROLE can revoke *any* role.

The following statement revokes the CREATE TABLE system privilege and the accts_rec role from tsmith:

```
REVOKE CREATE TABLE, accts_rec FROM tsmith;
```

Note: The ADMIN OPTION for a system privilege or role cannot be selectively revoked. The privilege or role must be revoked and then the privilege or role regranted without the ADMIN OPTION.

Revoking Object Privileges

To revoke an object privilege, you must fulfill one of the following conditions:

- You previously granted the object privilege to the user or role.
- You possess the GRANT ANY OBJECT PRIVILEGE system privilege that enables you to grant and revoke privileges on behalf of the object owner.

You can only revoke the privileges that you, the grantor, directly authorized, not the grants made by other users to whom you granted the GRANT OPTION. However, there is a cascading effect. The object privilege grants propagated using the GRANT OPTION are revoked if a grantor object privilege is revoked.

Assuming you are the original grantor, the following statement revokes the SELECT and INSERT privileges on the emp table from the users jfee and tsmith:

```
REVOKE SELECT, insert ON emp FROM jfee, tsmith;
```

The following statement revokes all object privileges for the dept table that you originally granted to the human_resource role

```
REVOKE ALL ON dept FROM human_resources;
```

> **Note:** The GRANT OPTION for an object privilege cannot be selectively revoked. The object privilege must be revoked and then regranted without the GRANT OPTION. Users cannot revoke object privileges from themselves.

Revoking Object Privileges on Behalf of the Object Owner

The GRANT ANY OBJECT PRIVILEGE system privilege enables you to revoke any specified object privilege where the object owner is the grantor. This occurs when the object privilege is granted by the object owner, or on behalf of the owner by any user holding the GRANT ANY OBJECT PRIVILEGE system privilege.

In a situation where the object privilege has been granted by both the owner of the object and the user executing the REVOKE statement (who has both the specific object privilege and the GRANT ANY OBJECT PRIVILEGE system privilege), Oracle only revokes the object privilege granted by the user issuing the REVOKE. This can be illustrated by continuing the example started in "Granting Object Privileges on Behalf of the Object Owner" on page 11-20.

At this point, blake has granted the SELECT privilege on hr.employees to clark. Even though blake possesses the GRANT ANY OBJECT PRIVILEGE system privilege, he also holds the specific object privilege, thus this grant is attributed to him. Assume that hr also grants the SELECT privilege on hr.employees to clark. A query of the DBA_TAB_PRIVS view shows that the following grants are in effect for the hr.employees table:

```
GRANTEE  OWNER GRANTOR PRIVILEGE    GRANTABLE
-------- ----- ------- -----------  ----------
BLAKE    HR    HR       SELECT       YES
CLARK    HR    BLAKE    SELECT       NO
CLARK    HR    HR       SELECT       NO
```

User blake now issues the following REVOKE statement:

```
REVOKE  SELECT ON hr.employees FROM clark;
```

Only the object privilege for clark granted by blake is removed. The grant by the object owner, hr, remains.

```
GRANTEE  OWNER GRANTOR PRIVILEGE    GRANTABLE
-------- ----- ------- -----------  ----------
BLAKE    HR    HR       SELECT      YES
CLARK    HR    HR       SELECT      NO
```

If blake issues the REVOKE statement again, then this time the effect will be to remove the object privilege granted by hr.

> **See Also:** "Granting Object Privileges on Behalf of the Object Owner" on page 11-20

Revoking Column-Selective Object Privileges

Although users can grant column-selective INSERT, UPDATE, and REFERENCES privileges for tables and views, they cannot selectively revoke column-specific privileges with a similar REVOKE statement. Instead, the grantor must first revoke the object privilege for all columns of a table or view, and then selectively regrant the column-specific privileges that should remain.

For example, assume that role human_resources has been granted the UPDATE privilege on the deptno and dname columns of the table dept. To revoke the UPDATE privilege on just the deptno column, issue the following two statements:

```
REVOKE UPDATE ON dept FROM human_resources;
GRANT UPDATE (dname) ON dept TO human_resources;
```

The REVOKE statement revokes UPDATE privilege on all columns of the dept table from the role human_resources. The GRANT statement then regrants UPDATE privilege on the dname column to the role human_resources.

Revoking the REFERENCES Object Privilege

If the grantee of the REFERENCES object privilege has used the privilege to create a foreign key constraint (that currently exists), then the grantor can revoke the privilege only by specifying the CASCADE CONSTRAINTS option in the REVOKE statement:

```
REVOKE REFERENCES ON dept FROM jward CASCADE CONSTRAINTS;
```

Any foreign key constraints currently defined that use the revoked REFERENCES privilege are dropped when the CASCADE CONSTRAINTS clause is specified.

Cascading Effects of Revoking Privileges

Depending on the type of privilege, there may be cascading effects when a privilege is revoked. This is discussed in the following sections:

- System Privileges
- Object Privileges

System Privileges

There are no cascading effects when revoking a system privilege related to DDL operations, regardless of whether the privilege was granted with or without the ADMIN OPTION. For example, assume the following:

1. The security administrator grants the CREATE TABLE system privilege to jfee with the ADMIN OPTION.
2. User jfee creates a table.
3. User jfee grants the CREATE TABLE system privilege to tsmith.
4. User tsmith creates a table.
5. The security administrator revokes the CREATE TABLE system privilege from jfee.

6. The table created by user jfee continues to exist. tsmith still has the table and the CREATE TABLE system privilege.

Cascading effects can be observed when revoking a system privilege related to a DML operation. If SELECT ANY TABLE is revoked from a user, then all procedures contained in the users schema relying on this privilege will fail until the privilege is reauthorized.

Object Privileges

Revoking an object privilege can have cascading effects that should be investigated before issuing a REVOKE statement.

- Object definitions that depend on a DML object privilege can be affected if the DML object privilege is revoked. For example, assume that the procedure body of the test procedure includes a SQL statement that queries data from the emp table. If the SELECT privilege on the emp table is revoked from the owner of the test procedure, then the procedure can no longer be executed successfully.
- When a REFERENCES privilege for a table is revoked from a user, any foreign key integrity constraints that are defined by the user and require the dropped REFERENCES privilege are automatically dropped. For example, assume that the user jward is granted the REFERENCES privilege for the deptno column of the dept table. This user now creates a foreign key on the deptno column in the emp table that references the deptno column of the dept table. If the REFERENCES privilege on the deptno column of the dept table is revoked, then the foreign key constraint on the deptno column of the emp table is dropped in the same operation.
- The object privilege grants propagated using the GRANT OPTION are revoked if the object privilege of a grantor is revoked. For example, assume that user1 is granted the SELECT object privilege with the GRANT OPTION, and grants the SELECT privilege on emp to user2. Subsequently, the SELECT privilege is revoked from user1. This REVOKE is cascaded to user2 as well. Any objects that depend on the revoked SELECT privilege of user1 and user2 can also be affected, as described earlier.

Object definitions that require the ALTER and INDEX DDL object privileges are not affected if the ALTER or INDEX object privilege is revoked. For example, if the INDEX privilege is revoked from a user that created an index on a table that belongs to another user, then the index continues to exist after the privilege is revoked.

Granting to and Revoking from the PUBLIC User Group

Privileges and roles can also be granted to and revoked from the user group PUBLIC. Because PUBLIC is accessible to every database user, all privileges and roles granted to PUBLIC are accessible to every database user.

Security administrators and database users should grant a privilege or role to PUBLIC only if every database user requires the privilege or role. This recommendation reinforces the general rule that at any given time, each database user should have only the privileges required to accomplish the current group tasks successfully.

Revoking a privilege from PUBLIC can cause significant cascading effects. If any privilege related to a DML operation is revoked from PUBLIC (for example, SELECT ANY TABLE, UPDATE ON emp), then all procedures in the database, including functions and packages, must be *reauthorized* before they can be used again. Therefore, exercise caution when granting and revoking DML-related privileges to or from PUBLIC.

See Also:

- Managing Object Dependencies in *Oracle Database Administrator's Guide* for more information about object dependencies
- A Security Checklist, in Chapter 7, "Security Policies"

When Do Grants and Revokes Take Effect?

Depending on what is granted or revoked, a grant or revoke takes effect at different times:

- All grants and revokes of system and object privileges to anything (users, roles, and PUBLIC) take immediate effect.
- All grants and revokes of roles to anything (users, other roles, PUBLIC) take effect only when a current user session issues a SET ROLE statement to reenable the role after the grant and revoke, or when a new user session is created after the grant and revoke.

You can see which roles are currently enabled by examining the SESSION_ROLES data dictionary view.

The SET ROLE Statement

During the session, the user or an application can use the SET ROLE statement any number of times to change the roles currently enabled for the session. You must already have been granted the roles that you name in the SET ROLE statement. The number of roles that can be concurrently enabled is limited by the initialization parameter MAX_ENABLED_ROLES.

This example enables the role clerk, which you have already been granted, and specifies the password.

```
SET ROLE clerk IDENTIFIED BY bicentennial;
```

You can disable all roles with the following statement:

```
SET ROLE NONE;
```

Specifying Default Roles

When a user logs on, Oracle enables all privileges granted explicitly to the user and all privileges in the default roles of the user.

A list of default roles for a user can be set and altered using the ALTER USER statement. The ALTER USER statement enables you to specify roles that are to be enabled when a user connects to the database, without requiring the user to specify the role passwords. The user must have already been directly granted the roles with a GRANT statement. You cannot specify as a default role any role managed by an external service including a directory service (external roles or global roles).

The following example establishes default roles for user jane:

```
ALTER USER jane DEFAULT ROLE payclerk, pettycash;
```

You cannot set default roles for a user in the CREATE USER statement. When you first create a user, the default user role setting is ALL, which causes all roles subsequently granted to the user to be default roles. Use the ALTER USER statement to limit the default user roles.

Caution: When you create a role (other than a user role), it is granted to you implicitly and added as a default role. You receive an error at login if you have more than MAX_ENABLED_ROLES. You can avoid this error by altering the default user roles to be less than MAX_ENABLED_ROLES. Therefore, you should change the DEFAULT ROLE settings of SYS and SYSTEM before creating user roles.

Restricting the Number of Roles that a User Can Enable

A user can enable as many roles as specified by the initialization parameter MAX_ENABLED_ROLES. All indirectly granted roles enabled as a result of enabling a primary role are included in this count. The database administrator can alter this limitation by modifying the value for this parameter. Higher values permit each user session to have more concurrently enabled roles, but these values also cause more memory to be used for each user session. This occurs because the PGA size requires four bytes for each role in each session. Determine the highest number of roles that will be concurrently enabled by any one user and use this value for the MAX_ENABLED_ROLES parameter.

Granting Roles Using the Operating System or Network

This section describes aspects of granting roles through your operating system or network, and contains the following topics:

- Using Operating System Role Identification
- Using Operating System Role Management
- Granting and Revoking Roles When OS_ROLES=TRUE
- Enabling and Disabling Roles When OS_ROLES=TRUE
- Using Network Connections with Operating System Role Management

Instead of a security administrator explicitly granting and revoking database roles to and from users using GRANT and REVOKE statements, the operating system that operates Oracle can grant roles to users at connect time. Roles can be administered using the operating system and passed to Oracle Database when a user creates a session. As part of this mechanism, the default roles of a user and the roles granted to a user with the ADMIN OPTION can be identified. If the operating system is used to authorize users for roles, then all roles must be created in the database and privileges assigned to the role with GRANT statements.

Roles can also be granted through a network service.

The advantage of using the operating system to identify the database roles of a user is that privilege management for an Oracle database can be externalized. The security facilities offered by the operating system control user privileges. This option may offer advantages of centralizing security for a number of system activities, such as the following situation:

- MVS Oracle administrators want RACF groups to identify database user roles.
- UNIX Oracle administrators want UNIX groups to identify database user roles.
- VMS Oracle administrators want to use rights identifiers to identify database user roles.

The main disadvantage of using the operating system to identify the database roles of a user is that privilege management can only be performed at the role level. Individual

privileges cannot be granted using the operating system, but can still be granted inside the database using GRANT statements.

A secondary disadvantage of using this feature is that, by default, users cannot connect to the database through the shared server or any other network connection if the operating system is managing roles. However, you can change this default as described in "Using Network Connections with Operating System Role Management" on page 11-29.

> **Note:** The features described in this section are available only on some operating systems. See your operating system specific Oracle documentation to determine if you can use these features.

Using Operating System Role Identification

To operate a database so that it uses the operating system to identify each user's database roles when a session is created, set the initialization parameter OS_ROLES to TRUE (and restart the instance, if it is currently running). When a user attempts to create a session with the database, Oracle Database initializes the user security domain using the database roles identified by the operating system.

To identify database roles for a user, each Oracle user's operating system account must have operating system identifiers (these may be called groups, rights identifiers, or other similar names) that indicate which database roles are to be available for the user. Role specification can also indicate which roles are the default roles of a user and which roles are available with the ADMIN OPTION. No matter which operating system is used, the role specification at the operating system level follows the format:

```
ora_ID_ROLE[_[d][a]]
```

where:

- ID has a definition that varies on different operating systems. For example, on VMS, ID is the instance identifier of the database, on MVS, it is the machine type, and on UNIX, it is the system ID.

  > **Note:** ID is case-sensitive to match your ORACLE_SID. ROLE is not case-sensitive.

- ROLE is the name of the database role.
- d is an optional character that indicates this role is to be a default role of the database user.
- a is an optional character that indicates this role is to be granted to the user with the ADMIN OPTION. This allows the user to grant the role to other roles only. Roles cannot be granted to users if the operating system is used to manage roles.

  > **Note:** If either the d or a character is specified, then they must be preceded by an underscore.

For example, an operating system account might have the following roles identified in its profile:

```
ora_PAYROLL_ROLE1
ora_PAYROLL_ROLE2_a
```

```
ora_PAYROLL_ROLE3_d
ora_PAYROLL_ROLE4_da
```

When the corresponding user connects to the `payroll` instance of Oracle, `role3` and `role4` are defaults, while `role2` and `role4` are available with the `ADMIN OPTION`.

Using Operating System Role Management

When you use operating system managed roles, it is important to note that database roles are being granted to an operating system user. Any database user to which the operating system user is able to connect will have the authorized database roles enabled. For this reason, you should consider defining all Oracle users as `IDENTIFIED EXTERNALLY` if you are using `OS_ROLES = TRUE`, so that the database accounts are tied to the operating system account that was granted privileges.

Granting and Revoking Roles When OS_ROLES=TRUE

If `OS_ROLES` is set to `TRUE`, then the operating system completely manages the grants and revokes of roles to users. Any previous grants of roles to users using `GRANT` statements do not apply, however, they are still listed in the data dictionary. Only the role grants made at the operating system level to users apply. Users can still grant privileges to roles and users.

Note: If the operating system grants a role to a user with the `ADMIN OPTION`, then the user can grant the role only to other roles.

Enabling and Disabling Roles When OS_ROLES=TRUE

If `OS_ROLES` is set to `TRUE`, then any role granted by the operating system can be dynamically enabled using the `SET ROLE` statement. This still applies, even if the role was defined to require a password or operating system authorization. However, any role not identified in a user's operating system account cannot be specified in a `SET ROLE` statement, even if a role has been granted using a `GRANT` statement when `OS_ROLES = FALSE`. (If you specify such a role, then Oracle ignores it.)

When `OS_ROLES = TRUE`, a user can enable as many roles as specified by the initialization parameter `MAX_ENABLED_ROLES`.

Using Network Connections with Operating System Role Management

If you choose to have the operating system to manage roles, then by default users cannot connect to the database through the shared server. This restriction is the default because a remote user could impersonate another operating system user over a nonsecure connection.

If you are not concerned with this security risk and want to use operating system role management with the shared server, or any other network connection, then set the initialization parameter `REMOTE_OS_ROLES` in the database's initialization parameter file to `TRUE`. The change will take effect the next time you start the instance and mount the database. The default setting of this parameter is `FALSE`.

Viewing Privilege and Role Information

To access information about grants of privileges and roles, you can query the following data dictionary views:

View	Description
DBA_COL_PRIVS ALL_COL_PRIVS USER_COL_PRIVS	DBA view describes all column object grants in the database. ALL view describes all column object grants for which the current user or PUBLIC is the object owner, grantor, or grantee. USER view describes column object grants for which the current user is the object owner, grantor, or grantee.
ALL_COL_PRIVS_MADE USER_COL_PRIVS_MADE	ALL view lists column object grants for which the current user is object owner or grantor. USER view describes column object grants for which the current user is the grantor.
ALL_COL_PRIVS_RECD USER_COL_PRIVS_RECD	ALL view describes column object grants for which the current user or PUBLIC is the grantee. USER view describes column object grants for which the current user is the grantee.
DBA_TAB_PRIVS ALL_TAB_PRIVS USER_TAB_PRIVS	DBA view lists all grants on all objects in the database. ALL view lists the grants on objects where the user or PUBLIC is the grantee. USER view lists grants on all objects where the current user is the grantee.
ALL_TAB_PRIVS_MADE USER_TAB_PRIVS_MADE	ALL view lists the all object grants made by the current user or made on the objects owned by the current user. USER view lists grants on all objects owned by the current user.
ALL_TAB_PRIVS_RECD USER_TAB_PRIVS_RECD	ALL view lists object grants for which the user or PUBLIC is the grantee. USER view lists object grants for which the current user is the grantee.
DBA_ROLES	This view lists all roles that exist in the database.
DBA_ROLE_PRIVS USER_ROLE_PRIVS	DBA view lists roles granted to users and roles. USER view lists roles granted to the current user.
DBA_SYS_PRIVS USER_SYS_PRIVS	DBA view lists system privileges granted to users and roles. USER view lists system privileges granted to the current user.
ROLE_ROLE_PRIVS	This view describes roles granted to other roles. Information is provided only about roles to which the user has access.
ROLE_SYS_PRIVS	This view contains information about system privileges granted to roles. Information is provided only about roles to which the user has access.
ROLE_TAB_PRIVS	This view contains information about object privileges granted to roles. Information is provided only about roles to which the user has access.
SESSION_PRIVS	This view lists the privileges that are currently enabled for the user.
SESSION_ROLES	This view lists the roles that are currently enabled to the user.

Some examples of using these views follow. For these examples, assume the following statements have been issued:

```
CREATE ROLE security_admin IDENTIFIED BY honcho;

GRANT CREATE PROFILE, ALTER PROFILE, DROP PROFILE,
    CREATE ROLE, DROP ANY ROLE, GRANT ANY ROLE, AUDIT ANY,
    AUDIT SYSTEM, CREATE USER, BECOME USER, ALTER USER, DROP USER
    TO security_admin WITH ADMIN OPTION;

GRANT SELECT, DELETE ON SYS.AUD$ TO security_admin;

GRANT security_admin, CREATE SESSION TO swilliams;

GRANT security_admin TO system_administrator;

GRANT CREATE SESSION TO jward;
```

```
GRANT SELECT, DELETE ON emp TO jward;

GRANT INSERT (ename, job) ON emp TO swilliams, jward;
```

> **See Also:** *Oracle Database Reference* for a detailed description of these data dictionary views

Listing All System Privilege Grants

The following query returns all system privilege grants made to roles and users:

```
SELECT * FROM DBA_SYS_PRIVS;

GRANTEE            PRIVILEGE                        ADM
--------------     -------------------------------- ---
SECURITY_ADMIN     ALTER PROFILE                    YES
SECURITY_ADMIN     ALTER USER                       YES
SECURITY_ADMIN     AUDIT ANY                        YES
SECURITY_ADMIN     AUDIT SYSTEM                     YES
SECURITY_ADMIN     BECOME USER                      YES
SECURITY_ADMIN     CREATE PROFILE                   YES
SECURITY_ADMIN     CREATE ROLE                      YES
SECURITY_ADMIN     CREATE USER                      YES
SECURITY_ADMIN     DROP ANY ROLE                    YES
SECURITY_ADMIN     DROP PROFILE                     YES
SECURITY_ADMIN     DROP USER                        YES
SECURITY_ADMIN     GRANT ANY ROLE                   YES
SWILLIAMS          CREATE SESSION                   NO
JWARD              CREATE SESSION                   NO
```

Listing All Role Grants

The following query returns all the roles granted to users and other roles:

```
SELECT * FROM DBA_ROLE_PRIVS;

GRANTEE            GRANTED_ROLE                       ADM
------------------ ---------------------------------- ---
SWILLIAMS          SECURITY_ADMIN                     NO
```

Listing Object Privileges Granted to a User

The following query returns all object privileges (not including column-specific privileges) granted to the specified user:

```
SELECT TABLE_NAME, PRIVILEGE, GRANTABLE FROM DBA_TAB_PRIVS
    WHERE GRANTEE = 'JWARD';

TABLE_NAME   PRIVILEGE    GRANTABLE
-----------  ------------ ----------
EMP          SELECT       NO
EMP          DELETE       NO
```

To list all the column-specific privileges that have been granted, use the following query:

```
SELECT GRANTEE, TABLE_NAME, COLUMN_NAME, PRIVILEGE
    FROM DBA_COL_PRIVS;

GRANTEE      TABLE_NAME     COLUMN_NAME      PRIVILEGE
-----------  ------------   -------------    --------------
```

```
SWILLIAMS    EMP           ENAME           INSERT
SWILLIAMS    EMP           JOB             INSERT
JWARD        EMP           NAME            INSERT
JWARD        EMP           JOB             INSERT
```

Listing the Current Privilege Domain of Your Session

The following query lists all roles currently enabled for the issuer:

```
SELECT * FROM SESSION_ROLES;
```

If `swilliams` has enabled the `security_admin` role and issues this query, then Oracle Database returns the following information:

```
ROLE
------------------------------
SECURITY_ADMIN
```

The following query lists all system privileges currently available in the security domain of the issuer, both from explicit privilege grants and from enabled roles:

```
SELECT * FROM SESSION_PRIVS;
```

If `swilliams` has the `security_admin` role enabled and issues this query, then Oracle returns the following results:

```
PRIVILEGE
----------------------------------------
AUDIT SYSTEM
CREATE SESSION
CREATE USER
BECOME USER
ALTER USER
DROP USER
CREATE ROLE
DROP ANY ROLE
GRANT ANY ROLE
AUDIT ANY
CREATE PROFILE
ALTER PROFILE
DROP PROFILE
```

If the `security_admin` role is disabled for `swilliams`, then the first query would return no rows, while the second query would only return a row for the `CREATE SESSION` privilege grant.

Listing Roles of the Database

The `DBA_ROLES` data dictionary view can be used to list all roles of a database and the authentication used for each role. For example, the following query lists all the roles in the database:

```
SELECT * FROM DBA_ROLES;

ROLE                 PASSWORD
----------------     --------
CONNECT              NO
RESOURCE             NO
DBA                  NO
SECURITY_ADMIN       YES
```

Listing Information About the Privilege Domains of Roles

The ROLE_ROLE_PRIVS, ROLE_SYS_PRIVS, and ROLE_TAB_PRIVS data dictionary views contain information on the privilege domains of roles. For example, the following query lists all the roles granted to the system_admin role:

```
SELECT GRANTED_ROLE, ADMIN_OPTION
   FROM ROLE_ROLE_PRIVS
   WHERE ROLE = 'SYSTEM_ADMIN';

GRANTED_ROLE              ADM
----------------          ----
SECURITY_ADMIN            NO
```

The following query lists all the system privileges granted to the security_admin role:

```
SELECT * FROM ROLE_SYS_PRIVS WHERE ROLE = 'SECURITY_ADMIN';

ROLE                    PRIVILEGE                      ADM
----------------------- ------------------------------ ---
SECURITY_ADMIN           ALTER PROFILE                 YES
SECURITY_ADMIN           ALTER USER                    YES
SECURITY_ADMIN           AUDIT ANY                     YES
SECURITY_ADMIN           AUDIT SYSTEM                  YES
SECURITY_ADMIN           BECOME USER                   YES
SECURITY_ADMIN           CREATE PROFILE                YES
SECURITY_ADMIN           CREATE ROLE                   YES
SECURITY_ADMIN           CREATE USER                   YES
SECURITY_ADMIN           DROP ANY ROLE                 YES
SECURITY_ADMIN           DROP PROFILE                  YES
SECURITY_ADMIN           DROP USER                     YES
SECURITY_ADMIN           GRANT ANY ROLE                YES
```

The following query lists all the object privileges granted to the security_admin role:

```
SELECT TABLE_NAME, PRIVILEGE FROM ROLE_TAB_PRIVS
    WHERE ROLE = 'SECURITY_ADMIN';

TABLE_NAME                   PRIVILEGE
---------------------------  ----------------
AUD$                         DELETE
AUD$                         SELECT
```

12

Configuring and Administering Auditing

Auditing is always about accountability, and is frequently done to protect and preserve privacy for the information stored in databases. Concern about privacy policies and practices has been rising steadily with the ubiquitous use of databases in businesses and on the Internet. Oracle Database provides a depth of auditing that readily enables system administrators to implement enhanced protections, early detection of suspicious activities, and finely-tuned security responses.

The types of auditing available in Oracle Database are described in Chapter 8, "Database Auditing: Security Considerations".

The present chapter explains how to choose the types of auditing you need, how to manage that auditing, and how to use the information gained, in the following sections:

- Actions Audited by Default
- Guidelines for Auditing
- What Information Is Contained in the Audit Trail?
- Managing the Standard Audit Trail
- Viewing Database Audit Trail Information
- Fine-Grained Auditing

Actions Audited by Default

Regardless of whether database auditing is enabled, Oracle Database *always* audits certain database-related operations and writes them to the operating system audit file. This fact is called mandatory auditing, and it includes the following operations:

- Connections to the instance with administrator privileges

  An audit record is generated that lists the operating system user connecting to Oracle Database as `SYSOPER` or `SYSDBA`. This provides for accountability of users with administrative privileges. Full auditing for these users can be enabled as explained in "Auditing Administrative Users" on page 12-3.

- Database startup

  An audit record is generated that lists the operating system user starting the instance, the user terminal identifier, and the date and time stamp. This data is stored in the operating system audit trail because the database audit trail is not available until after the startup has successfully completed.

- Database shutdown

An audit record is generated that lists the operating system user shutting down the instance, the user terminal identifier, and the date and time stamp.

Guidelines for Auditing

Oracle Database 10*g* enables you to send audit records to the database audit trail or the operating system audit trail, when the operating system is capable of receiving them. The database audit records can also be written to operating system files in XML format. The audit trail for database administrators, for example, is typically written to a secure location in the operating system. Writing audit trails to the operating system provides a way for a separate auditor who has `root` privileges on the operating system to hold all DBAs (who don't have root access) accountable for their actions. These options, added to the broad selection of audit options and customizable triggers or stored procedures, give you the flexibility to implement an auditing scheme that suits your specific business needs.

This section describes guidelines for auditing and contains the following topics:

- Keeping Audited Information Manageable
- Auditing Normal Database Activity
- Auditing Suspicious Database Activity
- Auditing Administrative Users
- Using Triggers
- Deciding Whether to Use the Database or Operating System Audit Trail

Keeping Audited Information Manageable

Although auditing is relatively inexpensive, limit the number of audited events as far as possible. Doing so minimizes the performance impact on the execution of audited statements and the size of the audit trail, making it easier to analyze and understand.

Use the following general guidelines when devising an auditing strategy:

- Evaluate the purpose for auditing.

  After you have a clear understanding of the reasons for auditing, you can devise an appropriate auditing strategy and avoid unnecessary auditing.

  For example, suppose you are auditing to investigate suspicious database activity. This information by itself is not specific enough. What types of suspicious database activity do you suspect or have you noticed? A more focused auditing purpose might be to audit unauthorized deletions from arbitrary tables in the database. This purpose narrows the type of action being audited and the type of object being affected by the suspicious activity.

- Audit knowledgeably.

  Audit the minimum number of statements, users, or objects required to get the targeted information. This prevents unnecessary audit information from cluttering the meaningful information and consuming valuable space in the `SYSTEM` tablespace. Balance your need to gather sufficient security information with your ability to store and process it.

  For example, if you are auditing to gather information about database activity, then determine exactly what types of activities you want to track, audit only the activities of interest, and audit only for the amount of time necessary to gather the

information that you desire. As another example, do not audit *objects* if you are only interested in logical I/O information for each session.

Auditing Normal Database Activity

When your purpose for auditing is to gather historical information about particular database activities, use the following guidelines:

- Audit only pertinent actions.

  To avoid cluttering meaningful information with useless audit records and reduce the amount of audit trail administration, only audit the targeted database activities.

- Archive audit records and purge the audit trail.

  After you have collected the required information, archive the audit records of interest and purge the audit trail of this information.

- Privacy considerations

  Privacy regulations often lead to additional business privacy policies. Most privacy laws require businesses to monitor access to personally identifiable information (PII), and such monitoring is implemented by auditing. A business-level privacy policy should address all relevant aspects of data access and user accountability, including technical, legal, and company-policy concerns.

Auditing Suspicious Database Activity

When you audit to monitor suspicious database activity, use the following guidelines:

- First audit generally, and then specifically.

  When starting to audit for suspicious database activity, it is common that not much information is available to target specific users or schema objects. Therefore, audit options must be set more generally at first. Once preliminary audit information is recorded and analyzed, the general audit options should be turned off and more specific audit options enabled. This process should continue until enough evidence is gathered to draw conclusions about the origin of the suspicious database activity.

- Protect the audit trail.

  When auditing for suspicious database activity, protect the audit trail so that audit information cannot be added, changed, or deleted without being audited.

See Also: "Protecting the Standard Audit Trail" on page 12-18

Auditing Administrative Users

Sessions for users who connect as `SYS` can be fully audited, including all users connecting as `SYSDBA` or `SYSOPER`. Use the `AUDIT_SYS_OPERATIONS` initialization parameter to specify whether such users are to be audited. For example, the following setting specifies that `SYS` is to be audited:

```
AUDIT_SYS_OPERATIONS = TRUE
```

The default value, `FALSE`, disables `SYS` auditing.

All audit records for `SYS` are written to the operating system file that contains the audit trail, and not to `SYS.AUD$` (also viewable as `DBA_AUDIT_TRAIL`).

- In Windows, for example, when AUDIT_TRAIL=OS, audit records are written as events to the Event Viewer log file. If either XML or XML, EXTENDED is specified, then audit records are written in the XML format.

  Notes: DB_UNIQUE_NAME is defined as the globally unique name for the database in *Oracle Database Reference*. Refer to this book for other details and naming rules.

  The adump directory is the first default location used if the AUDIT_FILE_DEST init.ora parameter is not set or does not point to a valid directory. If writing to that first default location fails, then the *$ORACLE_HOME/rdbms/audit* directory is used as the backup default destination. If that attempt fails, then the audited operation fails and a message is written to the alert log.

  When AUDIT_TRAIL=OS, audit filenames will continue to be of the following form:

  <*short_form_of_process_name*>_<processid>.aud

  For example, the short process name ora is used for dedicated server processes, and the names s001, s002, and so on are used for shared server processes.

  When AUDIT_TRAIL=XML, the same audit filenames have the extension xml instead of aud.

- If the AUDIT_FILE_DEST parameter is not specified, then the default location is $ORACLE_BASE/admin/$DB_UNIQUE_NAME/adump in Solaris and $ORACLE_BASE\admin\$DB_UNIQUE_NAME\adump in Windows.
- For other operating systems, refer to their audit trail documentation.

All SYS-issued SQL statements are audited indiscriminately and regardless of the setting of the AUDIT_TRAIL initialization parameter.

Consider the following SYS session:

```
CONNECT / AS SYSDBA;
ALTER SYSTEM FLUSH SHARED_POOL;
UPDATE salary SET base=1000 WHERE name='myname';
```

When SYS auditing is enabled, both the ALTER SYSTEM and UPDATE statements are displayed in the operating system audit file as follows:

```
Thu Jan 24 12:58:00 2002
ACTION: 'CONNECT'
DATABASE USER: '/'
OSPRIV: SYSDBA
CLIENT USER: jeff
CLIENT TERMINAL: pts/2
STATUS: 0

Thu Jan 24 12:58:00 2002
ACTION: 'alter system flush shared_pool'
DATABASE USER: ''
OSPRIV: SYSDBA
CLIENT USER: jeff
CLIENT TERMINAL: pts/2
STATUS: 0
```

```
Thu Jan 24 12:58:00 2002
ACTION: 'update salary set base=1000 where name='myname''
DATABASE USER: ''
OSPRIV: SYSDBA
CLIENT USER: jeff
CLIENT TERMINAL: pts/2
STATUS: 0
```

Because of the superuser privileges available to users who connect as SYSDBA, Oracle recommends that DBAs rarely use this connection and only when necessary. Normal day to day maintenance activity can usually be done by DBAs, who are regular database users with the DBA role, or by means of a DBA role (for example, mydba or jr_dba) that your organization customizes.

Using Triggers

You can often use triggers to record additional customized information that is not automatically included in audit records, thereby customizing your own audit conditions and record contents. For example, you could define a trigger on the EMP table to generate an audit record whenever the salary of an employee is increased by more than 10 percent. You can include selected information, such as the values of SALARY before and after it was changed:

```
CREATE TRIGGER audit_emp_salaries
AFTER INSERT OR DELETE OR UPDATE ON employee_salaries
for each row
begin
if (:new.salary> :old.salary * 1.10)
      then
      insert into emp_salary_audit values (
      :employee_no,
      :old.salary,
      :new.salary,
      user,
      sysdate);
      endif;
end;
```

Furthermore, you can use event triggers to enable auditing options for specific users on login, and disable them upon logoff.

However, while Oracle triggers can readily monitor DML actions such as INSERT, UPDATE, and DELETE, monitoring on SELECT can be costly and, in some cases, uncertain. Triggers do not enable businesses to capture the statement executed as well as the result set from a query. They also do not enable users to define their own alert action in addition to simply inserting an audit record into the audit trail.

For these capabilities, use fine-grained auditing, which provides an extensible auditing mechanism supporting definition of key conditions for granular audit as well as an event handler to actively alert administrators to misuse of data access rights. Refer to Fine-Grained Auditing on page 12-24.

Deciding Whether to Use the Database or Operating System Audit Trail

The data dictionary of every Oracle database has a table named SYS.AUD$, commonly referred to as the database **audit trail**. Depending on configuration choices, this table can reside in different schema, such as the traditional SYS schema in the SYSTEM tablespace. For example, when Oracle Label Security is active, the AUD$ table is moved

to the SYSTEM schema, with SYS.AUD$ becoming a synonym to SYSTEM.AUD$. The database audit trail is viewable as DBA_AUDIT_TRAIL, and its entries arise from auditing database statements, privileges, or schema objects.

You can optionally choose to store the database audit information in an operating system file. If auditing facility of your operating system writes audit records to a file that Oracle can also write to, then you can direct the database audit entries to this file. For example, the Windows operating system allows Oracle to write audit records as events to the Application Event Log, viewable by the Event Viewer.

Consider the advantages and disadvantages of using either the database or operating system audit trail to store database audit records.

Using the *database* audit trail offers the following advantages:

- You can view selected portions of the audit trail with the predefined audit trail views of the data dictionary, such as DBA_AUDIT_TRAIL.
- You can use Oracle tools (such as Oracle Reports) or third-party tools to generate audit reports.

Using the *operating system* audit trail offers the following advantages:

- Audit records stored in operating system files can be more secure than database-stored audit records because access can require file permissions that DBAs do not have. Greater availability is another advantage to operating system storage for audit records, in that they remain available even if the database is temporarily inaccessible.
- If init.ora specifies AUDIT_TRAIL=XML, then audit records are written to the operating system as XML files. A new dynamic view, V$XML_AUDIT_TRAIL, makes such XML audit records available to DBAs through SQL query, providing enhanced usability. Querying this view causes all XML files (all files with a .xml extension) in the AUDIT_FILE_DEST directory to be parsed and presented in relational table format.
- DBA_COMMON_AUDIT_TRAIL also includes the standard and fine grained audit trails as written to database tables, as well as the XML-format audit trail records. The DBA_COMMON_AUDIT_TRAIL view includes the contents of the V$XML_AUDIT_TRAIL dynamic view for standard and fine-grained audit records. DBA_COMMON_AUDIT_TRAIL also includes the standard and fine grained audit trails as written to database tables, as well as the XML-format standard, fine grained, SYS and mandatory audit trails.
- Using your operating system audit trail can enable you to consolidate audit records from multiple sources, including Oracle and other applications. Examining system activity can become more efficient with all audit records in one place. Using XML audit records permits use of any standard XML editing tool to review or extract information from those records.

See Also:

- Your operating-system-specific documentation for information about its auditing capabilities.
- Audit Trail Views on page 12-19

What Information Is Contained in the Audit Trail?

Oracle Database can write records to either the database audit trail, an operating system file, or both. This section describes what information the audit trail contains.

Database Audit Trail Contents

The database audit trail is a single table named `SYS.AUD$` in the `SYS` schema of each Oracle Database data dictionary. Several predefined views are provided to help you use the information in this table, such as `DBA_AUDIT_TRAIL`.

Audit trail records can contain different types of information, depending on the events audited and the auditing options set. The partial list in the following section shows columns that always appear in the audit trail: if the data they represent is available, then that data populates the corresponding column. (For certain columns, this list has the column name as it displays in the audit record, shown here inside parentheses.) The operating system audit trail has only those columns marked `Yes` in the corresponding column.

Table 12–1 Audit Trail Record Data

Data Populated in Database Audit Trail	In Operating System Audit Trail?
(*) Bind values used for the SQL statement, if any	Footnote 1
(*) SQL text (the SQL text that triggered the auditing)	Footnote 1
Completion code of the operation	Yes
Database user name (DATABASE USER)	Yes
Date and time stamp in UTC (Coordinated Universal Time) format	No
Distinguished name	Yes
Global User unique ID	No
Instance number	No
Name of the schema object accessed	Yes
Operating system login user name (CLIENT USER)	Yes
Operation performed or attempted (ACTION)	Yes
Process number	Footnote 2
Proxy Session audit ID	No
SCN (system change number) for the SQL statement	No
Session identifier	Yes
System privileges used(PRIVILEGE)	Yes
Terminal identifier	Yes
Transaction ID	No

Footnote 1: Asterisked (*) columns in Table 12–1 appear in the audit records only if your database initialization file, `init.ora`, specifies `AUDIT_TRAIL=DB, EXTENDED` or `AUDIT_TRAIL=XML, EXTENDED`. Also, for an array, the values recorded are only the last set of bind values.

Footnote 2: Process number is populated as `ProcessId` on Unix systems. In Windows systems, the label is `ProcessId:ThreadId` (or `ProcessId` if it is not running as a thread).

Note: If the AUDIT_TRAIL parameter in init.ora is set to XML or XML, EXTENDED, then standard audit records are sent to OS files in XML format. Because XML is a standard document format, many utilities are available to parse and analyze such XML data.

If the database destination for audit records becomes full or unavailable and therefore unable to accept new records, then an audited action cannot complete. Instead, it causes an error message and is not done. In most cases, using an operating system log as the audit trail destination allows such an action to complete.

See Also:

- Keeping Audited Information Manageable on page 12-2.
- Controlling the Growth and Size of the Standard Audit Trail on page 12-16

The audit trail does not store information about any data values that might be involved in the audited statement. For example, old and new data values of updated rows are not stored when an UPDATE statement is audited. However, this specialized type of auditing can be performed using fine-grained auditing methods.

The DBA_COMMON_AUDIT_TRAIL view combines standard and fine-grained audit log records.

You can use the Flashback Query feature to show the old and new values of the updated rows, subject to any auditing policy presently in force. The current policies are enforced even if the flashback is to an old query that was originally subject to a different policy. Current business access rules always apply.

See Also:

- Fine-Grained Auditing on page 12-24 for more information about methods of fine-grained auditing
- Flashback Queries in *Oracle Database Administrator's Guide*
- Flashback entries in the table of system privileges in Chapter 18 of *Oracle Database SQL Reference*

Note: To read from FLASHBACK_TRANSACTION_TABLE or V$LOGMNR_CONTENTS, the following system privilege is required: SELECT ANY TRANSACTION.

Audit Information Stored in an Operating System File

The operating system file that contains the audit trail can include any of the following data:

- Audit records generated by the operating system
- Database audit trail records
- Database actions that are always audited
- Audit records for administrative users (SYS)

Audit trail records written to an operating system audit trail may contain encoded information, but this information can be decoded using data dictionary tables and error messages as follows:

Encoded Information	How to Decode
Action code	Describes the operation performed or attempted, using codes listed in the `AUDIT_ACTIONS` data dictionary table, with their descriptions.
Privileges used	Describes any system privileges used to perform the operation, using codes listed in the `SYSTEM_PRIVILEGE_MAP` table, with their descriptions.
Completion code	Describes the result of the attempted operation, using codes listed in *Oracle Database Error Messages*, with their descriptions. Successful operations return a value of zero and unsuccessful operations return an Oracle error code corresponding to the reason the operation was unsuccessful.

Managing the Standard Audit Trail

This section describes various aspects of managing standard audit trail information, and contains the following topics:

- Enabling and Disabling Standard Auditing
- Standard Auditing in a Multitier Environment
- Enabling Standard Auditing Options
- Disabling Standard Audit Options
- Controlling the Growth and Size of the Standard Audit Trail
- Protecting the Standard Audit Trail
- Auditing the Standard Audit Trail

Enabling and Disabling Standard Auditing

Any authorized database user can set statement, privilege, and object auditing options at any time. However, Oracle Database does not generate audit information for the standard database audit trail unless database auditing is enabled. The security administrator is normally responsible for controlling auditing.

This section discusses the initialization parameters that enable and disable standard auditing.

Note:

- The initialization parameters `AUDIT_SYS_OPERATIONS` and `AUDIT_TRAIL` affecting standard auditing are static. `Static` means that if you change their values, you must shut down and restart your database for the new values to take effect.
- The `AUDIT_FILE_DEST` initialization parameter can be changed with `ALTER SYSTEM SET AUDIT_FILE_DEST = <dir> DEFERRED`, meaning the new destination will be effective for all subsequent sessions.

Setting the AUDIT_TRAIL Initialization Parameter

Database auditing is enabled and disabled by the AUDIT_TRAIL initialization parameter in the database initialization parameter file, init.ora. The parameter can be set to the following values:

Parameter Value	Meaning
DB	Enables database auditing and directs all audit records to the database audit trail (SYS.AUD$), except for records that are always written to the operating system audit trail
XML	All elements of the AuditRecord node except Sql_Text and Sql_Bind will be printed to the operating system XML audit file.
DB,EXTENDED	Does all actions of AUDIT_TRAIL=DB and also populates the SQL bind and SQL text CLOB-type columns of the SYS.AUD$ table, wherever possible. (These two columns are populated only when this parameter is specified.)
XML,EXTENDED	Does all actions of AUDIT_TRAIL=XML and also populates the SQL bind and SQL text CLOB-type columns of the SYS.AUD$ table, wherever possible. (These columns are populated only when this parameter is specified.)
OS	Enables database auditing and directs all audit records to an operating system file
NONE	Disables standard auditing (This value is the default.)

Note that changes altering what objects are audited do not require restarting the database. Restart is only required if a universal change is made, such as turning on or off *all* auditing.

Note: You do not need to set AUDIT_TRAIL to enable either fine-grained auditing or SYS auditing. For fine-grained auditing, you simply add and remove FGA policies as you see fit, applying them to the specific operations or objects you want to monitor. You can use the AUDIT_SYS_OPERATIONS parameter to enable and disable SYS auditing.

See Also:

- "Auditing Administrative Users" on page 12-3
- "Fine-Grained Auditing" on page 12-24

Specifying a Directory for the Operating System Auditing Trail

The AUDIT_FILE_DEST initialization parameter specifies an operating system directory into which the audit trail is written when either AUDIT_TRAIL=OS or AUDIT_TRAIL=XML is specified. Mandatory auditing information also goes into that directory, as do audit records for user SYS if the AUDIT_SYS_OPERATIONS initialization parameter is specified. AUDIT_FILE_DEST can be changed with ALTER SYSTEM SET AUDIT_FILE_DEST = <dir> DEFERRED, meaning the new destination will be effective for all subsequent sessions.

If the AUDIT_FILE_DEST parameter is not specified, then the default location on Solaris is $ORACLE_BASE/admin/$DB_UNIQUE_NAME/adump, and on Windows, $ORACLE_BASE\admin\$DB_UNIQUE_NAME\adump.

Notes:

- If your operating system supports an audit trail, then its location is operating-system-specific. For example, when the init.ora file contains AUDIT_TRAIL=OS, Windows operating systems write audit records as events to the application event log.
- When the init.ora file contains AUDIT_TRAIL=XML (or XML, EXTENDED), audit records are written to XML-formatted operating system files. Setting the AUDIT_FILE_DEST parameter on Windows causes the XML-format audit records to be stored in the directory specified by the parameter.

Specifying the Syslog Level

To enable syslog auditing, you assign a value of OS to the AUDIT_TRAIL initialization parameter, as described in "Setting the AUDIT_TRAIL Initialization Parameter" on page 12-10. You must also manually add the AUDIT_SYSLOG_LEVEL parameter to the database's initialization parameter file, init.ora. You assign to the AUDIT_SYSLOG_LEVEL parameter a facility and priority in the format AUDIT_SYSLOG_LEVEL=*facility.priority*. The *facility* argument describes the part of the operating system that is logging the message while the *priority* argument defines the severity of the message. The syslog daemon compares the value assigned to the facility argument of the AUDIT_SYSLOG_LEVEL parameter with the syslog.conf file in order to determine where to log information. For example, the following statement identifies the facility as local1 with a priority level of warning:

```
AUDIT_SYSLOG_LEVEL=local1.warning
```

Caution: You should have a strong understanding of how to work with syslog before enabling syslog auditing.

Standard Auditing in a Multitier Environment

In a multitier environment, Oracle preserves the identity of the client through all tiers, which enables auditing of actions taken on behalf of the client. To do such auditing, you use the BY *proxy* clause in your AUDIT statement.

This clause allows to do the following:

- Audit SQL statements issued by the specified proxy on its own behalf
- Audit statements executed on behalf of a specified user or users
- Audit all statements executed on behalf of any user

The following example audits SELECT TABLE statements issued on behalf of client jackson by the proxy application server appserve.

```
AUDIT SELECT TABLE
    BY appserve ON BEHALF OF jackson;
```

See Also: *Oracle Database Concepts* and*Oracle Database Application Developer's Guide - Fundamentals* for more information on proxies and multitier applications

Enabling Standard Auditing Options

You specify one of the four standard auditing options using the AUDIT statement:

Level	Effect
Statement	Causes auditing of specific SQL statements or groups of statements that affect a particular type of database object. For example, AUDIT TABLE audits the CREATE TABLE, TRUNCATE TABLE, COMMENT ON TABLE, and DELETE [FROM] TABLE statements.
Privilege	Audits SQL statements that are authorized by the specified system privilege. For example, AUDIT CREATE ANY TRIGGER audits statements issued using the CREATE ANY TRIGGER system privilege.
Object	Audits specific statements on specific objects, such as ALTER TABLE on the emp table.
Network	Audits unexpected errors in network protocol or internal errors in the network layer.

To use the AUDIT statement to set statement and privilege options, you must have the AUDIT SYSTEM privilege. To use it to set object audit options, you must own the object to be audited or have the AUDIT ANY privilege.

Audit statements that set statement and privilege audit options can include a BY clause to specify a list of users or application proxies to limit the scope of the statement and privilege audit options.

When setting auditing options, you can also specify the following conditions for auditing:

- BY SESSION/BY ACCESS

  BY SESSION causes Oracle Database to write a single record for all SQL statements of the same type issued in the same session. BY ACCESS causes Oracle to write one record for each access.

  Note: If AUDIT_TRAIL=OS or AUDIT_TRAIL=XML, then multiple records may still be written to the audit trail when BY SESSION is specified. Multiple records occur because while Oracle Database can write to the operating system file, the database cannot read it to detect that an audit entry already exists for the action.

- WHENEVER SUCCESSFUL/WHENEVER NOT SUCCESSFUL

  WHENEVER SUCCESSFUL chooses auditing only for statements that succeed. WHENEVER NOT SUCCESSFUL chooses auditing only for statements that fail or result in errors.

The implications of your choice of auditing options and the specification of AUDIT statement clauses are discussed in subsequent sections.

A new database session picks up auditing options from the data dictionary when the session is created. These auditing options remain in force for the duration of the

database connection. Setting new system or object auditing options causes all subsequent database sessions to use these options. Existing sessions continue using the audit options in place at session creation.

Caution: The `AUDIT` statement only specifies auditing options, it does not enable auditing as a whole. To turn auditing on and control whether Oracle Database generates audit records based on the audit options currently set, set the initialization parameter `AUDIT_TRAIL` as described in "Enabling and Disabling Standard Auditing" on page 12-9.

See Also: *Oracle Database SQL Reference* for a complete description of the `AUDIT` statement

Enabling Statement Auditing

Valid statement audit options that can be included in `AUDIT` and `NOAUDIT` statements are listed in the *Oracle Database SQL Reference*.

Two special cases of statement auditing are discussed in the following sections.

Auditing Connections and Disconnections The `SESSION` statement option is unique because it does not generate an audit record when a particular type of statement is issued. This option generates a single audit record for each session created by connections to an instance. An audit record is inserted into the audit trail at connect time and updated at disconnect time. Cumulative information about a session is stored in a single audit record that corresponds to the session. This record can include connection time, disconnection time, and logical and physical I/O processed, among other information.

To audit all successful and unsuccessful connections to and disconnections from the database, regardless of user, `BY SESSION` (the default and only value for this option), enter the following statement:

```
AUDIT SESSION;
```

You can set this option selectively for individual users also, as in the next example:

```
AUDIT SESSION
BY jeff, lori;
```

Auditing Statements That Fail Because an Object Does Not Exist The `NOT EXISTS` statement option specifies auditing of all SQL statements that fail because the target object does not exist.

Enabling Privilege Auditing

Privilege audit options exactly match the corresponding system privileges. For example, the option to audit use of the `DELETE ANY TABLE` privilege is `DELETE ANY TABLE`. To turn this option on, you use a statement similar to the following example:

```
AUDIT DELETE ANY TABLE
    BY ACCESS
    WHENEVER NOT SUCCESSFUL;
```

Oracle Database system privileges are listed in the *Oracle Database SQL Reference*.

To audit all successful and unsuccessful uses of the DELETE ANY TABLE system privilege, enter the following statement:

```
AUDIT DELETE ANY TABLE;
```

To audit all unsuccessful SELECT, INSERT, and DELETE statements on all tables and unsuccessful uses of the EXECUTE PROCEDURE system privilege, by all database users, and by individual audited statement, issue the following statement:

```
AUDIT SELECT TABLE, INSERT TABLE, DELETE TABLE, EXECUTE PROCEDURE
      BY ACCESS
      WHENEVER NOT SUCCESSFUL;
```

The AUDIT SYSTEM system privilege is required to set any statement or privilege audit option. Normally, the security administrator is the only user granted this system privilege.

Enabling Object Auditing

The *Oracle Database SQL Reference* lists valid object audit options and the schema object types for which each option is available.

A user can set any object audit option for the objects contained in the schema of the user. The AUDIT ANY system privilege is required to set an object audit option for an object contained in another user schema or to set the default object auditing option. Normally, the security administrator is the only user granted the AUDIT ANY privilege.

To audit all successful and unsuccessful DELETE statements on the jeff.emp table, BY SESSION (the default value), enter the following statement:

```
AUDIT DELETE ON jeff.emp;
```

To audit all successful SELECT, INSERT, and DELETE statements on the dept table owned by user jward, BY ACCESS, enter the following statement:

```
AUDIT SELECT, INSERT, DELETE
     ON jward.dept
     BY ACCESS
     WHENEVER SUCCESSFUL;
```

To set the default object auditing options to audit all unsuccessful SELECT statements, BY SESSION (the default), enter the following statement:

```
AUDIT SELECT
     ON DEFAULT
     WHENEVER NOT SUCCESSFUL;
```

Enabling Network Auditing

Valid statement audit options that can be included in AUDIT and NOAUDIT statements are listed in the *Oracle Database SQL Reference*.

The errors that network auditing uncovers (such as ACTION 122 Network Error in AUDIT_ACTIONS) are not connect failures, but rather can have several possible causes. One such possible cause could be an internal event set by an Oracle engineer purely for testing purposes. Other causes include conflicting configuration settings for encryption, such as the network not finding the information required to create or process expected encryption. Table 12–2 shows four such conditions.

Table 12–2 Auditable Network Error Conditions

Error	Cause	Action
TNS-02507 Encryption algorithm not installed	After picking an algorithm, the server was unable to find an index for it in its table of algorithms. This should be impossible because the algorithm was chosen (indirectly) from that list.	Not normally visible to the user. For further details, turn on tracing and rerun the operation. If error persists, then contact Oracle Support Services.
TNS-12648 Encryption or data integrity algorithm list empty	An Oracle Advanced Security list-of-algorithms parameter was empty.	Change the list to contain the name of at least one installed algorithm, or remove the list entirely if every installed algorithm is acceptable.
TNS-12649 Unknown encryption or data integrity algorithm	An Oracle Advanced Security list-of-algorithms parameter included an algorithm name that was not recognized.	Remove that algorithm name, correct it if it was misspelled, or install the driver for the missing algorithm.
TNS-12650 No common encryption or data integrity algorithm	The client and server have no algorithm in common for either encryption or data integrity or both.	Choose sets of algorithms that overlap. In other words, add one of the client algorithm choices to the server list or add one of the server list choices to the client algorithm.

Disabling Standard Audit Options

The `NOAUDIT` statement turns off the various audit options of Oracle Database 10*g*. Use it to reset statement and privilege audit options, and object audit options. A `NOAUDIT` statement that sets statement and privilege audit options can include the `BY` *`user`* or `BY` *`proxy`* option to specify a list of users to limit the scope of the statement and privilege audit options.

You can use a `NOAUDIT` statement to disable an audit option selectively using the `WHENEVER` clause. If the clause is not specified, then the auditing option is disabled entirely, for both successful and unsuccessful cases.

The `BY SESSION/BY ACCESS` option pair is *not* supported by the `NOAUDIT` statement. Audit options, no matter how they were turned on, are turned off by an appropriate `NOAUDIT` statement.

Caution: The `NOAUDIT` statement only specifies auditing options. It does not disable auditing as a whole. To turn auditing off and stop Oracle Database from generating audit records, set the initialization parameter `AUDIT_TRAIL` in the database's initialization parameter file as described in "Enabling and Disabling Standard Auditing" on page 12-9.

See Also: *Oracle Database SQL Reference* for a complete syntax listing of the `NOAUDIT` statement

Turning Off Statement and Privilege Auditing

The following statements turn off the corresponding audit options:

```
NOAUDIT session;
NOAUDIT session BY jeff, lori;
NOAUDIT DELETE ANY TABLE;
NOAUDIT SELECT TABLE, INSERT TABLE, DELETE TABLE,
    EXECUTE PROCEDURE;
```

The following statement turns off all statement audit options:

```
NOAUDIT ALL;
```

The following statement turns off all privilege audit options:

```
NOAUDIT ALL PRIVILEGES;
```

To disable statement or privilege auditing options, you must have the `AUDIT SYSTEM` system privilege.

Turning Off Object Auditing

The following statements turn off the corresponding auditing options:

```
NOAUDIT DELETE
   ON emp;
NOAUDIT SELECT, INSERT, DELETE
   ON jward.dept;
```

Furthermore, to turn off all object audit options on the `emp` table, enter the following statement:

```
NOAUDIT ALL
   ON emp;
```

To turn off all default object audit options, enter the following statement:

```
NOAUDIT ALL
   ON DEFAULT;
```

All schema objects that are created before this `NOAUDIT` statement is issued continue to use the default object audit options in effect at the time of their creation, unless overridden by an explicit `NOAUDIT` statement after their creation.

To disable object audit options for a specific object, you must be the owner of the schema object. To disable the object audit options of an object in another user's schema or to disable default object audit options, you must have the `AUDIT ANY` system privilege. A user with privileges to disable object audit options of an object can override the options set by any user.

Turning Off Network Auditing

The following statement turns off network auditing:

```
NOAUDIT NETWORK;
```

Recording of db link usage and login type stops.

Controlling the Growth and Size of the Standard Audit Trail

If the audit trail is full and no more audit records can be inserted, then audited statements cannot be successfully executed until the audit trail is purged. Warnings are returned to all users that issue audited statements. Therefore, the security administrator must control the growth and size of the audit trail.

When auditing is enabled and audit records are being generated, the audit trail grows according to two factors:

- The number of audit options turned on
- The frequency of execution of audited statements

To control the growth of the audit trail, you can use the following methods:

- Enable and disable database auditing. If it is enabled, then audit records are generated and stored in the audit trail. If it is disabled, then audit records are not generated.
- Be very selective about the audit options that are turned on. If more selective auditing is performed, then useless or unnecessary audit information is not generated and stored in the audit trail.
- Tightly control the ability to perform object auditing. This can be done in two different ways:
  - A security administrator owns all objects and the `AUDIT ANY` system privilege is never granted to any other user. Alternatively, all schema objects can belong to a schema for which the corresponding user does not have `CREATE SESSION` privilege.
  - All objects are contained in schemas that do not correspond to real database users (that is, the `CREATE SESSION` privilege is not granted to the corresponding user) and the security administrator is the only user granted the `AUDIT ANY` system privilege.

  In both scenarios, object auditing is controlled entirely by the security administrator.

The maximum size of the database audit trail (`SYS.AUD$` table) is determined by the default storage parameters of the `SYSTEM` tablespace, in which it is stored.

> **See Also:** Operating-system-specific Oracle documentation for more information about managing the operating system audit trail when directing audit records to that location

Purging Audit Records from the Audit Trail

After auditing is enabled for some time, the security administrator may want to delete records from the database audit trail both to free audit trail space and to facilitate audit trail management.

For example, to delete *all* audit records from the audit trail, enter the following statement:

```
DELETE FROM SYS.AUD$;
```

Alternatively, to delete all audit records from the audit trail generated as a result of auditing the table `emp`, enter the following statement:

```
DELETE FROM SYS.AUD$
     WHERE obj$name='EMP';
```

> **Note:** All deletes from the audit trail are audited without exception. Refer to "Auditing the Standard Audit Trail" on page 12-18 and "Auditing Administrative Users" on page 12-3.

Only the user SYS, a user who has the DELETE ANY TABLE privilege, or a user to whom SYS has granted DELETE privilege on SYS.AUD$ can delete records from the database audit trail.

> **Note:** If the audit trail is full and connections are being audited (that is, if the SESSION option is set), then typical users cannot connect to the database because the associated audit record for the connection cannot be inserted into the audit trail. In this case, the security administrator must connect as SYS (operations by SYS are not audited) and make space available in the audit trail.

Archiving Audit Trail Information

If audit trail information must be archived for historical purposes, then the security administrator can copy the relevant records to a normal database table (for example, using INSERT INTO *table* SELECT ... FROM SYS.AUD$ ...) or export the audit trail table to an operating system file.

> **See Also:** *Oracle Database Utilities* for information about exporting tables

Reducing the Size of the Audit Trail

As with any database table, after records are deleted from the database audit trail, the extents allocated for this table still exist.

If the database audit trail has many extents allocated for it, but many of them are not being used, then the space allocated to the database audit trail can be reduced by following these steps:

1. If you want to save information currently in the audit trail, then copy it to another database table or export it by using the EXPORT utility.
2. Connect as a user with administrator privileges.
3. Truncate SYS.AUD$ using the TRUNCATE statement.
4. Reload archived audit trail records generated in Step 1.

The new version of SYS.AUD$ is allocated only as many extents as are necessary to contain current audit trail records.

> **Note:** SYS.AUD$ is the only SYS object that should ever be directly modified.

Protecting the Standard Audit Trail

When auditing for suspicious database activity, protect the integrity of the audit trail's records to guarantee the accuracy and completeness of the auditing information.

Audit records generated as a result of object audit options set for the SYS.AUD$ table can only be deleted from the audit trail by someone connected with administrator privileges, which itself has protection against unauthorized use.

Auditing the Standard Audit Trail

If an application needs to give SYS.AUD$ access to regular users (non-SYSDBA users), then such access needs to be audited.

To do so, you turn on the relevant auditing options for SYS.AUD$, which work a little differently because they are auditing actions on the audit trail(aud$) itself:

1. CONNECT sys/*passw* AS SYSDBA
2. Issue the following command:

```
AUDIT SELECT, INSERT, UPDATE, DELETE ON sys.aud$ BY ACCESS;
```

Please note that this command will AUDIT actions performed by non-SYSDBA users only.

If a regular user has SELECT, UPDATE, INSERT, and DELETE privileges on SYS.AUD$ and executes a SELECT operation, then the audit trail will have a record of that operation. That is, SYS.AUD$ will have a row identifying the SELECT action on itself, as say row1.

If a user later tries to DELETE this row1 from SYS.AUD$, then the DELETE will succeed, because the user has the privilege to perform this action. However, this DELETE action on SYS.AUD$ is also recorded in the audit trail. Setting up this type of auditing acts as a safety feature, potentially revealing unusual or unauthorized actions. A logfile for an illustrative test case appears at the end of this chapter, at The SYS.AUD$ Auditing Table: Example.

Note: DELETE, INSERT, UPDATE, and MERGE operations on SYS.AUD$ table are always audited, and such audit records are not allowed to be deleted.

Viewing Database Audit Trail Information

The database audit trail (SYS.AUD$) is a single table in each Oracle database data dictionary. Several predefined views are available to present auditing information from this table in a meaningful way. If you decide not to use auditing, then you can later delete these views. The following subsections show you what is in these views, how to use them, and how to delete them:

Audit Trail Views

The following views are created upon installation:

View	Description
STMT_AUDIT_OPTION_MAP	Contains information about auditing option type codes. Created by the SQL.BSQ script at CREATE DATABASE time.
AUDIT_ACTIONS	Contains descriptions for audit trail action type codes.
ALL_DEF_AUDIT_OPTS	Contains default object-auditing options that will be applied when objects are created.
DBA_STMT_AUDIT_OPTS	Describes current system auditing options across the system and by user.
DBA_PRIV_AUDIT_OPTS	Describes current system privileges being audited across the system and by user.

View	Description
DBA_OBJ_AUDIT_OPTS USER_OBJ_AUDIT_OPTS	Describes auditing options on all objects. The USER view describes auditing options on all objects owned by the current user.
DBA_AUDIT_TRAIL USER_AUDIT_TRAIL	Lists all audit trail entries. The USER view shows audit trail entries relating to current user.
DBA_AUDIT_OBJECT USER_AUDIT_OBJECT	Contains audit trail records for all objects in the system. The USER view lists audit trail records for statements concerning objects that are accessible to the current user.
DBA_AUDIT_SESSION USER_AUDIT_SESSION	Lists all audit trail records concerning CONNECT and DISCONNECT. The USER view lists all audit trail records concerning connections and disconnections for the current user.
DBA_AUDIT_STATEMENT USER_AUDIT_STATEMENT	Lists audit trail records concerning GRANT, REVOKE, AUDIT, NOAUDIT, and ALTER SYSTEM statements throughout the database, or for the USER view, issued by the user.
DBA_AUDIT_EXISTS	Lists audit trail entries produced BY AUDIT NOT EXISTS.
DBA_AUDIT_POLICIES	Shows all the auditing policies on the system.
DBA_FGA_AUDIT_TRAIL	Lists audit trail records for value-based auditing.
DBA_COMMON_AUDIT_TRAIL	Combines standard and fine-grained audit log records, and includes SYS and mandatory audit records written in XML format.

See Also: *Oracle Database Reference* for detailed descriptions of the predefined views in Oracle Database

Using Audit Trail Views to Investigate Suspicious Activities

This section offers examples that demonstrate how to examine and interpret the information in the audit trail. Consider the following situation.

You would like to audit the database for the following suspicious activities:

- Passwords, tablespace settings, and quotas for some database users are altered without authorization.
- A high number of deadlocks occur, most likely because of users acquiring exclusive table locks.
- Rows are arbitrarily deleted from the emp table in jeff's schema.

You suspect the users jward and swilliams of several of these detrimental actions.

To investigate, you issue the following statements (in the order specified):

```
AUDIT ALTER, INDEX, RENAME ON DEFAULT
    BY SESSION;
CREATE VIEW jeff.employee AS SELECT * FROM jeff.emp;
AUDIT SESSION BY jward, swilliams;
AUDIT ALTER USER;
AUDIT LOCK TABLE
    BY ACCESS
    WHENEVER SUCCESSFUL;
AUDIT DELETE ON jeff.emp
    BY ACCESS
    WHENEVER SUCCESSFUL;
```

The following statements are subsequently issued by the user jward:

```
ALTER USER tsmith QUOTA 0 ON users;
```

```
DROP USER djones;
```

The following statements are subsequently issued by the user `swilliams`:

```
LOCK TABLE jeff.emp IN EXCLUSIVE MODE;
DELETE FROM jeff.emp WHERE mgr = 7698;
ALTER TABLE jeff.emp ALLOCATE EXTENT (SIZE 100K);
CREATE INDEX jeff.ename_index ON jeff.emp (ename);
CREATE PROCEDURE jeff.fire_employee (empid NUMBER) AS
  BEGIN
    DELETE FROM jeff.emp WHERE empno = empid;
  END;
/

EXECUTE jeff.fire_employee(7902);
```

The following sections display the information relevant to your investigation that can be viewed using the audit trail views in the data dictionary:

- Listing Active Statement Audit Options
- Listing Active Privilege Audit Options
- Listing Active Object Audit Options for Specific Objects
- Listing Default Object Audit Options
- Listing Audit Records
- Listing Audit Records for the AUDIT SESSION Option

Listing Active Statement Audit Options

The following query returns all the statement audit options that are set:

```
SELECT * FROM DBA_STMT_AUDIT_OPTS;

USER_NAME              AUDIT_OPTION         SUCCESS         FAILURE
---------------------  -------------------  ----------      ---------
JWARD                  SESSION              BY SESSION      BY SESSION
SWILLIAMS              SESSION              BY SESSION      BY SESSION
                       LOCK TABLE           BY ACCESS       NOT SET
```

Notice that the view reveals the statement audit options set, whether they are set for success or failure (or both), and whether they are set for `BY SESSION` or `BY ACCESS`.

Listing Active Privilege Audit Options

The following query returns all the privilege audit options that are set:

```
SELECT * FROM DBA_PRIV_AUDIT_OPTS;

USER_NAME           PRIVILEGE            SUCCESS      FAILURE
------------------- -------------------- ---------   ----------
ALTER USER          BY SESSION           BY SESSION
```

Listing Active Object Audit Options for Specific Objects

The following query returns all audit options set for any objects with names that start with the characters `emp` and that are contained in `jeff`'s schema:

```
SELECT * FROM DBA_OBJ_AUDIT_OPTS
    WHERE OWNER = 'JEFF' AND OBJECT_NAME LIKE 'EMP%';
```

```
OWNER OBJECT_NAME OBJECT_TY ALT AUD COM DEL GRA IND INS LOC ...
----- ----------- --------- --- --- --- --- --- --- --- --- ...
JEFF EMP         TABLE     S/S -/- -/- A/- -/- S/S -/- -/- ...
JEFF EMPLOYEE    VIEW      -/- -/- -/- A/- -/- S/S -/- -/- ...
```

Notice that the view returns information about all the audit options for the specified object. The information in the view is interpreted as follows:

- A dash (-) indicates that the audit option is not set.
- The S character indicates that the audit option is set, BY SESSION.
- The A character indicates that the audit option is set, BY ACCESS.
- Each audit option has two possible settings, WHENEVER SUCCESSFUL and WHENEVER NOT SUCCESSFUL, separated by a slash (/). For example, the DELETE audit option for jeff.emp is set BY ACCESS for successful delete statements and not set at all for unsuccessful delete statements.

Listing Default Object Audit Options

The following query returns all default object audit options:

```
SELECT * FROM ALL_DEF_AUDIT_OPTS;

ALT AUD COM DEL GRA IND INS LOC REN SEL UPD REF EXE FBK REA
--- --- --- --- --- --- --- --- --- --- --- --- --- --- ---
S/S -/- -/- -/- -/- S/S -/- -/- S/S -/- -/- -/- -/- /-  -/-
```

Notice that the view returns information similar to the USER_OBJ_AUDIT_OPTS and DBA_OBJ_AUDIT_OPTS views (refer to previous example).

Listing Audit Records

The following query lists audit records generated by statement and object audit options:

```
SELECT * FROM DBA_AUDIT_OBJECT;
```

Listing Audit Records for the AUDIT SESSION Option

The following query lists audit information corresponding to the AUDIT SESSION statement audit option:

```
SELECT USERNAME, LOGOFF_TIME, LOGOFF_LREAD, LOGOFF_PREAD,
    LOGOFF_LWRITE, LOGOFF_DLOCK
    FROM DBA_AUDIT_SESSION;

USERNAME   LOGOFF_TI LOGOFF_LRE LOGOFF_PRE LOGOFF_LWR LOGOFF_DLO
---------- --------- ---------- ---------- ---------- ----------
JWARD      02-AUG-91         53          2         24          0
SWILLIAMS  02-AUG-91       3337        256        630          0
```

Deleting the Audit Trail Views

If you disable auditing and no longer need the audit trail views, then delete them by connecting to the database as SYS and running the script file CATNOAUD.SQL. The name and location of the CATNOAUD.SQL script are operating-system-dependent.

The SYS.AUD$ Auditing Table: Example

The code in this section illustrates the auditing of changes made to `SYS.AUD$`.

```
SQL> @t
SQL>
SQL> SET FEEDBACK 1
SQL> SET NUMWIDTH 10
SQL> SET LINESIZE 80
SQL> SET TRIMSPOOL ON
SQL> SET TAB OFF
SQL> SET PAGESIZE 100
SQL>
SQL> column username format a10
SQL> column owner  format a10
SQL> column obj_name format a6
SQL> column action_name format a17
SQL> SET ECHO ON
SQL>
SQL> connect sys/newdbapassword as sysdba
Connected.
SQL> grant select, insert, update, delete on sys.aud$ to jeff;

Grant succeeded.

SQL> grant select on dba_audit_trail to jeff;

Grant succeeded.

SQL> audit select, update, delete on sys.aud$ by access;

Audit succeeded.

SQL> truncate table sys.aud$;

Table truncated.

SQL>
SQL> connect jeff/wolf
Connected.
SQL> select count(*) from emp

  COUNT(*)
----------
         0

1 row selected.

SQL>
SQL> select statementid,entryid,username,action_name,returncode,owner,
  2  obj_name,substr(priv_used,1,8) priv,  SES_ACTIONS
  3  from dba_audit_trail
  4  order by sessionid,entryid;

STATEMENTID    ENTRYID USERNAME   ACTION_NAME       RETURNCODE OWNER      OBJ_NA
----------- ---------- ---------- ----------------- ---------- ---------- ------
PRIV     SES_ACTIONS
-------- -------------------
          8          1 JEFF      SELECT                     0 SYS        AUD$
```

```
1 row selected.

SQL>
SQL> update sys.aud$ set userid = 0;

2 rows updated.

SQL> select statementid,entryid,username,action_name,returncode,owner,
  2  obj_name,substr(priv_used,1,8) priv,  SES_ACTIONS
  3  from dba_audit_trail
  4  order by sessionid,entryid;

STATEMENTID    ENTRYID USERNAME   ACTION_NAME       RETURNCODE OWNER      OBJ_NA
---------- ---------- ---------- ---------------- ---------- ---------- ------
PRIV     SES_ACTIONS
-------- -------------------
          8          1 0          SELECT                     0 SYS        AUD$

          9          2 0          SELECT                     0 SYS        AUD$

         10          3 JEFF       UPDATE                     0 SYS        AUD$

3 rows selected.

SQL>
SQL> delete from sys.aud$;

3 rows deleted.

SQL> select statementid,entryid,username,action_name,returncode,owner,
  2  obj_name,substr(priv_used,1,8) priv,  SES_ACTIONS
  3  from dba_audit_trail
  4  order by sessionid,entryid;

STATEMENTID    ENTRYID USERNAME   ACTION_NAME       RETURNCODE OWNER      OBJ_NA
---------- ---------- ---------- ---------------- ---------- ---------- ------
PRIV     SES_ACTIONS
-------- -------------------
         10          3 JEFF      UPDATE                     0 SYS        AUD$

         12          5 JEFF      DELETE                     0 SYS        AUD$

2 rows selected.

SQL>
SQL> connect sys/newdbapassword as sysdba
Connected.
SQL> noaudit insert, select, update, delete on sys.aud$;

Noaudit succeeded.

SQL>
SQL> spool off
```

Fine-Grained Auditing

As described earlier in this chapter and in Chapter 8, standard Oracle auditing is highly configurable. Its audit trail provides a fixed set of facts that monitor privileges,

object access, or (optionally) SQL usage, including information about the environment or query results. The scope of standard auditing can also be substantially expanded by using triggers, and providing additional customized information.

However, two auditing goals are not directly addressed by any mechanism in standard auditing: minimizing unhelpful audits, and proving that access rights were violated. Access logs, while helpful in reconstructing events, can often be inconclusive.

Fine-grained auditing addresses these needs, taking you beyond standard auditing and enabling you to minimize false or unhelpful audits by specifying more detailed audit conditions. You do not need to set `AUDIT_TRAIL` to enable fine-grained auditing. You simply add and remove FGA policies as you see fit, applying them to the specific operations or objects you want to monitor. A built-in audit mechanism in the database prevents users from bypassing the audit. Fine-grained auditing records are stored in `SYS.FGA_LOG$` table and are accessible through the `DBA_FGA_AUDIT_TRAIL` view.

Note: The `DBA_COMMON_AUDIT_TRAIL` view combines standard and fine-grained audit log records.

See Also: To add, drop, enable, or disable policies, use The DBMS_FGA Package

Policies in Fine-Grained Auditing

Policies you establish with fine-grained auditing can monitor data access based on content. Using policies, you can specify the columns and conditions that you want audit records for. Conditions can include limiting the audit to specific types of DML statements used in connection with the columns that you specify. You can also provide the name of the routine you want called when an audit event occurs. This routine can notify or alert administrators or handle errors and anomalies.

For example, most companies logically want to limit access to the specifications or test results for a product under development, and prefer that salary information remain private. Auditors need enough detail to be able to determine the data that was accessed.

Knowing only that `SELECT` privilege was used by a specific user on a particular table is not specific enough to provide accountability. A central tax authority has similar privacy concerns, needing to track access to tax returns so that employees do not snoop. Similarly, government agencies that use informants need detailed tracking of access to the database containing their identities. Such agencies also need enough detail to determine the data that was accessed, and not just information that the `SELECT` privilege was used by `JEFF` on the `TAXPAYERS` or `INFORMANTS` table.

Advantages of Fine-Grained Auditing over Triggers

Fine-grained auditing meets these needs by providing functionality (and efficiency) beyond triggers. Triggers incur a PL/SQL process call for every row processed and create an audit record only when a relevant column is changed by a DML statement.

A fine-grained auditing policy, on the other hand, does not incur this cost for every row. Instead, it audits only once for every policy. Specifically, it audits when a specified relevant column occurs in a specified type of DML statement, either being changed by the statement or being in its selection criteria. This combination of criteria uncovers users who hope their information gathering will be masked because they only use the selection criteria of a DML statement. Triggers also cannot monitor the

activity of another `instead-of` trigger on the same object, while fine-grained auditing supports tables and views.

Extensible Interface Using Event Handler Functions

Organizations can define fine-grained auditing policies to specify the data access conditions that are to trigger audit events. These policies can use flexible event handlers that notify administrators when a triggering event has occurred. For example, an organization may allow HR clerks to access employee salary information, but trigger an audit event when salaries greater than $500K are accessed. The audit policy (where `SALARY` > 500000) is applied to the `EMPLOYEES` table through an audit policy interface (`DBMS_FGA`, a PL/SQL package).

The audit function (`handler_module`) is an alerting mechanism for the administrator. The required interface for such a function is as follows:

```
PROCEDURE fname ( object_schema VARCHAR2, object_name VARCHAR2, policy_name
VARCHAR2 )  AS ...
```

Where:

- *`fname`* is the name of the procedure
- *`object_schema`* is the name of the schema of the table audited
- *`object_name`* is the name of the table to be audited
- *`policy_name`* is the name of the policy being enforced

See also:

- The DBMS_FGA Package on page 12-29
- *PL/SQL Packages and Types Reference*

Functions and Relevant Columns in Fine-Grained Auditing

For additional flexibility in implementation, organizations can employ a user-defined function to determine the policy condition and identify an audit column (called a *relevant column*) to further refine the audit policy. For example, the function could cause an audit record only when a salary greater than $250,000 is accessed.

Specifying a relevant column helps reduce the instances of false or unnecessary audit records, because the audit need only be triggered when a particular column is referenced in the query. For example, an organization may only wish to audit executive salary access when an employee name is accessed, because accessing salary information alone is not meaningful unless an HR clerk also selects the corresponding employee name. You can, however, specify that auditing occur only when all relevant columns are referenced.

If more than one relevant audit column is specified, then Oracle Database produces an audit record if the SQL statement references any of those audit columns.

The `DBMS_FGA` package administers these value-based audit policies. The security administrator creates an audit policy on the target object using the functions in the `DBMS_FGA` package.

See also:

- The DBMS_FGA Package (the next major section)
- *PL/SQL Packages and Types Reference*

Audit Records in Fine-Grained Auditing

If any rows returned from a query block match the audit condition, then an audit event entry is inserted into the fine-grained audit trail. This entry includes user name, SQL text, bind variable, policy name, session ID, time stamp, and other attributes. Only one row of audit information is inserted into the audit trail for every FGA policy that evaluates to true. As part of the extensibility framework, administrators can also optionally define an appropriate audit event handler to process the event, for example sending an alert page to the administrator.

NULL Audit Conditions

To guarantee auditing of the specified actions (`statement_types`) affecting the specified columns (`audit_column`), specify the audit_condition as `NULL` (or omit it), which is interpreted as `TRUE`. Only specifying `NULL` will guarantee auditing of the specified actions (`statement_types`) affecting the specified columns (`audit_column`). The former practice of specifying an audit condition of `1=1` to force such auditing should no longer be used and will not reliably achieve the desired result. `NULL` will cause audit even if no rows were processed, so that all actions on an `audit_column` with this policy are audited.

Note: Using an empty string is not equivalent to `NULL` and will not reliably cause auditing of all actions on a table with this policy.

The audit function is executed as an autonomous transaction, committing only the actions of the `handler_module` and not any user transaction. This function has no effect on any user SQL transaction.

If `NULL` or no audit condition is specified, then any action on a table with that policy causes an audit record to be created, whether or not rows are returned.

Defining FGA Policies

The administrator uses the `DBMS_FGA.ADD_POLICY` interface to define each FGA policy for a table or view, identifying any combination of `SELECT`, `UPDATE`, `DELETE`, or `INSERT` statements. Oracle supports `MERGE` statements as well, by auditing the underlying actions of `INSERT` and `UPDATE`. To audit `MERGE`s, set up FGA on these `INSERT`s and `UPDATE`s. Only one record is generated for each policy for successful `MERGE`s.

FGA policies associated with a table or view may also specify relevant columns, so that any specified statement type affecting a particular column is audited. More than one column can be included as relevant columns in a single FGA policy. Examples include privacy-relevant columns, such as those containing social security numbers, salaries, patient diagnoses, and so on. If no relevant column is specified, then auditing applies to all columns. That is, auditing occurs whenever any specified statement type affects any column, unless you specify in the policy that auditing is to occur only when all relevant columns are referenced.

An Added Benefit to Fine-Grained Auditing

In general, fine-grained auditing policies are based on simple user-defined SQL predicates on table objects as conditions for selective auditing. During fetching, whenever policy conditions are met for a returning row, the query is audited. Later, Oracle Database can run a user-defined audit event handler, if specified in the policy, using autonomous transactions to process the event.

Fine-grained auditing can be implemented in user applications using the DBMS_FGA package or by using database triggers.

The following example shows how you can audit statements (INSERT, UPDATE, DELETE, and SELECT) on table hr.emp to monitor any query that accesses the salary column of the employee records that belong to sales department:

```
DBMS_FGA.ADD_POLICY(
object_schema => 'hr',
object_name   => 'emp',
policy_name   => 'chk_hr_emp',
audit_condition => 'dept = ''SALES'' ',
audit_column => 'salary'
statement_types => 'insert,update,delete,select');
```

Then, any of the following SQL statements will cause the database to log an audit event record.

```
SELECT count(*) FROM hr.emp WHERE dept = 'SALES' and salary > 10000000;

SELECT salary FROM hr.emp WHERE dept = 'SALES';

DELETE from hr.emp where salary >1000000
```

With all the relevant information available, and a trigger-like mechanism to use, the administrator can define what to record and how to process the audit event.

Consider the following commands:

```
/* create audit event handler */
CREATE PROCEDURE sec.log_id (schema1 varchar2, table1 varchar2, policy1 varchar2)
AS
BEGIN
UTIL_ALERT_PAGER(schema1, table1, policy1);      -- send an alert note to my pager
END;

/* add the policy */
DBMS_FGA.ADD_POLICY(
object_schema => 'hr',
object_name   => 'emp',
policy_name   => 'chk_hr_emp',
audit_condition => 'dept = ''SALES'' ',
audit_column => 'salary',
handler_schema => 'sec',
handler_module => 'log_id',
enable               =>  TRUE);
```

Note: Because schema and table are reserved words, they cannot be used as variables without some alteration, such as appending 1 as is done here.

After the first row of interest is fetched, the event is recorded, and the SEC.LOG_ID audit function is run. The audit event record generated is stored in DBA_FGA_AUDIT_TRAIL, which is fga_log$ in the SYS schema in the SYSTEM tablespace. This table has reserved columns (such as SQL_TEXT and SQL_BIND) for recording SQL text, policy name, and other information. The SQLBIND and SQLTEXT values are recorded in the LSQLTEXT and LSQLBIND columns of fga_log$ only if the policy specifies audit_trail = DBMS_FGA.DB + DBMS_FGA.EXTENDED. If

the policy specifies AUDIT_TRAIL=DBMS_FGA.XML, then the audit records would be written to XML-formatted OS files.

Note:

- Fine-grained auditing is supported only with cost-based optimization. For queries using rule-based optimization, audit will check before applying row filtering, which could result in an unnecessary audit event trigger.
- Policies currently in force on an object involved in a flashback query are applied to the data returned from the specified flashback snapshot (based on time or SCN).

See Also:

- *Oracle Database Application Developer's Guide - Fundamentals* for information about using fine-grained auditing
- The DBMS_FGA chapter in *PL/SQL Packages and Types Reference*

The DBMS_FGA Package

The DBMS_FGA package provides fine-grained security functions. The execute privilege on DBMS_FGA is needed for administering audit policies. Because the audit function can potentially capture all user environment and application context values, policy administration should be executable by privileged users only.

This feature is available only for cost-based optimization. The rule-based optimizer may generate unnecessary audit records because audit monitoring can occur before row filtering. For both the rule-based optimizer and the cost-based optimizer, you can refer to DBA_FGA_AUDIT_TRAIL to analyze the SQL text and corresponding bind variables that are issued.

The procedures for this package are described in the following subsections:

- ADD_POLICY Procedure
- DISABLE_POLICY Procedure
- DROP_POLICY Procedure
- ENABLE_POLICY Procedure

The syntax, parameters, and usage notes accompanying each procedure description also discuss the defaults and restrictions that apply to it.

ADD_POLICY Procedure

This procedure creates an audit policy using the supplied predicate as the audit condition. The maximum number of FGA policies on any table or view object is 256.

Syntax

```
DBMS_FGA.ADD_POLICY(
   object_schema   VARCHAR2,
   object_name     VARCHAR2,
   policy_name     VARCHAR2,
   audit_condition VARCHAR2,
   audit_column    VARCHAR2,
```

```
handler_schema  VARCHAR2,
handler_module  VARCHAR2,
enable          BOOLEAN,
statement_types VARCHAR2,
audit_trail     BINARY_INTEGER IN DEFAULT,
audit_column_opts BINARY_INTEGER IN DEFAULT);
```

Parameters

Table 12–3 ADD_POLICY Procedure Parameters

Parameter	Description	Default Value
object_schema	The schema of the object to be audited. (If NULL, then the current login user schema is assumed.)	NULL
object_name	The name of the object to be audited.	-
policy_name	The unique name of the policy.	-
audit_condition	A condition in a row that indicates a monitoring condition. NULL is allowed and acts as TRUE.	NULL
audit_column	The columns to be checked for access. These can include hidden columns. The default, NULL, causes audit if any column is accessed or affected.	NULL
handler_schema	The schema that contains the event handler. The default, NULL, causes the current schema to be used.	NULL
handler_module	The function name of the event handler includes the package name if necessary. This function is called only after the first row that matches the audit condition in the query is processed. If the procedure fails with an exception, then the user SQL statement will fail as well.	NULL
enable	Whether the policy is to be enabled: TRUE means enable it.	TRUE
statement_types	The SQL statement types to which this policy is applicable: INSERT, UPDATE, DELETE, or SELECT only.	SELECT
audit_trail	Both where to write the fine-grained audit trail and whether or not to populate LSQLTEXT and LSQLBIND.	DB+EXTENDED
audit_column_opts	Whether a statement is audited when the query references *any* column specified in the audit_column parameter or only when *all* such columns are referenced.	ANY_COLUMNS

Usage Notes

Usage notes are described as follows:

- Sample command:

```
DBMS_FGA.ADD_POLICY(object_schema => 'scott', object_name=>'emp', policy_name
=> 'mypolicy1', audit_condition => 'sal < 100', audit_column =>'comm,
credit_card, expirn_date', handler_schema => NULL, handler_module => NULL,
enable => TRUE, statement_types=> 'INSERT, UPDATE', audit_trail =>
DBMS_FGA.DB+DBMS_FGA.EXTENDED, audit_column_opts => DBMS_FGA.ALL_COLUMNS);
```

- If object_schema is not specified, then the current login user schema is assumed.
- An FGA policy should not be applied to out-of-line columns such as LOB columns.

- Each audit policy is applied to the query individually. However, at most one audit record may be generated for each policy, no matter how many returned rows satisfy the audit_condition of the policy. In other words, whenever the number of rows returned satisfy an audit condition defined on the table, a single audit record is generated for each such policy.
- If a table with an FGA policy defined on it receives a Fast Path insert or a vectored update, then the hint is automatically disabled before any such operations. Disabling the hint allows auditing to occur according to the policy terms. (One example of a Fast Path insert is the statement INSERT-WITH-APPEND-hint.)
- The audit_condition must be a Boolean expression that can be evaluated using the values in the row being inserted, updated, or deleted. This condition can be NULL (or omitted), which is interpreted as TRUE, but it cannot contain the following elements:
  - Subqueries or sequences
  - Any direct use of SYSDATE, UID, USER or USERENV functions. However, a user-defined function and other SQL functions can use these functions to return the desired information.
  - Any use of the pseudocolumns LEVEL, PRIOR, or ROWNUM.

  Specifying an audit condition of 1=1 to force auditing of all specified statements (statement_types) affecting the specified column (audit_column) is no longer needed to achieve this purpose. NULL will cause audits even if no rows were processed, so that all actions on a table with this policy are audited.
- The audit function (handler_module) is an alerting mechanism for the administrator. The required interface for such a function is as follows:

  ```
  PROCEDURE <fname> ( object_schema VARCHAR2, object_name VARCHAR2, policy_name
  VARCHAR2 )  AS ...
  ```

  where fname is the name of the procedure, object_schema is the name of the schema of the table audited, object_name is the name of the table to be audited, and policy_name is the name of the policy being enforced.
- The audit_trail parameter specifies both where the fine-grained audit trail will be written and whether the audit trail should include the associated SQL Text and SQL Bind variable information (typically in columns named LSQLTEXT and LSQLBIND):
  - If audit_trail includes XML, then fine-grained audit records are written to XML-format operating system files, irrespective of the init.ora AUDIT_TRAIL parameter. The XML-format operating system files are stored in the directory specified by an AUDIT_FILE_DEST init.ora parameter. (The default AUDIT_FILE_DEST is $ORACLE_BASE/admin/$DB_UNIQUE_NAME/adump on Unix-based systems, and $ORACLE_BASE\admin\$DB_UNIQUE_NAME\adump on Windows systems.)
  - If audit_trail includes DB instead, then the audit records are written to the SYS.FGA_LOG$ table in the database.
  - If audit_trail includes EXTENDED, then the SQL Text and SQL Bind variable information for the query are included in the audit trail.
  - For example:

* Setting audit_trail to DBMS_FGA.DB sends the audit trail to the SYS.FGA_LOG$ table in the database and omits SQL Text and SQL Bind.
* Setting audit_trail to DBMS_FGA.DB+EXTENDED sends the audit trail to the SYS.FGA_LOG$ table in the database and includes SQL Text and SQL Bind.
* Setting audit_trail to DBMS_FGA.XML writes the audit trail in XML files sent to the operating system and omits SQL Text and SQL Bind.
* Setting audit_trail to DBMS_FGA.XML+EXTENDED writes the audit trail in XML files sent to the operating system and includes SQL Text and SQL Bind.

Note: When the init.ora file contains AUDIT_TRAIL=XML (or XML, EXTENDED), audit records are written to XML-formatted operating system files.

The audit_trail parameter appears in the ALL_AUDIT_POLICIES view.

- You can change the AUDIT_FILE_DESTparameter using the following command:

```
ALTER SYSTEM SET AUDIT_FILE_DEST = '<New Directory>' DEFERRED
```

- On many platforms, XML audit files are named as follows:

  <process_name>_<processId>.xml

  For example, ora_2111.xml, or s002_11.xml. On Windows, the XML audit files are named <process_name>_<ThreadId>.xml (or <process_name>_ProcessId>.xml if the process is not running as a thread).

- The audit_column_opts parameter establishes whether a statement is audited in any of the following conditions:
  - When the query references *any* column specified in the audit_column parameter (audit_column_opts = DBMS_FGA.ANY_COLUMNS)
  - Only when *all* such columns are referenced (audit_column_opts = DBMS_FGA.ALL_COLUMNS).

  Note: When audit_column_opts is set to DBMS_FGA.ALL_COLUMNS, a SQL statement is audited only when all the columns mentioned in audit_column have been explicitly referenced in the statement. These columns must have been referenced in the same SQL statement or subquery. Also, all these columns must refer to a single table/view or alias. Thus if a SQL statement selects the columns from different table aliases, then the statement will not be audited.

  The default is DBMS_FGA.ANY_COLUMNS.

  The ALL_AUDIT_POLICIES view also shows audit_column_opts.

V$XML_AUDIT_TRAIL View

The new values for the audit_trail parameter (XML and XML, EXTENDED) cause fine-grained auditing records to be written to operating system files in XML format.

Audit records stored in operating system files can be more secure than database-stored audit records because access can require file permissions that DBAs do not have. Operating system storage for audit records also offers higher availability, because such records remain available even if the database is temporarily inaccessible.

A new dynamic view, V$XML_AUDIT_TRAIL, makes such audit records from XML files available to DBAs through a SQL query, providing enhanced usability. Querying this view causes all XML files (all files with an .xml extension) in the AUDIT_FILE_DEST directory to be parsed and presented in relational table format.

The DBA_COMMON_AUDIT_TRAIL view includes the contents of the V$XML_AUDIT_TRAIL dynamic view for standard and fine-grained audit records.

Because the audit XML files are stored in files with .xml extension on all platforms, the dynamic view presents audit information similarly on all platforms, using the following schema:

Table 12–4 Elements in the V$XML_AUDIT_TRAIL Dynamic View

Element	Type
AUDIT_TYPE (refer to Note 1.)	NUMBER
CLIENT_ID	VARCHAR2(64)
COMMENT_TEXT	VARCHAR2(4000)
DB_USER	VARCHAR2(30)
ENTRYID	NUMBER
EXT_NAME	VARCHAR2(4000)
EXTENDED_TIMESTAMP (Refer to Note 1.)	TIMESTAMP(6) WITH TIME ZONE
GLOBAL_UID	VARCHAR2(32)
INSTANCE_NUMBER	NUMBER
OBJECT_NAME	VARCHAR2(128)
OBJECT_SCHEMA	VARCHAR2(30)
OS_PROCESS (Refer to Note 3.)	VARCHAR2(16)
OS_USER	VARCHAR2(255)
POLICY_NAME	VARCHAR2(30)
PROXY_SESSIONID	NUMBER (Refer to Note 3.)
SCN	NUMBER
SESSION_ID	NUMBER
SQL_BIND (Refer to Note 2.)	VARCHAR2(4000)
SQL_TEXT (Refer to Note 2.)	VARCHAR2(4000)
STATEMENT_TYPE	VARCHAR2(28)
STATEMENTID	NUMBER
TERMINAL	VARCHAR2(255)
TRANSACTIONID	RAW(8)
USERHOST	VARCHAR2(128)

Note 1: The AUDIT_TYPE column shows 1 for standard XML audit, 2 for fine-grained XML audit, 4 for SYS XML audit, and 8 for mandatory XML audit. Every XML audit record contains the elements Audit_Type and Extended_Timestamp, with the latter printed in UTC zone (with no time zone information). Values retrieved using V$XML_AUDIT_TRAIL view are converted to session time zone and printed.

Note 2: For SQL_TEXT and SQL_BIND element values (CLOB type columns), the dynamic view shows only the first 4000 characters. The underlying XML file may have more than 4000 characters for such SQL_TEXT and SQL_BIND values.

Note 3: OS_PROCESS on Unix systems. In Windows systems, the label is ProcessId:ThreadId (or ProcessId, if it is not running as a thread).

Note 4: For a large numbers of XML audit files, querying V$XML_AUDIT_TRAIL is faster when they are loaded into a database table using SQL*Loader or a similar tool. XML audit files are larger than the equivalent written to OS files when AUDIT_TRAIL=OS.

Error handling is the same as when AUDIT_TRAIL=OS. If any error occurs in writing an audit record to disk, including the directory identified by AUDIT_FILE_DEST being full, then the auditing operation fails. An alert message is logged.

Examples

```
DBMS_FGA.ADD_POLICY (object_schema => 'scott', object_name=>'emp', policy_name =>
'mypolicy1', audit_condition => 'sal < 100', audit_column =>'comm, credit_card,
expirn_date', handler_schema => NULL, handler_module => NULL, enable => TRUE,
statement_types=> 'INSERT, UPDATE', audit_trail => DBMS_FGA.DB+DBMS_FGA.EXTENDED,
audit_column_opts => DBMS_FGA.ALL_COLUMNS);
```

DISABLE_POLICY Procedure

This procedure disables an audit policy.

Syntax

```
DBMS_FGA.DISABLE_POLICY(
   object_schema  VARCHAR2,
   object_name    VARCHAR2,
   policy_name    VARCHAR2 );
```

Parameters

Table 12–5 DISABLE_POLICY Procedure Parameters

Parameter	Description
object_schema	The schema of the object to be audited (If NULL, then the current login user schema is assumed.)
object_name	The name of the object to be audited
policy_name	The unique name of the policy

The default value for object_schema is NULL. (If NULL, then the current login user schema is assumed.)

DROP_POLICY Procedure

This procedure drops an audit policy.

Syntax

```
DBMS_FGA.DROP_POLICY(
   object_schema  VARCHAR2,
   object_name    VARCHAR2,
   policy_name    VARCHAR2 );
```

Parameters

Table 12–6 DROP_POLICY Procedure Parameters

Parameter	Description
object_schema	The schema of the object to be audited. (If NULL, then the current log-on user schema is assumed.)
object_name	The name of the object to be audited.
policy_name	The unique name of the policy.

Usage Notes

The DBMS_FGA procedures cause current DML transactions, if any, to commit before the operation unless they are inside a DDL event trigger. With DDL transactions, the DBMS_FGA procedures are part of the DDL transaction. The default value for object_schema is NULL. (If NULL, then the current login user schema is assumed.)

ENABLE_POLICY Procedure

This procedure enables an audit policy.

Syntax

```
DBMS_FGA.ENABLE_POLICY(
   object_schema  VARCHAR2,
   object_name    VARCHAR2,
   policy_name    VARCHAR2,
   enable         BOOLEAN);
```

Parameters

Table 12–7 ENABLE_POLICY Procedure Parameters

Parameter	Description
object_schema	The schema of the object to be audited. (If NULL, then the current log-on user schema is assumed.)
object_name	The name of the object to be audited.
policy_name	The unique name of the policy.
enable	Defaults to TRUE to enable the policy.

13

Introducing Database Security for Application Developers

Creating an application security policy is the first step when writing secure database applications. An application security policy is a list of application security requirements and rules that regulate user access to database objects.

This chapter discusses aspects of application security and Oracle Database features that you should consider when drafting security policies for database applications. It contains the following topics:

About Application Security Policies

You should draft security policies for each database application. For example, each database application should have one or more database roles that provide different levels of security when executing the application. The database roles can be granted to user roles or directly to specific user names.

Applications that potentially allow unrestricted SQL statement execution (through tools such as SQL*Plus) also need security policies that prevent malicious access to confidential or important schema objects.

The following sections describe aspects of application security and the Oracle Database features that you can use to plan and develop secure database applications.

See Also:

- Chapter 7, "Security Policies" for an overview of database security policies
- "Application Developer Security" on page 7-7 for a discussion of application developer database privileges
- "Application Administrator Security" on page 7-9 for a description of the security-related tasks of an application administrator

Considerations for Using Application-Based Security

Two main issues to consider when you formulate and implement application security are covered in the following sections:

- Are Application Users Also Database Users?
- Is Security Enforced in the Application or in the Database?

Are Application Users Also Database Users?

Oracle recommends that, where possible, you build applications in which application users are database users. In this way, you can leverage the intrinsic security mechanisms of the database.

For many commercial packaged applications, application users are not database users. For these applications, multiple users authenticate themselves to the application, and the application then connects to the database as a single, highly-privileged user. We will call this the *One Big Application User* model.

Applications built in this fashion generally cannot use many of the intrinsic security features of the database, because the identity of the user is not known to the database.

For example, use of the following features is compromised by the One Big Application User model:

Oracle Feature	Limitations of One Big Application User Model
Auditing	A basic principle of security is accountability through auditing. If all actions in the database are performed by One Big Application User, then database auditing cannot hold individual users accountable for their actions. The application must implement its own auditing mechanisms to capture individual user actions.
Oracle Advanced Security enhanced authentication	Strong forms of authentication supported by Oracle Advanced Security (such as client authentication over SSL, tokens, and so on) cannot be used if the client authenticating to the database is the application, rather than an individual user.
Roles	Roles are assigned to database users. Enterprise roles are assigned to enterprise users who, though not created in the database, are known to the database. If application users are not database users, then the usefulness of roles is diminished. Applications must then craft their own mechanisms to distinguish between the privileges which various application users need to access data within the application.

Oracle Feature	Limitations of One Big Application User Model
Enterprise user management feature of Oracle Advanced Security	This feature enables an Oracle database to use the Oracle Identity Management Infrastructure by securely storing and managing user information and authorizations in an LDAP-based directory such as Oracle Internet Directory. While enterprise users do not need to be created in the database, they do need to be known to the database. The One Big Application User model cannot take advantage of Oracle Identity Management.

Is Security Enforced in the Application or in the Database?

Applications, whose users are also database users, can either build security into the application, or rely upon intrinsic database security mechanisms such as granular privileges, virtual private databases (fine-grained access control with application context), roles, stored procedures, and auditing (including fine-grained auditing). Oracle recommends that applications utilize the security enforcement mechanisms of the database as far as possible.

When security is enforced in the database itself, rather than in the application, it cannot be bypassed. The main shortcoming of application-based security is that security is bypassed if the user bypasses the application to access data. For example, a user who has SQL*Plus access to the database can execute queries without going through the Human Resources application. The user, therefore, bypasses all of the security measures in the application.

Applications that use the One Big Application User model must build security enforcement into the application rather than use database security mechanisms. Because it is the application, and not the database, that recognizes users, the application itself must enforce security measures for each user.

This approach means that each application that accesses data must reimplement security. Security becomes expensive, because organizations must implement the same security policies in multiple applications. Each new application requires an expensive reimplementation.

> **See Also:** "Use of Ad Hoc Tools: A Potential Security Problem" on page 14-15

Managing Application Privileges

Most database applications involve different privileges on different schema objects. Keeping track of which privileges are required for each application can be complex. In addition, authorizing users to run an application can involve many `GRANT` operations.

To simplify application privilege management, you can create a role for each application and grant that role all the privileges a user needs to run the application. In fact, an application might have a number of roles, each granted a specific subset of privileges that allow greater or lesser capabilities while running the application.

For example, suppose that every administrative assistant uses the Vacation application to record the vacation taken by members of the department. To best manage this application, you should:

1. Create a `VACATION` role.
2. Grant all privileges required by the Vacation application to the `VACATION` role.

3. Grant the VACATION role to all administrative assistants or to a role named ADMIN_ASSISTS (if previously defined).

Grouping application privileges in a role aids privilege management. Consider the following administrative options:

- You can grant the role, rather than many individual privileges, to those users who run the application. Then, as employees change jobs, you need to grant or revoke only one role, rather than many privileges.
- You can change the privileges associated with an application by modifying only the privileges granted to the role, rather than the privileges held by all users of the application.
- You can determine the privileges that are necessary to run a particular application by querying the ROLE_TAB_PRIVS and ROLE_SYS_PRIVS data dictionary views.
- You can determine which users have privileges on which applications by querying the DBA_ROLE_PRIVS data dictionary view.

> **See Also:** Chapter 11, "Administering User Privileges, Roles, and Profiles" for a complete discussion of creating, enabling, and disabling roles, and granting and revoking privileges

Creating Secure Application Roles

After database access privileges are grouped into roles, the roles are granted to the application user. Securing these roles can be accomplished in two ways:

- Embedding passwords inside the applications by creating what are called *application roles*
- Creating application roles and specifying which PL/SQL package is authorized to enable the roles, which are called *secure application roles*

  Within the package that implements the secure application role:

  - The application must do the necessary validation. For example, the application must validate that the user is in a particular department, the user session was created by proxy, the request comes from a particular IP address, or that the user was authenticated using an X.509 certificate. To perform the validation, applications can use session information accessible by using the SYS_CONTEXT SQL function with the USERENV namespace attributes ('userenv', <session_attribute>). The information returned by this function can indicate the way in which the user was authenticated, the IP address of the client, and whether the user was proxied.
  - The application must issue a SET_ROLE command by using dynamic SQL (DBMS_SESSION.SET ROLE).

Note: Because users cannot change the security domain inside **definer's rights procedures**, secure application roles can only be enabled inside **invoker's rights procedures**.

See Also:

- "Secure Application Roles" on page 5-20 for conceptual information about this topic
- Table 14–1, " Key to Predefined Attributes in USERENV Namespace" on page 14-8
- *PL/SQL User's Guide and Reference* for information about definer's rights versus invoker's rights procedures and how to create them

An Example of Creating a Secure Application Role

Steps to create a secure application role is as follows:

1. Create the roles as application roles and specify the authorized package that will enable the roles. In this example, `hr.hr_admin` is the example authorized package.

```
CREATE ROLE admin_role IDENTIFIED USING hr.hr_admin;
CREATE ROLE staff_role IDENTIFIED USING hr.hr_admin;
```

Note: You need to set up the following data structures for the examples in this section to work:

```
CREATE OR REPLACE PACKAGE hr_logon IS
PROCEDURE hr_set_responsibility;
END;
/

CREATE OR REPLACE PACKAGE BODY hr_logon IS
PROCEDURE hr_set_responsibility IS
   BEGIN
      DBMS_SESSION.SET_IDENTIFIER (1234);
   END;
END;
/
```

---

2. Create an invoker's right procedure.

```
/* Create a dedicated authentication function for manageability so that changes
in authentication policies would not affect the source code of the application
- this design is up to the application developers */
/* the only policy in this function is that current user must have been
authenticated using the proxy user 'SCOTT' */
CREATE OR REPLACE FUNCTION hr.MySecurityCheck RETURN BOOLEAN
AS
BEGIN
    /* a simple check to see if current session is authenticated
    by the proxy user 'SCOTT' */
    if (sys_context('userenv','proxy_user') = 'SCOTT')
    then
        return TRUE;
    else
        return FALSE;
    end IF;
END;
```

```
GRANT EXECUTE ON hr.MySecurityCheck TO PUBLIC;

/*Create the procedure*/
CREATE OR REPLACE PACKAGE hr_admin
AUTHID CURRENT_USER
IS
PROCEDURE hr_app_report;
END;
/
CREATE OR REPLACE PACKAGE BODY hr_admin IS
PROCEDURE hr_app_report IS
BEGIN
    /* set application context in 'responsibility' namespace */
    hr_logon.hr_set_responsibility;
    /* authentication check here */
    if (hr.MySecurityCheck = TRUE)
    then
         /* check 'responsibility' being set, then enable the roles without
         supplying the password */
         if (sys_context('hr','role') = 'admin' )
    then
         dbms_session.set_role('admin_role');
    else
         dbms_session.set_role('staff_role');
        end if;
    end if;
END;
END;
```

When enabling the secure application role, the database verifies that the authorized PL/SQL package is on the calling stack. This step verifies that the authorized PL/SQL package is issuing the command to enable the role. Also, when enabling the default user roles, no checking is performed for application roles.

You can use secure application roles to ensure a database connection. Because a secure application role is a role implemented by a package, the package can validate that users can connect to the database through a middle tier or from a specific IP address. In this way, the secure application role prevents users from accessing data outside an application. They are forced to work within the framework of the application privileges that they have been granted.

Associating Privileges with User Database Roles

A single user can use many applications and associated roles. However, you should ensure that the user has only the privileges associated with the current database role. Consider the following scenario:

- The `ORDER` role (for the Order application) contains the `UPDATE` privilege for the `INVENTORY` table
- The `INVENTORY` role (for the Inventory application) contains the `SELECT` privilege for the `INVENTORY` table
- Several order entry clerks have been granted both the `ORDER` and `INVENTORY` roles

In this scenario, an order entry clerk who has been granted both roles, can use the privileges of the `ORDER` role when running the `INVENTORY` application to update the `INVENTORY` table. The problem is that updating the `INVENTORY` table is not an

authorized action when using the INVENTORY application, but only when using the ORDER application.

To avoid such problems, consider using either the SET ROLE statement or the SET_ROLE procedure as explained in the following section. You can also use the secure application role feature to allow roles to be set based on criteria you define.

Topics in this section include:

- Using the SET ROLE Statement
- Using the SET_ROLE Procedure
- Examples of Assigning Roles with Static and Dynamic SQL

Using the SET ROLE Statement

Use a SET ROLE statement at the beginning of each application to automatically enable its associated role and to disable all others. In this way, each application dynamically enables particular privileges for a user only when required.

The SET ROLE statement simplifies privilege management. You control what information users can access and when they can access it. The SET ROLE statement also keeps users operating in a well-defined privilege domain. If a user obtains privileges only from roles, then the user cannot combine these privileges to perform unauthorized operations.

See Also:

- When Do Grants and Revokes Take Effect? on page 11-26 for information about enabling and disabling roles
- The SET ROLE Statement on page 11-26

Using the SET_ROLE Procedure

The PL/SQL package DBMS_SESSION.SET_ROLE is functionally equivalent to the SET ROLE statement in SQL. Roles are not supported in definer's rights procedures, so the DBMS_SESSION.SET_ROLE command cannot be called from them. However, the DBMS_SESSION.SET_ROLE command can be called from the following:

- Anonymous PL/SQL blocks
- Invoker's rights stored procedures (except those invoked from within definer's rights procedures)

SET ROLE takes effect only at execution time. Because anonymous blocks compile and execute simultaneously, roles are set before security checks are performed, so the block completes successfully. With respect to invoker's rights stored procedures, if they contain static SQL statements and access to objects in the SQL are authorized through roles, then the procedure may fail during compilation (Because the roles are not enabled until the procedure executes). To resolve this problem, replace static SQL with dynamic SQL by using the DBMS_SQL package. Then, security checks are performed at execution, while at the same time the SET ROLE statement enables roles.

Note: If you use DBMS_SESSION.SET_ROLE within an invoker's rights procedure, then the role remains in effect until you explicitly disable it. In keeping with the *least privilege* principle (that users should have the fewest privileges they need to do their jobs), you should explicitly disable roles set within an invoker's rights procedure, at the end of the procedure.

See Also: *PL/SQL Packages and Types Reference* for information about the DBMS_SESSION and the DBMS_SQL packages

Examples of Assigning Roles with Static and Dynamic SQL

This section shows how static and dynamic SQL affect the assignment of roles.

Note: You need to set up the following data structures for the examples in this section to work:

```
CONNECT system/manager
DROP USER joe CASCADE;
CREATE USER joe IDENTIFIED BY joe;
GRANT CREATE SESSION, RESOURCE, UNLIMITED TABLESPACE TO joe;
GRANT CREATE SESSION, RESOURCE, UNLIMITED TABLESPACE TO scott;
DROP ROLE acct;
CREATE ROLE acct;
GRANT acct TO scott;
ALTER USER scott DEFAULT ROLE ALL EXCEPT acct;

CONNECT joe/joe;
CREATE TABLE finance (empno NUMBER);
GRANT SELECT ON finance TO acct;
CONNECT scott/tiger
```

Suppose you have a role named ACCT that has been granted privileges allowing you to select from table FINANCE in the JOE schema. In this case, the following procedure that uses static SQL fails:

```
CREATE OR REPLACE PROCEDURE statSQL_proc
AUTHID CURRENT_USER AS
    n NUMBER;
BEGIN
    SYS.DBMS_SESSION.SET_ROLE('acct');
    SELECT empno INTO n FROM JOE.FINANCE;
END;
```

The procedure fails because the security check that verifies that you have the SELECT privilege on table JOE.FINANCE occurs at compile time. At compile time, however, the ACCT role is not yet enabled. The role is not enabled until the procedure is executed.

In contrast, the DBMS_SQL package, which uses dynamic SQL, is not subject to this restriction. When you use this package, the security checks are performed when the procedure executes, and not when it is compiled. Thus, the following block is successful:

```
CREATE OR REPLACE PROCEDURE dynSQL_proc
AUTHID CURRENT_USER AS
   n NUMBER;
BEGIN
```

```
   SYS.DBMS_SESSION.SET_ROLE('acct');
   EXECUTE IMMEDIATE 'select empno from joe.finance' INTO n;
    --other calls to SYS.DBMS_SQL
END;
/
```

> **See Also:** Choosing Between Native Dynamic SQL and the `DBMS_SQL` Package in *Oracle Database Application Developer's Guide - Fundamentals*

Protecting Database Objects by Using Schemas

A *schema* is a security domain that can contain database objects. The privileges granted to each user or role control access to these database objects. This section covers:

- Unique Schemas
- Shared Schemas

Unique Schemas

Most schemas can be thought of as user names: the accounts that enable users to connect to a database and access the database objects. However, *unique schemas* do not allow connections to the database, but are used to contain a related set of objects. Schemas of this sort are created as normal users, and yet are not granted the `CREATE SESSION` system privilege (either explicitly or through a role). However, you must temporarily grant the `CREATE SESSION` and `RESOURCE` privilege to such schemas if you want to use the `CREATE SCHEMA` statement to create multiple tables and views in a single transaction.

For example, the schema objects for a specific application might be owned by a given schema. If application users have the privileges to do so, then they can connect to the database using typical database user names and use the application and the corresponding objects. However, no user can connect to the database using the schema set up for the application. This configuration prevents access to the associated objects through the schema, and provides another layer of protection for schema objects. In this case, the application could issue an `ALTER SESSION SET CURRENT_SCHEMA` statement to connect the user to the correct application schema.

Shared Schemas

For many applications, users do not need their own accounts or schemas in a database. These users only need to access an application schema. For example, users John, Firuzeh, and Jane are all users of the Payroll application, and they need access to the `Payroll` schema on the `Finance` database. None of them need to create their own objects in the database. They need only access `Payroll` objects. To address this issue, Oracle Advanced Security provides enterprise users (schema-independent users).

Enterprise users, users managed in a directory service, do not need to be created as database users because they use a shared database schema. To reduce administration costs, an administrator can create an enterprise user once in the directory and point the user at a shared schema that many other enterprise users can also access.

> **See Also:** *Oracle Database Advanced Security Administrator's Guide* for information about how shared schemas are created and used for Enterprise User Security

Managing Object Privileges

As part of designing your application, you need to determine the types of users who will be working with the application and the level of access that they need to accomplish their designated tasks. You must categorize these users into role groups, and then determine the privileges that must be granted to each role. This section covers:

- What Application Developers Need to Know About Object Privileges
- SQL Statements Permitted by Object Privileges

What Application Developers Need to Know About Object Privileges

End users are typically granted object privileges. An object privilege allows a user to perform a particular action on a specific table, view, sequence, procedure, function, or package. Table 13–1 summarizes the object privileges available for each type of object.

Table 13–1 How Privileges Relate to Schema Objects

Object Privilege	Applies to Table?	Applies to View?	Applies to Sequence?	Applies to Procedure?[1]
ALTER	Yes	No	Yes	No
DELETE	Yes	Yes	No	No
EXECUTE	No	No	No	Yes
INDEX	Yes[2]	No	No	No
INSERT	Yes	Yes	No	No
REFERENCES	Yes	No	No	No
SELECT	Yes	Yes[3]	Yes	No
UPDATE	Yes	Yes	No	No

[1] Stand-alone stored procedures, functions, and public package constructs
[2] Privilege that cannot be granted to a role
[3] Can also be granted for snapshots

SQL Statements Permitted by Object Privileges

As you implement and test your application, you should create each necessary role. Test the usage scenario for each role to be certain that the users of your application will have proper access to the database. After completing your tests, coordinate with the administrator of the application to ensure that each user is assigned the proper roles.

Table 13–2 lists the SQL statements permitted by the object privileges shown in Table 13–1.

Table 13–2 SQL Statements Permitted by Database Object Privileges

Object Privilege	SQL Statements Permitted
ALTER	ALTER object (table or sequence)
	CREATE TRIGGER ON object (tables only)
DELETE	DELETE FROM object (table, view, or synonym)
EXECUTE	EXECUTE object (procedure or function)
	References to public package variables
INDEX	CREATE INDEX ON object (table, view, or synonym)
INSERT	INSERT INTO object (table, view, or synonym)
REFERENCES	CREATE or ALTER TABLE statement defining a FOREIGN KEY integrity constraint on object (tables only)
SELECT	SELECT...FROM object (table, view, synonym, or snapshot)
	SQL statements using a sequence

See Also: "Understanding User Privileges and Roles" on page 11-11 for a discussion of object privileges

14

Using Virtual Private Database to Implement Application Security Policies

Oracle Database provides the necessary tools to build secure applications. One such tool is Virtual Private Database (VPD), which is the combination of the following:

- Fine-grained access control, which enables you to associate security policies to database objects
- Application context, which enables you to define and access application or database session attributes

VPD combines these two features, enabling you to enforce security policies to control access at the row level. This control is based on application or session attributes, which can be made available during execution.

The following topics introduce these features and explain how and why you would use them:

About Virtual Private Database, Fine-Grained Access Control, and Application Context

Virtual Private Database (VPD) combines server-enforced fine-grained access control with a secure storage of application context values in the Oracle database server. VPD enables you to build applications that enforce row-level security policies at the object level. Policy execution dynamically appends predicates (`WHERE` clauses) to any SQL statements that query data you have identified as requiring protection.

The application context feature enables application developers to define, set, and access variable-length application attributes and their values. These attributes can then be used as predicate values for fine-grained access control policies. Two types of application contexts exist:

- Local (session-based) application context, stored in the UGA and invoked each time an application user connects to the database

- **Global application context** (non-session-based), stored in the SGA and used for multitiered environment users accessing databases through connection pools.

Although application context is an integral part of VPD, it can be implemented alone, without fine-grained access control. When implemented alone, application context can be used to access session information, such as the client identifier, to preserve user identity across multitiered environments.

The remainder of this chapter discusses how VPD works and introduces its main components, fine-grained access control and application context.

See Also:

- Chapter 15, "Implementing Application Context and Fine-Grained Access Control" for information about using local application context and global application context with or without VPD fine-grained access control policies
- "Using the CLIENT_IDENTIFIER Attribute to Preserve User Identity" on page 16-8 for information about using the client identifier attribute to preserve user identity across multitiered environments

Introduction to VPD

Virtual private database (VPD) enables you to enforce security, to a fine level of granularity, directly on tables, views, or synonyms. Because security policies are attached directly to tables, views, or synonyms and automatically applied whenever a user accesses data, there is no way to bypass security.

When a user directly or indirectly accesses a table, view, or synonym protected with a VPD policy, the server dynamically modifies the SQL statement of the user. The modification creates a `WHERE` condition (known as a predicate) returned by a function implementing the security policy. The statement is modified dynamically, transparently to the user, using any condition that can be expressed in or returned by a function. VPD policies can be applied to `SELECT`, `INSERT`, `UPDATE`, `INDEX`, and `DELETE` statements.

Note: Users need full table access to create table indexes. Consequently, users with privileges to maintain an index can see all the row data even if they do not have full table access under a regular query. To prevent this, apply VPD policies to `INDEX` statements.

Functions that return predicates can also include calls to other functions. Within your PL/SQL package, you can embed C or Java calls to access operating system information or to return `WHERE` clauses from an operating system file or central policy store. A policy function can return different predicates for each user, for each group of users, or for each application. Using policy functions over synonyms can substitute for maintaining a separate view for each user or class of users, saving substantial overhead in memory and processing resources.

Application context enables you to securely access the attributes on which you base your security policies. For example, users with the position attribute of `manager` would have a different security policy than users with the position attribute of `employee`.

Consider an HR clerk who is only allowed to see employee records in the Retail Division who initiates the following query:

```
SELECT * FROM emp;
```

The function implementing the security policy returns the predicate `division = 'RETAIL'`, and the database transparently rewrites the query. The query actually executed becomes:

```
SELECT * FROM emp WHERE division = 'RETAIL';
```

Column-Level VPD

Column-level VPD enables you to enforce row-level security when a security-relevant column is referenced in a query. You can apply column-level VPD to tables and views, but not to synonyms. By specifying the security-relevant column name with the `sec_relevant_cols` parameter of the `DBMS_RLS.ADD_POLICY` procedure, the security policy is applied whenever the column is referenced, explicitly or implicitly, in a query.

For example, users outside of the HR department typically are allowed to view only their own Social Security numbers. When a sales clerk initiates the following query:

```
SELECT fname, lname, ssn FROM emp;
```

The function implementing the security policy returns the predicate `ssn='my_ssn'` and the database rewrites the query and executes the following:

```
SELECT fname, lname, ssn FROM emp WHERE ssn = 'my_ssn';
```

> **See Also:** "Adding Policies for Column-Level VPD" on page 15-32 for information about how to add column-level VPD policies

Column-Level VPD with Column-masking Behavior

If a query references a sensitive column, then the default behavior of column-level VPD restricts the number of rows returned. With column-masking behavior, which can be enabled by using the `sec_relevant_cols_opt` parameter of the `DBMS_RLS.ADD_POLICY` procedure, all rows display, even those that reference sensitive columns. However, the sensitive columns display as `NULL` values.

To illustrate this, consider the results of the sales clerk query, described in the previous example. If column-masking behavior is used, then instead of seeing only the row containing the details and Social Security number of the sales clerk, the clerk would see all rows from `emp`, but the `ssn` column values would be returned as `NULL`. Note that this behavior is fundamentally different from all other types of VPD policies, which return only a subset of rows.

> **See Also:** "Column-masking Behavior" on page 15-33 for information about how to add column-level VPD policies with column-masking behavior.

VPD Security Policies and Applications

The security policy is applied within the database itself, rather than within an application. This means that the use of a different application will not bypass the security policy. Security can thus be built once, in the database, instead of being implemented again in multiple applications. VPD therefore provides far stronger security than application-based security, at a lower cost of ownership.

It may be desirable to enforce different security policies depending on which application is accessing data. Consider a situation in which two applications, Order Entry and Inventory, both access the `ORDERS` table. You may want to have the Inventory application use a policy that limits access based on type of product. At the same time, you may want to have the Order Entry application use a policy that limits access based on customer number.

In this case, you must partition the use of fine-grained access by application. Otherwise, both policies would be automatically `AND`ed together, which is not the desired result. You can specify one or more policy groups, and a driving application context that determines which policy group is in effect for a given transaction. You can also designate default policies that always apply to data access. In a hosted application, for example, data access should always be limited by subscriber ID.

Introduction to Fine-Grained Access Control

Fine-grained access control enables you to build applications that enforce security policies at a low level of granularity. These policies are also referred to as VPD policies. You can use it, for example, to restrict customers accessing an Oracle database server to see only their own accounts. A physician could be limited to seeing only the records of her own patients, or a manager to seeing only the records of employees who work for him.

When you use fine-grained access control, you create security policy functions attached to the table, view, or synonym on which you have based your application. Then, when a user enters a `SELECT` or DML statement (`INSERT`, `UPDATE`, or `DELETE`) on that object, Oracle Database dynamically modifies the statement, transparently to the user. The modification ensures that the statement implements the correct access control. You can also enforce security policies on index maintenance operations performed with the DDL statements `CREATE INDEX` and `ALTER INDEX`.

Features of Fine-Grained Access Control

Fine-grained access control provides the following capabilities:

- Security Policies Based on Tables, Views, and Synonyms
- Multiple Policies for Each Table, View, or Synonym
- Grouping of Security Policies
- High Performance
- Default Security Policies

Security Policies Based on Tables, Views, and Synonyms

Attaching security policies to tables, views, or synonyms rather than to applications provides greater security, simplicity, and flexibility.

Security Associating a policy with a table, view, or synonym overcomes a potentially serious application security problem. Suppose a user is authorized to use an application, and then drawing on the privileges associated with that application, wrongfully modifies the database by using an ad hoc query tool, such as SQL*Plus. By attaching security policies to tables, views, or synonyms, fine-grained access control ensures that the same security is in force, no matter how a user accesses the data.

Simplicity Adding the security policy to the table, view, or synonym means that you make the addition only once, rather than repeatedly adding it to each of your table-based, view-based, or synonym-based applications.

Flexibility You can have one security policy for `SELECT` statements, another for `INSERT` statements, and still others for `UPDATE` and `DELETE` statements. For example, you might want to enable Human Resources clerks to `SELECT` all employee records in their division, but to `UPDATE` only salaries for those employees in her division whose last names begin with `A` through `F`.

Note: Although you can define a policy against a table, you cannot select that table from within the policy that was defined against the table.

Multiple Policies for Each Table, View, or Synonym

You can establish several policies for the same table, view, or synonym. Suppose, for example, you have a base application for Order Entry, and each division of your company has its own special rules for data access. You can add a division-specific policy function to a table without having to rewrite the policy function of the base application.

Note that all policies applied to a table are enforced with `AND` syntax. If you have three policies applied to the `CUSTOMERS` table, then each policy is applied to any access of the table. You can use policy groups and a driving application context to partition fine-grained access control enforcement so that different policies apply, depending upon which application is accessing data. This eliminates the requirement for development groups to collaborate on policies and simplifies application development. You can also have a default policy group that is always applicable (for example, to enforce data separated by subscriber in a hosting environment).

Grouping of Security Policies

Because multiple applications with multiple security policies, can share the same table, view, or synonym, it is important to identify those policies that should be in effect when the table, view, or synonym is accessed.

For example, in a hosting environment, Company A can host the `BENEFIT` table for Company B and Company C. The table is accessed by two different applications, Human Resources and Finance, with two different security policies. The Human Resources application authorizes users based on ranking in the company, and the Finance application authorizes users based on department. To integrate these two policies into the `BENEFIT` table would require joint development of policies between the two companies, which is not a feasible option. By defining an application context to drive the enforcement of a particular set of policies to the base objects, each application can implement a private set of security policies.

To do this, you can organize security policies into groups. By referring to the application context, the Oracle Database server determines which group of policies should be in effect at run time. The server enforces all the policies that belong to that policy group.

High Performance

With fine-grained access control, each policy function for a given query is evaluated only once, at statement parse time. Also, the entire dynamically modified query is optimized and the parsed statement can be shared and reused. This means that

rewritten queries can take advantage of the high performance features of Oracle Database, such as dictionary caching and shared cursors.

Default Security Policies

While partitioning security policies by application is desirable, it is also useful to have security policies that are always in effect. In the previous example, a hosted application can always enforce data separation by `subscriber_ID`, whether you are using the Human Resources application or the Finance application. Default security policies allow developers to have base security enforcement under all conditions, while partitioning of security policies by application (using security groups) enables layering of additional, application-specific security on top of default security policies. To implement default security policies, you add the policy to the `SYS_DEFAULT` policy group.

> **See Also:** The following topics for information about how to implement fine-grained access control:
>
> - "How Fine-Grained Access Control Works" on page 15-23
> - "How to Establish Policy Groups" on page 15-24
> - "How to Add a Policy to a Table, View, or Synonym" on page 15-28
> - "Examples: Secure Application Context Within a Fine-Grained Access Control Function" on page 15-7

About Creating a VPD Policy with Oracle Policy Manager

To implement VPD, developers can use the `DBMS_RLS` package to apply security policies to tables and views. They can also use the `CREATE CONTEXT` command to create application contexts.

Alternatively, developers can use the Oracle Policy Manager graphical user interface (GUI), accessed from Oracle Enterprise Manager, to apply security policies to schema objects, such as tables and views, and to create application contexts. Oracle Policy Manager provides an easy-to-use interface to manage security policies and application contexts, and therefore makes VPD easier to develop.

To create VPD policies, users must provide the schema name, table (or view or synonym) name, policy name, the function name that generates the predicate, and the statement types to which the policy applies (that is, `SELECT`, `INSERT`, `UPDATE`, `DELETE`). Oracle Policy Manager then executes the function `DBMS_RLS.ADD_POLICY`. You create an application context by providing the name of the context and the package that implements the context.

Oracle Policy Manager is also the administration tool for Oracle Label Security. Oracle Label Security provides a functional, out-of-the-box VPD policy that enhances your ability to implement row-level security. It supplies an infrastructure, which is a label-based access control framework, whereby you can specify labels for users and data. It also enables you to create one or more custom security policies to be used for label access decisions. You can implement these policies without any knowledge of a programming language. There is no need to write additional code, but in a single step you can apply a security policy to a given table.

In this way, Oracle Label Security provides a straightforward, efficient way to implement row-level security policies using data labeling technology. Finally, the structure of Oracle Label Security labels provides a degree of granularity and

flexibility that cannot easily be derived from the application data alone. Oracle Label Security is thus a generic solution that can be used in many different circumstances.

> **See Also:** *Oracle Label Security Administrator's Guide* for information about using Oracle Policy Manager

Introduction to Application Context

Application context enables you to define, set, and access variable-length application attributes and values in a secure data cache available in User Global Area (UGA) and System Global Area (SGA).

Most applications contain the kind of information that can be used for access control. For example, in an order entry application, the `ORDER_NUMBER` and the `CUSTOMER_NUMBER` of the customer can be used as security attributes to restrict his access to his own orders.

As another example, consider a user running a human resources application. Part of the application initialization process is to determine the kind of responsibility that the user can assume based on user identity. This ID becomes part of the human resource application context. It affects what data the user can access throughout the session.

You configure application context by using the SQL function `SYS_CONTEXT` with the following syntax:

```
SYS_CONTEXT ('namespace','parameter'[,length])
```

The following subsections describe application context and how to use it:

- Features of Application Context
- Ways to Use Application Context with Fine-Grained Access Control

> **See Also:** *Oracle Database SQL Reference* for detailed information about using `SYS_CONTEXT`

Features of Application Context

Application context provides the following important security features:

- Specifying Attributes for Each Application
- Providing Access to Predefined Attributes Through the USERENV Namespace
- Externalized Application Contexts

Specifying Attributes for Each Application

Each application can have its own context with its own attributes. Suppose, for example, you have three applications: General Ledger, Order Entry, and Human Resources. You can specify different attributes for each application:

- For the General Ledger application context, you can specify the attributes `SET_OF_BOOKS` and `TITLE`.
- For the Order Entry application context, you can specify the attribute `CUSTOMER_NUMBER`.
- For the Human Resources application context, you can specify the attributes `ORGANIZATION_ID`, `POSITION`, and `COUNTRY`.

In each case, you can adapt the application context to your precise security needs.

Providing Access to Predefined Attributes Through the USERENV Namespace

Oracle database server provides a built-in application context namespace (USERENV) that provides access to predefined attributes. These attributes are session primitives, which is information that the database captures regarding a user session. Examples include the user name, the IP address from which the user connected, and a proxy user name if the user connection is proxied through a middle tier.

Predefined attributes are useful for access control. For example, a three-tier application creating lightweight user sessions through OCI or thick JDBC can access the PROXY_USER attribute in USERENV. This attribute enables you to determine if the user session was created by a middle tier application. Your policy function could allow a user to access data only for connections where the user is proxied. If users connect directly to the database, then they would not be able to access any data.

You can use the PROXY_USER attribute within VPD to ensure that users only access data through a particular middle-tier application. As a different approach, you can develop a secure application role to enforce your policy that users access the database only through a specific proxy.

See Also:

- Chapter 3, "Tip 4: Use Proxy Authentication and a Secure Application Role", on page 3-5
- Chapter 5, "Secure Application Roles", on page 5-20, and
- Chapter 13, "Creating Secure Application Roles", on page 13-4

You can access predefined attributes through the USERENV application context, but you cannot change them. They are listed in Table 14–1.

Use the following syntax to obtain information about the current session.

```
SYS_CONTEXT('userenv', 'attribute')
```

Note: The USERENV application context namespace replaces the USERENV function provided in earlier database releases.

See Also:

- Chapter 16, "Preserving User Identity in Multitiered Environments" for information about proxy authentication and about using the USERENV attribute, CLIENT_IDENTIFIER, to preserve user identity across multiple tiers
- SYS_CONTEXT in the *Oracle Database SQL Reference* for complete details about the USERENV namespace and its predefined attributes

Table 14–1 Key to Predefined Attributes in USERENV Namespace

Predefined Attribute	Function
AUDITED_CURSORID	Returns the fine-grained auditing cursor ID.

Table 14–1 (Cont.) Key to Predefined Attributes in USERENV Namespace

Predefined Attribute	Function
AUTHENTICATED_IDENTITY	Returns the identity used in authentication. In the list that follows, the type of user is followed by the value returned: ■ Kerberos-authenticated enterprise user: kerberos principal name ■ Kerberos-authenticated external user: kerberos principal name, same as the schema name ■ SSL authenticated enterprise user: the DN in the user's PKI certificate ■ SSL authenticated external user: the DN in the user's PKI certificate ■ Password authenticated enterprise user: nickname, same as the login name ■ Password authenticated database user: the database user name, same as the schema name ■ OS authenticated external user: the external operating system user name ■ Radius authenticated external user: the schema name ■ Proxy with DN: Oracle Internet Directory DN of the client ■ Proxy with certificate: certificated DN of the client ■ Proxy with username: database user name if client is a local database user, nickname if the client is an enterprise user ■ SYSDBA/SYSOPER using password file: login name ■ SYSDBA/SYSOPER using OS authentication: operating system user name
AUTHENTICATION_DATA	Returns the data being used to authenticate the login user. If the user has been authenticated through SSL, or if the user certificate was proxied to the database, then this includes the X.509 certificate of the user.
AUTHENTICATION_METHOD	Returns the method of authentication. In the list that follows, the type of user is followed by the method returned: ■ Password authenticated enterprise user, local database user, SYSDBA/SYSOPER using password file, proxy with username using password: PASSWORD ■ Kerberos authenticated enterprise or external user: KERBEROS ■ SSL authenticated enterprise or external user: SSL ■ Radius authenticated external user: RADIUS ■ OS authenticated external user or SYSDBA/SYSOPER: OS ■ Proxy with certificate, DN, or username without using password: NONE You can use IDENTIFICATION_TYPE to distinguish between external and enterprise users when the authentication method is PASSWORD, KERBEROS OR SSL.
BG_JOB_ID	Returns the background job ID.
CLIENT_IDENTIFIER	User-defined client identifier for the session.

Table 14–1 (Cont.) Key to Predefined Attributes in USERENV Namespace

Predefined Attribute	Function
CLIENT_INFO	Returns up to 64 bytes of user session information that can be stored by an application using the DBMS_APPLICATION_INFO package.
CURRENT_BIND	Returns the bind variables for fine-grained auditing. Maximum length is 4K.
CURRENT_SCHEMA	Returns the name of the default schema being used in the current session. This can be changed with an ALTER SESSION SET SCHEMA statement.
CURRENT_SCHEMAID	Returns the identifier of the default schema being used in the current session. This can be changed with an ALTER SESSION SET SCHEMAID statement.
CURRENT_SQL	Returns the SQL text of the query that triggers fine-grained audit or row-level security (RLS) policy functions or event handlers. Only valid inside the function or event handler.
CURRENT_SQL1 to CURRENT_SQL7	Returns 4K length substrings of the SQL query text that triggers fine-grained audit or row-level security (RLS) policy functions or audit event handlers. This is valid only inside the RLS policy function or event handler. Maximum length is 32K. For example, if a user issued a 32 K length SQL statement, then CURRENT_SQL returns 0 to 4K, CURRENT_SQL1 returns 5K to 8K, CURRENT_SQL2 returns 9K to 12K, and so on.
CURRENT_SQL_LENGTH	Returns the length of the current SQL statement that triggers fine-grained audit or row-level security (RLS) policy functions or event handlers. Only valid inside the function or event handler.
DB_DOMAIN	Returns the domain of the database as specified in the DB_DOMAIN initialization parameter
DB_NAME	Returns the name of the database as specified in the DB_NAME initialization parameter
ENTRYID	Returns available auditing entry identifier. Incremented for every audit record for a SQL statement. Note that there can be more than one audit record for the same SQL statement.
ENTERPRISE_IDENTITY	Returns the user's enterprise-wide identity: ■ For enterprise users: Oracle Internet Directory DN ■ For external users: the external identity (kerberos principal name, Radius schema name, OS user name, certificate DN) ■ For local users and SYSDBA/SYSOPER logins: NULL The value of the attribute differs by proxy method: ■ For a proxy with DN: Oracle Internet Directory DN of the client ■ For a proxy with certificate: certificate DN of the client for external users, Oracle Internet Directory DN for global users ■ For a proxy with username: Oracle Internet Directory DN if the client is an enterprise user, NULL if the client is a local database user
FG_JOB_ID	Returns the foreground job ID.
GLOBAL_CONTEXT_MEMORY	Amount of shared memory used by global application context, in bytes.

Table 14–1 (Cont.) Key to Predefined Attributes in USERENV Namespace

Predefined Attribute	Function
GLOBAL_UID	Returns the user login name from Oracle Internet Directory.
HOST	Returns the name of the host machine on which the database is running.
IDENTIFICATION_TYPE	Returns the way the user's schema was created in the database. Specifically, it reflects the IDENTIFIED clause in the CREATE/ALTER USER syntax. In the list that follows, the syntax used during schema creation is followed by the IDENTIFICATION_TYPE returned: ■ IDENTIFIED BY *password*: LOCAL ■ IDENTIFIED EXTERNALLY: EXTERNAL ■ IDENTIFIED GLOBALLY: GLOBAL SHARED ■ IDENTIFIED GLOBALLY AS *DN*: GLOBAL PRIVATE
INSTANCE	Returns instance identification number of the current instance.
INSTANCE_NAME	Returns the name of the instance.
IP_ADDRESS	Returns the IP address (when available) of the machine from which the client is connected.
ISDBA	Returns TRUE if you currently have the DBA role enabled and FALSE if not.
LANG	Returns the abbreviation for the language name
LANGUAGE	Returns the language and territory currently used by the session, along with the database character set in the form: *language_territory.characterset*
NETWORK_PROTOCOL	Returns the protocol named in the connect string (PROTOCOL=*protocol*).
NLS_TERRITORY	Returns the territory of the current session.
NLS_CURRENCY	Returns the currency symbol of the current session.
NLS_CALENDAR	Returns the calendar used for dates in the current session.
NLS_DATE_FORMAT	Returns the current date format of the current session.
NLS_DATE_LANGUAGE	Returns language used to express dates in the current session.
NLS_SORT	Indicates whether the sort base is binary or linguistic.
OS_USER	Returns the operating system user name of the client process that initiated the database session.
POLICY_INVOKER	Returns the invoker of row-level security policy functions.
PROXY_USER	Returns the name of the database user (typically middle tier) who opened the current session on behalf of SESSION_USER.
PROXY_USERID	Returns the identifier of the database user (typically middle tier) who opened the current session on behalf of SESSION_USER.
SERVER_HOST	Returns the host name of machine on which the instance is running.
SESSION_USER	Returns the database user name by which the current user is authenticated.

Table 14–1 (Cont.) Key to Predefined Attributes in USERENV Namespace

Predefined Attribute	Function
SESSION_USERID	Returns the identifier of the database user name by which the current user is authenticated.
SESSIONID	Returns the auditing session identifier.
SID	Returns the session number (different from the session ID).
STATEMENTID	Returns available auditing statement identifier. Incremented once for every SQL statement audited in a session.
TERMINAL	Returns the operating system identifier for the client of the current session. Virtual in TCP/IP.

Table 14–2 lists the attributes of namespace USERENV that have been deprecated. Oracle suggests that you use the alternatives suggested in the Comments column.

Table 14–2 Deprecated Attributes of Namespace USERENV

Predefined Attribute	Comments
AUTHENTICATION_TYPE	This parameter returned a value indicating how the user was authenticated. The same information is now available from the new AUTHENTICATION_METHOD parameter combined with IDENTIFICATION_TYPE.
CURRENT_USER	Use the SESSION_USER parameter instead.
CURRENT_USERID	Use the SESSION_USERID parameter instead.
EXTERNAL_NAME	This parameter returned the external name of the user. More complete information can now be obtained from the AUTHENTICATED_IDENTITY and ENTERPRISE_IDENTITY parameter.

Externalized Application Contexts

Many applications store attributes used for fine-grained access control within a database metadata table. For example, an EMPLOYEES table could include cost center, title, signing authority, and other information useful for fine-grained access control. Organizations also centralize user information for user management and access control in LDAP-based directories, such as Oracle Internet Directory. Application context attributes can be stored in Oracle Internet Directory and assigned to one or more enterprise users. They can be retrieved automatically upon login for an enterprise user and then used to initialize an application context.

Note: Enterprise User Security is a feature of Oracle Advanced Security.

See Also:

- "Initializing Secure Application Context Externally" on page 15-15 for information about initializing local application context through external resources such as an OCI interface, a job queue process, or a database link
- "Initializing Secure Application Context Globally" on page 15-16 for information about initializing local application context through a centralized resource, such as Oracle Internet Directory
- *Oracle Database Advanced Security Administrator's Guide* for information about enterprise users

Ways to Use Application Context with Fine-Grained Access Control

To simplify security policy implementation, you can use application context within a fine-grained access control function.

Application context can be used in the following ways with fine-grained access control:

- Secure Data Caching
- Returning a Specific Predicate (Security Policy)
- Providing Attributes Similar to Bind Variables in a Predicate

Secure Data Caching

Accessing an application context inside your fine-grained access control policy function is like writing down an often-used phone number and posting it next to your phone, where you can find it easily rather than looking it up every time you need it.

For example, suppose you base access to the `ORDERS_TAB` table on customer number. Rather than querying the customer number for a logged-in user each time you need it, you could store the number in the application context. In this way, the customer number is available in the session when you need it.

Application context is especially helpful if your security policy is based on multiple security attributes. For example, if a policy function bases a predicate on four attributes (such as employee number, cost center, position, spending limit), then multiple subqueries must execute to retrieve this information. Instead, if this data is available through application context, then performance is much faster.

Returning a Specific Predicate (Security Policy)

You can use application context to return the correct security policy, enforced through a predicate.

Consider an order entry application that enforces the following rules: customers only see their own orders, and clerks see all orders for all customers. These are two different policies. You could define an application context with a `Position` attribute, and this attribute could be accessed within the policy function to return the correct predicate, depending on the value of the attribute. Thus, you can enable a user in the `Clerk` position to retrieve all orders, but a user in the `Customer` position to see only those records associated with that particular user.

To design a fine-grained access control policy to return a specific predicate for an attribute, access the application context within the function that implements the

policy. For example, to limit customers to seeing only their own records, use fine-grained access control to dynamically modify user queries as from this:

```
SELECT * FROM Orders_tab
```

to the following query:

```
SELECT * FROM Orders_tab
   WHERE Custno = SYS_CONTEXT ('order_entry', 'cust_num');
```

Providing Attributes Similar to Bind Variables in a Predicate

Continuing with the preceding example, suppose you have 50,000 customers, and you do not want to have a different predicate returned for each customer. Customers all share the same predicate, which prescribes that they can only see their own orders. It is merely their customer numbers that are different.

Using application context, you can return one predicate within a policy function that applies to 50,000 customers. As a result, there is one shared cursor that executes differently for each customer, because the customer number is evaluated at execution time. This value is different for every customer. Use of application context in this case provides optimum performance, as well as row-level security.

Note that the `SYS_CONTEXT` function works much like a bind variable, only if the `SYS_CONTEXT` arguments are constants.

> **See Also:** "Examples: Secure Application Context Within a Fine-Grained Access Control Function" on page 15-7 that also provides a code example

Introduction to Global Application Context

In many application architectures, the middle-tier application is responsible for managing session pooling for application users. Users authenticate themselves to the application, which uses a single identity to log in to the database and maintains all the connections. In this environment, it is not possible to maintain application attributes using session-dependent application context (local application context) because of the sessionless model of the application.

Another scenario is when a user is connected to the database through an application (such as Oracle Forms), which then spawns other applications (such as Oracle Reports) to connect to the database. These applications may need to share the session attributes such that they appear to be sharing the same database session.

Global application context is a type of secure application context that can be shared among trusted sessions. In addition to driving the enforcement of the fine-grained access control policies, applications (especially middle-tier products) can use this support to manage application attributes securely and globally.

Note:

- Global application context is not available in Real Application Clusters.
- Oracle Connection Manager, a router provided with Oracle Net Services, cannot be used with global application context.

Enforcing Application Security

This section contains information about enforcing application security. This section consists of the following topics:

- Use of Ad Hoc Tools: A Potential Security Problem
- Restricting SQL*Plus Users from Using Database Roles
- VPD and Oracle Label Security Exceptions and Exemptions

Use of Ad Hoc Tools: A Potential Security Problem

Prebuilt database applications explicitly control the potential actions of a user, including the enabling and disabling of user roles while using the application. By contrast, ad hoc query tools such as SQL*Plus, allow a user to submit any SQL statement (which may or may not succeed), including the enabling and disabling of any granted role.

Potentially, an application user can exercise the privileges attached to that application to issue destructive SQL statements against database tables by using an ad hoc tool.

For example, consider the following scenario:

- The Vacation application has a corresponding `VACATION` role.
- The `VACATION` role includes the privileges to issue `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements against the `EMP_TAB` table.
- The Vacation application controls the use of privileges obtained through the `VACATION` role.

Now, consider a user who has been granted the `VACATION` role. Suppose that, instead of using the Vacation application, the user executes SQL*Plus. At this point, the user is restricted only by the privileges granted to him explicitly or through roles, including the `VACATION` role. Because SQL*Plus is an ad hoc query tool, the user is not restricted to a set of predefined actions, as with designed database applications. The user can query or modify data in the `EMP_TAB` table as he or she chooses.

Restricting SQL*Plus Users from Using Database Roles

This section presents features that you may use in order to restrict SQL*Plus users from using database roles and thus, prevent serious security problems. These features include the following:

- Limiting Roles Through PRODUCT_USER_PROFILE
- Using Stored Procedures to Encapsulate Business Logic
- Using VPD for Highest Security

Limiting Roles Through PRODUCT_USER_PROFILE

DBAs can use `PRODUCT_USER_PROFILE` to disable certain SQL and SQL*Plus commands in the SQL*Plus environment for each user. SQL*Plus, not the Oracle Database, enforces this security. DBAs can even restrict access to the `GRANT`, `REVOKE`, and `SET ROLE` commands in order to control user ability to change their database privileges.

The `PRODUCT_USER_PROFILE` table enables you to list roles that you do not want users to activate with an application. You can also explicitly disable the use of various commands, such as `SET ROLE`.

For example, you could create an entry in the PRODUCT_USER_PROFILE table to:

- Disallow the use of the CLERK and MANAGER roles with SQL*Plus
- Disallow the use of SET ROLE with SQL*Plus

Suppose user Jane connects to the database using SQL*Plus. Jane has the CLERK, MANAGER, and ANALYST roles. As a result of the preceding entry in PRODUCT_USER_PROFILE, Jane is only able to exercise her ANALYST role with SQL*Plus. Also, when Jane attempts to issue a SET ROLE statement, she is explicitly prevented from doing so because of the entry in the PRODUCT_USER_PROFILE table prohibiting use of SET ROLE.

Use of the PRODUCT_USER_PROFILE table does not completely guarantee security, for multiple reasons. In the preceding example, while SET ROLE is disallowed with SQL*Plus, if Jane had other privileges granted to her directly, then she could exercise these using SQL*Plus.

> **See Also:** *SQL*Plus User's Guide and Reference* for more information about the PRODUCT_USER_PROFILE table

Using Stored Procedures to Encapsulate Business Logic

Stored procedures encapsulate use of privileges with business logic so that privileges are only exercised in the context of a well-formed business transaction. For example, an application developer might create a procedure to update employee name and address in the EMPLOYEES table, which enforces that the data can only be updated in normal business hours. Also, rather than grant a human resources clerk the UPDATE privilege on the EMPLOYEES table, a developer (or application administrator) may grant the privilege on the procedure only. Then, the human resources clerk can exercise the privilege only in the context of the procedures, and cannot update the EMPLOYEES table directly.

Using VPD for Highest Security

Virtual Private Database (VPD) provides the benefit of strong security policies, which apply directly to data. When you use VPD, you can enforce security no matter how a user gets to the data: whether through an application, through a query, or by using a report-writing tool.

> **See Also:**
>
> - "Introduction to VPD" on page 14-2
> - "Ways to Use Application Context with Fine-Grained Access Control" on page 14-13
> - "How to Add a Policy to a Table, View, or Synonym" on page 15-28 for information about using the DBMS_RLS.ADD_POLICY procedure to add policies for VPD.

VPD and Oracle Label Security Exceptions and Exemptions

VPD and Oracle Label Security are not enforced during DIRECT path export. Also, VPD policies and Oracle Label Security policies cannot be applied to objects in the SYS schema. As a consequence, the SYS user and users making a DBA-privileged connection to the database (for example, CONNECT/AS SYSDBA) do not have VPD or Oracle Label Security policies applied to their actions. The database user SYS is thus always exempt from VPD or Oracle Label Security enforcement, regardless of the export mode, application, or utility used to extract data from the database. However,

SYSDBA actions can be audited by enabling such auditing upon installation and specifying that this audit trail be stored in a secure location in the operating system.

Similarly, database users granted the EXEMPT ACCESS POLICY privilege, either directly or through a database role, are exempt from VPD enforcements. They are also exempt from some Oracle Label Security policy enforcement controls, such as READ_CONTROL and CHECK_CONTROL, regardless of the export mode, application, or utility used to access the database or update its data. However, the following policy enforcement options remain in effect even when EXEMPT ACCESS POLICY is granted:

- INSERT_CONTROL, UPDATE_CONTROL, DELETE_CONTROL, WRITE_CONTROL, LABEL_UPDATE, and LABEL_DEFAULT
- If the Oracle Label Security policy specifies the ALL_CONTROL option, then all enforcement controls are applied except READ_CONTROL and CHECK_CONTROL.

EXEMPT ACCESS POLICY is a very powerful privilege and should be carefully managed. It is inadvisable to grant this privilege WITH ADMIN OPTION because very few users should have this exemption.

Note:

- The EXEMPT ACCESS POLICY privilege does not affect the enforcement of object privileges such as SELECT, INSERT, UPDATE, and DELETE. These privileges are enforced even if a user has been granted the EXEMPT ACCESS POLICY privilege.
- The SYS_CONTEXT values that VPD uses are not propagated to secondary databases for failover.

See Also: *Oracle Label Security Administrator's Guide*

User Models and VPD

Oracle enables applications to enforce fine-grained access control for each user, regardless of whether that user is a database user or an application user unknown to the database.

When application users are also database users, VPD enforcement is relatively simple. Users connect to the database, and the application sets up application contexts for each session. As each session is initiated under a different user name, it is simple to enforce different fine-grained access control conditions for each such user. Even proxy authentication permits different fine-grained access control for each user, because each session (OCI or thick JDBC) is a distinct database session with its own application context.

When proxy authentication is integrated with Enterprise User Security, user roles and other attributes can be retrieved from Oracle Internet Directory to enforce VPD. (In addition, globally initialized application context can also be retrieved from the directory.)

Applications connecting to the database as a single user on behalf of all users can also have fine-grained access control for each user. The user for that single session is often called *One Big Application User*. Within the context of that session, however, an application developer can create a global application context attribute to represent the individual application user (for example, REALUSER). Although all such database sessions and audit records are created for One Big Application User, each session

attributes can vary, depending on who the real end user is. This model works best for applications with a limited number of users and no reuse of sessions. The scope of roles and database auditing is diminished because each session is created as the same database user.

Web-based applications typically have hundreds if not thousands of users. Even when there are persistent connections to the database, supporting data retrieval for many user requests, these connections are not specific to particular Web-based users. Instead, Web-based applications typically set up and reuse connections, to provide scalability, rather than having different sessions for each user. For example, when Web users Jane and Ajit connect to a middle tier application, it may establish a single database session that it uses on behalf of both users. Typically, neither Jane nor Ajit is known to the database. The application is responsible for switching the user name on the connection, so that, at any given time, it is either Jane or Ajit using the session.

Oracle Database VPD facilitates connection pooling by allowing multiple connections to access more than one global application context. This capability makes it unnecessary to establish a separate application context for each distinct user session.

Table 14–3 summarizes how VPD applies to various user models.

Table 14–3 VPD in Different User Models

User Model Scenario	Individual DB Connection	Separate Application Context per User	Single DB Connection	Application Must Switch User Name
Application users are also database users	Yes	Yes	No	No
Proxy authentication using OCI or thick JDBC	Yes	Yes	No	No
Proxy authentication integrated with Enterprise User Security[1]	No	No	Yes	Yes
One Big Application User	No	No[2]	Yes	Yes[2]
Web-based applications	No	No	Yes	Yes

[1] User roles and other attributes, including globally initialized application context, can be retrieved from Oracle Internet Directory to enforce VPD.

[2] Application developers can create a global application context attribute representing individual application users (for example, `REALUSER`), which can then be used for controlling each session attributes, or for auditing.

15

Implementing Application Context and Fine-Grained Access Control

Application context can be used with fine-grained access control as part of Virtual Private Database (VPD) or by itself. Used alone, it enables application developers to define, set, and access application attributes by serving as a data cache. Such usage removes the repeated overhead of querying the database each time access to application attributes is needed.

This chapter discusses how to implement application context and fine-grained access control using the following sections:

About Using Application Context

Application context can be useful for the following purposes:

- Enforcing fine-grained access control
- Preserving user identity across multitier environments
- Serving as a secure data cache for attributes needed by an application for fine-grained auditing or for use in PL/SQL conditional statements or loops

This cache saves the repeated overhead of querying the database each time such attributes are needed.

- Serving as a holding area for attribute-value pairs that an application can define, modify, and access

Three types of application context are defined, differentiated by where the context data is stored and by how updates can be done:

- Secure session-based application contexts, for which data is stored in the database user session (UGA) in a namespace specified by an SQL command (`CREATE CONTEXT`).

  Example: `CREATE CONTEXT hr USING scott.package1;`

  Data in this secure, session-private data cache can only be set by the PL/SQL package in that command. In this example, only `scott.package1` can alter data in the `hr` namespace.

  Namespaces created in this way are maintained using the existing PL/SQL packages available in the `DBMS_SESSION` package.

  Session-based application contexts can be initialized either externally or globally, and either method stores the context information in the user session.

  - *External* initialization can come from an OCI interface, a job queue process, or a connected user database link.
  - *Global* initialization uses attributes and values from a centralized location, such as an LDAP directory.

  See Also:

  - Initializing Secure Application Context Externally on page 15-15
  - Initializing Secure Application Context Globally on page 15-16

  Note: The `init.ora` *`parameter _session_context_size`* limits the total number of (namespace, attribute) pairs used by all application contexts in the user session UGA. This limit includes the client namespace `CLIENTCONTEXT`, but excludes globally accessed context, which uses memory from the shared pool.

  Users can change the value *`for _session_context_size`*, the default value for which is 10,000.

  A new attribute, `SESSION_CONTEXT_SIZE`, which stores the number of sets of context information that are currently in the user session, is accessible as `SELECT SYS_CONTEXT('userenv', 'session_context_size') from dual;`

- The client session-based application context, using only the `CLIENTCONTEXT` namespace, updatable by any OCI client or by the existing `DBMS_SESSION API` for application context. No privilege or package security check is done.

  The `CLIENTCONTEXT` namespace enables a single application transaction to both change the user context information and use the same user session handle to service the new user request.

  - An OCI client uses the `OCIAppCtx` API to set variable length data for the namespace, on the `OCISessionHandle`. The OCI network single round-trip

transport sends all the information to the server in one round trip. On the server side, the application context information can be queried using the `SYS_CONTEXT` SQL function on the namespace.

- A JDBC client uses the `oracle.jdbc.internal.OracleConnection` API to achieve the same purposes.

See Also:

- Managing Scalable Platforms in *Oracle Call Interface Programmer's Guide* for details regarding the OCIAppCtx API
- Using Client Session-Based Application Context on page 15-19.

Any user can set, clear, or collect the information in the `CLIENTCONTEXT` namespace, because it is not protected by package-based security.

- Nonsession-based (global) application contexts, for which data is stored in the shared area (SGA).

See Also: How to Use Global Application Context on page 15-20

Table 15–1 summarizes the different types of application contexts.

Table 15–1 Types of Application Contexts

Application Context Type	Stored in UGA	Stored in SGA	Supports Connected User Database Links	Supports Centralized Storage of Users' Application Context	Supports Sessionless Multitier Applications
Application Context	X	--	--	--	--
Application Context Initialized Externally	X	--	X	--	--
Application Context Initialized Globally	X	--	--	X	--
CLIENTCONTEXT	X	--	X	--	X
Global Application Context	--	X	--	--	X

See Also:

- "Introduction to Application Context" on page 14-7 for conceptual information about session-based application context
- "Introduction to Global Application Context" on page 14-14 for conceptual information about nonsession-based application context
- "Using the CLIENT_IDENTIFIER Attribute to Preserve User Identity" on page 16-8 for a discussion about using the `CLIENT_IDENTIFIER` attribute of the predefined `USERENV` application context
- "Fine-Grained Auditing" on page 12-24 for information about using application context with fine-grained auditing

The following sections explain secure and client session-based application context:

- Using Secure Session-Based Application Context on page 15-4
- Using Client Session-Based Application Context on page 15-19

Using Secure Session-Based Application Context

To use secure session-based application context, you must perform the following tasks:

- Task 1: Create a PL/SQL Package that Sets the Secure Context for Your Application
- Task 2: Create a Unique Secure Context and Associate It with the PL/SQL Package
- Task 3: Set the Secure Context Before the User Retrieves Data
- Task 4: Use the Secure Context in a VPD Policy Function

Task 1: Create a PL/SQL Package that Sets the Secure Context for Your Application

Begin by creating a PL/SQL package with functions that set the secure context for your application. This section presents the syntax and behavior of the `SYS_CONTEXT` SQL function, followed by an example for creating the PL/SQL package.

Note: A login trigger can be used because the user context (information such as `EMPNO`, `GROUP`, `MANAGER`) should be set before the user accesses any data.

SYS_CONTEXT Syntax

The syntax for this function is:

```
SYS_CONTEXT ('namespace', 'attribute', [length])
```

This function returns the value of `attribute` as defined in the package currently associated with the context namespace. It is evaluated once for each statement execution, and is treated like a constant during type checking for optimization. You can use the predefined namespace `USERENV` to access primitive contexts such as userid and Globalization Support parameters.

See Also:

- "Providing Access to Predefined Attributes Through the USERENV Namespace" on page 14-8 for information about the `USERENV` application context namespace and a complete list of its predefined attributes
- *Oracle Database SQL Reference* for details about `USERENV` predefined attributes

SYS_CONTEXT Example

The following example creates the package `App_security_context`.

```
CREATE OR REPLACE PACKAGE App_security_context IS
   PROCEDURE Set_empno;
END;

CREATE OR REPLACE PACKAGE BODY App_security_context IS
   PROCEDURE Set_empno
   IS
   Emp_id NUMBER;
   BEGIN
    SELECT Empno INTO Emp_id FROM Emp
       WHERE Ename = SYS_CONTEXT('USERENV',
```

```
                                    'SESSION_USER');
    DBMS_SESSION.SET_CONTEXT('app_context', 'empno', Emp_id);
   END;
END;
```

> **See Also:** *PL/SQL Packages and Types Reference* for information about the `DBMS_SESSION.SET_CONTEXT` procedure.

Using Dynamic SQL with SYS_CONTEXT

> **Note:** This feature is applicable when `COMPATIBLE` is set to either 8.0 or 8.1.

During a session in which you expect a change in policy between executions of a given query, the query must use dynamic SQL. You must use dynamic SQL because static SQL and dynamic SQL parse statements differently.

- Static SQL statements are parsed at compile time. They are not reparsed at execution for performance reasons.
- Dynamic SQL statements are parsed every time they are executed.

Consider a situation in which policy A is in force when you compile a SQL statement, and then you switch to policy B and run the statement. With static SQL, policy A remains in force. The statement is parsed at compile time and not reparsed upon execution. With dynamic SQL, the statement is parsed upon execution, and so the switch to policy B takes effect.

For example, consider the following policy:

```
EMPLOYEE_NAME = SYS_CONTEXT ('USERENV', 'SESSION_USER')
```

The policy `EMPLOYEE_NAME` matches the database user name. It is represented in the form of a SQL predicate: the predicate is basically a policy. If the predicate changes, then the statement must be reparsed in order to produce the correct result.

> **See Also:** "Automatic Reparse" on page 15-36

Using SYS_CONTEXT in a Parallel Query

If `SYS_CONTEXT` is used inside a SQL function that is embedded in a parallel query, then the function picks up the application context.

Consider a user-defined function within a SQL statement, which sets the user ID to 5:

```
CREATE FUNCTION proc1 AS RETURN NUMBER;
BEGIN
     IF SYS_CONTEXT ('hr', 'id') = 5
     THEN RETURN 1; ELSE RETURN 2;
     END
END;
```

Now consider the statement:

```
SELECT * FROM EMP WHERE proc1( ) = 1;
```

When this statement is run as a parallel query, the user session, which contains the application context information, is propagated to the parallel execution servers (query slave processes).

Using SYS_CONTEXT with Database Links

Session-based local application context can be accessed by SQL statements within a user session by using the SYS_CONTEXT SQL function. When these SQL statements involve database links, then the SYS_CONTEXT SQL function is executed at the database link's initiating site and captures the context information there (at the initiating site).

If remote PL/SQL procedure calls are executed over a database link, then any SYS_CONTEXT function inside such a procedure is executed at the database link's destination site. In this case, only externally initialized application contexts are available at the database link destination site. For security reasons, only the externally initialized application context information is propagated to the destination site from the initiating database link site.

Task 2: Create a Unique Secure Context and Associate It with the PL/SQL Package

To perform this task, use the CREATE CONTEXT statement. Each context must have a unique attribute and belong to a namespace. That is, context names must be unique within the database, not just within a schema. Contexts are always owned by the SYS schema.

For example:

```
CREATE CONTEXT order_entry USING App_security_context;
```

Here, order_entry is the context namespace and App_security_context is the trusted package that can set attributes in the context namespace.

After you have created the context, you can set or reset the context attributes by using the DBMS_SESSION.SET_CONTEXT package. The values of the attributes you set remain either until you reset them or until the user ends the session.

You can only set the context attributes inside the trusted procedure you named in the CREATE CONTEXT statement. This prevents a malicious user from changing context attributes without proper attribute validation.

Alternatively, you can use the Oracle Policy Manager graphical user interface (GUI) to create a context and associate it with a PL/SQL package. Oracle Policy Manager, accessed from Oracle Enterprise Manager, enables you to apply policies to database objects and create application contexts. It also can be used to create and manage Oracle Label Security policies.

Task 3: Set the Secure Context Before the User Retrieves Data

Always use an event trigger on login to pull session information into the context. This sets the security-limiting attributes of the user for the database to evaluate, and thus enables it to make the appropriate security decisions.

Other considerations come into play if you have a changing set of books, or if positions change constantly. In these cases, the new attribute values may not be picked up right away, and you must force a cursor reparse to pick them up.

Task 4: Use the Secure Context in a VPD Policy Function

Now that you have set up the context and the PL/SQL package, your VPD policy functions can use the application context to make policy decisions based on different context values.

Examples: Secure Application Context Within a Fine-Grained Access Control Function

This section provides the following examples that use secure session-based application context within a fine-grained access control function.

Example 1: Implementing the Policy

This example uses secure application context to implement a policy, in which customers can see only their own orders, and builds the application using the following steps:

The procedure in this example assumes a one-to-one relationship between users and customers. It finds the user customer number (`Cust_num`) and caches the customer number in the application context. You can later refer to the `cust_num` attribute of the order entry context (`oe_ctx`) inside the security policy function.

Note that you could use a login trigger to set the initial context.

Step 1: Create a PL/SQL Package To Set the Secure Context for the Application

Create the package as follows:

> **Note:** You may need to set up the following data structures for the following examples to work:
>
> ```
> CREATE TABLE apps.customers (cust_no NUMBER(4), cust_name
> VARCHAR2(20));
> CREATE TABLE scott.orders_tab (order_no NUMBER(4));
> ```

```
CREATE OR REPLACE PACKAGE apps.oe_ctx AS
   PROCEDURE set_cust_num;
END;

CREATE OR REPLACE PACKAGE BODY apps.oe_ctx AS
   PROCEDURE set_cust_num IS
     custnum NUMBER;
   BEGIN
      SELECT cust_no INTO custnum FROM customers WHERE cust_name =
         SYS_CONTEXT('USERENV', 'SESSION_USER');
     /* SET cust_num attribute in 'order_entry' context */
         DBMS_SESSION.SET_CONTEXT('order_entry', 'cust_num', custnum);
         DBMS_SESSION.SET_CONTEXT('order_entry', 'cust_num', custnum);
   END set_cust_num;
END;
```

Note: This example does not treat error handling.

You can access predefined attributes, such as session user, by using `SYS_CONTEXT('USERENV', session_primitive)`.

For more information, see Table 14–1, " Key to Predefined Attributes in USERENV Namespace" on page 14-8 and *Oracle Database SQL Reference*.

Step 2: Create a Secure Application Context

Create an application context by entering:

```
CREATE CONTEXT Order_entry USING apps.oe_ctx;
```

Alternatively, you can use Oracle Policy Manager to create an application context.

Step 3: Access the Secure Application Context Inside the Package

Access the secure application context inside the package that implements the security policy on the database object.

Note: You may need to set up the following data structures for certain examples to work:

```
CREATE OR REPLACE PACKAGE Oe_security AS
FUNCTION Custnum_sec (D1 VARCHAR2, D2 VARCHAR2)
RETURN VARCHAR2;
END;
```

The package body appends a dynamic predicate to `SELECT` statements on the `ORDERS_TAB` table. This predicate limits the orders returned to those associated with the customer number of the user by accessing the `cust_num` context attribute, instead of using a subquery to the customers table.

```
CREATE OR REPLACE PACKAGE BODY Oe_security AS

/* limits select statements based on customer number: */
FUNCTION Custnum_sec (D1 VARCHAR2, D2 VARCHAR2) RETURN VARCHAR2
IS
    D_predicate VARCHAR2 (2000);
    BEGIN
     D_predicate := 'cust_no = SYS_CONTEXT(''order_entry'', ''cust_num'')';
     RETURN D_predicate;
    END Custnum_sec;
END Oe_security;
```

Step 4: Create the New Security Policy

Note: You may need to set up the following data structures for certain examples to work:

```
CONNECT sys/xIcf1T9u AS sysdba;
CREATE USER secusr IDENTIFIED BY secusr;
```

Create the policy as follows:

```
BEGIN
DBMS_RLS.ADD_POLICY ('scott', 'orders_tab', 'oe_policy', 'secusr',
                     'oe_security.custnum_sec', 'select');
END;
```

This statement adds a policy named OE_POLICY to the ORDERS_TAB table for viewing in the SCOTT schema. The SECUSR.OE_SECURITY.CUSTNUM_SEC function implements the policy, is stored in the SECUSR schema, and applies to SELECT statements only.

Now, any SELECT statement by a customer on the ORDERS_TAB table automatically returns only the orders of that particular customer. In other words, the dynamic predicate modifies the statement from:

```
SELECT * FROM Orders_tab;
```

to the following:

```
SELECT * FROM Orders_tab
   WHERE Custno = SYS_CONTEXT('order_entry','cust_num');
```

Note the following with regard to this example:

- In reality, you might have several predicates based on a user's position. For example, a sales representative would be able to see records only for his customers, and an order entry clerk would be able to see any customer order. You could expand the custnum_sec function to return different predicates based on the user position context value.
- The use of application context in a fine-grained access control package effectively gives you a bind variable in a parsed statement. For example:

  ```
  SELECT * FROM Orders_tab
     WHERE Custno = SYS_CONTEXT('order_entry', 'cust_num')
  ```

  This is fully parsed and optimized, but the evaluation of the CUST_NUM attribute value of the user for the ORDER_ENTRY context takes place at execution. This means that you get the benefit of an optimized statement that executes differently for each user who executes the statement.

  Note: You can improve the performance of the function in this example by indexing CUST_NO.

- You can set context attributes based on data from a database table or tables, or from a directory server using LDAP (Lightweight Directory Access Protocol).

  See Also:

  - Using Triggers in *Oracle Database Application Developer's Guide - Fundamentals*
  - "Optimizing Performance by Enabling Static and Context Sensitive Policies" on page 15-31
  - "Adding Policies for Column-Level VPD" on page 15-32

Compare and contrast this example, which uses an application context within the dynamically generated predicate, with "How Fine-Grained Access Control Works" on page 15-23, which uses a subquery in the predicate.

Example 2: Controlling User Access with an Application

This example uses secure application context to control user access with a Human Resources application. Each task that you need to perform is described more fully in the sections that follow:

- Step 1: Create a PL/SQL Package to Set the Secure Context
- Step 2: Create the Secure Context and Associate It with the Package
- Step 3: Create the Initialization Script for the Application

In this example, assume that the application context for the Human Resources application is assigned to the `HR_CTX` namespace.

Step 1: Create a PL/SQL Package to Set the Secure Context

Note: You may need to set up the following data structures for certain examples to work:

```
DROP USER apps CASCADE;
CREATE USER apps IDENTIFIED BY welcome1;

CREATE OR REPLACE PACKAGE apps.hr_sec_ctx IS
   PROCEDURE set_resp_id (respid NUMBER);
   PROCEDURE set_org_id (orgid NUMBER);
  /* PROCEDURE validate_respid (respid NUMBER); */
  /* PROCEDURE validate_org_id (orgid NUMBER); */
END hr_sec_ctx;
```

Create a PL/SQL package with a number of functions that set the secure context for the application. `APPS` is the schema that owns this package.

```
CREATE OR REPLACE PACKAGE BODY apps.hr_sec_ctx IS
/* function to set responsibility id */
PROCEDURE set_resp_id (respid NUMBER) IS
BEGIN

/* validate respid based on primitive and other context */
/*    validate_respid (respid); */
/* set resp_id attribute under namespace 'hr_ctx'*/

    DBMS_SESSION.SET_CONTEXT('hr_ctx', 'resp_id', respid);
END set_resp_id;

/* function to set organization id */
PROCEDURE set_org_id (orgid NUMBER) IS
BEGIN

/* validate organization ID */
/*    validate_org_id(orgid); /*
/* set org_id attribute under namespace 'hr_ctx' */

    DBMS_SESSION.SET_CONTEXT('hr_ctx', 'org_id', orgid);
END set_org_id;

/* more functions to set other attributes for the HR application */
END hr_sec_ctx;
```

Step 2: Create the Secure Context and Associate It with the Package

For example:

```
CREATE CONTEXT Hr_ctx USING apps.hr_sec_ctx;
```

Step 3: Create the Initialization Script for the Application

Suppose that the execute privilege on the HR_SEC_CTX package has been granted to the schema running the application. Part of the script will make calls to set various attributes of the HR_CTX context. Here, we do not show how the context is determined. Normally, it is based on the primitive context or other derived context.

```
APPS.HR_SEC_CTX.SET_RESP_ID(1);
APPS.HR_SEC_CTX.SET_ORG_ID(101);
```

The SYS_CONTEXT function can be used for data access control based on this application context. For example, the base table HR_ORGANIZATION_UNIT can be secured by a view that restricts access to rows based on the attribute ORG_ID:

```
CREATE VIEW Hr_organization_secv AS
   SELECT * FROM hr_organization_unit
      WHERE Organization_id = SYS_CONTEXT('hr_ctx','org_id');
```

Note: You may need to set up the following data structures for certain examples to work:

```
CREATE TABLE hr_organization_unit (organization_id NUMBER);
```

Example 3: Event Triggers, Secure Application Context, Fine-Grained Access Control, and Encapsulation of Privileges

This example illustrates use of the following security features in Oracle Database:

- Event triggers
- Application context (session-based)
- Fine-grained access control
- Encapsulation of privileges in stored procedures

In this example, we associate a security policy with the table called DIRECTORY, which has the following columns:

Column	Description
EMPNO	Identification number for each employee
MGRID	Employee identification number for the manager of each employee
RANK	Position of the employee in the corporate hierarchy

Note: You may need to set up the following data structures for certain examples to work:

```
CREATE TABLE Payroll(
   Srate  NUMBER,
   Orate  NUMBER,
   Acctno NUMBER,
   Empno  NUMBER,
   Name   VARCHAR2(20));
CREATE TABLE Directory_u(
   Empno NUMBER,
   Mgrno NUMBER,
   Rank  NUMBER);
CREATE SEQUENCE Empno_seq;
CREATE SEQUENCE Rank_seq;
```

The security policy associated with this table has two elements:

- All users can find the MGRID for a specific EMPNO. To implement this, we create a definer's right package in the human resources schema (HR) to perform SELECT on the table.
- Managers can update only those positions in the corporate hierarchy that are their direct subordinates. To do this, they must use only the designated application. You can implement this as follows:
  - Define fine-grained access control policies on the table based on EMPNO and application context.
  - Set EMPNO by using a login trigger.
  - Set the application context by using the designated package for processing the updates (event triggers and application context).

Note: In this example, we grant UPDATE privileges on the table to PUBLIC, because fine-grained access control prevents an unauthorized user from wrongly modifying a given row.

The following code implements this example as described:

```
CONNECT system/yJdg2U1v AS sysdba
GRANT CONNECT,RESOURCE,UNLIMITED TABLESPACE,CREATE ANY CONTEXT, CREATE PROCEDURE,
CREATE ANY TRIGGER TO HR IDENTIFIED BY HR;
CONNECT hr/hr;
CREATE TABLE Directory (Empno   NUMBER(4) NOT NULL,
                        Mgrno   NUMBER(4) NOT NULL,
                        Rank    NUMBER(7,2) NOT NULL);

CREATE TABLE Payroll (Empno  NUMBER(4) NOT NULL,
                      Name   VARCHAR(30) NOT NULL );

/* seed the tables with a couple of managers: */
INSERT INTO Directory VALUES (1, 1, 1.0);
INSERT INTO Payroll VALUES (1, 'KING');
INSERT INTO Directory VALUES (2, 1, 5);
INSERT INTO Payroll VALUES (2, 'CLARK');

/* Create the sequence number for EMPNO: */
```

```
CREATE SEQUENCE Empno_seq START WITH 5;

/* Create the sequence number for RANK:  */
CREATE SEQUENCE Rank_seq START WITH 100;

CREATE OR REPLACE CONTEXT Hr_app USING Hr.Hr0_pck;
CREATE OR REPLACE CONTEXT Hr_sec USING Hr.Hr1_pck;

CREATE OR REPLACE PACKAGE Hr0_pck IS
PROCEDURE adjustrankby1(Empno NUMBER);
END;

CREATE OR REPLACE PACKAGE BODY Hr0_pck IS
/* raise the rank of the empno by 1:  */
PROCEDURE Adjustrankby1(Empno NUMBER)
IS
   Stmt   VARCHAR2(100);
   BEGIN

   /*Set context to indicate application state */
   DBMS_SESSION.SET_CONTEXT('hr_app','adjstate',1);
   /* Now we can issue DML statement:  */
   Stmt := 'UPDATE Directory d SET Rank = Rank + 1 WHERE d.Empno = '
   || Empno;
   EXECUTE IMMEDIATE STMT;

/* Re-set application state: */
   DBMS_SESSION.SET_CONTEXT('hr_app','adjstate',0);
   END;
END;

CREATE OR REPLACE PACKAGE hr1_pck IS PROCEDURE setid;
END;
/
/* Based on userid, find EMPNO, and set it in application context */

CREATE or REPLACE PACKAGE BODY Hr1_pck IS
PROCEDURE setid
  IS
  id NUMBER;
  BEGIN
    SELECT Empno INTO id FROM Payroll WHERE Name =
      SYS_CONTEXT('userenv','session_user') ;
    DBMS_SESSION.SET_CONTEXT('hr_sec','empno',id);
    DBMS_SESSION.SET_CONTEXT('hr_sec','appid',id);
  EXCEPTION
    /* For purposes of demonstration insert into payroll table
    /  so that user can continue on and run example. */
    WHEN NO_DATA_FOUND THEN
      INSERT INTO Payroll (Empno, Name)
         VALUES (Empno_seq.NEXTVAL, SYS_CONTEXT('userenv','session_user'));
      INSERT INTO Directory (Empno, Mgrno, Rank)
         VALUES (Empno_seq.CURRVAL, 2, Rank_seq.NEXTVAL);
      SELECT Empno INTO id FROM Payroll WHERE Name =
        sys_context('userenv','session_user') ;
      DBMS_SESSION.SET_CONTEXT('hr_sec','empno',id);
      DBMS_SESSION.SET_CONTEXT('hr_sec','appid',id);
    WHEN OTHERS THEN
      NULL;
    /* If this is to be fired by using a "logon" trigger,
```

```
    /  you need to handle exceptions if you want the user to continue
    /  logging into the database. */
  END;
END;

GRANT EXECUTE ON Hr1_pck TO public;

CONNECT system/yJdg2U1v AS sysdba

CREATE OR REPLACE TRIGGER Databasetrigger

AFTER LOGON
ON DATABASE
BEGIN
   hr.Hr1_pck.Setid;
END;

/* Creates the package for finding the MGRID for a particular EMPNO
using definer's right (encapsulated privileges). Note that users are
granted EXECUTE privileges only on this package, and not on the table
(DIRECTORY) it is querying. */

CONNECT hr/hr

CREATE OR REPLACE PACKAGE hr2_pck IS
   FUNCTION Findmgr(Empno NUMBER) RETURN NUMBER;
END;

CREATE OR REPLACE PACKAGE BODY hr2_pck IS
   /* insert a new employee record: */
   FUNCTION findmgr(empno number) RETURN NUMBER IS
   Mgrid NUMBER;
   BEGIN
      SELECT mgrno INTO mgrid FROM directory WHERE mgrid = empno;
   RETURN mgrid;
   END;
END;

CREATE OR REPLACE FUNCTION secure_updates(ns varchar2,na varchar2)
  RETURN VARCHAR2 IS
     Results VARCHAR2(100);
    BEGIN
       /* Only allow updates when designated application has set the session
       state to indicate we are inside it. */
       IF (sys_context('hr_app','adjstate') = 1)
          THEN results := 'mgrno = SYS_CONTEXT("hr_sec","empno")';
       ELSE results := '1=2';
      END IF;
      RETURN Results;
   END;

/* Attaches fine-grained access policy to all update operations on
hr.directory */

CONNECT system/yJdg2U1v AS sysdba;
BEGIN
   DBMS_RLS.ADD_POLICY('hr','directory','secure_update','hr',
                       'secure_updates','update',TRUE,TRUE);
END;
```

Initializing Secure Application Context Externally

This feature lets you specify a special type of namespace that accepts initialization of attribute values from external resources and stores them in the local user session. Allowing secure application context to be initialized externally enhances performance and enables the automatic propagation of attributes from one session to another. Connected user database links are supported only by application contexts initialized from OCI-based external sources.

This section contains these topics:

Obtaining Default Values from Users

Sometimes it is desirable to obtain default values from users. Initially, these default values may serve as hints or preferences, and then after validation become trusted contexts. Similarly, it may be more convenient for clients to initialize some default values, and then rely on a login event trigger or applications to validate the values.

For job queues, the job submission routine records the context being set at the time the job is submitted, and restores it when executing the batched job. To maintain the integrity of the context, job queues cannot bypass the designated PL/SQL package to set the context. Rather, externally initialized application context accepts initialization of context values from the job queue process.

Automatic propagation of context to a remote session may create security problems. Developers or administrators can effectively handle this type of context that takes default values from resources other than the designated PL/SQL procedure by using login triggers to reset the context when users log in.

Obtaining Values from Other External Resources

In addition to using the designated trusted package, externally initialized secure application contexts can also accept initialization of attributes and values through external resources. Examples include an OCI interface, a job queue process, or a database link.

Externally initialized secure application context provides:

- For remote sessions, automatic propagation of context values that are in the externally initialized secure context namespace
- For job queues, restoration of context values that are in the externally initialized secure context namespace
- For OCI interfaces, a mechanism to initialize context values that are in the externally initialized secure context namespace

Although this type of namespace can be initialized by any client program using OCI, there are login event triggers that can verify the values. It is up to the application to interpret and trust the values of the attributes.

Middle-tier servers can actually initialize secure context values on behalf of database users. Context attributes are propagated for the remote session at initialization time, and the remote database accepts the values if the namespace is externally initialized.

See Also:

- *Oracle Database JDBC Developer's Guide and Reference*
- *Oracle Call Interface Programmer's Guide*

Initializing Secure Application Context Globally

This feature uses a centralized location to store the secure application context of the user, enabling applications to set up a user context during initialization based upon user identity. In particular, it supports Oracle Label Security labels and privileges. This feature makes it much easier for the administrator to manage contexts for large numbers of users and databases.

For example, many organizations want to manage user information centrally, in an LDAP-based directory. Enterprise User Security, a feature of Oracle Advanced Security, supports centralized user and authorization management in Oracle Internet Directory. However, there may be additional attributes an application wishes to retrieve from LDAP to use for VPD enforcement, such as the user title, organization, or physical location.

This section contains these topics:

Using Secure Application Context with LDAP

Secure session-based application context initialized globally uses the Lightweight Directory Access Protocol (LDAP), a standard, extensible, and efficient directory access protocol. The LDAP directory stores a list of users to which this application is assigned. An Oracle database server uses the directory service, typically Oracle Internet Directory, for authentication and authorization of enterprise users.

Note:

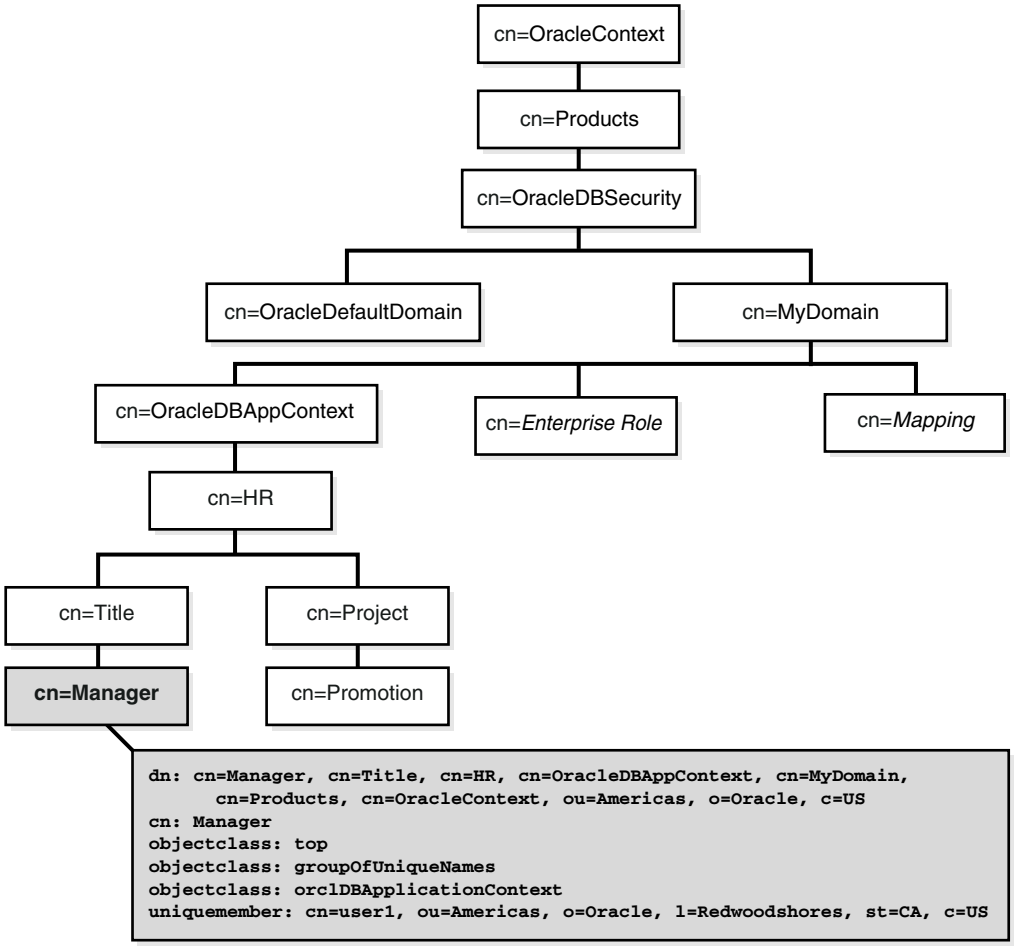
- Enterprise User Security requires Oracle Advanced Security.
- Third-party directories such as Microsoft Active Directory and Sun Microsystems iPlanet can also be used as the directory service.

The `orclDBApplicationContext` LDAP object (a subclass of `groupOfUniqueNames`) stores the application context values in the directory. The location of the application context object is described in Figure 15–1, which is based on the Human Resources example.

An internal C function is required to retrieve the `orclDBApplicationContext` value, which returns a list of application context values to the RDBMS.

Note: In this example, HR is the namespace, Title and Project are the attributes, and Manager and Promotion are the values.

Figure 15–1 Location of Application Context in LDAP Directory Information Tree



How Globally Initialized Secure Application Context Works

The administrator configures Enterprise User Security, a feature of Oracle Advanced Security. Then, the administrator sets up the secure application context values for the user in the database and the directory.

When a global user (enterprise user) connects to the database, the Oracle Advanced Security Enterprise User Security feature performs authentication to verify the identity of the user connecting to the database. After authentication, the global user roles and application context are retrieved from the directory. When the user logs on to the database, the global roles and initial application context are already set up.

> **See Also:** *Oracle Database Advanced Security Administrator's Guide* for a detailed discussion of Enterprise User Security and how to configure this feature

Example: Initializing Secure Application Context Globally

The initial application context for a user, such as department name and title, can be set up and stored in the LDAP directory. The values are retrieved during user login so that the context is set properly. In addition, any information related to the user is retrieved and stored in the `SYS_USER_DEFAULTS` application context namespace. The following example shows how this is done.

1. Create a secure application context in the database.

```
CREATE CONTEXT HR USING hrapps.hr_manage_pkg INITIALIZED GLOBALLY;
```

2. Create and add new entries in the LDAP directory.

   An example of the entries added to the LDAP directory follows. These entries create an attribute named `Title` with attribute value `Manager` for the application (namespace) `HR`, and assign user names `user1` and `user2`.

```
dn:
cn=OracleDBAppContext,cn=myDomain,cn=OracleDBSecurity,cn=Products,cn=OracleC
ontext,ou=Americas,o=oracle,c=US
changetype: add
cn: OracleDBAppContext
objectclass: top
objectclass: orclContainer

dn:
cn=HR,cn=OracleDBAppContext,cn=myDomain,cn=OracleDBSecurity,cn=Products,cn=O
racleContext,ou=Americas,o=oracle,c=US
changetype: add
cn: HR
objectclass: top
objectclass: orclContainer

dn:
cn=Title,cn=HR,cn=OracleDBAppContext,cn=myDomain,cn=OracleDBSecurity,cn=Prod
ucts,cn=OracleContext,ou=Americas,o=oracle,c=US
changetype: add
cn: Title
objectclass: top
objectclass: orclContainer

dn:
cn=Manager,cn=Title,cn=HR,cn=OracleDBAppContext,cn=myDomain,cn=OracleDBSecurity
,cn=Products,cn=OracleContext,ou=Americas,o=oracle,c=US
cn: Manager
objectclass: top
objectclass: groupofuniquenames
objectclass: orclDBApplicationContext
uniquemember: CN=user1,OU=Americas,O=Oracle,L=Redwoodshores,ST=CA,C=US
uniquemember: CN=user2,OU=Americas,O=Oracle,L=Redwoodshores,ST=CA,C=US
```

3. If an LDAP `inetOrgPerson` object entry exists for the user, then the connection will also retrieve all the attributes from `inetOrgPerson` and assign them to the namespace `SYS_LDAP_USER_DEFAULT`. The following is an example of an `inetOrgPerson` entry:

```
dn: cn=user1,ou=Americas,O=oracle,L=redwoodshores,ST=CA,C=US
changetype: add
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
cn: user1
sn: One
givenName: User
initials: UO
title: manager, product development
uid: uone
```

```
mail: uone@us.oracle.com
telephoneNumber: +1 650 123 4567
employeeNumber: 00001
employeeType: full time
```

4. Connect to the database.

   When `user1` connects to a database that belongs to the `myDomain` domain, `user1` will have his `Title` set to `Manager`. Any information related to `user1` will be retrieved from the LDAP directory. The value can be obtained using the following syntax:

   ```
   SYS_CONTEXT('namespace','attribute name')
   ```

   For example:

   ```
   DECLARE
   tmpstr1 VARCHAR2(30);
   tmpstr2 VARCHAR2(30);
   BEGIN
   tmpstr1 = SYS_CONTEXT('HR','TITLE);
   tmpstr2 = SYS_CONTEXT('SYS_LDAP_USER_DEFAULT','telephoneNumber');
   DBMS_OUTPUT.PUT_LINE('Title is ' || tmpstr1);
   DBMS_OUTPUT.PUT_LINE('Telephone Number is ' || tmpstr2);
   END;
   ```

   The output of this example is:

   ```
   Title is Manager
   Telephone Number is +1 650 123 4567
   ```

Using Client Session-Based Application Context

The `OCIAppCtx` API enables you to use client session-based application context. You can set or clear individual values for attributes in the `CLIENTCONTEXT` namespace, or clear all their values, by following the examples in the subsections that follow:

- Setting a Value in CLIENTCONTEXT on page 15-19
- Clearing a Particular Setting in CLIENTCONTEXT on page 15-20
- Clearing all Settings in CLIENTCONTEXT on page 15-20

Setting a Value in CLIENTCONTEXT

For OCI, use a command of the following form:

```
err = OCIAppCtxSet((void *) session_handle,(dvoid *)"CLIENTCONTEXT",(ub4) 13,
                   (dvoid *)attribute_name, length_of_attribute_name
                   (dvoid *)attribute_value, length_of_attribute_value, errhp,
                   OCI_DEFAULT);
```

where *`session_handle`* represents the `OCISessionHandle`, *`attribute_name`* could be, for example, "responsibility", with a length of 14, and *`attribute_value`* could be, for example, "manager", with a length of 7.

For JDBC, use a command of the following form:

```
public void setApplicationContext(String CLIENTCONTEXT, string attribute,
string value) throws SQLException;
```

where *attribute* represents the attribute whose value needs to be set, and *value* is the value to be assigned to that attribute.

Clearing a Particular Setting in CLIENTCONTEXT

For OCI, set the value to NULL or to an empty string by using one of the following command forms:

```
(void) OCIAppCtxSet((void *) session_handle, (dvoid *)"CLIENTCONTEXT", 13,
                    (dvoid *)attribute_name, length_of_attribute_name,
                    (dvoid *)0, 0,errhp,
                    OCI_DEFAULT);
```

or

```
(void) OCIAppCtxSet((void *) session_handle, (dvoid *)"CLIENTCONTEXT", 13
                    (dvoid *)attribute_name, length_of_attribute_name,
                    (dvoid *)"", 0,errhp,
                    OCI_DEFAULT);
```

For JDBC, use the following command:

```
public void setApplicationContext(String CLIENTCONTEXT, string attribute,
string value) throws SQLException;
```

where *attribute* represents the attribute whose value needs to be cleared and *value* is either 0 or the null string `""`.

Clearing all Settings in CLIENTCONTEXT

For OCI, use a command of the following form:

```
err = OCIAppCtxClearAll((void *) session_handle, (dvoid *)"CLIENTCONTEXT", 13,
                        errhp,
                        OCI_DEFAULT);
```

For JDBC, use a command of the following form:

```
public void clearAllApplicationContext(String CLIENTCONTEXT) throws
SQLException;
```

How to Use Global Application Context

Global application context stores context information in the System Global Area (SGA) so that it can be used for applications that use a sessionless model, such as middle-tier applications in a three-tiered architecture. These applications cannot use session-based application context because users authenticate to the application and then it typically connects to the database as a single identity. Global application context uses the `CLIENT_IDENTIFIER` `USERENV` namespace attribute, set using the `DBMS_SESSION` interface, to associate the database session with a particular user or group. The following sections explain how to use the `DBMS_SESSION` interface to set the `CLIENT_IDENTIFIER` and then provide examples:

- Using the DBMS_SESSION Interface to Manage Application Context in Client Sessions
- Examples: Global Application Context

See Also: "Introduction to Global Application Context" on page 14-14 for conceptual information about this feature

Using the DBMS_SESSION Interface to Manage Application Context in Client Sessions

The DBMS_SESSION interface for managing application context has a client identifier for each application context. In this way, application context can be managed globally, yet each client sees only his or her assigned application context. The following interfaces in DBMS_SESSION enable the administrator to manage application context in client sessions:

- SET_CONTEXT
- CLEAR_CONTEXT
- CLEAR_ALL_CONTEXT (can also be used with session-based application context)
- SET_IDENTIFIER
- CLEAR_IDENTIFIER

The middle-tier application server can use SET_CONTEXT to set application context for a specific client ID. Then, when assigning a database connection to process the client request, the application server needs to issue a SET_IDENTIFIER to denote the ID of the application session. From then on, every time the client invokes SYS_CONTEXT, only the context that was associated with the set identifier is returned.

See Also:

- *PL/SQL Packages and Types Reference* for reference information and a complete description of the DBMS_SESSION package
- "Using CLIENT_IDENTIFIER Independent of Global Application Context" on page 16-9 for information about setting this USERENV namespace attribute with the DBMS_SESSION interface

Examples: Global Application Context

This section provides two examples that use global application context.

- Example 1: Global Application Context Process
- Example 2: Global Application Context for Lightweight Users

Example 1: Global Application Context Process

The following steps outline the global application context process:

1. Consider the application server, AppSvr, that has assigned the client identifier 12345 to client SCOTT. It then issues the following statement to indicate that, for this client identifier, there is an application context called RESPONSIBILITY with a value of 13 in the HR namespace.

   ```
   DBMS_SESSION.SET_CONTEXT( 'HR', 'RESPONSIBILITY' , '13', 'SCOTT', '12345' );
   ```

   Note that HR must be a global context namespace created as follows:

   ```
   CREATE CONTEXT hr USING hr.init ACCESSED GLOBALLY;
   ```

2. Then, the following command is issued to indicate the connecting client identity each time SCOTT uses AppSvr to connect to the database:

```
DBMS_SESSION.SET_IDENTIFIER('12345');
```

3. When there is a `SYS_CONTEXT('HR','RESPONSIBILITY')` call within the database session, the database engine matches the client identifier `12345` to the global context, and returns the value `13`.
4. When exiting this database session, `AppSvr` clears the client identifier by issuing:

```
DBMS_SESSION.CLEAR_IDENTIFIER( );
```

> **Note:** After a client identifier in a session is cleared, it takes on a `NULL` value. This implies that subsequent `SYS_CONTEXT` calls only retrieve application contexts with `NULL` client identifiers, until the client identifier is set again using the `SET_IDENTIFIER` interface.

Example 2: Global Application Context for Lightweight Users

The following steps outline the global application context process for a lightweight user application:

1. The administrator creates the global context namespace by using the following statement:

```
CREATE CONTEXT hr USING hr.init ACCESSED GLOBALLY;
```

2. The `HR` application server (`AppSvr`) starts up and establishes multiple connections to the `HR` database as the `APPSMGR` user.
3. User `SCOTT` logs in to the `HR` application server.
4. `AppSvr` authenticates `SCOTT` to the application.
5. `AppSvr` assigns a temporary session ID (or simply uses the application user ID), `12345`, for this connection.
6. The session ID is returned to the browser used by `SCOTT` as part of a cookie or maintained by `AppSvr`.

> **Note:** If the application generates a session ID for use as a `CLIENT_IDENTIFIER`, then the session ID must be suitably random and protected over the network by encryption. If the session ID is not random, then a malicious user could guess the session ID and access the data of another user. If the session ID is not encrypted over the network, then a malicious user could retrieve the session ID and access the connection.

7. `AppSvr` initializes application context for this client by calling the `HR.INIT` package, which issues the following statements:

```
DBMS_SESSION.SET_CONTEXT( 'hr', 'id', 'scott', 'APPSMGR', 12345 );
DBMS_SESSION.SET_CONTEXT( 'hr', 'dept', 'sales', 'APPSMGR', 12345 );
```

8. `AppSvr` assigns a database connection to this session and initializes the session by issuing the following statement:

```
DBMS_SESSION.SET_IDENTIFIER( 12345 );
```

9. All SYS_CONTEXT calls within this database session will return application context values belonging only to the client session. For example, SYS_CONTEXT('hr','id') will return the value SCOTT.
10. When done with the session, AppSvr can issue the following statement to clean up the client identity:

```
DBMS_SESSION.CLEAR_IDENTIFIER ( );
```

Note that even if another database user (ADAMS) had logged into the database, he cannot access the global context set by AppSvr because AppSvr has specified that only the application with logged in user APPSMGR can see it. If AppSvr has used the following, then any user session with client ID set to 12345 can see the global context.

```
DBMS_SESSION.SET_CONTEXT( 'hr', 'id', 'scott', NULL , 12345 );
DBMS_SESSION.SET_CONTEXT( 'hr', 'dept', 'sales', NULL , 12345 );
```

This approach enables different users to share the same context.

> **Note:** Users must be aware of the security implication of different settings of the global context. NULL in the user name means that any user can access the global context. A NULL client ID in the global context means that only a session with an uninitialized client ID can access the global context.

Users can query the client identifier set in the session as follows:

```
SYS_CONTEXT('USERENV','CLIENT_IDENTIFIER')
```

The DBA can see which sessions have the client identifier set by querying the V$SESSION view for the CLIENT_IDENTIFIER and USERNAME.

When users want to see the amount of global context area (in bytes) being used, they can use SYS_CONTEXT('USERENV','GLOBAL_CONTEXT_MEMORY').

> **See Also:** For more information about using the CLIENT_IDENTIFIER predefined attribute of the USERENV application context:
>
> - "Using the CLIENT_IDENTIFIER Attribute to Preserve User Identity" on page 16-8
> - *Oracle Database SQL Reference*
> - *PL/SQL Packages and Types Reference*
> - *Oracle Database JDBC Developer's Guide and Reference*
> - *Oracle Call Interface Programmer's Guide*

How Fine-Grained Access Control Works

Fine-grained access control is based on dynamically modified statements. Suppose you want to associate the ORDERS_TAB table with the following security policy: Customers can see only their own orders. The process is described in this section.

1. Create a function to add a predicate to a DML statement run by a user.

> **Note:** A predicate is the `WHERE` clause (a selection criterion clause) based on one of the operators (`=`, `!=`, `IS`, `IS NOT`, `>`, `>=`, `EXIST`, `BETWEEN`, `IN`, `NOT IN`, and so on). For a complete list of operators, refer to the *Oracle Database SQL Reference*.

In this case, you might create a function that adds the following predicate:

```
Cust_no = (SELECT Custno FROM Customers WHERE Custname =
           SYS_CONTEXT ('userenv','session_user'))
```

2. A user enters the following statement:

```
SELECT * FROM Orders_tab;
```

3. The Oracle Database server calls the function you created to implement the security policy.
4. The function dynamically modifies the statement entered by the user to read as follows:

```
SELECT * FROM Orders_tab WHERE Custno = (
   SELECT Custno FROM Customers
       WHERE Custname = SYS_CONTEXT('userenv', 'session_user'))
```

5. The Oracle Database server runs the dynamically modified statement.

Upon execution, the function employs the user name returned by `SYS_CONTEXT ('userenv','session_user')` to look up the corresponding customer and to limit the data returned from the `ORDERS_TAB` table to that associated with that particular customer only.

> **See Also:** For more information on using fine-grained access control:
>
> - "Introduction to Fine-Grained Access Control" on page 14-4
> - "Introduction to Global Application Context" on page 14-14
> - *PL/SQL Packages and Types Reference*

How to Establish Policy Groups

A policy group is a set of security policies that belong to an application. You can designate an application context (known as a driving context) to indicate the policy group in effect. Then, when the table, view, or synonym column is accessed, the server looks up the driving context (which is also known as policy context) to determine the policy group in effect. It enforces all the associated policies which belong to that policy group.

This section contains the following topics:

The Default Policy Group: SYS_DEFAULT

In the Oracle Policy Manager tree structure, the Fine-Grained Access Control Policies folder contains the Policy Groups folder. The Policy Groups folder contains an icon for each policy group, as well as an icon for the SYS_DEFAULT policy group.

By default, all policies belong to the SYS_DEFAULT policy group. Policies defined in this group for a particular table, view, or synonym will always be executed along with the policy group specified by the driving context. The SYS_DEFAULT policy group may or may not contain policies. If you attempt to drop the SYS_DEFAULT policy group, then an error will be raised.

If, to the SYS_DEFAULT policy group, you add policies associated with two or more objects, then each such object will have a separate SYS_DEFAULT policy group associated with it. For example, the EMP table in the SCOTT schema has one SYS_DEFAULT policy group, and the DEPT table in the SCOTT schema has a different SYS_DEFAULT policy group associated with it. These are displayed in the tree structure as follows:

```
SYS_DEFAULT
  - policy1 (SCOTT/EMP)
  - policy3 (SCOTT/EMP)
SYS_DEFAULT
  - policy2 (SCOTT/DEPT)
```

Note: Policy groups with identical names are supported. When you select a particular policy group, its associated schema and object name are displayed in the property sheet on the right-hand side of the screen.

New Policy Groups

When adding the policy to a table, view, or synonym, you can use the DBMS_RLS.ADD_GROUPED_POLICY interface to specify the group to which the policy belongs. To specify which policies will be effective, you add a driving context using the DBMS_RLS.ADD_POLICY_CONTEXT interface. If the driving context returns an unknown policy group, then an error is returned.

If the driving context is not defined, then all policies are executed. Likewise, if the driving context is NULL, then policies from all policy groups are enforced. In this way, an application accessing the data cannot bypass the security setup module (which sets up application context) to avoid any applicable policies.

You can apply multiple driving contexts to the same table, view, or synonym, and each of them will be processed individually. In this way, you can configure multiple active sets of policies to be enforced.

Consider, for example, a hosting company that hosts Benefits and Financial applications, which share some database objects. Both applications are striped for hosting using a SUBSCRIBER policy in the SYS_DEFAULT policy group. Data access is partitioned first by subscriber ID, then by whether the user is accessing the Benefits or Financial applications (determined by a driving context). Suppose that Company A, which uses the hosting services, wants to apply a custom policy which relates only to its own data access. You could add an additional driving context (such as COMPANY A SPECIAL) to ensure that the additional, special policy group is applied for data access for Company A only. You would not apply this under the SUBSCRIBER policy, because the policy relates only to Company A, and it is more efficient to segregate the basic hosting policy from other policies.

How to Implement Policy Groups

To create policy groups, the administrator must do two things:

- Set up a driving context to identify the effective policy group.
- Add policies to policy groups as required.

The following example shows how to perform these tasks.

Note: You need to set up the following data structures for the examples in this section to work:

```
DROP USER finance CASCADE;
CREATE USER finance IDENTIFIED BY welcome2;
GRANT RESOURCE TO apps;
DROP TABLE apps.benefit;
CREATE TABLE apps.benefit (c NUMBER);
```

Step 1: Set Up a Driving Context

Begin by creating a namespace for the driving context. For example:

```
CREATE CONTEXT appsctx USING apps.apps_security_init;
```

Create the package that administers the driving context. For example:

```
CREATE OR REPLACE PACKAGE apps.apps_security_init IS
PROCEDURE setctx (policy_group VARCHAR2);
END;

CREATE OR REPLACE PACKAGE BODY apps.apps_security_init AS
PROCEDURE setctx ( policy_group varchar2 ) IS
BEGIN

REM  Do some checking to determine the current application.
REM  You can check the proxy if using the proxy authentication feature.
REM  Then set the context to indicate the current application.
.
.
.
DBMS_SESSION.SET_CONTEXT('APPSCTX','ACTIVE_APPS', policy_group);
END;
END;
```

Define the driving context for the table `APPS.BENEFIT`.

```
BEGIN
DBMS_RLS.ADD_POLICY_CONTEXT('apps','benefit','APPSCTX','ACTIVE_APPS');
END;
```

Step 2: Add a Policy to the Default Policy Group.

Create a security function to return a predicate to divide the data by company.

```
CREATE OR REPLACE FUNCTION by_company (sch varchar2, tab varchar2)
RETURN VARCHAR2 AS
BEGIN
  RETURN 'COMPANY = SYS_CONTEXT(''ID'',''MY_COMPANY'')';
END;
```

Because policies in SYS_DEFAULT are always executed (except for SYS, or users with the EXEMPT ACCESS POLICY system privilege), this security policy (named SECURITY_BY_COMPANY), will always be enforced regardless of the application running. This achieves the universal security requirement on the table: namely, that each company should see its own data regardless of the application that is running. The function APPS.APPS_SECURITY_INIT.BY_COMPANY returns the predicate to make sure that users can only see data related to their own company:

```
BEGIN
DBMS_RLS.ADD_GROUPED_POLICY('apps','benefit','SYS_DEFAULT',
'security_by_company',
'apps','by_company');
END;
```

Step 3: Add a Policy to the HR Policy Group

First, create the HR group:

```
CREATE OR REPLACE FUNCTION hr.security_policy
RETURN VARCHAR2
AS
BEGIN
 RETURN 'SYS_CONTEXT(''ID'',''TITLE'') = ''MANAGER'' ';
END;
```

The following creates the policy group and adds a policy named HR_SECURITY to the HR policy group. The function HR.SECURITY_POLICY returns the predicate to enforce security on the APPS.BENEFIT table:

```
BEGIN
DBMS_RLS.CREATE_POLICY_GROUP('apps','benefit','HR');
DBMS_RLS.ADD_GROUPED_POLICY('apps','benefit','HR',
'hr_security','hr','security_policy');
END;
```

Step 4: Add a Policy to the FINANCE Policy Group

Create the FINANCE policy:

```
CREATE OR REPLACE FUNCTION finance.security_policy
RETURN VARCHAR2
AS
BEGIN
 RETURN ('SYS_CONTEXT(''ID'',''DEPT'') = ''FINANCE'' ');
END;
```

Create a policy group named FINANCE and add the FINANCE policy to the FINANCE group:

```
BEGIN
DBMS_RLS.CREATE_POLICY_GROUP('apps','benefit','FINANCE');
DBMS_RLS.ADD_GROUPED_POLICY('apps','benefit','FINANCE',
'finance_security','finance', 'security_policy');
END;
```

As a result, when the database is accessed, the application initializes the driving context after authentication. For example, with the HR application:

```
execute apps.security_init.setctx('HR');
```

Validating the Application Used to Connect to the Database

The package implementing the driving context must correctly validate the application that is being used to connect to the database. Although the database always checks the call stack to ensure that the package implementing the driving context sets context attributes, inadequate validation can still occur within the package.

For example, in applications where database users or enterprise users are known to the database, the user needs the `EXECUTE` privilege on the package that sets the driving context. Consider a user who knows that:

- The `BENEFITS` application allows more liberal access than the `HR` application
- The `setctx` procedure (which sets the correct policy group within the driving context) does not perform any validation to determine which application is actually connecting. That is, the procedure does not check either the IP address of the incoming connection (for a three-tier system) or the `proxy_user` attribute of the user session.

Such a user could pass to the driving context package an argument setting the context to the more liberal `BENEFITS` policy group, and then access the `HR` application instead. Because the `setctx` does no further validation of the application, this user bypasses the normally more restrictive HR security policy.

By contrast, if you implement proxy authentication with VPD, then you can determine the identity of the middle tier (and the application) that is actually connecting to the database on behalf of a user. In this way, the correct policy will be applied for each application to mediate data access.

For example, a developer using the proxy authentication feature could determine that the application (the middle tier) connecting to the database is `HRAPPSERVER`. The package that implements the driving context can thus verify whether the `proxy_user` in the user session is `HRAPPSERVER`. If so, then it can set the driving context to use the `HR` policy group. If `proxy_user` is not `HRAPPSERVER`, then it can disallow access.

In this case, when the following query is executed

```
SELECT * FROM APPS.BENEFIT;
```

Oracle Database picks up policies from the default policy group (`SYS_DEFAULT`) and active namespace `HR`. The query is internally rewritten as follows:

```
SELECT * FROM APPS.BENEFIT WHERE COMPANY = SYS_CONTEXT('ID','MY_COMPANY') and SYS_
CONTEXT('ID','TITLE') = 'MANAGER';
```

How to Add a Policy to a Table, View, or Synonym

The `DBMS_RLS` package enables you to administer security policies by using its procedures for adding, enabling, refreshing, or dropping policies, policy groups, or application contexts. You need to specify the table, view, or synonym to which you are adding a policy, as well as the data pertinent to that policy, such as the policy name. Such data also includes names for the policy group and the function implementing the policy. You can also specify the types of statements the policy controls (`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE INDEX`, or `ALTER INDEX`). Table 15–2 lists the procedures in the `DBMS_RLS` package.

Table 15–2 DBMS_RLS Procedures

Procedure	Purpose
For Handling Individual Policies	

Table 15–2 (Cont.) DBMS_RLS Procedures

Procedure	Purpose
DBMS_RLS.ADD_POLICY	To add a policy to a table, view, or synonym
DBMS_RLS.ENABLE_POLICY	To enable (or disable) a policy you previously added to a table, view, or synonym
DBMS_RLS.REFRESH_POLICY	To invalidate cursors associated with non-static policies
DBMS_RLS.DROP_POLICY	To drop a policy from a table, view, or synonym
For Handling Grouped Policies	
DBMS_RLS.CREATE_POLICY_GROUP	To create a policy group
DBMS_RLS.DELETE_POLICY_GROUP	To drop a policy group
DBMS_RLS.ADD_GROUPED_POLICY	To add a policy to the specified policy group
DBMS_RLS.ENABLE_GROUPED_POLICY	To enable a policy within a group
DBMS_RLS.REFRESH_GROUPED_POLICY	To reparse the SQL statements associated with a refreshed policy
DBMS_RLS.DISABLE_GROUPED_POLICY	To disable a policy within a group
DBMS_RLS.DROP_GROUPED_POLICY	To drop a policy which is a member of the specified group
For Handling Application Context	
DBMS_RLS.ADD_POLICY_CONTEXT	To add the context for the active application
DBMS_RLS.DROP_POLICY_CONTEXT	To drop the context for the application

Alternatively, you can use Oracle Policy Manager to administer security policies.

> **See Also:** *PL/SQL Packages and Types Reference* for information about using the DBMS_RLS package and all of its procedures and parameters

DBMS_RLS.ADD_POLICY Procedure Policy Types

The execution of policy functions can consume a significant amount of system resources. If you can minimize the number of times that policy functions must run, then you can optimize your database server performance. To avoid unnecessary policy function execution, you can choose from five different policy types, which enable you to precisely specify how and how often a policy predicate should change. You can enable these different types of policies, which are listed in Table 15–3, by setting the `policy_type` parameter of the `DBMS_RLS.ADD POLICY` procedure.

Table 15–3 DBMS_RLS.ADD_POLICY Policy Types At a Glance

Policy Types	When Policy Function Executes...	Usage Example	Shared Across Multiple Objects?
STATIC	Once, then the predicate is cached in the SGA[1]	View replacement	No
SHARED_ STATIC	Same as STATIC	Hosting environments, such as data warehouses where the same predicate must be applied to multiple database objects	Yes
CONTEXT_ SENSITIVE	▪ At statement parse time ▪ At statement execution time when the local application context has changed since the last use of the cursor	3-tier, session pooling applications where policies enforce two or more predicates for different users or groups	No
SHARED_ CONTEXT_ SENSITIVE	First time the object is reference in a database session Predicates are cached in the private session memory UGA so policy functions can be shared among objects.	Same as CONTEXT_SENSITIVE, but multiple objects can share the policy function from the session UGA	Yes
DYNAMIC	Policy function re-executes every time a policy-protected database object is accessed.	Applications where policy predicates must be generated for each query, such as time-dependent policies where users are denied access to database objects at certain times during the day	No

[1] Each execution of the same cursor could produce a different row set even for the same predicate because the predicate may filter the data differently based on attributes such as SYS_CONTEXT or SYSDATE.

Static and context sensitive policies enable you to optimize server performance, because they do not run the policy function each time protected database objects are accessed. However, Oracle recommends that before you enable policies as either static or context-sensitive, you first test them as DYNAMIC policy types, which run every time. Testing policy functions as DYNAMIC policies first enables you to observe how the policy function affects each query, because nothing is cached. This ensures that the functions work properly before you enable them as static or context-sensitive policy types to optimize performance.

Dynamic policies are the system default. If you do not specify a policy type with the DBMS_RLS.ADD_POLICY procedure, then by default your policy will be dynamic. You can specifically configure a policy to be dynamic by setting the policy_type parameter of the DBMS_RLS.ADD_POLICY procedure to DYNAMIC. Refer to Example 15–1 for the syntax.

Example 15–1 Syntax for Enabling Policy Types with DBMS_RLS.ADD_POLICY

```
DBMS_RLS.ADD_POLICY (
.
.
.
policy_type => dbms_rls.POLICY_TYPE);
```

Note: The DBMS_RLS.ADD POLICY policy_type parameter is intended to replace the static_policy parameter, which may be deprecated in future releases.

See Also: The following topics for a more detailed discussion of static and context-sensitive policies:

- "About Static Policies" on page 15-31
- "About Context-Sensitive Policies" on page 15-32

Optimizing Performance by Enabling Static and Context Sensitive Policies

In previous releases, policies were dynamic, which means that the database runs the policy function for each query or DML statement. In addition to dynamic policies, the current release of Oracle Database provides static and context-sensitive policies. These policy types provide a means to improve server performance, because they do not always rerun policy functions for each DML statement and can be shared across multiple database objects.

Note: When using shared static and shared context-sensitive policies, ensure that the policy predicate does not contain attributes that are specific to a particular database object, such as a column name.

About Static Policies

Static policy predicates are cached in SGA, so policy functions do not rerun for each query, resulting in faster performance. When you specify a static policy, the same predicate is always enforced for all users in the instance. However, each execution of the same cursor could produce a different row set even for the same predicate, because the predicate may filter the data differently based on attributes such as `SYS_CONTEXT` or `SYSDATE`.

For example, suppose you enable a policy as either a `STATIC` or `SHARED_STATIC` policy type, which appends the following predicate to all queries made against policy protected database objects:

```
WHERE dept=SYS_CONTEXT ('HR_APP','deptno')
```

Although the predicate does not change for each query, it applies to the query based on session attributes of the `SYS_CONTEXT`. In the case of the preceding example, the predicate would return only those rows where the department number matches the `deptno` attribute of the `SYS_CONTEXT`, which would be the department number of the user who is querying the policy-protected database object.

You can enable static policies by setting the `policy_type` parameter of the `DBMS_RLS.ADD_POLICY` procedure to either `STATIC` or `SHARED_STATIC`, depending on whether or not you want the policy to be shared across multiple objects. (Refer to Example 15–1 on page 15-30 for the syntax.)

When to Use Static Policies Static policies are ideal for environments where every query requires the same predicate and fast performance is essential, such as hosting environments. For these situations when the policy function appends the same predicate to every query, rerunning the policy function each time adds unnecessary overhead to the system. For example, consider a data warehouse that contains market research data for customer organizations that are competitors of each other. The warehouse must enforce the policy that each organization can see only their own market research, which is expressed by the predicate `WHERE subscriber_id=SYS_CONTEXT('customer', 'cust_num')`. Using `SYS_CONTEXT` for the application context enables the database to dynamically change the rows that are returned. There

is no need to rerun the function, so the predicate can be cached in the SGA, thus conserving system resources and improving performance.

About Context-Sensitive Policies

In contrast to static policies, context-sensitive policies do not always cache the predicate. With context-sensitive policies, the server assumes that the predicate will change after statement parse time. But if there is no change in local application context, the server does not rerun the policy function within the user session. If there has been a change in context, then the server reruns the policy function to ensure that it captures any changes to the predicate since the initial parsing. These policies are useful where different predicates should apply depending on which user is executing the query. For example, consider the case where managers should always have the predicate `WHERE group=managers` and employees should always have the predicate `WHERE empno=emp_id`.

Shared context-sensitive policies operate in the same way as regular context-sensitive policies, except they can be shared across multiple database objects. For this policy type, all objects can share the policy function from the UGA, where the predicate is cached until the local session context changes.

You can enable context-sensitive policies by setting the `policy_type` parameter of the `DBMS_RLS.ADD_POLICY` procedure to either `CONTEXT_SENSITIVE` or `SHARED_CONTEXT_SENSITIVE`. (Refer to Example 15–1 on page 15-30 for the syntax.)

When to Use Context-Sensitive Policies This type of policy is useful when a predicate need not change for a user session, but the policy must enforce two or more different predicates for different users or groups. For example, consider a `SALES_HISTORY` table with a single policy, which states that analysts can see only their own products and regional employees can see only their own region. In this case, the server must rerun the policy function each time the type of user changes. The performance gain is realized when a user can log in and issue several DML statements against the protected object without causing the server to rerun the policy function.

Note: For session pooling where multiple clients share a database session, the middle tier must reset the context during client switches.

Adding Policies for Column-Level VPD

Column-level VPD, which can be applied to a table or a view, enables you to enforce security when a security-relevant column is referenced in a query, resulting in row-level security. Column-level VPD cannot be applied to a synonym.

It can be configured to produce two distinct behaviors as follows:

- Default Behavior
- Column-masking Behavior

The following example shows a VPD policy in which sales department users should not see the salaries of people outside their own department (department number 30). The relevant columns for such a policy are `SAL` and `COMM`. First, the VPD policy function is created and then added by using the `DBMS_RLS` PL/SQL package as shown in Example 15–2.

Example 15–2 Creating and Adding a Column-Level VPD Policy

```
*/Create a policy function which does not expose salaries of employees outside the
sales department (department 30)/*

CREATE OR REPLACE FUNCTION pf1 (oowner IN VARCHAR2, ojname IN VARCHAR2)
RETURN VARCHAR2 AS
con VARCHAR2 (200);
BEGIN
   con := 'deptno=30';
   RETURN (con);
END pf1;
```

Then the policy is added with the DBMS_RLS package as follows:

```
BEGIN
   DBMS_RLS.ADD_POLICY (object_schema=>'scott', object_name=>'emp',
                        policy_name=>'sp', function_schema=>'pol_admin',
                        policy_function=>'pf1',
                        sec_relevant_cols=>'sal,comm');
END;
```

The different behaviors of column-level VPD are discussed in the following sections using Example 15–2 on page 15-33 as a starting point for discussion.

Default Behavior

The default behavior for column-level VPD is to restrict the number of rows returned for a query that references columns containing sensitive information. These security-relevant columns are specified with the sec_relevant_cols parameter of the DBMS_RLS.ADD_POLICY procedure.

For an example of column-level VPD default behavior, consider sales department users with SELECT privilege on the emp table, which is protected with the column-level VPD policy created in Example 15–2. When users run the following query:

```
SELECT ENAME, d.dname, JOB, SAL, COMM from emp e, dept d
WHERE d.deptno = e.deptno;
```

the database returns a subset of rows as follows:

```
ENAME          DNAME          JOB                 SAL          COMM
-------------- -------------- ------------ ------------ -------------
ALLEN          SALES          SALESMAN             1600           300
WARD           SALES          SALESMAN             1250           500
MARTIN         SALES          SALESMAN             1250          1400
BLAKE          SALES          MANAGER              2850
TURNER         SALES          SALESMAN             1500             0
JAMES          SALES          CLERK                 950
```

Only those rows are displayed in which the user should have access to all columns.

Column-masking Behavior

In contrast to the default behavior of column-level VPD, column-masking displays all rows, but returns sensitive column values as NULL. To include column-masking in your policy, set the sec_relevant_cols_opt parameter of the DBMS_RLS.ADD_POLICY procedure to dbms_rls.ALL_ROWS. Also set the default behavior parameter.

Example 15–3 shows column-level VPD column-masking. It uses the same VPD policy as Example 15–2 on page 15-33 but with `sec_relevant_cols_opt` specified as `dbms_rls.ALL_ROWS`.

Example 15–3 Adding a Column-level VPD Policy with Column-masking Behavior

```
*/add the ALL_ROWS policy/*
BEGIN
     DBMS_RLS.ADD_POLICY(object_schema=>'scott', object_name=>'emp',
                         policy_name=>'sp', function_schema=>'pol_admin',
                         policy_function=>'pf1',
                         sec_relevant_cols=>'sal,comm',
                         sec_relevant_cols_opt=>dbms_rls.ALL_ROWS);
END;
```

Assume that a sales department user with `SELECT` privilege on the emp table runs the following query:

```
SELECT ENAME, d.dname, JOB, SAL, COMM from emp e, dept d
WHERE d.deptno = e.deptno;
```

The database returns all rows specified in the query, but with certain values masked because of the VPD policy:

```
ENAME          DNAME          JOB                  SAL          COMM
-------------- -------------- ------------ ------------ -------------
SMITH          RESEARCH       CLERK
ALLEN          SALES          SALESMAN             1600           300
WARD           SALES          SALESMAN             1250           500
JONES          RESEARCH       MANAGER
MARTIN         SALES          SALESMAN             1250          1400
BLAKE          SALES          MANAGER              2850
CLARK          ACCOUNTING     MANAGER
SCOTT          RESEARCH       ANALYST
KING           ACCOUNTING     PRESIDENT
TURNER         SALES          SALESMAN             1500             0
ADAMS          RESEARCH       CLERK
JAMES          SALES          CLERK                 950
FORD           RESEARCH       ANALYST
MILLER         ACCOUNTING     CLERK
```

Note that column-masking has returned all rows requested by the sales user query, but has made the `SAL` and `COMM` columns `NULL` for employees outside the sales department.

The following considerations apply to column-masking:

- Column-masking applies only to `SELECT` statements.
- Column-masking conditions generated by the policy function must be a simple Boolean expressions, unlike regular VPD predicates.
- For applications that perform calculations, or do not expect `NULL` values, use standard column-level VPD, specifying `sec_relevant_cols` rather than column-masking.
- Column-masking used with `UPDATE AS SELECT` will update only the columns that users are allowed to see.
- For some queries, column-masking may prevent some rows from displaying. For example:

```
SELECT * FROM employees
```

```
WHERE salary = 10
```

This query may not return rows if the `salary` column returns a `NULL` value, because the column-masking option has been set.

> **See Also:** The chapter on the `DBMS_RLS` package in *PL/SQL Packages and Types Reference* for a discussion of parameters and usage examples for the `DBMS_RLS.ADD_POLICY` procedure

Enforcing VPD Policies on Specific SQL Statement Types

VPD policies can be enforced for `SELECT`, `INSERT`, `UPDATE`, `INDEX`, and `DELETE` statements. Specify any combination of these statement types with the `DBMS_RLS.ADD_POLICY` procedure `statement_types` parameter as follows:

```
DBMS_RLS.ADD_POLICY (
.
.
.
statement_types=>'SELECT,INDEX');
```

Enforcing Policies on Index Maintenance

A user who has privileges to maintain an index can see all the row data even if the user does not have full table access under a regular query, such as `SELECT`. For example, a user can create a function-based index that contains a user-defined function with column values as its arguments. During index creation, the server passes column values of every row into the user function, making the row data available to the user who creates the index. Administrators can enforce VPD policies on index maintenance operations by specifying `INDEX` with the `statement_types` parameter as shown in the previous section.

How to Check for Policies Applied to a SQL Statement

`V$VPD_POLICY` allows one to perform a dynamic view in order to check what policies are being applied to a SQL statement. When debugging, in your attempt to find which policy corresponds to a particular SQL statement, you should use the following table.

Table 15–4 V$VPD_POLICY

Column Name	Type
ADDRESS	RAW(4|8)
PARADDR	RAW(4|8)
SQL_HASH	NUMBER
SQL_ID	VARCHAR2(13)
CHILD_NUMBER	NUMBER
OBJECT_OWNER	VARCHAR2(30)
OBJECT_NAME	VARCHAR2(30)
POLICY_GROUP	VARCHAR2(30)
POLICY	VARCHAR2(30)
POLICY_FUNCTION_OWNER	VARCHAR2(30)
PREDICATE	VARCHAR2(4000)

Table 15–4 (Cont.) V$VPD_POLICY

Column Name	Type
DBMS_RLS.REFRESH_GROUPED_POLICY	VARCHAR2(4096)

See Also: *Oracle Database Reference* for more information about the V$VPD_POLICY view

Users Exempt from VPD Policies

Two classes of users are exempt from VPD policies: the SYS user is exempt by default, and any other user can be exempt if granted the EXEMPT ACCESS POLICY system privilege. These two cases are discussed in the following sections.

SYS User Exempted from VPD Policies

The database user SYS is always exempt from VPD or Oracle Label Security policy enforcement, regardless of the export mode, application, or utility that is used to extract data from the database. However, SYSDBA actions can be audited.

EXEMPT ACCESS POLICY System Privilege

The system privilege EXEMPT ACCESS POLICY allows a user to be exempted from all fine-grained access control policies on any SELECT or DML operation (INSERT, UPDATE, and DELETE). This provides ease of use for administrative activities such as installation and import and export of the database through a non-SYS schema.

Also, regardless of the utility or application that is being used, if a user is granted the EXEMPT ACCESS POLICY privilege, then the user is exempt from VPD and Oracle Label Security policy enforcement. That is, the user will not have any VPD or Oracle Label Security policies applied to their data access.

Because EXEMPT ACCESS POLICY negates the effect of fine-grained access control, this privilege should only be granted to users who have legitimate reasons for bypassing fine-grained access control enforcement. This privilege should not be granted WITH ADMIN OPTION, so that users cannot pass on the EXEMPT ACCESS POLICY privilege to other users, and thus propagate the ability to bypass fine-grained access control.

Automatic Reparse

Note: This feature is applicable when COMPATIBLE is set to 9.0.1.

Starting from Oracle9*i*, queries against objects enabled with fine-grained access control always run the policy function to make sure that the most current predicate is used for each policy. For example, in the case of a time-based policy function, in which queries are only allowed between 8:00 a.m. and 5:00 p.m., a cursor execution parsed at noon cause the policy function to run, ensuring that the policy is consulted again for the query.

Automatic reparse does not occur under the following conditions:

- When you specify the STATIC_POLICY=TRUE while adding the policy to indicate that the policy function always returns the same predicate.

- When you set the `_dynamic_rls_policies` parameter to `FALSE` in the initialization parameters files. Typically, this parameter is set to `FALSE` for users whose security policies do not return different predicates within a database session to reduce the execution overhead.

For deployment environments where the latest application context value is always the desired value, the `_app_ctx_vers` parameter can be set to `FALSE` in the initialization parameters file to reduce the overhead of application context scoping. By default, it is set to `TRUE` and changes of value within a SQL statement are not visible. This default may change in the future, thus developers should be careful not to allow changes of application context values within a SQL statement using a user-defined function. In general, you should not depend on the sequence of SQL statement execution, which can yield inconsistent results depending on query plans.

See Also: "Using Dynamic SQL with SYS_CONTEXT" on page 15-5

VPD Policies and Flashback Query

By default, operations on the database use the most recent committed data available. The flashback query feature enables you to query the database as it was at some time in the past. To write an application that uses flashback query, you can use the `AS OF` clause in SQL queries to specify either a time or a system change number (SCN) and then query against the committed data from the specified time. You can also use the `DBMS_FLASHBACK` PL/SQL package, which requires more code, but enables you to perform multiple operations, all of which refer to the same past time.

Flashback queries return data as it stood at the time specified in the query. However, if you use flashback query against a database object that is protected with VPD policies, then the current policies are applied to the old data. Applying the current VPD policies to flashback query data is more secure because it reflects the most current business policy.

See Also:

- *Oracle Database Application Developer's Guide - Fundamentals* for more information about the flashback query feature and how to write applications that use it
- *PL/SQL Packages and Types Reference* for more information about the `DBMS_FLASHBACK` PL/SQL package

16

Preserving User Identity in Multitiered Environments

Enforcing security in multitiered environments can be challenging. This chapter discusses the risks associated with computing environments that span multiple tiers and explains how to implement proxy authentication and use client identifiers for preserving user identity.

This chapter contains the following sections:

Security Challenges of Three-Tier Computing

While three-tier computing provides many benefits, it raises a number of security issues. These issues are described in the following sections:

Who Is the Real User?

Most organizations want to know the identity of the actual user who is accessing the database for reasons of access control, resource monitoring, or auditing. User accountability is diminished if the identity of the users cannot be traced through all tiers of the application.

Furthermore, if only the application server knows who the user is, then all security enforcement for each user must be done by the application itself. Application-based security is very expensive. If each application that accesses the data must enforce security, then security must be reimplemented in each and every application. It is often preferable to build security on the data itself, and also to have accountability for each user enforced within the database.

Does the Middle Tier Have Too Many Privileges?

Some organizations are willing to accept three-tier systems within the enterprise. In this architecture, all-privileged middle tiers, such as transaction processing (TP) monitors, can perform all actions for all users. The middle tier connects to the database

as the same user for all application users. It, therefore, needs all privileges that application users need to do their jobs.

This computing model can be undesirable on the Internet, where the middle tier resides outside, on, or just inside a firewall. More desirable, in this context, is a *limited trust model*, in which the identity of the real client is known to the data server, and the application server (or other middle tier) has a restricted privilege set.

Also useful is the ability to limit the users on whose behalf a middle tier can connect, and the roles the middle tier can assume for the user. For example, many organizations would prefer that users have different privileges depending on where they are connecting from. A user connecting to a Web server or application server on the firewall might only be able to use very minimal privileges to access data, whereas a user connecting to a Web server or application server within the enterprise might be able to exercise all privileges she is otherwise entitled to have.

How to Audit? Whom to Audit?

Accountability through auditing is a basic principle of information security. Most organizations want to know on whose behalf a transaction was accomplished, not just that a particular application server performed a transaction. A system must therefore be able to differentiate between a user performing a transaction, and an application server performing a transaction on behalf of a user.

Auditing in three-tier systems should be tied to the issue of knowing the real user: if you cannot preserve user identity through the middle tier of a three-tier application, then you cannot audit actions on behalf of the user.

What Are the Authentication Requirements for Three-Tier Systems?

In client/server systems, authentication tends to be straightforward: the client authenticates to the server. In three-tier systems authentication is more difficult, because there are many types of authentication.

- Client to Middle Tier Authentication
- Middle Tier to Database Authentication
- Client Reauthentication Through Middle Tier to Database

Client to Middle Tier Authentication

Client authentication to the middle tier is clearly required if a system must conform to basic security principles. The middle tier is typically the first gateway to useful information that the user can access. Users must, therefore, authenticate to the middle tier. Note that such authentication can be mutual, that is, the middle tier authenticates to the client just as the client authenticates to the middle tier.

Middle Tier to Database Authentication

Because the middle tier must typically initiate a connection to a database to retrieve data (whether on its own behalf or on behalf of the user), this session clearly must be authenticated. In fact, Oracle Database does not allow unauthenticated sessions. Again, middle tier to database authentication can also be mutual if using a protocol that supports it, such as SSL.

Client Reauthentication Through Middle Tier to Database

Client reauthentication from the middle tier to the database is problematic in three-tier systems. The user name might not be the same on the middle tier and the database. In

this case, users may need to reenter a user name and credential, which the middle tier uses to connect on their behalf. Or, more commonly, the middle tier may need to map the username provided to a database username. This mapping is often done in an LDAP-compliant directory service, such as Oracle Internet Directory.

For the client to reauthenticate itself to the database, the middle tier either needs to ask the user for a credential (which it must be trusted to pass to the database), or the middle tier must retrieve a credential for the user and use that to authenticate the user. Both approaches involve security risks, because the middle tier is provided with user credentials.

Reauthenticating the client to the back-end database is not always beneficial. First, two sets of authentication handshakes for each user involves considerable network overhead. Second, you must trust the middle tier to have authenticated the user. It is therefore reasonable for the database to simply accept that the middle tier has performed proper authentication. In other words, the database accepts the identity of the real client without requiring the real client to authenticate herself.

For some authentication protocols, client reauthentication is just not possible. For example, many browsers and application servers support the Secure Sockets Layer (SSL) protocol. Both the Oracle Database (through Oracle Advanced Security) and Oracle Application Server support the use of SSL for client authentication. However, SSL is a point-to-point protocol, not an end-to-end protocol. It cannot be used to reauthenticate a browser client (through the middle tier) to the database.

In short, organizations deploying three-tier systems require flexibility for the reauthentication of the client.

Oracle Database Solutions for Preserving User Identity

Many organizations would like to know who the user is through all tiers of an application without sacrificing the benefits of a middle tier. Oracle Database supports the following ways of preserving user identity through the middle tier of an application:

- Proxy Authentication

  Oracle Database provides proxy authentication in OCI or thick JDBC for database users or enterprise users. Enterprise users are those who are managed in Oracle Internet Directory and who access a shared schema in the database.

- Client Identifiers

  Oracle Database provides the `CLIENT_IDENTIFIER` attribute of the built-in `USERENV` application context namespace for application users. These users are known to an application but unknown to the database. The `CLIENT_IDENTIFIER` attribute can be used to capture any value that the application uses for identification or access control and pass it to the database. `CLIENT_IDENTIFIER` is supported in OCI, thick JDBC, and thin JDBC.

Proxy Authentication

The following sections explain how proxy authentication works and how to use it:

- Passing Through the Identity of the Real User by Using Proxy Authentication
- Limiting the Privilege of the Middle Tier
- Reauthenticating the User Through the Middle Tier to the Database
- Auditing Actions Taken on Behalf of the Real User

- Advantages of Proxy Authentication

Passing Through the Identity of the Real User by Using Proxy Authentication

For enterprise users or database users, OCI or thick JDBC enables a middle tier to set up a number of user sessions within a single database connection, each of which uniquely identifies a connected user. This is commonly referred to as *connection pooling*. These sessions reduce the network overhead of creating separate network connections from the middle tier to the database.

Authentication Process from Clients through Middle Tiers to the Database The full authentication sequence from the client to the middle tier to the database occurs as follows:

1. The client authenticates to the middle tier, using whatever form of authentication the middle tier will accept. For example, the client could authenticate to the middle tier by using a username and password or an X.509 certificate by means of SSL.
2. The middle tier authenticates itself to the database by using whatever form of authentication the database accepts. This could be a password or an authentication mechanism supported by Oracle Advanced Security, such as a **Kerberos ticket** or an X.509 certificate (SSL).
3. The middle tier then creates one or more sessions for users using OCI or thick JDBC.
   - If the user is a database user, then the session must, as a minimum, include the database user name. If the database requires it, then the session may also include a password (which the database verifies against the password store in the database). The session may also include a list of database roles for the user.
   - If the user is an enterprise user, then the session may provide different information depending on how the user is authenticated. For example:
     - If the user authenticates to the middle tier using SSL, then the middle tier can provide the DN from the X.509 certificate of the user, or the certificate itself in the session. The database uses the DN to look up the user in Oracle Internet Directory.
     - If the user is a password-authenticated enterprise user, then the middle tier must provide, as a minimum, a globally unique name for the user. The database uses this name to look up the user in Oracle Internet Directory. If the session also provides a password for the user, then the database will verify the password against Oracle Internet Directory. User roles are automatically retrieved from Oracle Internet Directory after the session is established.
   - The middle tier may optionally provide a list of database roles for the client. These roles are enabled if the proxy is authorized to exercise the roles on behalf of the client.
4. The database verifies that the middle tier has the privilege to create sessions on behalf of the user.

   The `OCISessionBegin` call will fail if the application server is not allowed to proxy on behalf of the client by the administrator, or if the application server is not allowed to activate the specified roles.

Limiting the Privilege of the Middle Tier

Least privilege is the principle that users should have the fewest privileges necessary to perform their duties and no more. As applied to middle tier applications, this means that the middle tier should not have more privileges than it needs. The Oracle Database enables you to limit the middle tier such that it can connect only on behalf of certain database users, using only specific database roles. You can limit the *privilege* of the middle tier to connect on behalf of an enterprise user, stored in an LDAP directory, by granting to the middle tier the privilege to connect as the mapped database user. For instance, if the enterprise user is mapped to the `APPUSER` schema, then you must at least grant to the middle tier the ability to connect on behalf of `APPUSER`. Otherwise, attempts to create a session for the enterprise user will fail.

However, you cannot limit the *ability* of the middle tier to connect on behalf of enterprise users. For example, suppose that user Sarah wants to connect to the database through a middle tier, `appsrv` (which is also a database user). Sarah has multiple roles, but it is desirable to restrict the middle tier to exercise only the `clerk` role on her behalf.

A DBA could effectively grant permission for `appsrv` to initiate connections on behalf of Sarah using her `clerk` role only, using the following syntax:

```
ALTER USER sarah GRANT CONNECT THROUGH appsrv WITH ROLE clerk;
```

By default, the middle tier cannot create connections for any client. The permission must be granted for each user.

To allow `appsrv` to use all of the roles granted to the client Sarah, the following statement would be used:

```
ALTER USER sarah GRANT CONNECT THROUGH appsrv;
```

Each time a middle tier initiates an OCI or thick JDBC session for another database user, the database verifies that the middle tier is authorized to connect for that user by using the role specified.

Note: Instead of using default roles, create your own and assign only necessary privileges to them. Creating your own roles enables you to control the privileges granted by them and protects you if Oracle changes or removes default roles. For example, the CONNECT role now has only the `CREATE SESSION` privilege, the one most directly needed when connecting to a database.

However, CONNECT formerly provided several additional privileges, often not needed or appropriate for most users. Extra privileges can endanger the security of your database and applications. These have now been removed from `CONNECT`, and both `CONNECT` and `RESOURCE` roles will be deprecated in future Oracle Database versions

---

Reauthenticating the User Through the Middle Tier to the Database

Administrators can specify that authentication is required by using the `AUTHENTICATION REQUIRED` proxy clause with the `ALTER USER` SQL statement. In this case, the middle tier must provide user authentication credentials.

For example, suppose that user Sarah wants to connect to the database through a middle tier, `appsrv`. A DBA could require that `appsrv` provides authentication credentials for Sarah by using the following syntax:

```
ALTER USER sarah GRANT CONNECT THROUGH appsrv AUTHENTICATION REQUIRED;
```

The AUTHENTICATION REQUIRED clause ensures that authentication credentials for the user must be presented when the user is authenticated through the specified proxy.

> **Note:** For backward compatibility, if a DBA uses the AUTHENTICATED USING PASSWORD proxy clause, then the system transforms it to AUTHENTICATION REQUIRED.

Using Password-Based Proxy Authentication When using password-based proxy authentication, the password of the client is passed to the middle-tier server. The middle-tier server then passes the password as an attribute to the data server for verification. The main advantage to this is that the client machine does not have to have Oracle software actually installed on it to perform database operations.

To pass over the password of the client, the middle-tier server calls the OCIAttrSet() function with the following pseudo interface:

```
OCIAttrSet (OCISession *session_handle,
OCI_HTYPE_SESSION,
lxstp *password,
(ub4) 0,
OCI_ATTR_PASSWORD,
OCIError *error_handle);
```

Using Proxy Authentication with Enterprise Users If the middle tier connects to the database as a client who is an enterprise user, then either the distinguished name, or the X.509 certificate containing the distinguished name is passed over instead of the database user name. If the user is a password-authenticated enterprise user, then the middle tier must provide, as a minimum, a globally unique name for the user. The database uses this name to look up the user in Oracle Internet Directory.

To pass over the distinguished name of the client, the application server would call OCIAttrSet() with the following pseudo interface.

```
OCIAttrSet(OCISession *session_handle,
OCI_HTYPE_SESSION,
lxstp *distinguished_name,
(ub4) 0,
OCI_ATTR_DISTINGUISHED_NAME,
OCIError *error_handle);
```

To pass over the entire certificate, the middle tier would use the following pseudo interface:

```
OCIAttrSet(OCISession *session_handle,
OCI_HTYPE_SESSION,
ub1 *certificate,
ub4 certificate_length,
OCI_ATTR_CERTIFICATE,
OCIError *error_handle);
```

If the type is not specified, then the server will use its default certificate type of X.509.

Note:

- OCI_ATTR_CERTIFICATE is DER encoded.
- Certificate based proxy authentication using OCI_ATTR_CERTIFICATE will not be supported in future Oracle Database releases. Use the OCI_ATTR_DISTINGUISHED_NAME or OCI_ATTR_USERNAME attribute instead

If using proxy authentication for password-authenticated enterprise users, then use the same OCI attributes as for database users authenticated by password (OCI_ATTR_USERNAME). The database first checks the user name against the database. If no user is found, then the database checks the user name in the directory. This user name must be globally unique.

Auditing Actions Taken on Behalf of the Real User

The proxy authentication features of Oracle Database enable you to audit actions that a middle tier performs on behalf of a user. For example, suppose an application server hrappserver creates multiple sessions for users Ajit and Jane. A DBA could enable auditing for SELECTs on the bonus table that hrappserver initiates for Jane as follows:

```
AUDIT SELECT TABLE BY hrappserver ON BEHALF OF Jane;
```

Alternatively, the DBA could enable auditing on behalf of multiple users (in this case, both Jane and Ajit) connecting through a middle tier as follows:

```
AUDIT SELECT TABLE BY hrappserver ON BEHALF OF ANY;
```

This auditing option only audits SELECT statements being initiated by hrappserver on behalf of other users. A DBA can enable separate auditing options to capture SELECTs against the bonus table from clients connecting directly to the database:

```
AUDIT SELECT TABLE;
```

For audit actions taken on behalf of the real user, you cannot audit CONNECT ON BEHALF OF *DN*, because the user in the LDAP directory is not known to the database. However, if the user accesses a shared schema (for example, APPUSER), then you can audit CONNECT ON BEHALF OF APPUSER.

See Also: Chapter 12, "Configuring and Administering Auditing"

Advantages of Proxy Authentication

In multitier environments, proxy authentication enables you to control the security of middle-tier applications by preserving client identities and privileges through all tiers and by auditing actions taken on behalf of clients. For example, this feature allows the identity of a user using a Web application (which acts as a *proxy*) to be passed through the application to the database server.

Three-tier systems provide many benefits to organizations.

- Organizations can separate application logic from data storage, partitioning the former in application servers and the latter in databases.
- Application servers and Web servers enable users to access data stored in databases.

- Users like using a familiar, easy-to-use browser interface.
- Organizations can also lower their cost of computing by replacing many *fat clients* with a number of *thin clients* and an application server.

In addition, Oracle proxy authentication provides the following security benefits:

- A limited trust model, by controlling the users on whose behalf middle tiers can connect and the roles that the middle tiers can assume for the user
- Scalability, by supporting user sessions through OCI and thick JDBC, and eliminating the overhead of reauthenticating clients
- Accountability, by preserving the identity of the real user through to the database, and enabling auditing of actions taken on behalf of the real user
- Flexibility, by supporting environments in which users are known to the database, and in which users are merely application users of which the database has no awareness

---

Note: Oracle Database supports this proxy authentication functionality in three tiers only. It does not support it across multiple middle tiers.

Client Identifiers

The following sections explain how using client identifiers works and how to use them:

- Support for Application User Models by Using Client Identifiers
- Using the CLIENT_IDENTIFIER Attribute to Preserve User Identity
- Using CLIENT_IDENTIFIER Independent of Global Application Context

Support for Application User Models by Using Client Identifiers

Many applications use session pooling to set up a number of sessions to be reused by multiple application users. Users authenticate themselves to a middle-tier application, which uses a single identity to log in to the database and maintains all the user connections. In this model, application users are users who are authenticated to the middle tier of an application, but who are not known to the database. Oracle Database supports use of a `CLIENT_IDENTIFIER` attribute that acts like an application user proxy for these types of applications.

In this model, the middle tier passes a client identifier to the database upon the session establishment. The client identifier could actually be anything that represents a client connecting to the middle tier, for example, a cookie or an IP address. The client identifier, representing the application user, is available in user session information and can also be accessed with an application context (by using the `USERENV` naming context). In this way, applications can set up and reuse sessions, while still being able to keep track of the *application user* in the session. Applications can reset the client identifier and thus reuse the session for a different user, enabling high performance.

Using the CLIENT_IDENTIFIER Attribute to Preserve User Identity

The `CLIENT_IDENTIFIER`, a predefined attribute of the built-in application context namespace, `USERENV`, can be used to capture the application user name for use with global application context or it can be used independently. When used independent of global application context, `CLIENT_IDENTIFIER` can be set with the `DBMS_SESSION`

interface. The ability to pass a CLIENT_IDENTIFIER to the database is supported in OCI and thick JDBC.

When CLIENT_IDENTIFIER is used with global application context, it provides flexibility and high performance for building applications. For example, suppose a Web-based application that provides information to business partners has three types of users: gold partner, silver partner, and bronze partner, representing different levels of information available. Instead of each user having his own session set up with individual application contexts, the application could set up global applications contexts for gold partners, silver partners, and bronze partners. Then, use the CLIENT_IDENTIFIER to point the session at the correct context in order to retrieve the appropriate type of data. The application need only initialize the three global contexts once and use the CLIENT_IDENTIFIER to access the correct application context to limit data access. This provides performance benefits through session reuse and through accessing global application contexts set up once, instead of having to initialize application contexts for each session individually.

> **See Also:** "How to Use Global Application Context" on page 15-20 for a discussion about implementing global application contexts and an example of using the CLIENT_IDENTIFIER attribute with it

Using CLIENT_IDENTIFIER Independent of Global Application Context

Using the CLIENT_IDENTIFIER attribute is especially useful for those applications in which the users are unknown to the database. In these situations, the application typically connects as a single database user and all actions are taken as that user. Because all user sessions are created as the same user, this security model makes it very difficult to achieve data separation for each user. These applications can use the CLIENT_IDENTIFIER attribute to preserve the real application user identity through to the database.

With this approach, sessions can be reused by multiple users by changing the value of the CLIENT_IDENTIFIER attribute, which is used to capture the name of the real application user. This avoids the overhead of setting up a separate session and separate attributes for each user and enables reuse of sessions by the application. When the CLIENT_IDENTIFIER attribute value changes, the change is piggybacked on the next OCI or thick JDBC call for additional performance benefits.

For example, a user Daniel connects to a Web Expense application. Daniel is not a database user, he is a typical Web Expense application user. The application accesses the built-in application context namespace and sets DANIEL as the CLIENT_IDENTIFIER attribute value. Daniel completes his Web Expense form and exits the application. Then Ajit connects to the Web Expense application. Instead of setting up a new session for Ajit, the application reuses the session that currently exists for Daniel, by changing the CLIENT_IDENTIFIER to AJIT. This avoids the overhead of setting up a new connection to the database and the overhead of setting up a global application context. The CLIENT_IDENTIFIER attribute can be set to any value on which the application bases access control. It does not have to be the application user name.

To use the DBMS_SESSION package to set and clear the CLIENT_IDENTIFIER on the middle tier, use the following interfaces:

- SET_IDENTIFIER
- CLEAR_IDENTIFIER

The middle tier uses `SET_IDENTIFIER` to associate the database session with a particular user or group. Then, the `CLIENT_IDENTIFIER` is an attribute of the session and can be viewed in session information.

To set the `CLIENT_IDENTIFIER` attribute with OCI, use the `OCI_ATTR_CLIENT_IDENTIFIER` attribute in the call to `OCIAttrSet()`. Then, on the next request to the server, the information is propagated and stored in the server sessions. For example:

```
OCIAttrSet (session,
OCI_HTYPE_SESSION,
(dvoid *) "appuser1",
(ub4)strlen("appuser1"),
OCI_ATTR_CLIENT_IDENTIFIER,
OCIError *error_handle);
```

For applications that use JDBC, in a connection pooling environment, the client identifier can be used to identify which light-weight user is currently using the database session. To set the `CLIENT_IDENTIFIER` for JDBC applications, use the following `oracle.jdbc.OracleConnection` interface methods:

- `setClientIdentifier()`: Sets the client identifier for a connection
- `clearClientIdentifier()`: Clears the client identifier for a connection

See Also:

- The chapter on the `DBMS_SESSION` interface in the *PL/SQL Packages and Types Reference*
- The section on `OCI_ATTR_CLIENT_IDENTIFIER` user session handle attribute in the *Oracle Call Interface Programmer's Guide*
- The section on the `oracle.jdbc.OracleConnection` interface in the *Oracle Database JDBC Developer's Guide and Reference* for information about the `setClientIdentifer()` and the `clearClientIdentifier()` methods

17

Developing Applications Using Data Encryption

In addition to controlling access, you can also encrypt data to reduce your security risks. However, data encryption is not an infallible solution. This chapter discusses the appropriate uses of data encryption and provides examples of using data encryption in applications. It contains the following topics:

Securing Sensitive Information

While the Internet poses new challenges in information security, many of them can be addressed by the traditional arsenal of security mechanisms:

- Strong user authentication to identify users
- Granular access control to limit what users can see and do
- Auditing for accountability
- Network encryption to protect the confidentiality of sensitive data in transmission

Encryption is an important component of several of these solutions. For example, Secure Sockets Layer (SSL), an Internet-standard network encryption and authentication protocol, uses encryption to strongly authenticate users by means of X.509 digital certificates. SSL also uses encryption to ensure data confidentiality, and cryptographic checksums to ensure data integrity. Many of these uses of encryption are relatively transparent to a user or application. For example, many browsers support SSL, and users generally do not need to do anything special to enable SSL encryption.

Oracle has provided network encryption between database clients and the Oracle database since version 7. Oracle Advanced Security, an option to the Oracle Database server, provides encryption and cryptographic checksums for integrity checking with any protocol supported by the database, including Oracle Net, Java Database Connectivity (JDBC—both thick and thin JDBC), and the Internet Intra-Orb Protocol

(IIOP). Oracle Advanced Security also supports SSL for Oracle Net, thick JDBC, and IIOP connections.

While encryption is not a security cure-all, it is an important tool in addressing specific security threats. In particular, the rapid growth of e-business has spurred increased encryption of stored data, such as credit card numbers. While SSL is typically used to protect these numbers in transit to a Web site, where data is not protected as it is in storage, the file system or database storing them often does so as clear text (unencrypted). Information stored in the clear is then directly accessible to anyone who can break into the host and gain root access, or gain illicit access to the database.

Databases can be made quite secure through proper configuration, but they can also be vulnerable to host break-ins if the host is misconfigured. In well-publicized break-ins, a hacker obtained a large list of credit card numbers by breaking into a database. Had the data been encrypted, the stolen information would have been useless. Encryption of stored data can thus be an important tool in limiting information loss even in the normally rare occurrence that access controls are bypassed.

Principles of Data Encryption

While there are many good reasons to encrypt data, there are many reasons not to. Encryption does not solve all security problems, and may even make some problems worse. The following sections describe some misconceptions about encryption of stored data:

- Principle 1: Encryption Does Not Solve Access Control Problems
- Principle 2: Encryption Does Not Protect Against a Malicious DBA
- Principle 3: Encrypting Everything Does Not Make Data Secure

Principle 1: Encryption Does Not Solve Access Control Problems

Most organizations need to limit data access to those who have a need to know. For example, a human resources system may limit employees to viewing only their own employment records, while allowing managers of employees to see the employment records of subordinates. Human resource specialists may also need to see employee records for multiple employees.

This type of security policy limiting data access to those with a need to see it is typically addressed by access control mechanisms. Oracle Database has provided strong, independently-evaluated access control mechanisms for many years. It enables access control enforcement to an extremely fine level of granularity through its Virtual Private Database capability.

Because human resource records are considered sensitive information, it is tempting to think that all information should be encrypted for better security. However, encryption cannot enforce granular access control, and it may actually hinder data access. For example, an employee, his manager, and a human resources clerk may all need to access an employee record. If all employee data is encrypted, then all three must be able to access the data in un-encrypted form. Therefore, the employee, the manager and the HR clerk would have to share the same encryption key to decrypt the data. Encryption would therefore not provide any additional security in the sense of better access control, and the encryption might actually hinder the proper or efficient functioning of the application. An additional issue is that it is very difficult to securely transmit and share encryption keys among multiple users of a system.

A basic principle behind encrypting stored data is that it must not interfere with access control. For example, a user who has the SELECT privilege on EMP should not be limited by the encryption mechanism from seeing all the data he is otherwise allowed to see. Similarly, there is little benefit to encrypting part of a table with one key and part of a table with another key if users need to see all encrypted data in the table. It merely adds to the overhead of decrypting the data before users can read it. If access controls are implemented well, then encryption adds little additional security within the database itself. Any user who has privilege to access data within the database has no more nor any less privilege as a result of encryption. Therefore, encryption should never be used to solve access control problems.

Principle 2: Encryption Does Not Protect Against a Malicious DBA

Some organizations, concerned that a malicious user might gain elevated (DBA) privileges by guessing a password, like the idea of encrypting stored data to protect against this threat. However, the correct solution to this problem is to protect the DBA account, and to change default passwords for other privileged accounts. The easiest way to break into a database is by using a default password for a privileged account that an administrator has allowed to remain unchanged. One example is SYS/CHANGE_ON_INSTALL.

While there are many destructive things a malicious user can do to a database after gaining DBA privilege, encryption will not protect against many of them. Examples include corrupting or deleting data, exporting user data to the file system to mail the data back to himself so as to run a password cracker on it, and so on.

Some organizations are concerned that DBAs, typically having all privileges, are able to see all data in the database. These organizations feel that the DBAs should merely administer the database, but should not be able to see the data that the database contains. Some organizations are also concerned about concentrating so much privilege in one person, and would prefer to partition the DBA function, or enforce two-person access rules.

It is tempting to think that encrypting all data (or significant amounts of data) will solve these problems, but there are better ways to protect against these threats. For example, Oracle Database does support limited partitioning of DBA privileges. Oracle Database provides native support for SYSDBA and SYSOPER users. SYSDBA has all privileges, but SYSOPER has a limited privilege set (such as startup and shutdown of the database).

Furthermore, an organization can create smaller roles encompassing a number of system privileges. A JR_DBA role might not include all system privileges, but only those appropriate to a junior DBA (such as CREATE TABLE, CREATE USER, and so on).

Oracle Database also enables auditing the actions taken by SYS (or SYS-privileged users) and storing that audit trail in a secure operating system location. Using this model, a separate auditor who has root privileges on the operating system can audit all actions by SYS, enabling the auditor to hold all DBAs accountable for their actions.

> **See Also:** "Auditing Administrative Users" on page 12-3 for information about using the AUDIT_SYS_OPERATIONS parameter.

The DBA function by its nature is a trusted position. Even organizations with the most sensitive data such as intelligence agencies do not typically partition the DBA function. Instead, they vet their DBAs strongly, because it is a position of trust. Periodic auditing can help to uncover inappropriate activities.

Encryption of stored data must not interfere with the administration of the database, because otherwise, larger security issues can result. For example, if by encrypting data you corrupt the data, then you create a security problem, the data itself becomes uninterpretable, and it may not be recoverable.

Encryption can be used to limit the ability of a DBAor other privileged user to see data in the database. However, it is not a substitute for vetting a DBA properly, or for controlling the use of powerful system privileges. If untrustworthy users have significant privileges, then they can pose multiple threats to an organization, some of them far more significant than viewing unencrypted credit card numbers.

Principle 3: Encrypting Everything Does Not Make Data Secure

A common error is to think that if encrypting some data strengthens security, then encrypting everything makes all data secure.

As the discussion of the prior two principles illustrates, encryption does not address access control issues well, and it is important that encryption not interfere with normal access controls. Furthermore, encrypting an entire production database means that all data must be decrypted to be read, updated, or deleted. Encryption is inherently a performance-intensive operation, encrypting all data will significantly affect performance.

Availability is a key aspect of security. If encrypting data makes data unavailable, or adversely affects availability by reducing performance, then encrypting everything will create a new security problem. Availability is also adversely affected by the database being inaccessible when encryption keys are changed, as good security practices require on a regular basis. When the keys are to be changed, the database is inaccessible while data is decrypted and re-encrypted with a new key or keys.

However, there may be advantages to encrypting data stored off-line. For example, an organization may store backups for a period of six months to a year off-line, in a remote location. Of course, the first line of protection is to secure the facility storing the data, by establishing physical access controls. Encrypting this data before it is stored may provide additional benefits. Because it is not being accessed on-line, performance need not be a consideration. While an Oracle database server does not provide this capability, there are vendors who can provide such encryption services. Before embarking on large-scale encryption of backup data, organizations considering this approach should thoroughly test the process. It is essential to verify that data encrypted before off-line storage can be decrypted and re-imported successfully.

Stored Data Encryption Using DBMS_CRYPTO

The `DBMS_CRYPTO` package provides several means for addressing the security issues that have been discussed. (For backward compatibility, `DBMS_OBFUSCATION_TOOLKIT` is also provided.) This section includes these topics:

- DBMS_CRYPTO Hashing and Encryption Capabilities
- Data Encryption Challenges

DBMS_CRYPTO Hashing and Encryption Capabilities

While encryption is not the ideal solution for addressing a number of security threats, it is clear that selectively encrypting sensitive data before storage in the database does improve security. Examples of such data could include:

- Credit card numbers

- National identity numbers

To address these needs, Oracle Database provides the PL/SQL package DBMS_CRYPTO to encrypt and decrypt stored data. This package supports several industry-standard encryption and hashing algorithms, including the Advanced Encryption Standard (AES) encryption algorithm. AES has been approved by the National Institute of Standards and Technology (NIST) to replace the Data Encryption Standard (DES).

The DBMS_CRYPTO package enables encryption and decryption for common Oracle data types, including RAW and large objects (LOBs), such as images and sound. Specifically, it supports BLOBs and CLOBs. In addition, it provides Globalization Support for encrypting data across different database character sets.

The following cryptographic algorithms are supported:

- Data Encryption Standard (DES), Triple DES (3DES, 2-key)
- Advanced Encryption Standard (AES)
- SHA-1 Cryptographic Hash
- SHA-1 Message Authentication Code (MAC)

Block cipher modifiers are also provided with DBMS_CRYPTO. You can choose from several padding options, including Public Key Cryptographic Standard (PKCS) #5, and from four block cipher chaining modes, including Cipher Block Chaining (CBC). Padding must be done in multiples of eight bytes.

Note:

- DES is no longer recommended by the National Institute of Standards and Technology (NIST).
- Usage of SHA-1 is more secure than MD5.
- Keyed MD5 is not vulnerable.

Table 17–1 compares the DBMS_CRYPTO package features to the other PL/SQL encryption package, the DBMS_OBFUSCATION_TOOLKIT.

Table 17–1 DBMS_CRYPTO and DBMS_OBFUSCATION_TOOLKIT Feature Comparison

Package Feature	DBMS_CRYPTO	DBMS_OBFUSCATION_TOOLKIT
Cryptographic algorithms	DES, 3DES, AES, RC4, 3DES_2KEY	DES, 3DES
Padding forms	PKCS5, zeroes	None supported
Block cipher chaining modes	CBC, CFB, ECB, OFB	CBC
Cryptographic hash algorithms	SHA-1	MD5
Keyed hash (MAC) algorithms	HMAC_MD5, HMAC_SH1	None supported
Cryptographic pseudo-random number generator	RAW, NUMBER, BINARY_INTEGER	RAW, VARCHAR2
Database types	RAW, CLOB, BLOB	RAW, VARCHAR2

DBMS_CRYPTO is intended to replace the obfuscation toolkit, because it is easier to use and supports a range of algorithms accommodating both new and existing systems. Although 3DES_2KEY and MD4 are provided for backward compatibility, you achieve better security using 3DES, AES, or SHA-1. Therefore, 3DES_2KEY is not recommended.

The DBMS_CRYPTO package includes cryptographic checksum capabilities (MD5), which are useful for compares, and the ability to generate a secure random number (the RANDOMBYTES function). Secure random number generation is an important part of cryptography, predictable keys are easily-guessed keys, and easily-guessed keys may lead to easy decryption of data. Most cryptanalysis is done by finding weak keys or poorly stored keys, rather than through brute force analysis (cycling through all possible keys).

Note:

- Do not use DBMS_RANDOM as it is unsuitable for cryptographic key generation.
- For more detailed descriptions of both DBMS_CRYPTO and DBMS_OBFUSCATION_TOOLKIT, also refer to the *PL/SQL Packages and Types Reference.*

Key management is programmatic. That is, the application (or caller of the function) must supply the encryption key. This means that the application developer must find a way of storing and retrieving keys securely. The relative strengths and weaknesses of various key management techniques are discussed in the sections that follow. The DBMS_OBFUSCATION_TOOLKIT package, which can handle both string and raw data, requires the submission of a 64-bit key. The DES algorithm itself has an effective key length of 56-bits.

Note: The DBMS_OBFUSCATION_TOOLKIT is granted to PUBLIC by default. Oracle strongly recommends that you revoke this grant.

While the DBMS_OBFUSCATION_TOOLKIT package can take either VARCHAR2 or RAW data types, it is preferable to use the RAW data type for keys and encrypted data. Storing encrypted data as VARCHAR2 can cause problems if it passes through Globalization Support routines. For example, when transferring database to a database that uses another character set.

To convert between VARCHAR2 and RAW data types, use the CAST_TO_RAW and CAST_TO_VARCHAR2 functions of the UTL_RAW package.

See Also: *PL/SQL Packages and Types Reference* for detailed documentation of the DBMS_CRYPTO, DBMS_OBFUSCATION_TOOLKIT and UTL_RAW packages

Data Encryption Challenges

Even in cases where encryption can provide additional security, come with associated technical challenges, as described in the following sections:

- Encrypting Indexed Data
- Key Generation
- Key Transmission
- Key Storage

- Changing Encryption Keys
- BLOBS

Encrypting Indexed Data

Special difficulties arise in handling encrypted data that is indexed. For example, suppose a company uses a national identity number, such as the U.S. Social Security number (SSN), as the employee number for its employees. The company considers employee numbers to be very sensitive data and therefore wants to encrypt data in the `EMPLOYEE_NUMBER` column of the `EMPLOYEES` table. Because `EMPLOYEE_NUMBER` contains unique values, the database designers want to have an index on it for better performance.

However, if `DBMS_CRYPTO` or the `DBMS_OBFUSCATION_TOOLKIT` (or another mechanism) is used to encrypt data in a column, then an index on that column will also contain encrypted values. Although such an index can be used for equality checking (for example, `'SELECT * FROM emp WHERE employee_number = '123245'`), if the index on that column contains encrypted values, then the index is essentially unusable for any other purpose. Oracle therefore recommends that developers not encrypt indexed data.

Given the privacy issues associated with overuse of national identity numbers (for example, identity theft), the fact that some allegedly unique national identity numbers have duplicates (as with U.S. Social Security numbers), and the ease with which a sequence can generate a unique number, there are many good reasons to avoid using national identity numbers as unique IDs.

Key Generation

Encrypted data is only as secure as the key used for encrypting it. An encryption key must be securely generated using secure cryptographic key generation. Oracle Database provides support for secure random number generation, with the `RANDOMBYTES` function of `DBMS_CRYPTO`. (This function replaces the capabilities provided by the `GetKey` procedure of the earlier `DBMS_OBFUSCATION_TOOLKIT`.) `DBMS_CRYPTO` calls the secure random number generator (RNG) previously certified by RSA.

> **Note:** Developers should not, under any circumstances use the `DBMS_RANDOM` package. The `DBMS_RANDOM` package generates pseudo-random numbers, which, as RFC-1750 states that the use of pseudo-random processes to generate secret quantities can result in pseudo-security.

Be sure to provide the correct number of bytes when you encrypt a key value. For example, you must provide a 16-byte key for the `ENCRYPT_AES128` encryption algorithm.

Key Transmission

If the key is to be passed by the application to the database, then it must be encrypted. Otherwise, a snooper could grab the key as it is being transmitted. Use of network encryption, such as that provided by Oracle Advanced Security, will protect all data in transit from modification or interception, including cryptographic keys.

Key Storage

Key storage is one of the most important, yet difficult, aspects of encryption. To recover data encrypted with a symmetric key, the key must be accessible to an authorized application or user seeking to decrypt the data. At the same time, the key must be inaccessible to someone who is maliciously trying to access encrypted data that he is not supposed to see.

The options available to a developer are:

- Storing the Keys in the Database
- Storing the Keys in the Operating System
- Users Managing Their Own Keys
- Using Transparent Database Encryption

Storing the Keys in the Database

Storing the keys in the database cannot always provide infallible security if you are trying to protect against the DBA accessing encrypted data. An all-privileged DBA could still access tables containing encryption keys. However, it can often provide quite good security against the casual snooper or against someone compromising the database file on the operating system.

As a trivial example, suppose you create a table (`EMP`) that contains employee data. You want to encrypt employee Social Security Number (SSN) stored in one of the columns. You could encrypt employee SSN using a key that is stored in a separate column. However, anyone with `SELECT` access on the entire table could retrieve the encryption key and decrypt the matching SSN.

While this encryption scheme seems easily defeated, with a little more effort you can create a solution that is much harder to break. For example, you could encrypt the SSN using a technique that performs some additional data transformation on the `employee_number` before using it to encrypt the SSN. This technique might be something as simple as XORing the `employee_number` with the birth date of the employee.

As additional protection, PL/SQL source code performing encryption can be wrapped, (using the `WRAP` utility) which obfuscates the code. The `WRAP` utility processes an input SQL file and obfuscates the PL/SQL units in it. For example, the following command uses the `keymanage.sql` file as the input:

```
wrap iname=/mydir/keymanage.sql
```

A developer can subsequently have a function in the package call the `DBMS_OBFUSCATION_TOOLKIT` with the key contained in the wrapped package.

Oracle Database 10*g* Release 2 (10.2) allows you to obfuscate dynamically generated PL/SQL code. The `DBMS_DDL` package contains two subprograms which allow you to obfuscate dynamically generated PL/SQL program units. For example, the following block uses the `DBMS_DDL.CREATE_WRAPPED` procedure to wrap dynamically generated PL/SQL code.

```
BEGIN
......
SYS.DBMS_DDL.CREATE_WRAPPED(function_returning_PLSQL_code());
......
END;
```

While wrapping is not unbreakable, it makes it harder for a snooper to get the key. Even in cases where a different key is supplied for each encrypted data value, not embedding the key value within a package, wrapping the package that performs key management (that is, data transformation or padding) is recommended.

> **See Also:** *PL/SQL User's Guide and Reference* for additional information on the `WRAP` command line utility and the `DBMS_DDL` subprograms for dynamic wrapping

An alternative would be to have a separate table in which to store the encryption key and to envelope the call to the keys table with a procedure. The key table can be joined to the data table using a primary key to foreign key relationship. For example, `EMPLOYEE_NUMBER` is the primary key in the `EMPLOYEES` table that stores employee information and the encrypted SSN. `EMPLOYEE_NUMBER` is a foreign key to the `SSN_KEYS` table that stores the encryption keys for employee SSN. The key stored in the `SSN_KEYS` table can also be transformed before use (by using XORing), so the key itself is not stored unencrypted. The procedure itself should be wrapped, to hide the way in which keys are transformed before use.

The strengths of this approach are:

- Users who have direct table access cannot see the sensitive data unencrypted, nor can they retrieve the keys to decrypt the data.
- Access to decrypted data can be controlled through a procedure that selects the encrypted data, retrieves the decryption key from the key table, and transforms it before it can be used to decrypt the data.
- The data transformation algorithm is hidden from casual snooping by wrapping the procedure, which obfuscates the procedure code.
- `SELECT` access to both the data table and the keys table does not guarantee that the user with this access can decrypt the data, because the key is transformed before use.

The weakness in this approach is that a user who has `SELECT` access to both the key table and the data table, and who can derive the key transformation algorithm, can break the encryption scheme.

The preceding approach is not infallible, but it is good enough to protect against easy retrieval of sensitive information stored in clear text.

Storing the Keys in the Operating System

Storing keys in a flat file in the operating system is another option. Oracle Database enables you to make callouts from PL/SQL, which you could use to retrieve encryption keys. However, if you store keys in the operating system and make callouts to it, then your data is only as secure as the protection on the operating system. If your primary security concern driving you to encrypt data stored in the database is that the database can be broken into from the operating system, then storing the keys in the operating system arguably makes it easier for a hacker to retrieve encrypted data than storing the keys in the database itself.

Users Managing Their Own Keys

Having the user supply the key assumes the user will be responsible with the key. Considering that 40% of help desk calls are from users who have forgotten their passwords, you can see the risks of having users manage encryption keys. In all

likelihood, users will either forget an encryption key, or write the key down, which then creates a security weakness. If a user forgets an encryption key or leaves the company, then your data is irrecoverable.

If you do elect to have user-supplied or user-managed keys, then you need to make sure you are using network encryption so that the key is not passed from the client to the server in the clear. You also must develop key archive mechanisms, which is also a difficult security problem. Key archives or backdoors create the security weaknesses that encryption is attempting to address in the first place.

Using Transparent Database Encryption

Transparent database encryption provides secure encryption with automatic key management for the encrypted tables. If the application requires protection of sensitive column data stored on the media, then transparent data encryption is a simple and fast way of achieving this.

> **See Also:** *Oracle Database Advanced Security Administrator's Guide* for more information on transparent data encryption

Changing Encryption Keys

Prudent security practice dictates that you periodically change encryption keys. For stored data, this requires periodically unencrypting the data, and reencrypting it with another well-chosen key. This would likely have to be done while the data is not being accessed, which creates another challenge. This is especially true for a Web-based application encrypting credit card numbers, because you do not want to shut down the entire application while you switch encryption keys.

BLOBS

Certain data types require more work to encrypt. For example, Oracle Database supports storage of Binary Large Objects (BLOBs), which let users store very large objects (for example, multiple gigabytes) in the database. A BLOB can be either stored internally as a column, or stored in an external file.

For an example of using `DBMS_CRYPTO` on BLOB data, refer to the section entitled Example of Encryption and Decryption Procedures for BLOB Data on page 17-12.

Example of a Data Encryption Procedure

The following sample PL/SQL program (`dbms_crypto.sql`) illustrates encrypting data. This example code does the following:

- DES-encrypts a string (`VARCHAR2` type) after first converting it into `RAW` type.

  This step is necessary because encrypt and decrypt functions and procedures in `DBMS_CRYPTO` package work on the `RAW` data type only, unlike functions and packages in the `DBMS_OBFUSCATION_TOOLKIT` package.
- Shows how to create a 160-bit hash using SHA-1 algorithm.
- Demonstrates how MAC, a key-dependent one-way hash, can be computed using MD5 algorithm.

The `dbms_crypto.sql` procedure follows:

```
DECLARE
    input_string     VARCHAR2(16) := 'tigertigertigert';
    raw_input        RAW(128) :=
```

```
UTL_RAW.CAST_TO_RAW(CONVERT(input_string,'AL32UTF8','US7ASCII'));
    key_string       VARCHAR2(8)  := 'scottsco';
    raw_key          RAW(128) :=
UTL_RAW.CAST_TO_RAW(CONVERT(key_string,'AL32UTF8','US7ASCII'));
    encrypted_raw    RAW(2048);
    encrypted_string VARCHAR2(2048);
    decrypted_raw    RAW(2048);
    decrypted_string VARCHAR2(2048);
-- 1. Begin testing Encryption
BEGIN
    dbms_output.put_line('> Input String                      : ' ||
    CONVERT(UTL_RAW.CAST_TO_VARCHAR2(raw_input),'US7ASCII','AL32UTF8'));
    dbms_output.put_line('> ========= BEGIN TEST Encrypt =========');
    encrypted_raw := dbms_crypto.Encrypt(
        src => raw_input,
        typ => DBMS_CRYPTO.DES_CBC_PKCS5,
        key => raw_key);
        dbms_output.put_line('> Encrypted hex value             : ' ||
        rawtohex(UTL_RAW.CAST_TO_RAW(encrypted_raw)));
decrypted_raw := dbms_crypto.Decrypt(
        src => encrypted_raw,
        typ => DBMS_CRYPTO.DES_CBC_PKCS5,
        key => raw_key);
    decrypted_string :=
    CONVERT(UTL_RAW.CAST_TO_VARCHAR2(decrypted_raw),'US7ASCII','AL32UTF8');
dbms_output.put_line('> Decrypted string output          : ' ||
        decrypted_string);
if input_string = decrypted_string THEN
    dbms_output.put_line('> String DES Encyption and Decryption successful');
END if;
dbms_output.put_line('');
dbms_output.put_line('> ========= BEGIN TEST Hash =========');
    encrypted_raw := dbms_crypto.Hash(
        src => raw_input,
        typ => DBMS_CRYPTO.HASH_SH1);
dbms_output.put_line('> Hash value of input string       : ' ||
        rawtohex(UTL_RAW.CAST_TO_RAW(encrypted_raw)));
dbms_output.put_line('> ========= BEGIN TEST Mac =========');
    encrypted_raw := dbms_crypto.Mac(
        src => raw_input,
        typ => DBMS_CRYPTO.HMAC_MD5,
        key => raw_key);
dbms_output.put_line('> Message Authentication Code      : ' ||
        rawtohex(UTL_RAW.CAST_TO_RAW(encrypted_raw)));
dbms_output.put_line('');
dbms_output.put_line('> End of DBMS_CRYPTO tests  ');
END;
/
```

See Also: *PL/SQL User's Guide and Reference*

Example of AES 256-Bit Data Encryption and Decryption Procedures

The following PL/SQL block block demonstrates how to encrypt and decrypt a predefined variable named `input_string` using the AES 256-bit algorithm with Cipher Block Chaining and PKCS #5 padding.

```
declare
   input_string       VARCHAR2 (200) := 'Secret Message';
```

```
   output_string      VARCHAR2 (200);
   encrypted_raw      RAW (2000);             -- stores encrypted binary text
   decrypted_raw      RAW (2000);             -- stores decrypted binary text
   num_key_bytes      NUMBER := 256/8;        -- key length 256 bits (32 bytes)
   key_bytes_raw      RAW (32);               -- stores 256-bit encryption key
   encryption_type    PLS_INTEGER :=          -- total encryption type
                         DBMS_CRYPTO.ENCRYPT_AES256
                       + DBMS_CRYPTO.CHAIN_CBC
                       + DBMS_CRYPTO.PAD_PKCS5;
begin
   DBMS_OUTPUT.PUT_LINE ('Original string: ' || input_string);
   key_bytes_raw := DBMS_CRYPTO.RANDOMBYTES (num_key_bytes);
   encrypted_raw := DBMS_CRYPTO.ENCRYPT
      (
         src => UTL_I18N.STRING_TO_RAW (input_string, 'AL32UTF8'),
         typ => encryption_type,
         key => key_bytes_raw
      );
    -- The encrypted value in the encrypted_raw variable can be used here
   decrypted_raw := DBMS_CRYPTO.DECRYPT
      (
         src => encrypted_raw,
         typ => encryption_type,
         key => key_bytes_raw
      );
   output_string := UTL_I18N.RAW_TO_CHAR (decrypted_raw, 'AL32UTF8');
   DBMS_OUTPUT.PUT_LINE ('Decrypted string: ' || output_string);
end;
```

Example of Encryption and Decryption Procedures for BLOB Data

The following sample PL/SQL program (`blob_test.sql`) illustrates encrypting and decrypting BLOB data. This example code does the following, and prints out its progress (or problems) at each step:

- Creates a table for the BLOB column
- Inserts the raw values into that table
- Encrypts the raw data
- Decrypts the encrypted data

The `blob_test.sql` procedure follows:

```
-- Create a table for BLOB column.
create table table_lob (id number, loc blob);

-- insert 3 empty lobs for src/enc/dec
insert into table_lob values (1, EMPTY_BLOB());
insert into table_lob values (2, EMPTY_BLOB());
insert into table_lob values (3, EMPTY_BLOB());

set echo on
set serveroutput on

declare
    srcdata    RAW(1000);
    srcblob    BLOB;
    encrypblob BLOB;
    encrypraw  RAW(1000);
    encrawlen  BINARY_INTEGER;
```

```
    decrypblob BLOB;
    decrypraw  RAW(1000);
    decrawlen  BINARY_INTEGER;

    leng       INTEGER;

begin

    -- RAW input data 16 bytes
    srcdata := hextoraw('6D6D6D6D6D6D6D6D6D6D6D6D6D6D6D6D');

    dbms_output.put_line('---');
    dbms_output.put_line('input is ' || srcdata);
    dbms_output.put_line('---');

    -- select empty lob locators for src/enc/dec
    select loc into srcblob from table_lob where id = 1;
    select loc into encrypblob from table_lob where id = 2;
    select loc into decrypblob from table_lob where id = 3;

    dbms_output.put_line('Created Empty LOBS');
    dbms_output.put_line('---');

    leng := DBMS_LOB.GETLENGTH(srcblob);
    IF leng IS NULL THEN
        dbms_output.put_line('Source BLOB Len NULL ');
    ELSE
        dbms_output.put_line('Source BLOB Len ' || leng);
    END IF;

    leng := DBMS_LOB.GETLENGTH(encrypblob);
    IF leng IS NULL THEN
        dbms_output.put_line('Encrypt BLOB Len NULL ');
    ELSE
        dbms_output.put_line('Encrypt BLOB Len ' || leng);
    END IF;

    leng := DBMS_LOB.GETLENGTH(decrypblob);
    IF leng IS NULL THEN
        dbms_output.put_line('Decrypt  BLOB Len NULL ');
    ELSE
        dbms_output.put_line('Decrypt BLOB Len ' || leng);
    END IF;

    -- write source raw data into blob
    DBMS_LOB.OPEN (srcblob, DBMS_LOB.lob_readwrite);
    DBMS_LOB.WRITEAPPEND (srcblob, 16, srcdata);
    DBMS_LOB.CLOSE (srcblob);

    dbms_output.put_line('Source raw data written to source blob');
    dbms_output.put_line('---');

    leng := DBMS_LOB.GETLENGTH(srcblob);
    IF leng IS NULL THEN
        dbms_output.put_line('source BLOB Len NULL ');
    ELSE
        dbms_output.put_line('Source BLOB Len ' || leng);
    END IF;

    /*
```

```
    * Procedure Encrypt
    * Arguments: srcblob -> Source BLOB
    *            encrypblob -> Output BLOB for encrypted data
    *            DBMS_CRYPTO.AES_CBC_PKCS5 -> Algo : AES
    *                                         Chaining : CBC
    *                                         Padding : PKCS5
    *            256 bit key for AES passed as RAW
    *                -> 
    hextoraw('000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F')
    *            IV (Initialization Vector) for AES algo passed as RAW
    *                -> hextoraw('00000000000000000000000000000000')
    */

    DBMS_CRYPTO.Encrypt(encrypblob,
                srcblob,
                DBMS_CRYPTO.AES_CBC_PKCS5,
                hextoraw
('000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F'),
                hextoraw('00000000000000000000000000000000'));


    dbms_output.put_line('Encryption Done');
    dbms_output.put_line('---');

    leng := DBMS_LOB.GETLENGTH(encrypblob);
    IF leng IS NULL THEN
        dbms_output.put_line('Encrypt BLOB Len NULL');
    ELSE
        dbms_output.put_line('Encrypt BLOB Len ' || leng);
    END IF;

    -- Read encrypblob to a raw
    encrawlen := 999;

    DBMS_LOB.OPEN (encrypblob, DBMS_LOB.lob_readwrite);
    DBMS_LOB.READ (encrypblob, encrawlen, 1, encrypraw);
    DBMS_LOB.CLOSE (encrypblob);

    dbms_output.put_line('Read encrypt blob to a raw');
    dbms_output.put_line('---');

    dbms_output.put_line('Encrypted data is (256 bit key) ' || encrypraw);
    dbms_output.put_line('---');

    /*
    * Procedure Decrypt
    * Arguments: encrypblob -> Encrypted BLOB to decrypt
    *            decrypblob -> Output BLOB for decrypted data in RAW
    *            DBMS_CRYPTO.AES_CBC_PKCS5 -> Algo : AES
    *                                         Chaining : CBC
    *                                         Padding : PKCS5
    *            256 bit key for AES passed as RAW (same as used during Encrypt)
    *                -> 
    hextoraw('000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F')
    *            IV (Initialization Vector) for AES algo passed as RAW (same as
                 used during Encrypt)
    *                -> hextoraw('00000000000000000000000000000000')
    */

    DBMS_CRYPTO.Decrypt(decrypblob,
```

```
                encrypblob,
                DBMS_CRYPTO.AES_CBC_PKCS5,
                hextoraw
           ('000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F'),
                hextoraw('00000000000000000000000000000000'));

    leng := DBMS_LOB.GETLENGTH(decrypblob);
    IF leng IS NULL THEN
        dbms_output.put_line('Decrypt BLOB Len NULL');
    ELSE
        dbms_output.put_line('Decrypt BLOB Len ' || leng);
    END IF;

    -- Read decrypblob to a raw
    decrawlen := 999;

    DBMS_LOB.OPEN (decrypblob, DBMS_LOB.lob_readwrite);
    DBMS_LOB.READ (decrypblob, decrawlen, 1, decrypraw);
    DBMS_LOB.CLOSE (decrypblob);

    dbms_output.put_line('Decrypted data is (256 bit key) ' || decrypraw);
    dbms_output.put_line('---');

    DBMS_LOB.OPEN (srcblob, DBMS_LOB.lob_readwrite);
    DBMS_LOB.TRIM (srcblob, 0);
    DBMS_LOB.CLOSE (srcblob);

    DBMS_LOB.OPEN (encrypblob, DBMS_LOB.lob_readwrite);
    DBMS_LOB.TRIM (encrypblob, 0);
    DBMS_LOB.CLOSE (encrypblob);

    DBMS_LOB.OPEN (decrypblob, DBMS_LOB.lob_readwrite);
    DBMS_LOB.TRIM (decrypblob, 0);
    DBMS_LOB.CLOSE (decrypblob);

end;
/

truncate table table_lob;
drop table table_lob;
```

Part IV

Appendixes

This part contains the following appendixes:

- Appendix A, "Addressing The CONNECT Role Change"
- Appendix B, "Verifying Data Integrity with DBMS_SQLHASH"

A

Addressing The CONNECT Role Change

The CONNECT role was introduced with Oracle Database version 7, which added new and robust support for database roles. The CONNECT role is used in sample code, applications, documentation, and technical papers. The CONNECT role was established with the following privileges :

Privileges Originally Associated	with the CONNECT Role
Alter Session	Create Session
Create Cluster	Create Synonym
Create Database Link	Create Table
Create Sequence	Create View

However, beginning in Oracle Database 10*g* Release 2 (10.2), the CONNECT role has only the CREATE SESSION privilege, all other privileges are removed.

Although the CONNECT role has frequently been used when provisioning new accounts in the Oracle database, simply connecting to the database does not require all those privileges. Making this change enables new and existing database customers to enforce good security practices more easily.

Each user should have only those privileges appropriate to the tasks she needs to do, an idea termed the principle of least privilege. Least privilege mitigates risk by limiting privileges, so that it remains easy to do what is needed while concurrently reducing the ability to do inappropriate things, either inadvertently or maliciously.

This Appendix discusses the effects of changed CONNECT privileges in the following sections:

- Approach 2 - Restore CONNECT privileges
- Approach 3 - Conduct least privilege analysis

How Applications Are Affected

The effects of the changes to the CONNECT role can be seen in database upgrades, account provisioning, and installation of applications using new databases.

Database Upgrade

Upgrading your existing Oracle database to Oracle Database 10gR2 automatically changes the `CONNECT` role to have only the `CREATE SESSION` privilege. Most applications are not affected because the applications objects already exist: no new tables, views, sequences, synonyms, clusters, or database links need be created.

Applications that create tables, views, sequences, synonyms, clusters, or database links, or that use the `ALTER SESSION` command dynamically, may fail due to insufficient privileges.

Account Provisioning

If your application or DBA grants the `CONNECT` role as part of the account provisioning process, then no privileges beyond `CREATE SESSION` are included. Any additional privilege must be granted either directly or through another role.

This issue can be addressed by creating a new customized database role.

See Also: Approaches to Addressing the CONNECT Role Change on page A-3

Installation of Applications Using New Databases

New databases created using the Oracle Database 10gR2 Utility (DBCA), or using database creation templates generated from DBCA, define the `CONNECT` role with only the `CREATE SESSION` privilege. Installing an application to use such a new database may fail if the database schema used for the application is granted privileges solely through the `CONNECT` role.

How Users Are Affected

The change to the `CONNECT` role affects three classes of users differently: general users, application developers, and client/server applications.

General Users

The new `CONNECT` supplies only the `CREATE SESSION` privilege. Therefore users who connect to the database to use an application are not affected, because the `CONNECT` role still has the `CREATE SESSION` privilege.

However, appropriate privileges will not be present for a certain set of users if they are provisioned solely with `CONNECT`. These are users who create tables, views, sequences, synonyms, clusters, or database links, or use the `ALTER SESSION` command. The privileges they need are no longer provided with the `CONNECT` role. To authorize the additional privileges needed, the database administrator must create and apply additional roles for the appropriate privileges, or grant them directly to the users who need them.

Note that the ALTER SESSION privilege is required for setting events. Very few database users should require the alter session privilege.

```
SQL> ALTER SESSION SET EVENTS ........
```

The alter session privilege is *not* required for other alter session commands.

```
SQL> ALTER SESSION SET NLS_TERRITORY = FRANCE;
```

Application Developers

Similarly, application developers provisioned solely with CONNECT will not have appropriate privileges to create tables, views, sequences, synonyms, clusters, or database links, nor to use the alter session command. The database administrator must either create and apply additional roles for the appropriate privileges, or grant them directly to the application developers who need them.

Client Server Applications

Most traditional client/server applications using dedicated user accounts will not be affected by this change. However, applications that create private synonyms or temporary tables using dynamic SQL in the user schema during account provisioning or run time operations will be affected. They will require additional roles or directly grants to acquire the system privileges appropiate to their activities.

Approaches to Addressing the CONNECT Role Change

Three approaches are recommended for addressing the impact of this change.

Approach 1 - Create a new database role

The privileges removed from the CONNECT role can be easily managed by creating a new database role.

First, connect to the upgraded Oracle database and create a new database role. The following example uses a role called my_app_developer:

```
SQL> CREATE ROLE my_app_developer;
SQL> GRANT CREATE TABLE, CREATE VIEW, CREATE SEQUENCE, CREATE SYNONYM, CREATE
CLUSTER, CREATE DATABASE LINK, ALTER SESSION TO my_app_developer;
SQL>
```

Second, determine which users or database roles have the CONNECT role and grant the new role to these users or roles.

```
SQL> SELECT user$.name, admin_option, default_role
     FROM user$, sysauth$, dba_role_privs
     WHERE privilege# =
     (SELECT user# from user$ WHERE name = 'CONNECT')
     AND user$.user# = grantee#
     AND grantee = user$.name
     AND granted_role = 'CONNECT';

NAME                           ADMIN_OPTI DEF
------------------------------ ---------- ---
R1                             YES        YES
R2                             NO         YES

SQL> GRANT my_app_developer TO R1 WITH ADMIN OPTION;
SQL> GRANT my_app_developer TO R2;
```

You can determine the privileges that users require by using Oracle Auditing. The audit information can then be analyzed and used to create additional database roles with finer granularity.

Privileges not used can then be revoked for specific users. Note that prior to auditing, the database initialization parameter AUDIT_TRAIL must be initialized and the database restarted.

```
SQL> AUDIT CREATE TABLE, CREATE SEQUENCE, CREATE SYNONYM, CREATE DATABASE LINK,
CREATE CLUSTER, CREATE VIEW, ALTER SESSION;
```

Database privilege usage can now be monitored periodically.

```
SQL> SELECT userid, name FROM aud$, system_privilege_map
WHERE - priv$used = privilege;
USERID                         NAME
------------------------------ ----------------
ACME                           CREATE TABLE
ACME                           CREATE SEQUENCE
ACME                           CREATE TABLE
ACME                           ALTER SESSION
APPS                           CREATE TABLE
APPS                           CREATE TABLE
APPS                           CREATE TABLE
APPS                           CREATE TABLE
8 rows selected.
SQL>
```

Approach 2 - Restore CONNECT privileges

Starting with 10*g* Release 2 (10.2), Oracle provides a script called `rstrconn.sql` located in the `$ORACLE_HOME/rdbms/admin` directory. After a database upgrade or new database creation, this script can be used to grant back the privileges removed from the CONNECT role in Oracle Database 10*g* Release 2 (10.2).

If this approach is used, then privileges that are not used should be revoked from users who do not need them. To identify such privileges and users, the database must be restarted with the database initialization parameter AUDIT_TRAIL initialized, for example, AUDIT_TRAIL=DB. Oracle Database auditing should then be turned on to monitor what privileges are used as follows:

```
SQL> AUDIT CREATE TABLE, CREATE SEQUENCE, CREATE SYNONYM, CREATE DATABASE LINK,
CREATE CLUSTER, CREATE VIEW, ALTER SESSION;
```

Database privilege usage can also be monitored periodically.

```
SQL> SELECT userid, name FROM aud$, system_privilege_map
WHERE - priv$used = privilege;
USERID                         NAME
------------------------------ ----------------
ACME                           CREATE TABLE
ACME                           CREATE SEQUENCE
ACME                           CREATE TABLE
ACME                           ALTER SESSION
APPS                           CREATE TABLE
APPS                           CREATE TABLE
APPS                           CREATE TABLE
APPS                           CREATE TABLE
```

```
8 rows selected.
SQL>
```

New View Showing CONNECT Grantees

A new view enables administrators who decide to continue using the old CONNECT role with its many privileges to see quickly which users have CONNECT.

The new view, DBA_CONNECT_ROLE_GRANTEES, has the following columns:

Table A–1 Columns and Contents for DBA_CONNECT_ROLE_GRANTEES

Column Name	Contents
Grantee	User granted the CONNECT role
Path_of_connect_role_grant	Role (or nested roles) by which the user is granted CONNECT
Admin_opt	VARCHAR2(3), YES if user has ADMIN OPTION on CONNECT, else NO

Approach 3 - Conduct least privilege analysis

Oracle partners and application providers should use this approach to deliver more secure products to the Oracle customer base. The principle of least privilege mitigates risk by limiting privileges to the minimum set required to perform a given function.

For each class of users that the analysis shows need the same set of privileges, create a role with exactly those privileges. Remove all other privileges from those users, and assign that role to those users. As needs change, additional privileges can be granted either directly or through these new roles, or new roles can be created to meet new needs. At any given time, however, there is a greater assurance that inappropriate privileges have been limited, thereby reducing the risk of inadvertent or malicious harm.

B

Verifying Data Integrity with DBMS_SQLHASH

This appendix discusses verifying data integrity using the DBMS_SQLHASH package. It includes the following sections:

Overview of the DBMS_SQLHASH Package

The DBMS_SQLHASH package can check data integrity by making use of hash algorithms. It provides an interface to generate the hash value of the result set returned by a SQL query. Hash values are like data fingerprints and are used to ensure data integrity. DBMS_SQLHASH provides support for several industry standard hashing algorithms, including MD4, MD5, and SHA-1 cryptographic hashes.

Oracle Database installs the DBMS_SQLHASH package in the SYS schema. You can then grant package access to existing users and roles as required.

DBMS_SQLHASH includes the GETHASH function that is used to retrieve the hash value of a query result set. The GETHASH function runs one of the supported cryptographic hash algorithms against the result set of the SQL statement to arrive at a hash value.

You can compare hash values to check whether data has been altered. For example, before storing data, Laura runs the DBMS_SQLHASH.GETHASH function against the SQL statement to create a hash value of the SQL result set. When she retrieves the stored data at a later date, she reruns the hash function against the SQL statement using the same algorithm. If the second hash value is identical to the first one, then data has not been altered. Any modification to the result set data would cause the hash value to be different.

The DBMS_SQLHASH.GETHASH Function

This function applies one of the supported cryptographic hash algorithms to the result set of the SQL statement.

Syntax

```
DBMS_SQLHASH.GETHASH(
    sqltext IN varchar2,
    digest_type IN BINARY_INTEGER,
    chunk_size IN number DEFAULT 134217728)
   RETURN raw;
```

Parameters

Table B–1 lists the GETHASH parameters and their descriptions.

Table B–1 GETHASH Function Parameters

Parameter Name	Description
sqltext	The SQL statement whose result is hashed
digest_type	Hash algorithm used: HASH_MD4, HASH_MD5 or HASH_SH1
chunk_size	Size of the result chunk when getting the hash When the result set size is large, the GETHASH function will break it into chunks having a size equal to chunk_size. It will generate the hash for each chunk and then use hash chaining to calculate the final hash. The default chunk_size is 128 MB.

Glossary

application roles

Database roles that are granted to application users and that are secured by embedding passwords inside the application. See also **secure application roles**

certificate

An ITU x.509 v3 standard data structure that securely binds an identify to a public key.

A certificate is created when an entity's public key is signed by a trusted identity, a certificate authority. The certificate ensures that the entity's information is correct and that the public key actually belongs to that entity.

A certificate contains the entity's name, identifying information, and public key. It is also likely to contain a serial number, expiration date, and information about the rights, uses, and privileges associated with the certificate. Finally, it contains information about the certificate authority that issued it.

certificate revocation lists

(CRLs) Signed data structures that contain a list of revoked **certificate**s. The authenticity and integrity of the CRL is provided by a digital signature appended to it. Usually, the CRL signer is the same entity that signed the issued certificate.

definer's rights procedures

Definer's rights procedures execute with the privileges of their owner, not their current user. Such definer's rights sub-programs are bound to the schema in which they reside. For example, assume that user `blake` and user `scott` each have a table called `dept` in their respective user schemas. If user `blake` calls a definer's rights procedure, which is owned by user `scott`, to update the `dept` table, then this procedure will update the `dept` table in the `scott` schema because this procedure executes with the privileges of the user who owns (defined) the procedure.

encryption

The process of disguising a message rendering it unreadable to any but the intended recipient.

Forwardable Ticket Granting Ticket

A special Kerberos ticket that can be forwarded to proxies permits the proxy to obtain additional Kerberos tickets on behalf of the client for proxy authentication. See also **Kerberos ticket**

integrity

The guarantee that the contents of the message received were not altered from the contents of the original message sent.

invoker's rights procedures

Invoker's rights procedures execute with the privileges of the current user, that is, the user who invokes the procedure. Such procedures are not bound to a particular schema. They can be run by a variety of users and allow multiple users to manage their own data by using centralized application logic. Invoker's rights procedures are created with the `AUTHID` clause in the declaration section of the procedure code.

KDC

See **Key Distribution Center**

Kerberos ticket

A temporary set of electronic credentials that verify the identity of a client for a particular service. Also referred to as a service ticket.

Key Distribution Center

(KDC) A machine that issues Kerberos tickets. See also **Kerberos ticket**

salt

In cryptography, generally speaking, "salt" is a way to strengthen the security of encrypted data. Salt is a random string that is added to the data before it is encrypted, making it more difficult for attackers to steal the data by matching patterns of ciphertext to known ciphertext samples. Salt is often also added to passwords, before the passwords are encrypted, to avoid dictionary attacks, a method that unethical hackers (attackers) use to steal passwords. The encrypted salted values make it difficult for attackers to match the hash value of encrypted passwords (sometimes called verifiers) with their dictionary lists of common password hash values.

secure application roles

Like an application roles, a secure application role is a database role that is granted to application users, but it is secured by using an Invoker's Right stored procedure to retrieve the role password from a database table. A secure application role password is not embedded in the application. See also **application roles**

service ticket

See **Kerberos ticket**

wallet

A wallet is a data structure used to store and manage security credentials for an individual entity.

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X