|  |  |
| --- | --- |
| Advanced PL/SQL – Preventing SQL Injection AttacksA book on a tableAn introduction and overview |  |
| This booklet will cover best practices for preventing SQL Injection attacks, including input validation, parameterized queries, stored procedures, and the principle of least privilege. By following these guidelines, developers can significantly reduce the risk of SQL Injection and ensure the security and integrity of their applications.By Randall Fadler September 2024 |  |

Table of Contents

[1. Introduction 3](#_Toc177224191)

[Overview of SQL Injection Attacks 3](#_Toc177224192)

[Importance of Preventing SQL Injection Attacks 3](#_Toc177224193)

[Signs of a SQL Injection Attack 4](#_Toc177224194)

[2. Best Practices for Preventing SQL Injection Attacks 5](#_Toc177224195)

[Use Bind Variables 5](#_Toc177224196)

[Parameterized Queries 5](#_Toc177224197)

[Benefits of Using Bind Variables and Parameterized Queries 6](#_Toc177224198)

[Examples of Vulnerable Dynamic SQL 7](#_Toc177224199)

[Dynamic SQL with Concatenation, example 1: 7](#_Toc177224200)

[Dynamic SQL with Concatenation, example 2: 7](#_Toc177224201)

[Vulnerable Login Query: 8](#_Toc177224202)

[Search Query with User Input: 8](#_Toc177224203)

[Use DBMS\_ASSERT Package 9](#_Toc177224204)

[Use Static SQL Whenever Possible 9](#_Toc177224205)

[Validate and Sanitize User Inputs 9](#_Toc177224206)

[Use Least Privilege Principle 10](#_Toc177224207)

[Role-Based Access Control (RBAC): 10](#_Toc177224208)

[Granting Specific Privileges: 10](#_Toc177224209)

[Using Secure Application Roles: 10](#_Toc177224210)

[Limiting System Privileges: 11](#_Toc177224211)

[Using Fine-Grained Access Control (FGAC): 11](#_Toc177224212)

[Use PL/SQL Packages and Procedures 12](#_Toc177224213)

[Regular Security Audits and Code Reviews 12](#_Toc177224214)

[Using the DBMS\_SQL package 13](#_Toc177224215)

[List of Procedures 14](#_Toc177224216)

[Examples of using DBMS\_SQL: 15](#_Toc177224217)

[Executing a Dynamic Query: 15](#_Toc177224218)

[Handling Dynamic Columns: 15](#_Toc177224219)

[Using the DBMS\_ASSERT Package 16](#_Toc177224220)

[List of Procedures 17](#_Toc177224221)

[Examples of using DBMS\_ASSERT 17](#_Toc177224222)

[Validating a Single Quoted Literal: 17](#_Toc177224223)

[Validating a Qualified SQL Name: 18](#_Toc177224224)

[Validating a SQL Object Name: 18](#_Toc177224225)

# 1. Introduction

## Overview of SQL Injection Attacks

SQL Injection is a type of cyber-attack where an attacker exploits vulnerabilities in an application’s software by injecting malicious SQL code into a query. This can occur when user inputs are not properly sanitized or validated, allowing the attacker to manipulate the SQL query to execute arbitrary commands. The consequences of SQL Injection can be severe, including unauthorized access to sensitive data, data corruption, and even complete control over the database server. Attackers can use SQL Injection to bypass authentication mechanisms, retrieve hidden data, and execute administrative operations. It is a critical security concern for any application that interacts with a database, and developers must be vigilant in implementing protective measures.

## Importance of Preventing SQL Injection Attacks

Preventing SQL Injection attacks is crucial for maintaining the security and integrity of any application that interacts with a database. SQL Injection can lead to unauthorized access to sensitive information, such as personal data, financial records, and intellectual property. For example, an attacker might exploit a vulnerable login form by injecting malicious SQL code to bypass authentication and gain access to user accounts. This could result in identity theft, financial fraud, or exposure of confidential information. Another example is an attacker manipulating a search query to retrieve hidden data, such as customer lists or proprietary business information, which could be used for competitive advantage or sold on the black market. Additionally, SQL Injection can be used to execute administrative operations, such as deleting or modifying data, which can disrupt business operations and cause significant financial losses. By implementing robust security measures, such as input validation, parameterized queries, and least privilege principles, developers can protect their applications from SQL Injection attacks and ensure the confidentiality, integrity, and availability of their data

## Signs of a SQL Injection Attack

Detecting a SQL Injection attack can be challenging, but there are several common signs that may indicate an attack is occurring:

* Unexpected Database Errors: If your application suddenly starts generating unusual or unexpected database errors, it could be a sign of SQL Injection. These errors might reveal information about your database structure or query syntax.
* Unusual Activity in Logs: Monitoring your database and application logs for unusual activity, such as unexpected queries or a high volume of requests, can help identify potential SQL Injection attempts.
* Changes in Application Behavior: If your application starts behaving erratically, such as displaying incorrect data, allowing unauthorized access, or experiencing performance issues, it could be due to a SQL Injection attack.
* Unauthorized Data Access: If you notice that sensitive data is being accessed or modified without proper authorization, it could be a result of SQL Injection.
* Unusual User Activity: Monitoring user activity for unusual patterns, such as repeated failed login attempts or accessing parts of the application they shouldn’t, can help identify potential SQL Injection attacks.

By being vigilant and monitoring for these signs, you can detect and respond to SQL Injection attacks more effectively. Implementing robust security measures, such as input validation and parameterized queries, can also help prevent these attacks from occurring in the first place.

# 2. Best Practices for Preventing SQL Injection Attacks

Preventing SQL injection in PL/SQL code is crucial for maintaining the security and integrity of your database. Here are some of the best practices to prevent SQL injection:

## Use Bind Variables

Bind variables are placeholders in SQL statements that are replaced with actual values at runtime. This approach ensures that user inputs are treated as data, not executable code. Here’s an example in PL/SQL:

PROCEDURE GetEmployeeDetails(p\_emp\_id IN NUMBER) IS

 v\_emp\_name VARCHAR2(100);

BEGIN

 SELECT emp\_name INTO v\_emp\_name FROM employees WHERE emp\_id = :emp\_id;

 DBMS\_OUTPUT.put\_line('Employee Name: ' || v\_emp\_name);

END;

## Parameterized Queries

Parameterized queries are another way to safely include user inputs in SQL statements. Using parameterized queries is a great way to prevent SQL Injection attacks and ensure the security of your database. Here’s an example of how to use a parameterized query in Oracle using PL/SQL:

DECLARE

 v\_empno NUMBER := 7369; -- Example employee number

 v\_ename VARCHAR2(50);

BEGIN

 -- Using a parameterized query with bind variables

 SELECT ename

 INTO v\_ename

 FROM emp

 WHERE empno = :empno;

 DBMS\_OUTPUT.PUT\_LINE('Employee Name: ' || v\_ename);

EXCEPTION

 WHEN NO\_DATA\_FOUND THEN

 DBMS\_OUTPUT.PUT\_LINE('No employee found with the given employee number.');

 WHEN OTHERS THEN

 DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM);

END;

In this example:

* The v\_empno variable is used to store the employee number.
* The SELECT statement uses a bind variable :empno to safely include the employee number in the query.
* The INTO clause is used to store the result of the query in the v\_ename variable.
* The DBMS\_OUTPUT.PUT\_LINE procedure is used to display the employee name.

By using bind variables, you ensure that user input is safely included in the SQL statement, reducing the risk of SQL Injection attacks. This booklet will cover more examples and best practices for using parameterized queries to help you secure your Oracle PL/SQL applications.

## Benefits of Using Bind Variables and Parameterized Queries

* Prevents SQL Injection: By treating user inputs as data, it prevents malicious inputs from being executed as part of the SQL statement.
* Improves Performance: Bind variables can improve performance by allowing the database to reuse execution plans for similar queries with different inputs.
* Enhances Security: It ensures that user inputs are properly escaped and handled, reducing the risk of SQL injection attacks.

## Examples of Vulnerable Dynamic SQL

Here are some examples of a vulnerable dynamic SQL statement that should be modified to prevent attacks:

### Dynamic SQL with Concatenation, example 1:

PROCEDURE GetEmployeeDetails(p\_emp\_id IN NUMBER) IS

 v\_sql VARCHAR2(200);

 v\_emp\_name VARCHAR2(100);

BEGIN

 v\_sql := 'SELECT emp\_name INTO v\_emp\_name FROM employees WHERE emp\_id = ' || p\_emp\_id;

 EXECUTE IMMEDIATE v\_sql;

 DBMS\_OUTPUT.put\_line('Employee Name: ' || v\_emp\_name);

END;

In this example, if p\_emp\_id contains a malicious input like 1 OR 1=1, it could lead to an SQL injection attack.

### Dynamic SQL with Concatenation, example 2:

DECLARE

 v\_query VARCHAR2(1000);

 v\_username VARCHAR2(50) := 'user\_input';

BEGIN

 v\_query := 'SELECT \* FROM users WHERE username = ''' || v\_username || '''';

 EXECUTE IMMEDIATE v\_query;

END;

In this example, if v\_username contains malicious input like user\_input' OR '1'='1, the query becomes:

SELECT \* FROM users WHERE username = 'user\_input' OR '1'='1'

This would return all rows from the users table, bypassing any authentication checks.

### Vulnerable Login Query:

DECLARE

 v\_query VARCHAR2(1000);

 v\_username VARCHAR2(50) := 'user\_input';

 v\_password VARCHAR2(50) := 'pass\_input';

BEGIN

 v\_query := 'SELECT \* FROM users WHERE username = ''' || v\_username || ''' AND password = ''' || v\_password || '''';

 EXECUTE IMMEDIATE v\_query;

END;

If an attacker inputs user\_input' OR '1'='1 for the username and anything for the password, the query becomes:

SELECT \* FROM users WHERE username = 'user\_input' OR '1'='1' AND password = 'anything'

This would allow the attacker to bypass the login authentication.

### Search Query with User Input:

DECLARE

 v\_query VARCHAR2(1000);

 v\_search\_term VARCHAR2(50) := 'search\_input';

BEGIN

 v\_query := 'SELECT \* FROM products WHERE name LIKE ''%' || v\_search\_term || '%''';

 EXECUTE IMMEDIATE v\_query;

END;

If v\_search\_term contains search\_input%' OR '1'='1, the query becomes:

SELECT \* FROM products WHERE name LIKE '%search\_input%' OR '1'='1'

This would return all rows from the products table, potentially exposing sensitive data.

To prevent SQL Injection attacks, it is essential to use parameterized queries, bind variables, and input validation. This booklet will cover these best practices in detail to help you secure your Oracle PL/SQL applications.

### Use DBMS\_ASSERT Package

The DBMS\_ASSERT package provides functions to validate and sanitize user inputs.

PROCEDURE GetEmployeeDetails(p\_emp\_id IN VARCHAR2) IS

 v\_emp\_id NUMBER;

 v\_emp\_name VARCHAR2(100);

BEGIN

 v\_emp\_id := DBMS\_ASSERT.simple\_sql\_name(p\_emp\_id);

 SELECT emp\_name INTO v\_emp\_name FROM employees WHERE emp\_id = v\_emp\_id;

 DBMS\_OUTPUT.put\_line('Employee Name: ' || v\_emp\_name);

END;

### Use Static SQL Whenever Possible

Static SQL is precompiled and does not allow for dynamic user input, reducing the risk of SQL injection.

PROCEDURE GetEmployeeDetails(p\_emp\_id IN NUMBER) IS

 v\_emp\_name VARCHAR2(100);

BEGIN

 SELECT emp\_name INTO v\_emp\_name FROM employees WHERE emp\_id = p\_emp\_id;

 DBMS\_OUTPUT.put\_line('Employee Name: ' || v\_emp\_name);

END;

## Validate and Sanitize User Inputs

Always validate and sanitize user inputs to ensure they conform to expected formats and values.

PROCEDURE ValidateInput(p\_input IN VARCHAR2) IS

BEGIN

 IF NOT REGEXP\_LIKE(p\_input, '^[0-9]+$') THEN

 RAISE\_APPLICATION\_ERROR(-20001, 'Invalid input');

 END IF;

END;

## Use Least Privilege Principle

The principle of least privilege is a fundamental security concept that involves granting users and applications the minimum level of access necessary to perform their tasks. In the context of Oracle databases, this principle helps to minimize the risk of unauthorized access, data breaches, and other security incidents. Here are some key aspects and examples of implementing the least privilege principle in Oracle:

### Role-Based Access Control (RBAC):

* + Create roles with specific privileges and assign them to users based on their job functions.
	+ Example:

-- Create a role with specific privileges

CREATE ROLE read\_only\_role;

GRANT SELECT ON employees TO read\_only\_role;

-- Assign the role to a user

GRANT read\_only\_role TO user1;

### Granting Specific Privileges:

* + Instead of granting broad privileges like DBA, grant only the specific privileges required for a user to perform their tasks.
	+ Example:

-- Grant specific privileges to a user

GRANT SELECT, INSERT ON employees TO user2;

### Using Secure Application Roles:

* + Create application roles that are enabled only when certain conditions are met, such as when a user is authenticated through the application.
	+ Example:

-- Create a secure application role

CREATE ROLE app\_role IDENTIFIED USING app\_role\_pkg;

-- Grant privileges to the role

GRANT SELECT, UPDATE ON employees TO app\_role;

-- Package to enable the role

CREATE OR REPLACE PACKAGE app\_role\_pkg AS

 FUNCTION enable\_role RETURN BOOLEAN;

END app\_role\_pkg;

/

CREATE OR REPLACE PACKAGE BODY app\_role\_pkg AS

 FUNCTION enable\_role RETURN BOOLEAN IS

 BEGIN

 -- Logic to verify if the role should be enabled

 RETURN TRUE;

 END enable\_role;

END app\_role\_pkg;

/

### Limiting System Privileges:

* + Avoid granting powerful system privileges like CREATE ANY TABLE or DROP ANY TABLE unless absolutely necessary.
	+ Example:

-- Grant limited system privileges

GRANT CREATE SESSION TO user3;

### Using Fine-Grained Access Control (FGAC):

* + Implement FGAC to enforce row-level security based on user roles or other criteria.
	+ Example:

-- Create a policy function

CREATE OR REPLACE FUNCTION emp\_policy (schema\_name IN VARCHAR2, table\_name IN VARCHAR2)

RETURN VARCHAR2 IS

BEGIN

 RETURN 'dept\_id = SYS\_CONTEXT(''USERENV'', ''SESSION\_USER'')';

END emp\_policy;

/

-- Apply the policy to the table

BEGIN

 DBMS\_RLS.ADD\_POLICY(

 object\_schema => 'HR',

 object\_name => 'EMPLOYEES',

 policy\_name => 'EMP\_POLICY',

 function\_schema => 'HR',

 policy\_function => 'emp\_policy'

 );

END;

/

By adhering to the principle of least privilege, you can significantly enhance the security of your Oracle database environment. This booklet will cover detailed examples and best practices for implementing least privilege principles to help you build secure and robust PL/SQL applications.

## Use PL/SQL Packages and Procedures

Encapsulate SQL statements within PL/SQL packages and procedures to control access and reduce the risk of SQL injection.

PACKAGE EmployeePackage IS

 PROCEDURE GetEmployeeDetails(p\_emp\_id IN NUMBER);

END EmployeePackage;

PACKAGE BODY EmployeePackage IS

 PROCEDURE GetEmployeeDetails(p\_emp\_id IN NUMBER) IS

 v\_emp\_name VARCHAR2(100);

 BEGIN

 SELECT emp\_name INTO v\_emp\_name FROM employees WHERE emp\_id = p\_emp\_id;

 DBMS\_OUTPUT.put\_line('Employee Name: ' || v\_emp\_name);

 END;

END EmployeePackage;

## Regular Security Audits and Code Reviews

Conduct regular security audits and code reviews to identify and fix potential vulnerabilities in your PL/SQL code.

# Using the DBMS\_SQL package

The Oracle package DBMS\_SQL is a powerful tool for executing dynamic SQL statements and PL/SQL blocks that you cannot otherwise do with NDS. Here are some reasons why you should consider using DBMS\_SQL, along with examples:

* **Flexibility**: DBMS\_SQL allows you to construct and execute SQL statements dynamically at runtime. This is particularly useful when the structure of the SQL statement is not known until runtime, such as when building complex queries based on user input or application logic.
* **Handling Dynamic Queries**: With DBMS\_SQL, you can handle dynamic queries that involve *varying* numbers of columns or parameters. This is especially useful for applications that need to generate SQL statements on the fly based on different conditions.
* **Binding Variables**: DBMS\_SQL supports binding variables, which helps in preventing SQL Injection attacks. By using bind variables, you can safely include user input in your SQL statements without risking the integrity of your database.
* **Performance**: DBMS\_SQL can improve performance in certain scenarios by allowing you to parse and execute SQL statements separately. This can be beneficial when executing the same statement multiple times with different bind variables.
* **Error Handling**: DBMS\_SQL provides detailed error handling capabilities, allowing you to capture and handle exceptions that occur during the execution of dynamic SQL statements. This can help in building robust and reliable applications.
* **Compatibility**: DBMS\_SQL is compatible with all versions of Oracle Database, making it a versatile choice for developers working with different Oracle environments.
* **Advanced Features**: DBMS\_SQL offers advanced features such as bulk operations, cursor management, and the ability to describe columns and rows dynamically. These features can help you build more efficient and scalable applications.

## List of Procedures

Here is a list of common procedures in the DBMS\_SQL package:

Procedure Description

* OPEN\_CURSOR Opens a new cursor and returns its handle.
* PARSE Parses a SQL statement.
* BIND\_VARIABLE Binds a value to a variable in a SQL statement.
* EXECUTE Executes a parsed SQL statement.
* FETCH\_ROWS Fetches rows from the result set of a query.
* COLUMN\_VALUE Retrieves the value of a column in the current row of the result set.
* CLOSE\_CURSOR Closes a cursor.
* DEFINE\_COLUMN Defines a column in the result set.
* VARIABLE\_VALUE Retrieves the value of a bind variable.
* DEFINE\_ARRAY Defines an array for bulk fetch operations.
* BIND\_ARRAY Binds an array to a variable in a SQL statement for bulk operations.
* DESCRIBE\_COLUMNS Describes the columns in a query result set.
* DESCRIBE\_COLUMNS2 Describes the columns in a query result set, including column types and lengths.
* IS\_OPEN Checks if a cursor is open.
* LAST\_ERROR\_POSITION Returns the position of the last error in a SQL statement.
* LAST\_ROW\_COUNT Returns the number of rows affected by the last executed SQL statement.
* LAST\_ROW\_ID Returns the row ID of the last row affected by the last executed SQL statement.
* TO\_CURSOR\_NUMBER Converts a cursor handle to a cursor number.
* TO\_REFCURSOR Converts a cursor number to a ref cursor.
* CLOSE\_ALL Closes all open cursors.

## Examples of using DBMS\_SQL:

## Executing a Dynamic Query:

DECLARE

 v\_cursor NUMBER;

 v\_query VARCHAR2(1000);

 v\_emp\_name VARCHAR2(50);

BEGIN

 v\_cursor := DBMS\_SQL.OPEN\_CURSOR;

 v\_query := 'SELECT ename FROM emp WHERE empno = :empno';

 DBMS\_SQL.PARSE(v\_cursor, v\_query, DBMS\_SQL.NATIVE);

 DBMS\_SQL.BIND\_VARIABLE(v\_cursor, ':empno', 7369);

 DBMS\_SQL.DEFINE\_COLUMN(v\_cursor, 1, v\_emp\_name, 50);

 DBMS\_SQL.EXECUTE(v\_cursor);

 IF DBMS\_SQL.FETCH\_ROWS(v\_cursor) > 0 THEN

 DBMS\_SQL.COLUMN\_VALUE(v\_cursor, 1, v\_emp\_name);

 DBMS\_OUTPUT.PUT\_LINE('Employee Name: ' || v\_emp\_name);

 END IF;

 DBMS\_SQL.CLOSE\_CURSOR(v\_cursor);

EXCEPTION

 WHEN OTHERS THEN

 IF DBMS\_SQL.IS\_OPEN(v\_cursor) THEN

 DBMS\_SQL.CLOSE\_CURSOR(v\_cursor);

 END IF;

 RAISE;

END;

In this example, a dynamic query is constructed and executed using DBMS\_SQL, with a bind variable to prevent SQL Injection.

## Handling Dynamic Columns:

DECLARE

 v\_cursor NUMBER;

 v\_query VARCHAR2(1000);

 v\_col\_value VARCHAR2(50);

 v\_col\_count INTEGER;

 v\_col\_name VARCHAR2(50);

BEGIN

 v\_cursor := DBMS\_SQL.OPEN\_CURSOR;

 v\_query := 'SELECT \* FROM emp WHERE empno = :empno';

 DBMS\_SQL.PARSE(v\_cursor, v\_query, DBMS\_SQL.NATIVE);

 DBMS\_SQL.BIND\_VARIABLE(v\_cursor, ':empno', 7369);

 DBMS\_SQL.EXECUTE(v\_cursor);

 v\_col\_count := DBMS\_SQL.COLUMN\_COUNT(v\_cursor);

 FOR i IN 1..v\_col\_count LOOP

 DBMS\_SQL.DESCRIBE\_COLUMN(v\_cursor, i, v\_col\_name, v\_col\_value);

 DBMS\_SQL.DEFINE\_COLUMN(v\_cursor, i, v\_col\_value, 50);

 END LOOP;

 IF DBMS\_SQL.FETCH\_ROWS(v\_cursor) > 0 THEN

 FOR i IN 1..v\_col\_count LOOP

 DBMS\_SQL.COLUMN\_VALUE(v\_cursor, i, v\_col\_value);

 DBMS\_OUTPUT.PUT\_LINE(v\_col\_name || ': ' || v\_col\_value);

 END LOOP;

 END IF;

 DBMS\_SQL.CLOSE\_CURSOR(v\_cursor);

EXCEPTION

 WHEN OTHERS THEN

 IF DBMS\_SQL.IS\_OPEN(v\_cursor) THEN

 DBMS\_SQL.CLOSE\_CURSOR(v\_cursor);

 END IF;

 RAISE;

END;

This example demonstrates how to handle dynamic columns using DBMS\_SQL, allowing you to describe and fetch column values dynamically.

By leveraging the capabilities of DBMS\_SQL, you can create dynamic, flexible, and secure PL/SQL applications that can adapt to a wide range of requirements. This booklet will cover detailed examples and best practices for using DBMS\_SQL to help you get the most out of this powerful package.

# Using the DBMS\_ASSERT Package

The DBMS\_ASSERT package in Oracle is a valuable tool for enhancing the security of your PL/SQL applications by helping to prevent SQL Injection attacks. Here are some reasons why you should use DBMS\_ASSERT, along with examples:

1. **Input Validation**: DBMS\_ASSERT provides functions to validate and sanitize user inputs, ensuring that they conform to expected formats. This helps prevent malicious inputs from being executed as part of SQL statements.
2. **Preventing SQL Injection**: By using DBMS\_ASSERT functions, you can ensure that user inputs do not contain harmful SQL code. This is crucial for protecting your database from SQL Injection attacks.
3. **Ease of Use**: The package offers a straightforward way to validate inputs without requiring complex custom validation logic. This makes it easier to implement security measures in your applications.
4. **Standardization**: Using DBMS\_ASSERT helps standardize input validation across your application, ensuring consistent security practices.

## List of Procedures

|  |  |
| --- | --- |
| Procedure Name | Description |
| ENQUOTE\_LITERAL | Enquotes a string literal by adding leading and trailing single quotes. |
| ENQUOTE\_NAME | Ensures that a string is enclosed by quotation marks and checks its validity. |
| NOOP | Returns the input value without any checking. |
| QUALIFIED\_SQL\_NAME | Verifies that the input string is a qualified SQL name. |
| SCHEMA\_NAME | Verifies that the input string is an existing schema name. |
| SIMPLE\_SQL\_NAME | Verifies that the input string is a simple SQL name. |
| SQL\_OBJECT\_NAME | Verifies that the input string is a qualified SQL identifier of an existing SQL object. |

## Examples of using DBMS\_ASSERT

### Validating a Single Quoted Literal:

DECLARE

 v\_input VARCHAR2(100) := 'user\_input';

 v\_validated\_input VARCHAR2(100);

BEGIN

 v\_validated\_input := DBMS\_ASSERT.SIMPLE\_SQL\_NAME(v\_input);

 DBMS\_OUTPUT.PUT\_LINE('Validated Input: ' || v\_validated\_input);

EXCEPTION

 WHEN OTHERS THEN

 DBMS\_OUTPUT.PUT\_LINE('Invalid Input');

END;

In this example, DBMS\_ASSERT.SIMPLE\_SQL\_NAME ensures that v\_input is a valid SQL name, preventing SQL Injection.

### Validating a Qualified SQL Name:

DECLARE

 v\_input VARCHAR2(100) := 'schema.table';

 v\_validated\_input VARCHAR2(100);

BEGIN

 v\_validated\_input := DBMS\_ASSERT.QUALIFIED\_SQL\_NAME(v\_input);

 DBMS\_OUTPUT.PUT\_LINE('Validated Input: ' || v\_validated\_input);

EXCEPTION

 WHEN OTHERS THEN

 DBMS\_OUTPUT.PUT\_LINE('Invalid Input');

END;

This example uses DBMS\_ASSERT.QUALIFIED\_SQL\_NAME to validate that v\_input is a properly qualified SQL name.

### Validating a SQL Object Name:

DECLARE

 v\_input VARCHAR2(100) := 'table\_name';

 v\_validated\_input VARCHAR2(100);

BEGIN

 v\_validated\_input := DBMS\_ASSERT.SQL\_OBJECT\_NAME(v\_input);

 DBMS\_OUTPUT.PUT\_LINE('Validated Input: ' || v\_validated\_input);

EXCEPTION

 WHEN OTHERS THEN

 DBMS\_OUTPUT.PUT\_LINE('Invalid Input');

END;

Here, DBMS\_ASSERT.SQL\_OBJECT\_NAME ensures that v\_input is a valid SQL object name.

By incorporating DBMS\_ASSERT into your PL/SQL code, you can enhance the security of your applications and protect against SQL Injection attacks. This booklet will cover best practices for using DBMS\_ASSERT to help you secure your Oracle PL/SQL applications effectively.