Safeguarding Your Code Against SQL Injection Attacks
Objectives

After completing this lesson, you should be able to do the following:

• Describe SQL injections
• Reduce attack surfaces
• Use DBMS_ASSERT
• Design immune code
• Test code for SQL injection flaws
Lesson Agenda

• Understanding SQL injection
• Reducing the attack surface
• Avoiding dynamic SQL
• Using bind arguments
• Filtering input with DBMS_ASSERT
• Designing code immune to SQL injections
• Testing code for SQL injection flaws
Understanding SQL Injection

SQL injection is a technique for maliciously exploiting applications that use client-supplied data in SQL statements.

- Attackers trick the SQL engine into executing unintended commands.
- SQL injection techniques may differ, but they all exploit a single vulnerability in the application.
- To immunize your code against SQL injection attacks, use bind arguments or validate and sanitize all input concatenated to dynamic SQL.
## Identifying Types of SQL Injection Attacks

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>First-order attack</td>
<td>The attacker can simply enter a malicious string and cause the modified code to be executed immediately.</td>
</tr>
<tr>
<td>Second-order attack</td>
<td>The attacker injects into persistent storage (such as a table row), which is deemed a trusted source. An attack is subsequently executed by another activity.</td>
</tr>
</tbody>
</table>
SQL Injection: Example

```sql
-- First order attack
CREATE OR REPLACE PROCEDURE GET_EMAIL
    (p_last_name VARCHAR2 DEFAULT NULL)
AS
    TYPE    cv_custtyp IS REF CURSOR;
    cv      cv_custtyp;
    v_email customers.cust_email%TYPE;
    v_stmt  VARCHAR2(400);
BEGIN
    v_stmt := 'SELECT cust_email FROM customers
    WHERE cust_last_name = ''' || p_last_name || '''';
    DBMS_OUTPUT.PUT_LINE('SQL statement: ' || v_stmt);
    OPEN cv FOR v_stmt;
    LOOP
        FETCH cv INTO v_email;
        EXIT WHEN cv%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Email: ' || v_email);
    END LOOP;
    CLOSE cv;
    EXCEPTION WHEN OTHERS THEN
        dbms_output.PUT_LINE(sqlerrm);
        dbms_output.PUT_LINE('SQL statement: ' || v_stmt);
END;
```

String literals that are incorrectly validated or not validated are concatenated into a dynamic SQL statement, and interpreted as code by the SQL engine.
Assessing Vulnerability

- Code
  - Contains dynamic SQL?
    - Yes
    - No
  - Uses bind arguments for all dynamic components
    - Yes
    - No
  - Lateral injection vulnerability
    - No
    - Yes
  - Filters input correctly?
    - Yes
    - No
- Safe
- Vulnerable
## Avoidance Strategies Against SQL Injection

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce the attack surface</td>
<td>Ensure that all excess database privileges are revoked and that only those routines that are intended for end-user access are exposed. Though this does not entirely eliminate SQL injection vulnerabilities, it does mitigate the impact of the attacks.</td>
</tr>
<tr>
<td>Avoid dynamic SQL with concatenated input</td>
<td>Dynamic SQL built with concatenated input values presents the easiest entry point for SQL injections. Avoid constructing dynamic SQL this way.</td>
</tr>
<tr>
<td>Use bind arguments</td>
<td>Parameterize queries using bind arguments. Not only do bind arguments eliminate the possibility of SQL injections, they also enhance performance.</td>
</tr>
<tr>
<td>Filter and sanitize input</td>
<td>The Oracle-supplied <code>DBMS_ASSERT</code> package contains a number of functions that can be used to sanitize user input and to guard against SQL injection in applications that use dynamic SQL built with concatenated input values. If your filtering requirements cannot be satisfied by the <code>DBMS_ASSERT</code> package, create your own filter.</td>
</tr>
</tbody>
</table>
Protecting Against SQL Injection: Example

CREATE OR REPLACE PROCEDURE GET_EMAIL
    (p_last_name VARCHAR2 DEFAULT NULL)
AS
    BEGIN
        FOR i IN
            (SELECT cust_email
             FROM customers
             WHERE cust_last_name = p_last_name)
        LOOP
            DBMS_OUTPUT.PUT_LINE('Email: ' || i.cust_email);
        END LOOP;
    END;

EXECUTE get_email('Andrews')
Email: Ajay.Andrews@YELLOWTHROAT.COM
Email: Dianne.Andrews@TURNSTONE.COM

PL/SQL procedure successfully completed.

EXECUTE get_email('x'' union select username from all_users where ''x''=''x'')

PL/SQL procedure successfully completed.

This example avoids dynamic SQL with concatenated input values.
Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with `DBMS_ASSERT`
- Designing code immune to SQL injections
- Testing code for SQL injection flaws
Reducing the Attack Surface

Use the following strategies to reduce attack surface:

- Expose the database only via a PL/SQL API.
- Use invoker’s rights.
- Reduce arbitrary inputs.
- Strengthen database security.
Expose the Database Only Via PL/SQL API

- Expose the database to clients only via a PL/SQL API.
- When you design a PL/SQL package that accesses the database, use the following paradigm:
  - Establish a database user as the *only* one to which a client may connect. Hypothetically, let us call this user `myuser`.
  - `myuser` may own *only* synonyms and these synonyms may denote *only* PL/SQL units owned by other users.
  - Grant the Execute privilege on *only* the denoted PL/SQL units to `myuser`. 
Using Invoker’s Rights

• Using invoker’s rights:
  – Helps to limit the privileges
  – Helps to minimize the security exposure.

• The following example does not use invoker's rights:

```sql
CREATE OR REPLACE
PROCEDURE change_password(p_username VARCHAR2 DEFAULT NULL,
                           p_new_password VARCHAR2 DEFAULT NULL)
IS
  v_sql_stmt VARCHAR2(500);
BEGIN
  v_sql_stmt := 'ALTER USER ' || p_username || ' IDENTIFIED BY ' || p_new_password;
  EXECUTE IMMEDIATE v_sql_stmt;
END change_password;
```

GRANT EXECUTE ON change_password to OE, HR, SH;

Note the use of dynamic SQL with concatenated input values.
Using Invoker’s Rights

- OE is successful at changing the SYS password, because, by default, `CHANGE_PASSWORD` executes with SYS privileges:

```sql
EXECUTE sys.change_password ('SYS', 'mine')
```

- Add the `AUTHID` to change the privileges to the invokers:

```sql
CREATE OR REPLACE PROCEDURE change_password(p_username VARCHAR2 DEFAULT NULL,
                                            p_new_password VARCHAR2 DEFAULT NULL)
AUTHID CURRENT_USER
IS
  v_sql_stmt VARCHAR2(500);
BEGIN
  v_sql_stmt := 'ALTER USER ''||p_username ||'' IDENTIFIED BY '' || p_new_password;
  EXECUTE IMMEDIATE v_sql_stmt;
END change_password;
```
Reducing Arbitrary Inputs

• Reduce the end-user interfaces to only those that are actually needed.
  – In a Web application, restrict users to accessing specified Web pages.
  – In a PL/SQL API, expose only those routines that are intended for customer use.

• Where user input is required, make use of language features to ensure that only data of the intended type can be specified.
  – Do not specify a `VARCHAR2` parameter when it will be used as a number.
  – Do not use numbers if you need only positive integers; use natural instead.
Strengthen Database Security

• Here is a list of some of the practices to observe when you secure the Oracle database:
  – Encrypt sensitive data so that it cannot be viewed.
  – Avoid the following:
    – PUBLIC privileges
    – EXECUTE ANY PROCEDURE privilege
    – Privileges WITH ADMIN option
  – Do not allow wide access to any standard Oracle packages that can operate on the operating system.
  – Lock the database default accounts and expire the default passwords.
  – Enforce password management.
  – Lock and expire the default user accounts and change the default user password.
Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with `DBMS_ASSERT`
- Designing code immune to SQL injections
- Testing code for SQL injection flaws
Using Static SQL

- Eliminates SQL injection vulnerability
- Creates schema object dependencies upon successful compilation
- Can improve performance, when compared with DBMS_SQL
CREATE OR REPLACE PROCEDURE list_products_dynamic
(p_product_name VARCHAR2 DEFAULT NULL)
AS

TYPE cv_prodtyp IS REF CURSOR;
cv   cv_prodtyp;
v_prodname product_information.product_name%TYPE;
v_minprice product_information.min_price%TYPE;
v_listprice product_information.list_price%TYPE;
v_stmt  VARCHAR2(400);
BEGIN
  v_stmt := 'SELECT product_name, min_price, list_price
            FROM product_information WHERE product_name LIKE
            ''%'||p_product_name||'%''';
  OPEN cv FOR v_stmt;
  DBMS_OUTPUT.PUT_LINE(v_stmt);
  LOOP
    FETCH cv INTO v_prodname, v_minprice, v_listprice;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Product Info: ''|'||v_prodname ||'', ''|'
                           v_minprice ||'', ''| | v_listprice);
  END LOOP;
  CLOSE cv;
END;
Using Static SQL

- To use static SQL, accept the user input, and then concatenate the necessary string to a local variable.
- Pass the local variable to the static SQL statement.

```sql
CREATE OR REPLACE PROCEDURE list_products_static
  (p_product_name VARCHAR2 DEFAULT NULL)
AS
  v_bind   VARCHAR2(400);
BEGIN
  v_bind := '%' || p_product_name || '%';
  FOR i in (
    SELECT product_name, min_price, list_price
    FROM product_information
    WHERE product_name like v_bind)
  LOOP
    DBMS_OUTPUT.PUT_LINE('Product Info: ' || i.product_name || ', ' || i.min_price || ', ' || i.list_price);
  END LOOP;
END list_products_static;
```
Using Dynamic SQL

- Dynamic SQL may be unavoidable in the following types of situations:
  - You do not know the full text of the SQL statements that must be executed in a PL/SQL procedure.
  - You want to execute DDL statements and other SQL statements that are not supported in purely static SQL programs.
  - You want to write a program that can handle changes in data definitions without the need to recompile.

- If you must use dynamic SQL, try not to construct it through concatenation of input values. Instead, use bind arguments.
Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with `DBMS_ASSERT`
- Designing code immune to SQL injections
- Testing code for SQL injection flaws
Using Bind Arguments with Dynamic SQL

You can rewrite the following statement as the following dynamic SQL with a placeholder (:1) by using a bind argument (p_sales_rep_id):

```sql
v_stmt :=
    'SELECT ' || filter(p_column_list) || ' FROM customers ' ||
    'WHERE account_mgr_id = '''|| p_sales_rep_id ||'''';

EXECUTE IMMEDIATE v_stmt;
```

```sql
v_stmt :=
    'SELECT ' || filter(p_column_list) || ' FROM customers ' ||
    'WHERE account_mgr_id = :1';

EXECUTE IMMEDIATE v_stmt USING p_sales_rep_id;
```
Using Bind Arguments with Dynamic PL/SQL

If you must use dynamic PL/SQL, try to use bind arguments. For example, you can rewrite the following dynamic PL/SQL with concatenated string values:

```sql
v_stmt :=
'BEGIN
    get_phone (''|| p_fname ||
    '::', ''|| p_lname ||''') ; END;';
EXECUTE IMMEDIATE v_stmt;
```

as the following dynamic PL/SQL with placeholders (:1, :2) by using bind arguments (p_fname, p_lname):

```sql
v_stmt :=
'BEGIN
    get_phone(:1, :2) ; END;';
EXECUTE IMMEDIATE v_stmt USING p_fname, p_lname;
```
What If You Cannot Use Bind Arguments?

- Bind arguments cannot be used with:
  - DDL statements
  - Oracle identifiers
- If bind arguments cannot be used with the dynamic SQL or PL/SQL, you must filter and sanitize all input concatenated to the dynamic statement.
Lesson Agenda

• Understanding SQL injection
• Reducing the attack surface
• Avoiding dynamic SQL
• Using bind arguments
• Filtering input with `DBMS_ASSERT`
• Designing code immune to SQL injections
• Testing code for SQL injection flaws
Understanding DBMS_ASSERT

DBMS_ASSERT functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENQUOTE_LITERAL</td>
<td>Encloses string literal in single quotes</td>
</tr>
<tr>
<td>SIMPLE_SQL_NAME</td>
<td>Verifies that the string is a simple SQL name</td>
</tr>
</tbody>
</table>
Formatting Oracle Identifiers

- **Example 1:** The object name used as an identifier:

  ```sql
  SELECT count(*) records FROM orders;
  ```

- **Example 2:** The object name used as a literal:

  ```sql
  SELECT num_rows FROM user_tables
  WHERE table_name = 'ORDERS';
  ```

- **Example 3:** The object name used as a quoted (normal format) identifier:
  - The "orders" table referenced in example 3 is a different table compared to the `orders` table in examples 1 and 2.
  - It is vulnerable to SQL injection.

  ```sql
  SELECT count(*) records FROM "orders";
  ```
Working with Identifiers in Dynamic SQL

• For your identifiers, determine:
  1. Where will the input come from: user or data dictionary?
  2. What verification is required?
  3. How will the result be used, as an identifier or a literal value?

• These three factors affect:
  – What preprocessing is required (if any) prior to calling the verification functions
  – Which DBMS_ASSERT verification function is required
  – What post-processing is required before the identifier can actually be used
# Choosing a Verification Route

<table>
<thead>
<tr>
<th>Identifier Type</th>
<th>Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL literal</td>
<td>Verify whether the literal is a well-formed SQL literal by using <code>DBMS_ASSERT.ENQUOTE_LITERAL</code>.</td>
</tr>
<tr>
<td>Simple SQL name</td>
<td>Verify that the input string conforms to the basic characteristics of a simple SQL name by using <code>DBMS_ASSERT.SIMPLE_SQL_NAME</code>.</td>
</tr>
</tbody>
</table>
| Qualified SQL name  | Step 1: Decompose the qualified SQL name into its simple SQL names by using `DBMS_Utility.Name_Tokenize()`.  
                              Step 2: Verify each of the simple SQL names using `DBMS_ASSERT.SIMPLE_SQL_NAME`. |
Avoiding Injection by Using
DBMS_ASSERT.ENQUOTE_LITERAL

CREATE OR REPLACE PROCEDURE Count_Rows(w in varchar2)
authid definer as
Quote constant varchar2(1) := ''';
Quote_Quote constant varchar2(2) := Quote||Quote;
Safe_Literal varchar2(32767) :=
    Quote||replace(w,Quote,Quote_Quote)||Quote;
Stmt constant varchar2(32767) :=
    'SELECT count(*) FROM t WHERE a='||
    DBMS_ASSERT.ENQUOTE_LITERAL(Safe_Literal);
Row_Count number;
BEGIN
    EXECUTE IMMEDIATE Stmt INTO Row_Count;
    DBMS_OUTPUT.PUT_LINE(Row_Count||' rows');
END;/

Verify whether the literal is well-formed.
Avoiding Injection by Using
\texttt{DBMS\_ASSERT\_SIMPLE\_SQL\_NAME}

\begin{verbatim}
CREATE OR REPLACE PROCEDURE show_col2 (p_colname varchar2,
p_tablename   varchar2) AS
type t is varray(200) of varchar2(25);
Results t;
Stmt CONSTANT VARCHAR2(4000) :=
  'SELECT ' || dbms_assert.simple_sql_name( p_colname ) || ' FROM '
  || dbms_assert.simple_sql_name( p_tablename ) ;
BEGIN
  DBMS_Output.Put_Line ('SQL Stmt: ' || Stmt);
  EXECUTE IMMEDIATE Stmt bulk collect into Results;
  for j in 1..Results.Count() loop
    DBMS_Output.Put_Line(Results(j));
  end loop;
END show_col2;
\end{verbatim}

Verify that the input string conforms to the basic characteristics of a simple SQL name.
DBMS_ASSERT Guidelines

• Do not perform unnecessary uppercase conversions on identifiers.

---Bad:
SAFE_SCHEMA := sys.dbms_assert.SIMPLE_SQL_NAME(UPPER(MY_SCHEMA));
---Good:
SAFE_SCHEMA := sys.dbms_assert.SIMPLE_SQL_NAME(MY_SCHEMA);
---Best:
SAFE_SCHEMA := sys.dbms_assert.ENQUOTE_NAME(
SAFE_SCHEMA := sys.dbms_assert.ENQUOTE_LITERAL(
    sys.dbms_assert.SIMPLE_SQL_NAME(MY_SCHEMA));

• When using ENQUOTE_LITERAL, do not add unnecessary double quotes around identifiers.

---Bad:
my_trace_routine(''||sys.dbms_assert.ENQUOTE_LITERAL(
my_procedure_name)||'');'||...
---Good:
my_trace_routine(''||sys.dbms_assert.ENQUOTE_LITERAL(
replace(my_procedure_name,'''','''''')||'');'||...
**DBMS_ASSERT Guidelines**

- Check and reject **NULL** or empty return results from `DBMS_ASSERT` (test for **NULL**, `' '`, and `' " " ' `).
- Prefix all calls to `DBMS_ASSERT` with the owning schema, `SYS`.
- Protect all injectable parameters and code paths.
DBMS_ASSERT Guidelines

• If DBMS_ASSERT exceptions are raised from a number of input strings, define and raise exceptions explicitly to ease debugging during application development.

-- Bad
CREATE OR REPLACE PROCEDURE change_password3
  (username VARCHAR2, password VARCHAR2)
AS
BEGIN
  ...
EXCEPTION WHEN OTHERS THEN
  RAISE;
END;
Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with `DBMS_ASSERT`
- Designing code immune to SQL injections
- Testing code for SQL injection flaws
Using Bind Arguments

• Most common vulnerability:
  – Dynamic SQL with string concatenation

• Your code design must:
  – Avoid input string concatenation in dynamic SQL
  – Use bind arguments, whether automatically via static SQL or explicitly via dynamic SQL statements
Avoiding Privilege Escalation

- Give out privileges appropriately.
- Run code with invoker’s rights when possible.
- Ensure that the database privilege model is upheld when using definer’s rights.
Beware of Filter Parameters

• Filter parameter:
  – `P_WHERE_CLAUSE` is a filter.
  – It is difficult to protect against SQL injection.

```sql
stmt := 'SELECT session_id FROM sessions WHERE' || p_where_clause;
```

• Prevention methods:
  – Do not specify APIs that allow arbitrary query parameters to be exposed.
  – Any existing APIs with this type of functionality must be deprecated and replaced with safe alternatives.
Trapping and Handling Exceptions

• Design your code to trap and handle exceptions appropriately.
• Before deploying your application:
  – Remove all code tracing
  – Remove all debug messages
Lesson Agenda

• Understanding SQL injection
• Reducing the attack surface
• Avoiding dynamic SQL
• Using bind arguments
• Filtering input with `DBMS_ASSERT`
• Designing code immune to SQL injections
• Testing code for SQL injection flaws
Coding Review and Testing Strategy

• Test:
  – Dynamic testing
  – Static testing

• Review:
  – Peer and self reviews
  – Analysis tools
## Reviewing Code

<table>
<thead>
<tr>
<th>Language</th>
<th>Look for...</th>
</tr>
</thead>
</table>
| **PL/SQL** | **EXECUTE IMMEDIATE**  
**OPEN cursor_var FOR ...**  
**DBMS_SQL**  
**DBMS_SYS_SQL** |
| **C** | String substitutions such as:  
`static const oratext createsq[] =  
"CREATE SEQUENCE ""%.*s"".""%.*s"" start with %.*s  
increment by  
%.*s";`  
Followed by usage such as:  
**DISCARD lstprintf(sql_txt, createsq, ownerl, owner,  
seqnam1, seqnam, sizeof(start), start,  
sinc_by1, sincrement_by);** |
| **Java** | String concatenations such as:  
`sqltext = "DROP VIEW " + this.username + "." +  
this.viewName;` |
Running Static Code Analysis

- Generally performed by an automated tool
- Can be performed on some versions of the source code
- Can be performed on some forms of the object code
- Should be used as one of the initial steps of testing code
Testing with Fuzzing Tools

• Is a software testing technique that provides random data (“fuzz”) to the inputs of a program
• Can enhance software security and software safety, because it often finds odd oversights and defects that human testers would fail to find
• Must not be used as a substitute for exhaustive testing or formal methods
• While tools can be used to automate fuzz testing, it may be necessary to customize the test data and application context to get the best results from such tools
Generating Test Cases

- Test each input parameter individually.
- When testing each parameter, leave all the other parameters unchanged with valid data as their arguments.
- Omitting parameters or supplying bad arguments to other parameters while you are testing another for SQL injection can break the application in ways that prevent you from determining whether SQL injection is possible.
- Always use the full parameter line, supplying every parameter, except the one that you are testing, with a legitimate value.
- Certain object and schema names help you uncover SQL injection vulnerabilities.
Quiz

Code that is most vulnerable to SQL Injection attack contains:

a. Input parameters
b. Dynamic SQL with bind arguments
c. Dynamic SQL with concatenated input values
d. Calls to external functions
Quiz

By default, a stored procedure executes with the privileges of its owner (definer’s rights).

a. True
b. False
Quiz

If you must use dynamic SQL, avoid using input concatenation to build the dynamic SQL.

a. True
b. False
Quiz

In the statement, `SELECT total FROM orders WHERE ord_id=p_ord_id`, the table name `orders` is being used as which of the following?

a. A literal  
b. An identifier  
c. A placeholder  
d. An argument
Summary

In this lesson, you should have learned how to:

- Detect SQL injection vulnerabilities
- Reduce attack surfaces
- Use `DBMS_ASSERT`
- Design immune code
- Test code for SQL injection flaws
Practice 12: Overview

This practice covers the following topics:

- Testing your knowledge of SQL injection
- Rewriting code to protect against SQL injection