

How to write SQL injection proof PL/SQL

An Oracle White Paper
October 2008

NOTE

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

How to write SQL injection proof PL/SQL

CONTENTS

- Abstract** 1
- Introduction** 2
 - Periodic revision of this paper 3
- Definition of SQL injection** 4
 - Introducing a new notion: SQL syntax template 4
 - Distinguishing between compile-time-fixed SQL statement text and run-time-created SQL statement text 6
 - Distinguishing between a static SQL syntax template and a dynamic SQL syntax template 7
 - Definition of static SQL syntax template 7
 - Definition of dynamic SQL syntax template 8
 - SQL injection (finally) defined 11
- How can SQL injection happen?** 12
 - Example 1: user-supplied column-comparison value 12
 - Example 2: user-supplied table name 14
 - Counter-example 3: user-supplied where clause 17
 - Counter-example 4: SQL syntax template with a questionable intent 17
- Ensuring the safety of a SQL literal or a simple SQL name** 19
 - Ensuring the safety of a SQL literal 19
 - Ensuring the safety of a SQL *text* literal 19
 - Ensuring the safety of a SQL *datetime* literal 20
 - Ensuring the safety of a SQL *numeric* literal 23
 - Ensuring the safety of a simple SQL name 25
- Rules for cost-effective, guaranteed prevention of SQL injection** 28
 - Expose the database to clients only via a PL/SQL API 28
 - Use compile-time-fixed SQL statement text unless you cannot 28
 - Use a static SQL syntax template for run-time-created SQL statement text unless you cannot 30
 - Replacement of a value placeholder in a SQL syntax template 30
 - Replacement of a simple SQL name placeholder in a SQL syntax template 31
 - Don't confuse the need to use a dynamic SQL syntax template with the need for dynamic text 32
 - Formal sufficient prescription for guaranteed safety 33
 - Static text 33
 - Dynamic text 34
 - Safe dynamic text 34
 - Safe SQL statement text 35
 - Establish the safety of run-time-created SQL statement text in the code that immediately precedes its execution 36

Scenarios	38
Make a like predicate by adding leading and trailing % characters	38
In list with number of elements not known until run-time	39
Query by example form	40
Callback	43
Analysis and hardening of extant code	47
Conclusion	48
Appendix A:	
Change History	50
Appendix B:	
Definitions of new terms of art introduced by this paper	51
common SQL name	51
exotic SQL name	51
compile-time-fixed SQL statement text	51
run-time-created SQL statement text	51
SQL syntax template	51
value placeholder	52
simple SQL name placeholder	52
static SQL syntax template	52
dynamic SQL syntax template	52
static text	53
dynamic text	53
safe dynamic text	53
safe SQL statement text	54
top level PL/SQL block	54
Appendix C:	
Summary of SQL injection prevention rules	55
Appendix D:	
Additional Oracle-supplied subprograms that implement dynamic SQL ...	58
<i>DBMS_UTILITY.Exec_DDL_Statement()</i>	58
<i>DBMS_DDL.Create_Wrapped()</i>	59
<i>DBMS_HS_Passthrough</i>	59
<i>DBMS_HS_Passthrough.Execute_Immediate()</i>	59
<i>DBMS_HS_Passthrough.Parse()</i>	59
<i>OWA_Util</i>	59
<i>OWA_Util.Bind_Variables()</i>	60
<i>OWA_Util.ListPrint()</i>	60
<i>OWA_Util.TablePrint()</i>	60
Appendix E:	
Self-contained code to illustrate implementing callback using dynamic polymorphism.	61

How to write SQL injection proof PL/SQL

ABSTRACT

An internet search for “SQL injection” gets about 4 million hits. The topic excites interest and superstitious fear. This whitepaper dymystifies the topic and explains a straightforward approach to writing database PL/SQL programs that provably guarantees their immunity to SQL injection.

Only when a PL/SQL subprogram executes SQL that it creates at run time is there a risk of SQL injection; and you’ll see that it’s easier than you might think to freeze the SQL at PL/SQL compile time. Then you’ll understand that you need the rules which prevent the risk only for the rare scenarios that do require run-time-created SQL. It turns out that these rules are simple to state and easy to follow.

INTRODUCTION

Make sure that you're reading the latest copy of this paper. Check the URL given at the top of each page.

At the time of writing, An internet search for “SQL injection” gets about 4 million hits. The topic excites interest and superstitious fear. This whitepaper demystifies the topic and explains a straightforward approach to writing database PL/SQL programs that provably guarantees their immunity to SQL injection.

The scope of the discussion is strictly limited to PL/SQL units that are stored in the database. Similar principles apply in the discussion of languages, such as C or Java, used to implement client-side programs, but it is very much harder to control access to such programs. It is even harder to ensure that access to the database is made *only* using such client-side programs.

In order best to understand the discussion, the reader must have sound understanding of the various ways that SQL may be executed from a database PL/SQL unit. The whitepaper *Doing SQL from PL/SQL: Best and Worst Practices*¹, addresses this topic in detail. Its study is, therefore, recommended as a prerequisite for the study of this whitepaper. In particular, the *Doing SQL from PL/SQL* whitepaper argues for the strategy that bans direct SQL access to the database and exposes it to the client *only* via a strictly minimal PL/SQL API². If this strategy is adopted, then the proofing against SQL injection is the sole responsibility of database PL/SQL; and a sufficient solution is possible in this regime.

Of course, one cannot avoid what one cannot define — and so we start with the section “*Definition of SQL injection*” on [page 4](#). We use this definition, in the section “*How can SQL injection happen?*” on [page 12](#), to examine some famous examples of code that is vulnerable. We also examine some counter-examples in order to prove our definition of SQL injection.

The discussion in these two sections leads to the understanding that SQL injection is possible only when a PL/SQL subprogram executes a SQL statement whose text it has created at run time using what, here, we can loosely call unchecked user input³. Clearly, then, the best way to avoid SQL injection is to execute only SQL statements whose text derives entirely from the source code of the PL/SQL program that executes it.

However, when the watertight approach will not meet the requirements, it is, after all, necessary to handle user input — and to do so safely. A careful study of this topic is presented in the section “*Ensuring the safety of a SQL literal or a simple SQL name*” on [page 19](#).

The material in these first three sections supports the rationale for, and the understanding of, what follows in the section “*Rules for cost-effective, guaranteed prevention of SQL injection*” on [page 28](#). Indeed, if this paper makes an original contribution, it is in the development of the conceptual framework, and the associated terms of art, that then allow the rules to be stated compactly and precisely. The new terms of art introduced and defined at appropriate points in

-
1. *Doing SQL from PL/SQL: Best and Worst Practices* is published on the Oracle Technology Network website here: www.oracle.com/technology/tech/pl_sql/pdf/doing_sql_from_plsql.pdf
 2. This is discussed in the section “*Expose the database to clients only via a PL/SQL API*” on [page 28](#).
 3. This notion will be formally defined in the section “*Dynamic text*” on [page 34](#).

the discussion. But it proved very hard to find an order of exposition that didn't rely on forward reference to terms not yet defined. Therefore, *Appendix B: Definitions of new terms of art introduced by this paper* on [page 51](#) lists these terms, defines each briefly, and cross-references to the section of the paper where it is introduced.

Next, the section “*Scenarios*” on [page 38](#) discusses some requirements scenarios against which the concepts and the rules developed in the preceding sections can be tested. They seem (to the beginner) to require the use of SQL statements built from user input. However, such scenarios are very much fewer than many programmers think; many can be implemented satisfactorily using SQL statements whose text derives entirely from the source code. Several such scenarios are illustrated.

This paper focuses unashamedly on writing injection-proof *de novo* PL/SQL code. Finally, in the section “*Analysis and hardening of extant code*” on [page 47](#), it turns briefly to the topic of extant, and possibly vulnerable, code.

The rules that this paper explains, and insists on, are reproduced⁴ for quick reference, in *Appendix C: Summary of SQL injection prevention rules* on [page 55](#).

They guarantee proof against injection and yet are surprisingly easy to follow. Moreover, as a bonus, they ensure semantic correctness in edge cases that programmers often overlook.

Periodic revision of this paper

Sadly, but realistically, this paper is likely to have minor spelling and grammar errors. For that reason alone, it is bound to be revised periodically⁵. Feedback from readers might suggest improving some explanations. Therefore, before settling down to study the paper, readers should ensure that they have the latest copy — for which the URL is given in the page's header.

URLs sometimes change. But this one will always take you to the Oracle Technical Network's PL/SQL Technology Center:

www.oracle.com/technology/tech/pl_sql

Even in the unlikely event that the paper is moved, it will still be easy to find from that page.

-
4. This paper was prepared using Adobe Framemaker 8.0. Its cross-reference feature allows the text of a source paragraph to be included by reference at the destination. The reader can be certain, therefore, that the wording of each rule in the quick-reference summary is identical to the wording where it is stated. (Sadly, the mechanism does not preserve font nuances.)
 5. This document's change history is listed at the end. See *Appendix A: Change History* on [page 50](#).

DEFINITION OF SQL INJECTION

The definition of SQL injection requires an informal understanding of the syntax of the SQL language and of how it is parsed.

Consider the two putative SQL statements shown in *Code_1*

```
-- Code_1
select c1 from t where c2 = 'Smith'
```

and *Code_2*.

```
-- Code_2
select c1 from t wear c2 = 'Smith'
```

It is clear, without connecting to an Oracle Database, that *Code_1* is syntactically correct but *Code_2* has a syntax error⁶. Consider next the syntactically correct SQL statements shown in *Code_3*

```
-- Code_3
select a1 from r where a2 = 'Jones'
```

and *Code_4*.

```
-- Code_4
select b1 from s where b2 = :1
```

The token *:1* is a placeholder. If the table *s*, with columns *b1* and *b2*, is accessible, then *Code_4* will parse without error⁷.

Introducing a new notion: SQL syntax template

The SQL statements in *Code_1*, *Code_3*, and *Code_4* are different instances of the same *SQL syntax template*; *Template_1* shows it⁸.

```
-- Template_1
select &&1 from &&2 where &&3 = &4
```

The SQL syntax template notion and the notation used in *Template_1* are invented for the purposes of this whitepaper.

The notion belongs in the domain of discourse of the Design Specification document. This document would list, with the notation used in *Template_1*, the SQL syntax template or templates that are prescribed for a particular purpose. Then the implementer would ensure, using the programming techniques that this paper explains, that only those SQL statements that were instances of the prescribed SQL syntax templates could be issued at the call site that implements that part of the design.

6. The attempt to execute *Code_2* will always cause *ORA-00933: SQL command not properly ended* while the attempt to execute *Code_1* might cause *ORA-00942: table or view does not exist*. *ORA-00933* is a syntax error and *ORA-00942* is a semantic error. Of course, if table *t(c1 varchar2(30), c2 varchar2(30))* is accessible to the current user, then *Code_1* will be parsed without error.

7. This is confirmed by executing this PL/SQL statement:

```
DBMS_Sql.Parse(Cur, 'select b1 from s where b2 = :1', DBMS_Sql.Native);
```

in an appropriately written PL/SQL anonymous block.

8. To emphasize the difference between an ordinary SQL statement and a SQL syntax template, the latter will be rendered using a proportionally spaced italic font.

The `&` syntax device denotes what we will call a *value placeholder*, and the `&&` syntax device⁹ denotes what we will call a *simple SQL name placeholder*. Notice that the value placeholder in a SQL syntax template is not the same notion as a regular placeholder in an ordinary SQL statement. A value placeholder in a SQL syntax template stands for *either* a well-formed SQL literal *or* a regular placeholder in a SQL statement.

A particular SQL syntax template is a specific sequence of intermixed specific keywords, specific operators, simple SQL name placeholders, and value placeholders. The simple SQL name placeholders, and value placeholders imply the possibility of various specializations of the template and so we extend the SQL syntax template notion to include examples where concrete identifiers, literals, and regular placeholders are in use. This is illustrated by the SQL syntax templates shown in [Template_2](#); each is a specialization of the most generic form shown in [Template_1](#).

```
-- Template_2
select c1 from &&1 where c2 = &1

select &&1 from t where &&2 = 99

select c1 from &&1 where c2 = :1

select c1 from t where c2 = :1
```

Notice that the only freedoms available when composing a particular SQL statement as an instance of a particular SQL syntax template are the textual substitutions¹⁰ of value placeholders and simple SQL name placeholders.

- A value placeholder may be replaced with a regular placeholder¹¹ or a well-formed SQL literal whose datatype is either *text*, *datetime*, or *numeric*, according to the specified syntax rules¹².

9. Don't confuse the use of `&` here with its use in the SQL*Plus scripting language. The choice of `&` was, however, made in *homage* to SQL*Plus's use. In both cases, the syntax implies early textual substitution before the "real" processing happens.

10. Notice that by saying "textual substitution" we don't mean that the SQL syntax template is represented as such by a value in PL/SQL source code — so that substitution is done programmatically with, for example, `Replace()`. Rather, we mean that a human who reads an example of an actual SQL statement that the PL/SQL program uses at run time will be able to see that it is an instantiation of the SQL syntax template that the Design Specification document prescribes, achieved by the textual substitution we discuss.

With respect to the auditing of this substitution, we consider whitespace to be insignificant. The Design Specification document can lay out a SQL syntax template as it pleases. And the program is free to use a different layout. Ordinary comments of both styles are just a special case of whitespace. However, when the Design Specification document prescribes the special comment that starts with `*/+` and ends with `/*`, and that expresses a SQL hint, this must be reproduced faithfully in the actual SQL statements that the program instantiates for this SQL syntax template. Just as is the case for, for example, a keyword, a SQL hint is definitely not a candidate for replacement at instantiation time.

11. The freedom to replace a value placeholder with either a regular placeholder or a SQL literal is only of formal interest; it helps to clarify thought. It is very unlikely indeed that a real program would be specified to implement the replacement of value placeholder with a regular placeholder at run time. Such a design would give cause for concern.

12. These rules are specified in the SQL Language Reference book.

- A simple SQL name placeholder may be replaced only with a simple SQL name¹³.
- It is not allowed to substitute any other elements.

In particular, and *by definition* of the notion that this paper introduces, the . and @ characters that are used to form qualified SQL names must be an explicit part of the SQL syntax template; a qualified SQL name may not be used to replace a simple SQL name placeholder¹⁴.

Distinguishing between compile-time-fixed SQL statement text and run-time-created SQL statement text

We define the term *compile-time-fixed SQL statement text* to mean the text of a SQL statement that cannot change at run time and that can be confidently determined by reading the source code. More precisely, it is the text of a SQL statement that is a PL/SQL static *varchar2* expression¹⁵. The value of a PL/SQL static *varchar2* expression cannot change at run time and could be pre-computed at compile time.

The SQL statement text for embedded SQL is composed by the PL/SQL compiler and cannot change at run time. Therefore, embedded SQL definitely executes only compile-time-fixed SQL statement text¹⁶.

However, it can easily be arranged that any of PL/SQL's methods for executing dynamic SQL will, at a particular call site, execute only compile-time-fixed SQL statement text; it is necessary only to establish the statement text as a PL/SQL static *varchar2* expression.

We define the term *run-time-created SQL statement text* to mean the text of a SQL statement that is not compile-time-fixed SQL statement text.

As we shall see, it is the distinction between compile-time-fixed SQL statement text and run-time-created SQL statement text that is significant in the discussion of SQL injection. It is more important to focus on the property of the SQL text than on the method used for its execution. To (we hope!) avoid confusion, the

13. It is simplest to define a simple SQL name as that which the *DBMS_Assert.Simple_Sql_Name()* function will return without raising an exception. Examples are *SCOTT* (which will be treated the same as, for example, *Scott*) and *My Table*. *SCOTT* is an example of what this paper calls a common SQL name; and *My Table* is an example of what it calls an exotic SQL name. These notions are discussed in the section *“Example 2: user-supplied table name”* on page 14.

DBMS_Assert.Simple_Sql_Name() is discussed in the section *“Ensuring the safety of a simple SQL name”* on page 25.

14. We shall defend this strict approach in the section *“Ensuring the safety of a simple SQL name”* on page 25.

15. The term *PL/SQL static varchar2* expression is defined in the PL/SQL Language Reference book. This also defines a *PL/SQL static varchar2* constant as a variable that is declared using the *constant* keyword and that is initialized using a PL/SQL static *varchar2* expression. The definition is recursive: a *PL/SQL static varchar2* constant can be used in the composition of a PL/SQL static *varchar2* expression. We shall return to this notion in the section *“Static text”* on page 33.

16. The fact that this text is characterized in a different domain of discourse (the text of the compiled PL/SQL unit) than text which is a *PL/SQL static varchar2* expression (here, the domain is the source text of the unit) is unimportant with respect to establishing the notion of compile-time-fixed SQL statement text.

execution methods will be referred to as *embedded SQL*¹⁷, *native dynamic SQL* and *the DBMS_Sql API*¹⁸.

Embedded SQL always executes compile-time-fixed SQL statement text. Dynamic SQL may execute compile-time-fixed SQL statement text or may execute run-time-created SQL statement text.

Distinguishing between a static SQL syntax template and a dynamic SQL syntax template

Sometimes a PL/SQL subprogram executes a SQL statement whose SQL syntax template is frozen at compile time. But sometimes the SQL statement must be composed at run time in such a way that the set of SQL syntax templates to which these must conform is too large to be written down explicitly in the Design Specification document. This section addresses this important distinction.

Definition of static SQL syntax template

Consider the definer's rights¹⁹ function $f()$ as shown in [Code_5](#).

```
-- Code_5
function f(PK in t.PK%type, Wait_Time in pls_integer)
  return t.c1%type
  authid Definer
is
  c1 t.c1%type;
  Stmt constant varchar2(32767) :=
    'select c1 from t where PK = :b for update wait '
    || Wait_Time;
begin
  execute immediate Stmt into c1 using PK;
  return c1;
end f;
```

Its purpose is to return the value of column $c1$ from the table t for a particular value of the primary key, PK , and to lock the row for future update. The function should not fail if another session presently has locked the row — and nor should it wait indefinitely in such a case. Rather, it should allow the caller to specify the maximum wait time. As it happens, SQL syntax does not allow a placeholder for the value that determines the timeout period²⁰.

-
17. We prefer the term *embedded SQL* to the more familiar *static SQL*. As we shall see, *dynamic SQL* may be used to execute *static text*. The term *static* is heavily overloaded.
 18. Procedural APIs other than the *DBMS_Sql* API support the execution of complete SQL statements of restricted kinds or accept fragments of SQL text and concatenate these unchecked to compose and then execute run-time-created SQL statement text. These are listed in [Appendix D: Additional Oracle-supplied subprograms that implement dynamic SQL](#) on [page 58](#). The rules for using these safely are identical to those for using native dynamic SQL and the *DBMS_Sql* API safely. Therefore, no further mention will be made of these other APIs in the body of this paper.
 19. A belief seems to have arisen that an invoker's rights unit is safe and that a definer's rights unit is risky. This viewpoint is naïve. The proper choice depends on the purpose of the subprogram. Here, the purpose of $f()$ is to select from a specific table using the privileges of the table's owner. It's quite likely that *Execute* on $f()$ will be granted to a user other than its owner, who has no direct privileges on the table, specifically to give the grantee tightly controlled access to the table. Invoker's rights, on the other hand, is appropriate when the purpose of the subprogram is to perform a parameterized, but powerful, operation using the privileges of the user that invokes it. A risk would, of course, arise when such a subprogram is definer's rights and is owner by, for example, *Sys!*

Manual inspection is sufficient to show us that, while $f()$ does not execute a compile-time-fixed SQL statement text, it *does* execute a SQL statement whose SQL syntax template is frozen at compile time as that shown in [Template_3](#)²¹.

```
-- Template_3
select c1 from t where PK = :b for update wait &1
```

We shall call such a SQL syntax template a *static SQL syntax template*.

Notice that, as a consequence of the definitions, compile-time-fixed SQL statement text always conforms to a static SQL syntax template. But run-time-created SQL statement text may, or may not, conform to a static SQL syntax template.

Definition of dynamic SQL syntax template

Now consider the procedure $x.p()$ as shown in [Code_6](#).

```
-- Code_6
package x is
  type cw is varray(20) of boolean;
  procedure p(PK in "My Table".PK%type, Wanted in cw);
end x;
```

Its purpose is to report the values of an arbitrary subset of all the columns in the table with the exotic SQL name *My Table*²² for a particular value of the primary key, PK . The author of code that will call $x.p()$ knows the purpose of *My Table*, and the names and significance of all its columns. In particular, he knows the order in which these are listed in its external documentation. The n^{th} element of the *in* formal parameter *Wanted* determines if the n^{th} column of *My Table* is to be included in the report.

Let's say that Functional Specification document requires that the order of the columns in the report be the same as that in the external documentation of the table. If the table has N columns, then the number of distinct *select* lists is therefore the sum of the number of ways of choosing 1 item from N , then 2 items from N , through to $N-1$ items from N , and finally choosing all the items. This²³ is famously 2^N-1 . For 3 columns, there are 7 distinct *select* lists, for 10 columns there are 1023 distinct *select* lists, and for 20 columns, there are just over

20. This code is used to make a teaching point. It would be an unacceptable design in a real world application. Each executed *select* statement would probably be textually different from any that had been seen before; and this would cause a hard parse explosion. A compromise would be to let the caller chose between, say, four values for the wait time: zero, short, long, and infinite. The corresponding formal parameter would express the choice. And the *select* statement would be build using one of four PL/SQL static *varchar2* expressions for the wait time.

21. This relies on the fact that the intended timeout value is represented by a *pls_integer* and that when a value of this type is concatenated to a *varchar2*, then it is implicitly converted to that datatype using the single argument overload of *To_Char()*. The result of this conversion is guaranteed to be well-formed SQL *numeric* literal. Because it's an integral quantity, there's never decimal separator. The second actual (that determines the format model) cannot be influenced through the environment; its default requests a decimal separator but no group separator. And the third actual, that determines the characters used for the decimal separator and the group separator, and that can be set using this statement:

```
alter session set NLS_Numeric_Characters
```

therefore has no effect. This is a subtle but crucial point. Had the datatype of the *in* formal parameter *Wait_Time* been *number*, then the SQL syntax template of *Stmt* would have been unpredictable. We shall return to it in the section "*Ensuring the safety of a SQL numeric literal*" on [page 23](#).

one million distinct *select* lists. The number grows exponentially²⁴ with the number of columns.

Procedure *x.p()* models the implementation of a very common requirement: the customizable report²⁵. And an underlying table with many more than 20 columns is in no way unusual. It is therefore not feasible to provide each possible statement as compile-time-fixed SQL statement text. Rather, the required SQL statement must be composed programmatically.

The implementation of *p()* in the body of package *x* is shown in [Code_7](#). The *Col_List()* inner function composes the *select* list. In this design, *column list* is perhaps not the best mnemonic; as [Code_8](#) shows, the *select* list is in fact a single item built by concatenating the right-blank-padded values of the chosen columns. This approach allows the simple use of *execute immediate* rather than the more complicated use of the *DBMS_Sql* API for a *select* list whose composition is unknown until run time.

```
-- Code_7
procedure p(PK in "My Table".PK%type, Wanted in cw) is

    function Col_List return varchar2;

    Stmt constant varchar2(32767) :=
        'select '
        || Col_List()
        || ' Report from "My Table" where PK = :b';

    Report varchar2(32767);

    function Col_List return varchar2 is ... end Col_List;

begin
    execute immediate Stmt into Report using PK;
    DBMS_Output.Put_Line(Report);
end p;
```

Notice that *Stmt* is declared using the *constant* keyword and that this means that it must be initialized as part of the declaration. This is not essential, but it is a highly recommended approach.

Rule_1

When it is necessary to compose a SQL statement programmatically, the code usually needs, or at least benefits from the use of, variables for intermediate results. Aim to declare these as *constant*, assigning the values in the declarations. This sometimes requires the use of nested block-statements or forward-declared functions. This technique makes code review easier because

When composing a SQL statement programmatically, help the code reviewer by declaring variables used for intermediate results as *constant*.

-
22. We use the exotic SQL name *My Table* just as a subliminal reminder that such names can occur. We need do no more in the code discussed in this section than surround its use, as is shown, with double quote characters. The uses occur only in ordinary PL/SQL source code (in the declaration of the formal parameter *PK*) and in a PL/SQL static *varchar2* expression. We shall return to this point in the section "[Query by example form](#)" on [page 40](#).
 23. The internet searching required to confirm this is left as an exercise for the reader.
 24. The expression *to grow exponentially* is often used metaphorically; here, it is used literally and correctly!
 25. Another common requirement is to let the user specify comparison criteria for any subset of the columns (the so-called query-by-example paradigm). We shall return to this in the section "[Don't confuse the need to use a dynamic SQL syntax template with the need for dynamic text](#)" on [page 32](#) and then in the section "[Query by example form](#)" on [page 40](#).

the reader can be sure that the value of a variable cannot change between its initial assignment and its use.

For completeness, the implementation of the *Col_List()* function is shown in [Code_8](#).

```
-- Code_8
function Col_List return varchar2 is
  type cn is varray(20) of varchar2(30);
  Col_Names constant cn :=
    cn('c1', 'c2', 'c3', 'c4', ..., 'c20');
  Seen_One boolean := false;
  List varchar2(32767);
begin
  for j in 1..Wanted.Count() loop
    if Wanted(j) then
      List :=
        List
          || case Seen_One when true then '||'
             else
                end
          || 'Rpad('||Col_Names(j)||', 10)';
      Seen_One := true;
    end if;
  end loop;
  return List;
end Col_List;
```

Notice that the elements from which *List* is composed are all PL/SQL static *varchar2* expressions²⁶.

Suppose that, as the use of the type constructor *varray(20)* suggests, the table *My Table* has 20 columns. The set of the more than one million distinct SQL statements that *x.p()* might execute is too numerous to allow each one to be inspected. Rather, the programmer (and auditor) must reason, from the self-evident predictability of the possible return values of the *Col_List* function, what the possible members are — and from that deduce what the possible SQL syntax templates are²⁷. In this example, there are 20 distinct SQL syntax templates. [Template_4](#) shows some.

```
-- Template_4
select Rpad(&&1, 10) Report
from "My Table" where PK = :b

select Rpad(&&1, 10)||Rpad(&&2, 10)||Rpad(&&3, 10) Report
from "My Table" where PK = :b

select Rpad(&&1, 10)||Rpad(&&2, 10)|| ... ||Rpad(&&20, 10) Report
from "My Table" where PK = :b
```

We shall call each such a SQL syntax template a *dynamic SQL syntax template*.

A static SQL syntax template can, by trivial inspection of the PL/SQL source code that composes it, be written down by the auditor with certainty. A dynamic SQL syntax template is one of a large set of such templates that are composed for execution at a particular call site where the set is too large to allow each to be written down but where, nevertheless, the set can be described with certainty.

26. The term PL/SQL static *varchar2* expression is defined in the PL/SQL Language Reference book and its significance is discussed in the section “*Static text*” on [page 33](#).

27. It would be possible to describe the universe of possible SQL statements and possible SQL syntax templates using the regular expression syntax.

Understand what is meant by the term SQL syntax template and the difference between a static SQL syntax template and a dynamic SQL syntax template.

Rule_2

Understand what is meant by the term SQL syntax template. Apply this understanding to the design of any code that constructs run-time-created SQL statement text. Understand the difference between a static SQL syntax template and a dynamic SQL syntax template.

SQL injection (finally) defined

We define SQL injection, as it might occur from a PL/SQL subprogram, to mean the execution, by that subprogram, of a SQL statement with a SQL syntax template which differs, at a particular call site, from that which the subprogram's author intended for that site. It differs because it has a SQL syntax template, intended by the attacker, embedded within the SQL syntax template that the author intended in such a way that the resulting SQL syntax template is legal.

Self-evidently, it cannot occur from a site that uses embedded SQL. Nor can it occur from a site that uses dynamic SQL to execute compile-time-fixed SQL statement text.

The injected text may come directly via one of the subprogram's formal parameters (a so-called first order attack); or it may come indirectly, for example via a table which the subprogram reads, and trusts, to obtain a component of the SQL statement that it is building, but into which the attacker has contrived to insert a rouge value (a so-called second order attack).

Rule_3

Understand that SQL injection is the execution of a SQL statement with an unintended SQL syntax template and that the risk can occur only when run-time-created SQL statement text is executed using dynamic SQL.

Understand how to define the term SQL injection as the execution of a SQL statement with an unintended SQL syntax template. Know that only run-time-created SQL statement text that, therefore, must be executed using dynamic SQL is potentially vulnerable to SQL injection.

HOW CAN SQL INJECTION HAPPEN?

This is best demonstrated by examples. These will also enable us to tighten the definition of SQL injection. In this section, we will not ask why the programmer does not use compile-time-fixed SQL statement text²⁸; we shall see later that there are indeed use cases that do require run-time-created SQL statement text. The fragments shown here, though not plausible, illustrate the SQL injection techniques.

Example 1: user-supplied column-comparison value

Consider the PL/SQL fragment shown in [Code_9](#).

```
-- Code_9
...
q constant varchar2(1) := 'q';
SQL_VC2_Literal constant varchar2(32767) :=
q||Raw_User_Input||q;
begin
  Stmt :=
    'select c2 from t where c1 = '||SQL_VC2_Literal;
  execute immediate Stmt bulk collect into v;
  ...
```

Presumably *Raw_User_Input* holds a PL/SQL *text* value that the programmer expects to a possible value for the column *t.c2*. Is this vulnerable to SQL injection? It is tempting to reply “no” because the SQL syntax template seems to be fixed at compile time to this:

```
-- Template_5
select c1 from t where c2 = &1
```

If *Raw_User_Input* has this value:

```
-- Value_1
Smith
```

then *Stmt* will now be this:

```
-- Value_2
select c1 from t where c2 = 'Smith'
```

All seems to be well. But suppose that *Raw_User_Input* has this value:

```
-- Value_3
O'Brien
```

The value of the PL/SQL variable *Stmt* will now be this:

```
-- Value_4
select c1 from t where c2 = 'O'Brien'
```

While this is unremarkable as a PL/SQL *text* value, it is syntactically incorrect as a SQL statement (the single quote characters don't balance) and will cause *ORA-00933: SQL command not properly ended* when it is parsed. The programmer

28. Of course, the intent of [Template_5](#) can be achieved using embedded SQL (which, under the covers, generates at compile time and uses at run time a compile-time-fixed SQL statement text that includes a placeholder). [Code_9](#) therefore violates the principles advocated in the section “Use compile-time-fixed SQL statement text unless you cannot” on [page 28](#). However, we have seen code where native dynamic SQL has been used where embedded SQL was sufficient. Sometimes the explanation is nothing more than what ordinarily explains poorly conceived and written programs; sometimes, it might be the work of a disgruntled and malicious employee.

intended to construct a SQL *text* literal from a PL/SQL *text* value by surrounding it with an opening and a closing single quote character²⁹. But he forgot that *O'Brien* is a perfectly plausible value for a *text* column and that, therefore, the PL/SQL *text* value might legally contain interior single quote characters. The rules for constructing a SQL *text* literal from a PL/SQL *text* value require that each interior single quote character be escaped by doubling it *before* surrounding it with an opening and a closing single quote character³⁰. Because the programmer forgot this, the singleton interior single quote character closes the SQL *text* literal, and *Brien'* is parsed as two SQL tokens, *Brien* and *'*.

This might seem to be an ordinary bug — and, of course, it should be avoided for ordinary reasons. However, if present, it could go undetected indefinitely; and it has a shocking manifestation when an ingeniously, and maliciously, contrived value for *Raw_User_Input* is provided.

Suppose that *Raw_User_Input* has this value:

```
-- Value_5
',
union
select Username c1 from All_Users --
```

Stmt will now be this:

```
-- Value_6
select c1 from t where c2 = ''
union
select Username c1 from All_Users --'
```

Notice that the would-be closing single quote character that the programmer adds, and that would cause a syntax error if it were unbalanced, is disarmed because *Value_5* finishes with the `--` token that starts a single line comment.

The result is a legal SQL statement. Because, in Oracle Database, the empty string is the same as *null*, and because an equality comparison with *null* always results in *null*, no rows are selected from *t*; so the SQL statement reduces to this:

```
-- Value_7
select Username c1 from All_Users
```

This SQL statement an instance of a very different SQL syntax template than [Template_5](#) shows — and is surely not what the programmer intended. We have seen, therefore, that the subprogram that executes *Stmt* is indeed vulnerable to SQL injection.

This is the canonical example of SQL injection. The superstitious fear is that *any* program that issues run-time-created SQL statement text is vulnerable to such

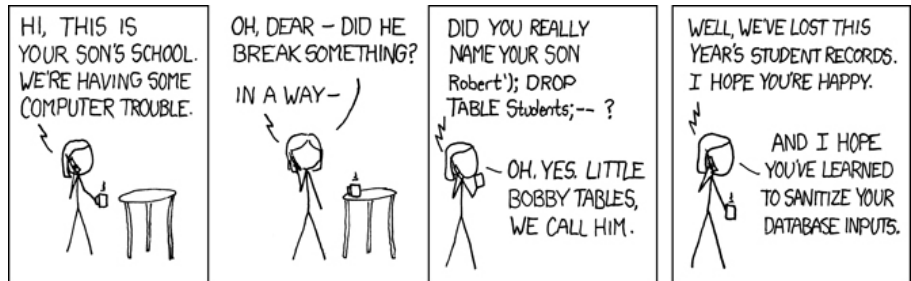
29. The point here, of course, is that the programmer is writing the source code of a PL/SQL program whose purpose is to create, at run-time, the source text of another program (a SQL statement) and then to execute that.

30. Oracle Database 10g introduced an alternative quoting syntax for both SQL and PL/SQL — the so-called alternative quoting mechanism (sometimes know as the user-defined quoting mechanism, or the q-quote syntax). It aims to increase usability when the value itself contains single quote characters. Here is an example in PL/SQL:

```
v varchar2(80) := q'{You can't do that}';
```

The opening *q'* and the closing *'* are non-negotiable; The user chooses the inner parentheses, in this example { and }. We will see in the section *“Ensuring the safety of a SQL text literal”* on [page 19](#) that this mechanism should be avoided.

an attack. However, as the cartoon³¹ suggests, such attacks can always be prevented by following simple immunization practices.



The proper practice in this example is clear: when a PL/SQL *text* value is to be converted to a SQL *text* literal, then the result must be a string that starts and ends with a single quote character; and between these, there must be no singleton single quote characters and no runs of an odd number of single quote characters. This is discussed formally in the section “*Ensuring the safety of a SQL literal*” on [page 19](#).

Example 2: user-supplied table name

Consider the PL/SQL fragment shown in [Code_10](#).

```
-- Code_10
...
Stmt :=
  'select c1 from '||Raw_User_Input||' where c2 = 'Smith'';
execute immediate Stmt bulk collect into v;
...
```

Presumably *Raw_User_Input* holds a value that the programmer expects to be the name of a table or view. Is this vulnerable to SQL injection? As with [Example 1](#), it is tempting to reply “no” because the SQL syntax template seems to be fixed at compile time to this:

```
-- Template_6
select c1 from &&1 where c2 = 'Smith'
```

If *Raw_User_Input* has this value:

```
-- Value_8
t
```

then *Stmt* will now be this:

```
-- Value_9
select c1 from t where c2 = 'Smith'
```

31. The original of this cartoon is to be found at xkcd.com/327/. A footnote states “...you're free to copy and share these comics (but not to sell them)”.

Again, all seems to be well. But recall that, though few programmers take advantage of it³², the following SQL*Plus script runs without error.

```
-- Code_11
drop table "a /"
/
create table "a /" ("?" number, "a'b" number, " " number)
/
insert into "a /" ("?", "a'b", " ") values (1, 2, 3)
/
select * from "a /" where "?" = 1
/
select      '['||Table_Name||']' x from User_Tables
union
select      '['||Column_Name||']' x from User_Tab_Cols
where      Table_Name = 'a /'
order by   x
/
```

A SQL identifier can consist of any sequence of one or more characters in the database character set provided that the representation needs no more than 30 bytes³³. The final *select* in *Code_10* produces an output that includes these rows:

```
[ ]
[?]
[a /]
[a'b]
```

Notice how the values end up in metadata in the catalog views; and notice how, when these views are queried, the rules are no different from those for any other tables and views. Especially, because an identifier may legally contain singleton single quote characters, the same care must be used when constructing a SQL *text* literal to represent one in a metadata query.

It helps to think that SQL ordinarily requires that every identifier is surrounded with double quote characters. Only if the identifier happens to start with an alphabetic character, and happens thereafter to contain only alphanumeric characters or *underscore*, # or \$, then (as a usability bonus) the double quote characters may be omitted. And if they are omitted, then the SQL parser upper-cases the identifier.

There seem to be no established terms of art to capture this distinction. We will use the term *common SQL name* to denote one that starts with an upper case alphabetic character in the range *A..Z* and then has only upper case alphanumeric characters in the range *A..Z*, or *underscore*, # or \$. A common SQL name doesn't need to be surrounded by double quote characters in a SQL statement; and if it is not so surrounded, the case with which it is written doesn't matter. It may, however, be so surrounded. If it is, then because the SQL parser preserves its case, it must be written in all upper case. We will use the term *exotic SQL name* to denote one that violates the rules for a common SQL name

32. GUI tools for the DBA and the developer are becoming increasingly popular. These support object creation without typing the SQL directly. It is therefore more likely these days to encounter exotic SQL names (like *Line Items*) than it used to be.

33. There is one exception: this DDL statement:

```
create table "a""b"(n number)
```

fails with *ORA-03001: unimplemented feature*. An identifier may not contain the double quote character. Oracle Database no longer allows this. The text of the message might seem to suggest a possibility that a later version of Oracle Database might, again, allow this. This is misleading. There is no intention to do this.

and that *must*, therefore, be surrounded by double quote characters in a SQL statement.

Suppose, then, that the value for the simple SQL name placeholder in *Template_6* is *a /* rather than *t*; *Raw_User_Input* will have this value:

```
-- Value_10
a /
```

Stmt will now be this:

```
-- Value_11
select c1 from a / where c2 = 'Smith'
```

This is syntactically incorrect and will cause *ORA-00933: SQL command not properly ended* when it is parsed.

The programmer intended to construct a SQL identifier from a PL/SQL *text* value. But he forgot about the possibility of getting an exotic SQL name. The rules for constructing a SQL identifier from a PL/SQL *text* value need to take account of this. This is discussed formally in the section “*Ensuring the safety of a simple SQL name*” on [page 25](#).

Again, this might seem to be an ordinary bug; and again it could go undetected indefinitely. (The use of exotic SQL names is relatively rare.) But again the bug has a shocking manifestation when an ingeniously, and maliciously, contrived value for *Raw_User_Input* is provided.

Suppose that *Raw_User_Input* has this value:

```
-- Value_12
t where 1=2
union
select Username c1 from All_Users --
```

Stmt will now be this:

```
-- Value_13
select c1 from t where 1=2
union
select Username c1 from All_Users -- where c2 = 'Smith'
```

The code expects only a common SQL name. But the caller has supplied an exotic SQL name. It has been cunningly designed so that the result is a legal SQL statement. As in *Example 1*, again the SQL statement reduces to this:

```
-- Value_14
select Username c1 from All_Users
```

Again, this is an instance of a very different SQL syntax template than *Template_6* shows and again it is surely not what the programmer intended. Of course, then, the subprogram that executes *Stmt* is vulnerable to SQL injection.

The proper practice in this example, too, is clear: when a PL/SQL *text* value is to be used to compose a SQL identifier, then the conversion must acknowledge the difference between a common SQL name and an exotic SQL name, must handle case accordingly, and must surround the final result with double quote characters.

Counter-example 3: user-supplied where clause

Consider next the PL/SQL fragment shown in [Code_12](#).

```
-- Code_12
...
Stmt := 'select c1 from t where '||Raw_User_Input;
execute immediate Stmt bulk collect into v;
...
```

Assuming that the PL/SQL identifier is appropriately named, then the intended SQL syntax template is simply unspecified. Formally, therefore, it falls outside the scope of the discussion of SQL injection — just as does SQL*Plus, which is specified to be able to execute any arbitrary SQL statement.

The programmer probably did have an informal notion of the set of possible SQL syntax templates that *Stmt* may instantiate, for example an equality comparison with a literal for each of any arbitrary combination of columns from table *t*. However, unlike checking that a string is a well-formed SQL literal or SQL identifier, it is very difficult to check that a putative complete *where* clause is an instantiation of one of a set of intended SQL syntax templates.

A Functional Specification document for a PL/SQL program that offers the powerful flexibility implied by [Code_12](#) must be rejected unless it can be guaranteed that it can be executed only by a user who anyway could connect directly to the database and execute arbitrary *select* statements against the same table.

We shall return to this point in the section “*Use a static SQL syntax template for run-time-created SQL statement text unless you cannot*” on [page 30](#).

Counter-example 4: SQL syntax template with a questionable intent

Consider next the procedure shown in [Code_13](#).

```
-- Code_13
procedure Make_DBA(Raw_User_Input in varchar2)
  authid Definer -- Current_User
is
  Double_Quote_Test constant char(3) := '%"%"';
begin
  if Raw_User_Input like Double_Quote_Test then
    Raise_Application_Error(-20000, 'Interior " is illegal');
  end if;

  declare
    Username constant varchar2(32767) :=
      '""||Raw_User_Input||""';
    Stmt constant varchar2(32767) :=
      'grant DBA to '
      || Username
      || ' identified by x with Admin Option';
  begin
    DBMS_Output.Put_Line(Stmt);
    execute immediate Stmt;
  end;
end Make_DBA;
```

Because a value of *Raw_User_Input* that has an interior double quote character³⁴ causes an error, and because a value that does not is then surrounded by double

34. A simple SQL name must not contain a double quote character. An attempt to use such a name in a SQL statement causes *ORA-03001: unimplemented feature*.

quotes³⁵, there is no risk that the SQL syntax template will be anything other than this:

```
-- Template_7
grant DBA to &&1 identified by x
```

There is, therefore, no risk of SQL injection³⁶, and further discussion of it is beyond the scope of this whitepaper.

Not to leave it unsaid, though, the intent of this SQL syntax template does signal the need for some serious analysis³⁷. Because *Make_DBA()* is an invoker's rights procedure, then it can be executed without error only by a user that already has the *DBA* role *with Admin Option*. Nominally, then, it should be safe to grant *Execute* on this to *public*. There is, however, a subtle secondary point to consider. If there exists a subprogram in the database, whose owner has the *DBA* role *with Admin Option*, and that is vulnerable to injection, then it might be possible to inject an invocation of this procedure³⁸ where it would not be possible to inject the SQL that it executes. Clearly, then, the existence of a procedure like *Make_DBA()* makes a security audit harder.

-
35. This crude programming device is used in this example in order not to pre-empt the discussion, in the section "*Ensuring the safety of a simple SQL name*" on page 25, of the proper approach. We shall see there that the proper approach is to use *DBMS_Assert.Simple_Sql_Name()*.
36. You might question the usability of the interpretation of *Raw_User_Input*. If you wanted to specify a user whose name was the common SQL name *SCOTT*, then you would have to be careful to spell it in all upper case. This is not the convention in SQL. However, the usability is not interesting in this example. The example is designed to illuminate the discussion of SQL injection.
37. To keep the code simple, the password is specified as an explicit identifier. But the point of this example would be unchanged even if the password were specified by user input that not only was surrounded with double quotes but also was required to pass a stringent password strength test.
38. This would require wrapping it in an invoker's rights function that was defined with the *Autonomous_Transaction* pragma.

ENSURING THE SAFETY OF A SQL LITERAL OR A SIMPLE SQL NAME

Only two kinds of element in a SQL syntax template may be replaced with dynamic text³⁹: a value placeholder may be replaced with a SQL literal and a simple SQL name placeholder may be replaced with a simple SQL name. However, as we have seen in the section “*Example 1: user-supplied column-comparison value*” on page 12 and the section “*Example 2: user-supplied table name*” on page 14, it is precisely in this replacement that the risk of SQL injection occurs. This section, therefore, prescribes the approaches that guarantee immunity to that risk.

Ensuring the safety of a SQL literal

There are three kinds of SQL literal: *text*, *datetime*, and *numeric*⁴⁰. Each deserves separate attention.

Ensuring the safety of a SQL *text* literal

If the value *O'Brien* is to be presented as a SQL *text* literal, then it can be written either using the default mechanism shown in [Code_14](#) or using the so-called alternative quoting mechanism⁴¹ shown in [Code_15](#).

```
-- Code_14
...where Last_Name = 'O'Brien'

-- Code_15
...where Last_Name = q'{O'Brien}'
```

While the alternative quoting mechanism can improve readability for values that have interior singleton occurrences of the single quote character, its use must be banned for the purpose of composing a safe SQL *text* literal⁴².

The *DBMS_Assert* package⁴³ exposes the function *Enquote_Literal()*. It has a single formal parameter, *Str*, of datatype *varchar2* and mode *in*; and its return datatype is *varchar2*. When the input is a well-formed SQL *text* literal, then the output is identical to the input; but when the input is not a well-formed SQL *text* literal, then the predefined exception *Standard.Value_Error* is raised⁴⁴.

These, then, are the rules for composing a safe SQL *text* literal from a PL/SQL *text* value:

- Replace each singleton occurrence, within the PL/SQL *text* value, of the single quote character with two consecutive single quote characters.

39. The term *dynamic text* is formally defined in the section “*Dynamic text*” on page 34. For now, it is sufficient to assume the intuitive meaning: text whose composition cannot be traced back to only PL/SQL static *varchar2* expressions.

40. This syntax definition for these items is given in the SQL Language Reference book.

41. Support for the alternative quoting mechanism was introduced in Oracle Database 10g.

42. The reason is simple: the *DBMS_Assert* package, through Oracle Database 11g, does not provide a function to assert the safety of this syntax.

43. The *DBMS_Assert* package was first documented in Oracle Database 11g. It has since been backported to each of these versions: 8.1, 9.2, 10.1, and 10.2. In some cases it is ordinarily available in the latest patchset; in others it is available in a CPU. Contact Oracle Support for more information.

- Concatenate one single quote character before the start of the value and one single quote character after the end of the value.
- Assert that the result is safe with `DBMS_Assert.Enquote_Literal()`⁴⁵.

Notice that the mandate in the third bullet is the crucial one. It is this one that guarantees immunity to injection; the first mandate prevents annoying run-time errors.

Ensuring the safety of a SQL *datetime* literal

A SQL *datetime* literal is a SQL *text* literal which must conform to the additional condition that it must be capable of conversion to a value of a *datetime* datatype. Obviously, then, the safety of a SQL *datetime* literal must be asserted with `DBMS_Assert.Enquote_Literal()` in exactly the same way as a SQL *text* literal.

It is interesting to see what can happen if this simple rule is forgotten, and if the documented behaviors of the single argument `To_Char(d)`, where *d* has a *datetime* datatype, and the single argument `To_Date(t)`, where *t* has a *text* datatype, are forgotten too. Consider the contrived procedure whose first few lines are shown in [Code_16](#).

```
-- Code_16
procedure p is
  q constant varchar2(1) := ''';
  d constant date :=
    To_Date('2008-09-22 17:30:00', 'yyyy-mm-dd hh24:mi:ss');
  Stmt constant varchar2(32767) :=
    'select t.PK from t where t.d > ' || q||d||q;
  ...
begin
  execute immediate Stmt bulk collect into Results;
  ...
```

The author might understand that, with the code as presented, the single argument `To_Char()` is implicitly invoked by PL/SQL when *d* is concatenated into *Stmt*; and that, next, the single argument `To_Date()` is implicitly invoked by SQL when *Stmt* is executed. He might even remember that both the output of the single argument `To_Char()` and that of the single argument `To_Date()` are affected by the environment setting of the `NLS_Date_Format` parameter — and might reason that the second conversion is the antidote to the first, and that there is, therefore, no cause for concern.

However, this understanding is naïve. At the very least, he has programmed an ordinary bug that can lead to unreliable query results because of a loss of date

44. There is one exception: when the input has only properly doubled interior single quote characters, but neither starts nor ends with a single quote character, then `Enquote_Literal()` will quietly add these. There are those (for example, Bryn Llewellyn!) who feel that this was an unfortunate design and that the function would have been better conceived of as a pure asserter. However, it is what it is; and backwards compatibility considerations prohibit changing it. This paper encourages the approach, which is always possible, that uses it as a pure asserter.

45. In all the code examples in this paper, the name of this package is always qualified with its owner, as `Sys.DBMS_Assert`, rather than using the bare public synonym. This measure is essential for safety. It ensures that the simple name cannot be captured by a private synonym or package in the same schema as the PL/SQL unit which refers to what is intended to be the Oracle-supplied `DBMS_Assert` package.

precision. Suppose that table *t* had been populated by the SQL*Plus script shown in [Code_17](#).

```
-- Code_17
alter session set NLS_Date_Format = 'yyyy-mm-dd hh24:mi:ss'
/
begin
  insert into t(PK, d) values(1, '2008-09-23 17:30:00');
  insert into t(PK, d) values(2, '2008-09-21 17:30:00');
  commit;
end;
/
```

And suppose, now, that *p()* is executed using the SQL*Plus script shown in [Code_18](#).

```
-- Code_18
alter session set NLS_Date_Format = 'dd-Mon-yy hh24:mi:ss'
/
begin p(); end;
/
alter session set NLS_Date_Format = 'yyyy'
/
begin p(); end;
/
```

The first execution of *p()*, though it runs with a different setting of *NLS_Date_Format* than was current when table *t* was populated, still produces the expected result: only the row with *PK=1* is selected. (This is because it honors the precision with which the dates were stored.) However, the second execution produces what is almost certainly an unexpected result: both the row with *PK=1* and the row with *PK=2* are selected. The reason, of course, is that the conversion of the value in the PL/SQL variable *d*, after lossy conversion to *text* and then conversion back to a *datetime* datatype, has ended up as (what would display as) *2008-01-01 00:00:00*. This no longer honors the intended precision.

One who has followed the discussion of the case presented in the section “*Example 1: user-supplied column-comparison value*” on [page 12](#), will not be surprised to learn that procedure *p()* is not only ordinarily buggy; it is also vulnerable to SQL injection. Suppose that *p()* is executed using the SQL*Plus script shown in [Code_19](#).

```
-- Code_19
alter session set
  NLS_Date_Format = ''' and Scott.Evil()=1--'
/
begin p(); end;
/
```

The value shown in [Value_15](#) is assigned to *Stmt*.

```
-- Value_15
select t.PK from t where t.d > '' and Scott.Evil()=1--'
```

This is surely not an instance of the intended SQL syntax template. Therefore, SQL injection has occurred. The explanation is a twist on the usual one: a singleton occurrence of the single quote character has been injected into the run-time composed SQL statement, but this time *from the side*⁴⁶ — via an NLS environment parameter — rather than directly. Of course, all bets are off now.

46. This form of attack has been referred to as *lateral* SQL injection by [David Litchfield](#). The article is available on the internet here:

www.databassecurity.com/dbsec/lateral-sql-injection.pdf

The invocation of the malicious function is injected, and the would-be closing single quote character that *p()* adds is disarmed by the `--` token that starts a single line comment. *Scott.Evil()* is invoked!

While the argument just presented might seem tortuous, the conclusion is simple to understand and simple to implement. These, then, are the rules for composing a safe SQL *datetime* literal from a PL/SQL *datetime* value:

- Use the two-parameter overload, for an input of datatype *date*, *To_Char(d, Fmt)*, to compose a SQL *datetime* literal⁴⁷, *t*. (This, of course, will be a PL/SQL *varchar2*.) Use a value for *Fmt* that is consistent with what the Functional Specification document requires for the precision.
- Concatenate one single quote character before the start of this value and one single quote character after its end.
- Assert that the result is safe with *DBMS_Assert.Enquote_Literal()*.
- Compose the date predicate in the SQL statement using the two-parameter overload for *To_Date(t, Fmt)* and using the identical value for *Fmt* as was used to compose *t*.

Notice that the mandate in the third bullet is the crucial one. It is this one that guarantees immunity to injection; the first two and the fourth mandates prevent annoying run-time errors.

The procedure *p_Safe()*, whose first few lines are shown in [Code_20](#), implements this approach.

```
-- Code_20
procedure p_Safe(d in date) is
  q constant varchar2(1) := ''';

  -- Choose precision according to purpose.
  Fmt constant varchar2(32767) := 'J hh24:mi:ss';

  Safe_Date_Literal constant varchar2(32767) :=
    Sys.DBMS_Assert.Enquote_Literal(q||To_Char(d, Fmt)||q);

  Fmt_Literal constant varchar2(32767) := q||Fmt||q;

  Safe Stmt constant varchar2(32767) :=
    ' insert into t(d) values(To_Date('
    || Safe_Date_Literal
    || ', '
    || Fmt_Literal
    || '))';
begin
  execute immediate Safe Stmt;
  ...
```

When *p_Safe()* is invoked using *Sysdate()*, a value like that shown in [Value_16](#)⁴⁸ is assigned to *Safe Stmt*.

```
-- Value_16
insert into t(d) values(To_Date('2454723 18:01:05', 'J hh24:mi:ss'))
```

47. Of course, *date* is not the only *datetime* datatype. The same reasoning applies for, for example, a *timestamp* literal.

48. The Julian date 2454723 is 13-Sep-2008.

The composition of *Safe Stmt* is immune to the effect of changes to the *NLS_Date_Format* parameter.

Ensuring the safety of a SQL *numeric* literal

Unlike a SQL *datetime* literal, which is a specialized kind of SQL *text* literal, a SQL *numeric* literal has specific syntax. It always uses a dot as the decimal character and never contains a group separator. Here are some examples (notice that there are no surrounding single quote characters)⁴⁹:

```
42
-1
+6.34
0.5
-123.4567
25e-03
25f
+6.34F
0.5d
-1D
```

The SQL syntax does not recognize national differences. However, the output of the overload of the *To_Char()* function for a *numeric* datatype does. This function has three sub-overloads with one, two, and three formal parameters.

For the overload with three formal parameters, the second formal parameter, called *Fmt*, specifies the format model and the third formal parameter, called *NLSparam*, specifies the actual characters that will be used for notions like the decimal character, the group separator, the currency symbol, and so on.

For the overload with two formal parameters, the second formal parameter specifies the format model. Notably, if this overload is used, the value for (the components of) *NLSparam* is determined by the *NLS_Numeric_Characters*, *NLS_Currency*, and *NLS_ISO_Currency* environment parameters. Even more notably, if the overload with one formal parameter is used, then the value for *Fmt* has a fixed default: it cannot be controlled from the environment. But the value for *NLSparam* is *still* determined by the environment and *still* has an effect.

Suppose that table *t* has been populated using the *insert* statement shown in [Code_21](#).

```
-- Code_21
insert into t(n) values (123456.789)
```

49. These are taken from the SQL Language Reference book.

Then the SQL*Plus script shown in [Code_22](#)

```
-- Code_22
select n from t
/
alter session set NLS_Numeric_Characters = ',.!'
/
alter session set NLS_Currency = 'NOK '
/
select To_Char(n, 'L999G999G999D999') n from t
/
select n from t
/
select To_Char(n, 'TM', 'NLS_Numeric_Characters = '!'.') n
from t
/
alter session set NLS_Numeric_Characters = '!'.!'
/
select 'c1 = '||n x from t
/
```

will produce this output:

```
123456.789
          NOK 123.456,789
123456,789
123456!789
c1 = 123456'789
```

If a SQL *numeric* literal is not composed carefully, there is a risk of generating a SQL statement with a syntax error — in other words, a different SQL syntax template than was intended. The ability to inject a singleton single quote character is particularly alarming. This is another example of lateral SQL injection, and the moral is clear.

These, then, are the rules for composing a safe SQL *numeric* literal from a PL/SQL *numeric* value:

- Use explicit conversion with the *To_Char()* overload with three formal parameters. This overload requires that a value be supplied for *Fmt*. Explicitly provide the value that supplies the default when the overload with one formal parameter is used. This is *'TM'*.⁵⁰
- Explicitly provide the value that supplies the default for the *NLS_Numeric_Characters* parameter when the one of the overloads with one or two formal parameters is used. This is *!',!*.

We shall return to this point in the section “*Safe dynamic text*” on [page 34](#), where we introduce the shorthand *To_Char(x,f, n)* for this overload.

50. *'TM'* is the so-called text minimum number format model. It returns the smallest number of characters possible in fixed notation unless the output exceeds 64 characters. In that case, the number is returned as if the model had been *'TMe'*. *'TMe'* returns the smallest number of characters possible in scientific notation. If the Functional Specification document suggests this, it is safe also to specify the *'TMe'* model explicitly. In fact, *any* format model is safe provided that it is chosen deliberately, together with the value for the *NLSparam* argument, in the light of the requirements for precision given in the Functional Specification document.

Ensuring the safety of a simple SQL name

Suppose that a user exists with the exotic SQL name⁵¹ *O'Brien* in whose schema there exists a procedure with the common SQL name *PROC* and a package with the common SQL name *PKG*. Suppose that this package has a procedure with the exotic SQL name *Do it*. Finally, suppose there exists, in a different database, a link to this one with the common SQL name *LNK*. Here are some examples of the various qualified SQL names that, in different contexts, might be used in a SQL statement:

```
Proc
'O'Brien'.Pkg
Pkg."Do it"
Proc@"Lnk"
'O'Brien'.Pkg."Do it"
'O'Brien'.Pkg@lnk
'O'Brien'.Pkg."Do it"@LNK
```

This is the general form of a qualified SQL name:

```
a [ .b [ .c ]][ @dblink ]
```

Each of the items *a*, *b*, *c*, and *DBlink* is a simple SQL name. A simple SQL name is either a common SQL name or an exotic SQL name and an exotic SQL name must be surrounded with double quote characters. Notice that this is just a prescription for the syntax; *a.b* might denote the element *b* in the package *a* (as in *Proc."Do it"*) or it might denote the object *b* owner by the user *a* (as in *"O'Brien".Pkg*).

The ability to compose a qualified SQL name by assembling simple SQL names with the punctuation characters `.` and `@` delivers a powerful generality in the context of a SQL statement. However, we have argued in this paper that the power of expression of SQL should be fully available only to the author of a subprogram⁵² that executes it. This logic applies no less to the assembly of a qualified SQL name. This is the reason that we insist that the `.` and `@` punctuation characters are part of the SQL syntax template and that the simple SQL name placeholder be exactly that: this placeholder must not be replaced by a qualified SQL name. This rule will be restated formally in the section “*Safe SQL statement text*” on [page 35](#).

It might be argued that the usability of a subprogram which allows the caller to choose the object it operates on is improved if the familiar syntax of the qualified SQL name can be expressed in a single actual argument. There is a simple way to accommodate such a requirement. The Oracle-supplied procedure `DBMS_Utility.Name_Tokenize()` decomposes a qualified SQL name into its simple SQL names. It takes no account of semantics: the denoted object need not exist;

51. The rules for composing a common SQL name and an exotic SQL name are given in the SQL Language Reference book; however, as was mentioned in the section “*Example 2: user-supplied table name*” on [page 14](#), these terms of art were invented for this paper.

52. To be precise, we should use the term *top level PL/SQL block*, as is defined in the section “*Static text*” on [page 33](#), rather than *subprogram*.

and it therefore gives no information about whether *a.b* is the element *b* in the package *a* or the object *b* owner by the user *a*. [Code_23](#) shows how to invoke it.

```
-- Code_23
procedure p(Qualified_SQL_Name in varchar2) is
  a varchar2(32767);
  b varchar2(32767);
  c varchar2(32767);
  DBlink varchar2(32767);
  Dummy pls_integer;
begin
  DBMS_Utility.Name_Tokenize(
    Qualified_SQL_Name,
    a,
    b,
    c,
    DBlink,
    Dummy);
```

The utility's first formal parameter (for which the actual *Qualified_SQL_Name* is used) has the mode *in*; and the remaining formal parameters (for which the actuals *a*, *b*, *c*, *DBlink*, and *Dummy* are used) have the mode *out*. The returned value of *a* will never be *null*. The values of *b*, *c*, and *DBlink* might be *null*; but *c* will never be *not null* unless *b* is also *not null*. *Dummy* carries no useful information: it is always equal to *Length(Qualified_SQL_Name)*.

An arguably better API design for a subprogram which allows the caller to choose the object it operates on is to provide an explicit formal parameter for each simple SQL name that will be used to compose the qualified SQL name. However, the choice of approach is unimportant with respect to this paper's focus.

Notice that by allowing the replacement of a simple SQL name placeholder in a SQL syntax template only with a simple SQL name, we force the programmer to take a deliberate decision in the design about, for example, whether an object in a remote database may be referenced. This self-evidently improves the safety of the design⁵³.

These, then, are the rules for composing a safe simple SQL name from a PL/SQL *numeric* value:

- Design the API so that the caller provides the putative name, in the PL/SQL *numeric* value, using exactly the syntax that would be used in a SQL statement: a common SQL name may be presented without enclosing double quote characters (when it will then be treated case-insensitively); an exotic SQL name must be presented with enclosing double quote characters.
- Ensure the safety of the name with *DBMS_Assert.Simple_Sql_Name()*.

The *DBMS_Assert.Simple_Sql_Name()* is a pure asserter: either the output is identical to the input; or the exception *DBMS_Assert.Invalid_Sql_Name* is raised. If the input *is not* surrounded by double quote characters, it is taken to be a common SQL name; here, the exception is raised if it violates the rules for such

53. There have been some discussions in public internet forums about how a program that uses the *DBMS_Assert* functions inappropriately can be vulnerable to injection. See, for example, the paper *Bypassing Oracle DBMS_Assert* by Alexander Kornbrust and discussions about it. This paper insists that the safety of object identification be asserted always and only with *DBMS_Assert.Simple_Sql_Name()*. Of course, other functions from the package may be used, *before the final check*, for their ordinary utility value.

a name. If the input *is* surrounded by double quote characters, it is taken to be an exotic SQL name; here, the exception is raised only if the name contains a singleton double quote character or a consecutive run of an odd number of such characters. However, a name that contains only properly doubled double quote characters will fail when it is used⁵⁴.

Finally, in this section, we should note that the *DBMS_Assert* package has functions to establish whether a putative first class object or a putative schema actually exists. While these might be convenient utilities for use in some very well-controlled scenarios, they have no value with respect to immunizing against the risk of SQL injection. Oracle Database is famously a multiuser, multiaccess environment. Therefore, a check for object existence might report a fact which no longer holds true at the moment an action is taken that should supposedly depend of the outcome of that check.

54. The attempt will cause *ORA-03001: unimplemented feature*.

RULES FOR COST-EFFECTIVE, GUARANTEED PREVENTION OF SQL INJECTION

Like any rules of best practice, the following do not prescribe the only way to achieve the goal. These rules are recommended because they are simple to state, easy to follow, easy to audit in a manual code review⁵⁵, and — most importantly — because they guarantee the objective: they make PL/SQL database code that adheres to them proof against SQL injection. The rules necessarily limit freedom. But we argue that the limited freedom that remains is sufficient to allow all reasonable application requirements to be supported.

Expose the database to clients only via a PL/SQL API

Briefly, the paradigm is to establish a database user as the *only* one to which a client may connect⁵⁶. This user may own *only* private synonyms; and these may denote *only* PL/SQL units. Necessarily, by application of this rule, the denoted PL/SQL units are owned by users as whom the client may not connect. The *Execute* privilege on exactly and only these denoted PL/SQL units is granted to the user as whom the client may connect.

This approach formally defines the API — subject to the caveat that objects that are accessible via privileges granted to *public* are therefore part of the API. (If appropriate, the scheme can be extended to several such users, for access by different kinds of client end-user, where each user has appropriate privileges.)

The API-defining PL/SQL units may be definer's rights or invoker's rights according to their purpose.

This paradigm locates the responsibility to prevent SQL injection where it belongs: in the subsystem of the overall application stack that executes the SQL. And it offers the only approach whose safety can be proved. The paradigm is a natural extension of the thinking that places all code that is responsible for the enforcement of data integrity in the database⁵⁷.

Rule_4

Expose the database to clients only via a PL/SQL API. Carefully control privileges so that the client has no direct access to the application's objects of other kinds — especially tables and views.

Expose the database to clients only via a PL/SQL API.

Use compile-time-fixed SQL statement text unless you cannot

If the complete text of a SQL statement, and not just the SQL syntax template, is fixed at compile time, then it takes no effort to prove that the site that issues that statement is proof against SQL injection: because the statement text is fixed, then so, in turn, is the SQL syntax template. We don't even need to discuss whether dynamic text⁵⁸ has been made safe, and the auditing task is therefore trivial (and correspondingly cheap).

55. They also allow the possibility, in a future release of Oracle Database, of at least partial mechanical auditing.

56. The passwords of all the other users are closely guarded and are not revealed to engineers who implement client-side code.

57. Notice that, as a side-benefit, the paradigm liberates the designer from requiring to use triggers to implement data integrity logic. When all data changes are made through PL/SQL subprograms, these can *directly* implement the data integrity logic.

Notice that while using embedded SQL guarantees a compile-time-fixed SQL statement text, it is not the only way to guarantee this. [Code_24](#) shows an obvious alternative.

```
-- Code_24
declare
  Stmt constant varchar2(80) :=
    'alter session
     set NLS_Date_Format = ''AD yyyy-mm-dd hh24:mi:ss''';
begin
  execute immediate Stmt;
  ...
end;
```

Here, we imagine that the Functional Specification document for the application insists that a report which lists some *datetime* values do so in a particular way⁵⁹.

The use of *constant* with an assignment statement that uses only PL/SQL static *varchar2* expressions⁶⁰ provably fixes the SQL statement text at compile time. The auditor need not study whatever expanse of code lies between the declaration of *Stmt* and its use as the argument of *execute immediate*; the PL/SQL compiler will refuse to compile the unit if it includes code that would change *Stmt*.

It is important, therefore, to separate the discussion of the method for executing the SQL at a particular site from the discussion of the SQL syntax template that will be executed at that site. Embedded SQL supports only these kinds of statement: *select*, *insert*, *update*, *delete*, *merge*, *lock table*, *commit*, *rollback*, *savepoint*, and *set transaction*.

For all other kinds of statement, one of PL/SQL's methods for dynamic SQL must be used. Moreover, for these, the *execute immediate* statement is sufficient in almost every case⁶¹.

There are plausible use cases that require SQL statements of the kinds that embedded SQL does not support in ordinary application code. But these occur very much more rarely for ordinary application code than for system code.

Rule_5

Insist that the Design Specification document for ordinary application code defends any proposal to execute SQL using any method other than Embedded SQL.

When the Design Specification document for ordinary application code proposes to use anything other than embedded SQL, insist on examining the rationale very carefully when the document is reviewed. The design may well be defensible. But this defense must be made explicitly.

Notice that the converse is true: when a statement of one of the kinds that Embedded SQL does support is to be executed, and when the statement text can be fixed at compile time, then dynamic SQL is never needed. Therefore,

58. The notion of dynamic text is defined formally in the section "*Dynamic text*" on [page 34](#).

59. For example, the report could be generated in a marked-up format for the specific purpose of import into a spreadsheet where the convention for representing dates has already been established.

60. The term PL/SQL static *varchar2* expression is defined in the PL/SQL Language Reference book and its significance is discussed in the section "*Static text*" on [page 33](#).

61. The cases where *execute immediate* is not sufficient are very rare indeed, and all need special consideration with respect to security. One example is when DDL statement must be executed in a remote database. In such cases, the *DBMS_Sql* API is needed.

embedded SQL should always be used for these cases. We understand now why the example in [Code_9](#) is a violation of the rule stated in this section.

There is, however, a practical consideration which forces us to moderate the statement of this rule. Sometimes, while the universe of possible SQL syntax templates that may be issued from a particular site is fixed at compile time, the number of these is so great (because they are generated programatically, albeit by assembling components all of which are fixed at compile time) that they cannot be set up at compile time as embedded SQL statements. This use case was discussed in the section “*Distinguishing between a static SQL syntax template and a dynamic SQL syntax template*” on [page 7](#). And it is illustrated in the section “*Query by example form*” on [page 40](#).

Rule_6

Use compile-time-fixed SQL statement text unless you cannot. Use Embedded SQL or execute immediate with a PL/SQL static varchar2 expression.

Use compile-time-fixed SQL statement text unless you cannot. Use embedded SQL when the SQL statement is of a kind that this supports. Otherwise, use *execute immediate* with single PL/SQL *constant* argument composed using only PL/SQL static *varchar2* expressions. When you conclude that you cannot, review the Functional Specification document and the Design Specification document carefully with colleagues and then explain in the latter exactly why compile-time-fixed SQL statement text cannot be used.

Use a static SQL syntax template for run-time-created SQL statement text unless you cannot

There are only very few requirements scenarios that might be used to justify a deliberate violation of [Rule_6](#). Each must be examined very carefully when the Design Specification document is reviewed.

Replacement of a value placeholder in a SQL syntax template

This requirement arises for two different reasons.

- There is no SQL syntax for binding to a regular placeholder for the required parameterization.
- A literal value is required to encourage an optimal execution plan⁶².

There are three reasons why binding to a placeholder might not be supported.

- While the SQL statement is of the kind that embedded SQL supports, there is a requirement to parameterize a value like that which determines the timeout period in a *select... for update* statement⁶³. SQL does not support the use of a regular placeholder for this purpose.
- There is a requirement to execute a SQL statement of the kind that requires Dynamic SQL and to parameterize it in some way⁶⁴. An example is provided

62. This requirement is plausible in the implementation of an application to expose the information in a datawarehouse. In such cases, the tables are enormous and the queries are complex. Moreover, the number of concurrent users is relatively low. The classical wisdom (bind to placeholders rather than using literals to reduce the frequency of the so-called SQL hard parse) doesn't hold. Parse time can be tiny in comparison to execution time, and contention is unimportant. Here, the use of a literal in a predicate, allowing more specific advantage to be taken of statistics, can be critical for getting acceptably short execution times.

63. Code to implement this use case was shown in [Code_5](#) on [page 7](#).

by an extension to the requirement that led to the implementation shown in [Code_24](#). Suppose, now, that *NLS_Date_Format* must be set at run-time in response to conditions that are first discovered then.

- The SQL statement (or fragment, like a *where clause*) is to be executed by one of the subprograms listed in [Appendix D: Additional Oracle-supplied subprograms that implement dynamic SQL](#) on [page 58](#) and this subprogram does not support binding to placeholders.

The safety of the dynamic text that is used to replace a value placeholder must be ensured according to the principles established in the section “[Ensuring the safety of a SQL literal](#)” on [page 19](#). This is stated compactly in [Rule_10](#) on [page 35](#).

Consider the implied Functional Specification document that led to the injectable implementation shown in [Code_12](#) on [page 17](#). Because the *from* list specifies exactly one item, it is likely that the requirement is to let the user mention some subset of the item’s columns and, for each, to express a comparison with a value. And it is likely that the combination of the predicates can be constrained to either uniformly mutual *or* (any of the predicates is satisfied) or mutual *and* (all of the predicates is satisfied). With about a dozen columns, the number of distinct SQL syntax templates is several thousand, so it is easy to see why it is tempting to let the user supply the *where clause* as string. However, there is no general way to define an algorithm that checks if such a string is indeed one of the several thousand allowed templates. There is, then, only one acceptable approach; this discussion is taken up in the section “[Don’t confuse the need to use a dynamic SQL syntax template with the need for dynamic text](#)” on [page 32](#).

Rule_7

Insist on a justification for a design that proposes to replace a value placeholder in a SQL syntax template.

A Design Specification document that proposes to replace a value placeholder in a SQL syntax template (whether this is a static SQL syntax template or a dynamic SQL syntax template) should be regarded with extreme suspicion. It must present a convincing argument for this approach before it is signed off.

Replacement of a simple SQL name placeholder in a SQL syntax template

The general characterization of the use case is that the application needs to operate on an object (which is typically a table) whose shape and purpose is given in the Design Specification document but whose identity is not known until run time.

While the safe approach for this scenario has been prescribed in the section “[Ensuring the safety of a simple SQL name](#)” on [page 25](#), it is still useful, as each new Design Specification document is written, to determine if an alternative approach is possible which uses only compile-time-fixed SQL statement text, and (for accessing a table) can therefore be implemented with embedded SQL.

Consider the sort of utility that is aimed manipulating data in one of a set of schemas where each such schema is populated by objects with the same names

-
64. It so happens that none of the kinds of SQL statement that embedded SQL does not support allows the use of a regular placeholder.

However it is relatively rare for ordinary application code to need to execute the kind of SQL that isn’t supported by embedded SQL. This need occurs only within special kinds of program. The higher level Functional Specification document should be very carefully reviewed to ensure that the proposed approach is essential.

and purposes. A naïve approach would encode the value returned by the *Sys_Context()* function for *Current_Schema* in the *Unserem* namespace into a qualified SQL name for each object to be accessed and would execute these statements with Dynamic SQL. The ideal solution, though, is to encapsulate the SQL statements that access the objects in an Invoker's Rights unit. The access can then be implemented ideally with embedded SQL which uses unqualified names. The code can be pointed at the intended objects by changing the *Current_Schema* — either programatically or by simply letting it follow the *Current_User*.

Sometimes an application needs to handle a very large quantity of data (more than can safely be held in a PL/SQL collection) but does not need to retain it beyond the lifetime of a session. Historically, applications used a pool of nominally temporary tables together with a management system to allocate, and later reclaim, such a table for use by a particular session. This, of course, meant that the table name was not known until run time. Oracle8i Database introduced the global temporary table. It is perfect for this purpose because it can be accessed using embedded SQL but no session sees another session's data.

However, there are some compelling use cases that non-negotiably require the replacement of a simple SQL name placeholder in a SQL syntax template with a simple SQL name. They tend to arise in the design of what is loosely referred to as system software. A convincing example is given by an implementation of a domain index that uses the ODCI framework. Each different index will use a set of tables whose names are typically derived from that of that index. The ODCI client code will need to derive such names at run time when it is called, by the Oracle system, with the name of the current index of interest.⁶⁵

Rule_8

Insist on a justification for a design that proposes to replace a simple SQL name placeholder in a SQL syntax template.

A Design Specification document that proposes to replace a simple SQL name placeholder in a SQL syntax template (whether this is a static SQL syntax template or a dynamic SQL syntax template) should be regarded with some suspicion. It must present a convincing argument for this approach before it is signed off.

Don't confuse the need to use a dynamic SQL syntax template with the need for dynamic text

The kind of Functional Specification document that cannot be implemented using a static SQL syntax template has already been briefly discussed⁶⁶ and the concept for the safe implementation has been illustrated with code⁶⁷. There, the requirement was to handle a *select* list whose composition is not known until run time. The requirement often goes hand-in-hand with one optionally to specify a restriction criterion for each column by supplying both the value and the comparison operator (*exact equals*, *like*, *less than*, and so on). There is also often a requirement to specify which columns to use, in which order, and with *ascending* or *descending* for each, to sort the results.

65. Oracle Text works this way.

66. See the section "Definition of dynamic SQL syntax template" on page 8.

67. See *Code_6* on page 8, *Code_7* on page 9, and *Code_8* on page 10.

The requirement to handle a *where* clause whose composition is not known until run time can lead junior programmers to decide that binding to placeholders when the requirement is not known at compile time is not feasible⁶⁸. More mature programmers realize that the *DBMS_Sql* API provides exactly the primitives that are needed to implement this. The approach is illustrated in the section “*Query by example form*” on page 40.

The coding effort is certainly nontrivial. But the size of that effort is never enough to justify abandoning binding in favor of encoding literals directly into the SQL statement.

In other words, the requirement to encode literals directly into the SQL statement (in this use case only for the reason of improving the execution plan) is orthogonal to the requirement to implement binding to a set of placeholders whose composition emerges first at run time.

Rule 9

Don't confuse the requirement to bind to a set of placeholders whose composition emerges first at run time, which is fully supported by the *DBMS_Sql* API, with a requirement to use directly encoded literals in pursuit of optimal query execution performance.

Don't confuse the need to use a dynamic SQL syntax template with the need to replace value placeholders in the template

Formal sufficient prescription for guaranteed safety

The foregoing exposition has aimed to show how a PL/SQL application that is vulnerable to SQL injection is also ordinarily buggy. It focussed, therefore, on how to write bug-free code that has the pleasant bonus of being proof against SQL injection. This section takes the auditor's viewpoint whose aim is to certify proof against SQL injection by inspecting extant code.

First, we need to establish some definitions.

Static text

Static text is either

- a PL/SQL static *varchar2* expression as defined in the PL/SQL Language Reference book⁶⁹, or

68. You need only to try to write the code to do this using *execute immediate* to realize that, because its *using* clause is fixed at compile time, the task is impossible.

69. This is the list as it appears in the Oracle Database 11g edition of the book:

- a simple literal, for example, *'abcdef'*
- a concatenation of simple literals, for example, *'abc' || 'def'*
- the literal *null*
- *To_Char(x)*, where *x* is a *pls_integer* static expression
- *To_Char(x, f, n)* where *x* is a *pls_integer* static expression and *f* and *n* are PL/SQL static *varchar2* expressions
- *x | y* where each of *x* and *y* is either a PL/SQL static *varchar2* expression or a *pls_integer* static expression.

The list is extended to include a *constant* declared in the specification of a package that has been assigned with PL/SQL static *varchar2* expressions.

Of course, *pls_integer* static expression is also defined in the same book.

The value of a PL/SQL static *varchar2* expressions is known, and therefore fixed, at compile time.

- an expression formed by an arbitrary concatenation of static text items, or
- the value of a local variable⁷⁰ that has been visibly assigned with static text.

This definition is intentionally recursive. A variable can be assigned some static text and then that variable (strictly speaking, its value⁷¹) can be used to build another yet larger static text. The tower can be built as high as one likes; but its foundation must consist only of PL/SQL static *varchar2* expressions.

Notice that the concatenation may be controlled by tests whose outcome is not known until run time. But it must be self-evident, following trivial human inspection, that every possible concatenation will result only in static text according to the rules stated in the preceding three bullets; this is what we mean by *visibly assigned*. (One might say that you must be able to see the foundation when you are standing on the top of a static text tower.)

[Code_25](#) shows an extract from the declarations in function *f()* shown in [Code_26](#) on [page 36](#).

```
-- Code_25
Tab_1 constant varchar2(32767) := 'Tab_1';
Tab_2 constant varchar2(32767) := 'Tab_2';
...
Tab constant varchar2(32767) :=
  case b
    when true then Tab_1
    else          Tab_2
  end;
```

Notice that *b* is a formal parameter and so the value of *Tab* will not be known until run time and will be, in general, different for each invocation of *f()*. Nevertheless, *Tab* is a static text according to the definition given above.

Dynamic text

Dynamic text is any text that is not static text. Obvious examples are formal parameters, variables declared at top level in a package without the *constant* keyword, and ordinary variables that are assigned by executing a SQL statement or as the actual argument for the *Buffer* formal parameter to *Utl_File.Get_Line()*.

Safe dynamic text

Safe dynamic text is the output of either

- *DBMS_Assert.Enquote_Literal()*, or
- *To_Char(x,f, 'NLS_Numeric_Chars = ",,")* where *x* is a variable of a numeric datatype and *f* is an the explicit format model *'TM'*, or
- *DBMS_Assert.Simple_Sql_Name()*.

70. A local variable is one that is declared within the present top level PL/SQL block. A top level PL/SQL block is one of the following: a schema-level function; a schema-level procedure; a function or procedure defined at top level within a package body or a type body; a package's initialization block; or the implementation of a trigger. Thus a variable that is declared at top level in a package or package body is, by definition, not a local variable.

71. The distinction between a variable and its (current) value becomes uninteresting when the declaration of the variable uses the *constant* keyword. It is always possible, with appropriate use of nested block statements, to declare all the variables that are used for the composition of static text this way. This practice is strongly encouraged because it makes the task of the human auditor very much easier.

Hereinafter, the shorthand $To_Char(x, f, n)$ will be used to denote $To_Char(x, f, 'NLS_Numeric_Characters = \", \")$. The background for this rule was established in the section “*Ensuring the safety of a SQL numeric literal*” on [page 23](#).

Notice that it follows from the definition of dynamic text that the safety of safe dynamic text must be established within the present top level PL/SQL block.

Safe SQL statement text

Safe SQL statement text is an arbitrary concatenation of static text and safe dynamic text. (A careful definition would require similar wording to that used in the definition of static text. Local variables may be used for intermediate results or for the final result; and it must be self-evident that every possible concatenation will result only in safe SQL statement text.)

We can now define a necessary prescription for guaranteed safety:

Rule_10

When a SQL statement represented by a PL/SQL *text* expression is executed using one of PL/SQL’s APIs for dynamic SQL, then the expression must be safe SQL statement text. Safe SQL statement text is a concatenation of static text and safe dynamic text. Static text is composed only of PL/SQL static *varchar2* expressions. Dynamic text is anything that isn’t static text. Safe dynamic text is the output of one of exactly three Oracle-supplied functions: `DBMS_Assert.Simple_Sql_Name()`, `DBMS_Assert.Enquote_Literal()`, and $To_Char(x, f, n)$.

Notice that the definition of safe SQL statement text pays no attention to the semantics of the resulting SQL statement and is, therefore, not sufficient.

Nevertheless, this rule is useful because, as it is easy to imagine, the PL/SQL compiler, in a future version of Oracle Database, could be enhanced to detect a violation of [Rule_10](#). Application code that violates the rule is definitely suspect and must be studied.

We must also guarantee that the composed SQL syntax template is only one of those that the programmer intended. We need to add [Rule_11](#) to [Rule_10](#) to give the sufficient prescription for guaranteed safety.

Rule_11

The safe dynamic text produced by `DBMS_Assert.Enquote_Literal()` or by $To_Char(x, f, n)$ should be used only at a spot within the SQL statement where a SQL literal is intended; and the safe dynamic text produced by `DBMS_Assert.Simple_Sql_Name()` should be used only at a spot within the SQL statement where a simple SQL name is intended.

It is very much harder to imagine how adherence to the stricter [Rule_11](#) could be mechanically policed. It would require a run-time check, for every single dynamic execution of SQL, by a subsystem that implemented complete knowledge of SQL syntax.

Anyway, in Oracle Database 11g, both [Rule_10](#) and [Rule_11](#) must be policed by ordinary human code review.

Dynamic SQL may execute only a concatenation of static text and safe dynamic text. Safe dynamic text is the output of one of exactly three Oracle-supplied functions: `Simple_Sql_Name()`, `Enquote_Literal()`, and $To_Char(x, f, n)$.

Use `Simple_Sql_Name()`, to ensure the safety of a SQL name. Use `Enquote_Literal()` to ensure the safety of a string or datetime literal. Use $To_Char(x, f, n)$ to ensure the safety of a numeric literal.

Code_26 shows an example⁷² that honors *Rule_10* and *Rule_11*. Function *f()* is guaranteed always to compose a SQL statement that conforms to the SQL syntax template shown in *Template_8*.

```
-- Template_8
select PK from &&1 where VC2 = &2

-- Code_26
function f(b in boolean, VC2 in varchar2) return number is
  Quote          constant varchar2(1) := ''';
  Doubled_Quote  constant varchar2(2) := Quote||Quote;

  Start_         constant varchar2(32767) := 'select PK from ';
  Tab_1         constant varchar2(32767) := 'Tab_1';
  Tab_2         constant varchar2(32767) := 'Tab_2';
  Where_        constant varchar2(32767) := ' where VC2 = ';

  Tab constant varchar2(32767) :=
    case b
      when true then Tab_1
      else          Tab_2
    end;

  SQL_Text_Literal constant varchar2(32767) :=
    Quote||Replace(VC2, Quote, Doubled_Quote)||Quote;

  Stmt constant varchar2(32767) :=
    Start_||
    Tab||
    Where_||
    Sys.DBMS_Assert.Enquote_Literal(SQL_Text_Literal);

  PK number;
begin
  execute immediate Stmt into PK;
  return PK;
end f;
```

When *f()* is invoked with actual arguments with values *true* for *b* and *O'Brien* for *VC2*, then the safe SQL statement text shown in *Value_17* is composed.

```
-- Value_17
select PK from Tab_1 where VC2 = 'O'Brien'
```

Establish the safety of run-time-created SQL statement text in the code that immediately precedes its execution

The definition established for safe dynamic text⁷³ insists that it be composed in the same top level PL/SQL block that executes the SQL statement, into which it is concatenated, using dynamic SQL. This is a sufficient condition to allow the future possibility of a mechanical check. However, the task of the human auditor is made very much easier if all calls to *DBMS_Assert.Simple_Sql_Name()*, *DBMS_Assert.Enquote_Literal()*, or to *To_Char(x,f,n)* are made in code that composes the SQL statement immediately before the PL/SQL statement that executes it.

Code_26 honors this principle.

72. Because this example replaces a value placeholder in a SQL syntax template, we must assume that this approach was deliberately chosen in preference to binding to a regular placeholder in order to increase the probability of a good execution plan as was discussed in the section "Replacement of a value placeholder in a SQL syntax template" on page 30.

73. See the section "Safe dynamic text" on page 34.

Keep the code that invokes
***Simple_Sql_Name()*,**
Enquote_Literal()*, or *To_Char(x f, n)
very close to the code that executes
the SQL statement that these calls
make safe.

Rule_12

Make the job of the auditor easy by establishing the safety of run-time-created SQL statement text in the code that immediately precedes the PL/SQL statement that executes it.

SCENARIOS

Each of the following requirements scenarios can be implemented very well using embedded SQL. Yet (incredibly, in the case of the first scenario) we have seen real production code where dynamic SQL has been used unsafely because the author didn't know how to program the embedded SQL solution.

Make a *like* predicate by adding leading and trailing % characters

[Code_27](#) shows an embedded SQL statement that implements a *like* predicate.

```
-- Code_27
select t.PK, t.c1 bulk
collect into Results
from t where c1 like '%'||x||'%';
```

It's impossible to speculate on the origin of the strange myth that says that this needs run-time-created SQL statement text.

It might help to see the SQL statement that the PL/SQL compiler constructs from this source code. The simplest way to observe this is, on one's own private development database, to encapsulate the code shown in [Code_27](#) in a procedure, to execute it, and to query the *v\$sql* view using a restriction that is selective enough to filter out irrelevant results⁷⁴. [Code_28](#) shows such a query.

```
-- Code_28
select   Sql_Text
from     v$sql
where    Lower(Sql_Text) not like '%v$sql%'
and      Lower(Sql_Text) like 'select%t.pk%t.c1%'
```

This is the output of [Code_28](#):

```
SELECT T.PK, T.C1 FROM T WHERE C1 LIKE '%'||:B1 ||'%'
```

Notice that the regular placeholder *:B1* has been established in the position where the variable *x* was used in the embedded SQL statement. The PL/SQL compiler generates appropriate code to do the binding.

As the companion paper, *Doing SQL from PL/SQL: Best and Worst Practices*, explains at some length, PL/SQL's embedded SQL and PL/SQL's dynamic SQL are both processed the same way at run time: the text of the SQL statement is submitted by the PL/SQL run-time system to the SQL subsystem for execution. The difference is that the PL/SQL compiler generates the text of the SQL statement from an embedded SQL statement at compile time and stores it with the unit's compiled code; but the SQL statement executed by dynamic SQL is composed at run time when the unit executes. Of course, as this paper has advocated, that composition might be no more demanding than the initialization of a PL/SQL *constant* with a PL/SQL static *varchar2* expression⁷⁵.

A programmer who thought that embedded SQL could not support a *like* predicate must have thought, if pushed to explain his thoughts, that the expression `'%'||:B1||'%'` was not legal in SQL; it most certainly is.

74. This is easy to arrange by inventing an unusual name for the test table. An alternative is simply to flush the shared pool before executing the test procedure.

75. It turns out that the optimizing PL/SQL compiler can do this at compile time.

In list with number of elements not known until run-time

This use case is described in the companion paper, *Doing SQL from PL/SQL: Best and Worst Practices*. For convenience, the account is reproduced here — slightly reworded to suit the needs of this paper’s pedagogy.

If, for some relatively improbable reasons, you need a query whose *where clause* uses an *in list* with, say, exactly five items, then you can write the embedded SQL statement without noticing the challenge that will present itself later when the requirements change to reflect the more probable use case. [Code_29](#) shows this unlikely statement.

```
-- Code_29
select          a.PK, a.v1
bulk collect into b.Results
from           t a
where          a.v1 in (b.p1, b.p2, b.p3, b.p4, b.p5);
```

[Code_30](#) expresses the intention of the far more likely statement.

```
-- Code_30
select          a.PK, a.v1
bulk collect into b.Results
from           t a
where          a.v1 in (b.ps(1), b.ps(2), b.ps(3),
                       b.ps(4), b.ps(5), b.ps(6),
                       b.ps(7), b.ps(8), b.ps(9),
                       ...
                       );
```

The problem is immediately apparent: we cannot bear to write an explicit reference to each element in a collection using a literal for the index value — and even if we could, the text would become unmanageably voluminous. Rather, we need a syntax that expresses the notion “all the elements in this collection, however many that might be”. Such a syntax exists and is supported in embedded SQL; [Code_31](#) shows it.

```
-- Code_31
ps Strings_t;
begin
  select          a.PK, a.v1
  bulk collect into b.Results
  from           t a
  where          a.v1 in (select Column Value
                        from table(b.ps));
```

However, this seems to be relatively little known, possibly because it uses the *table operator*⁷⁶. The datatype of *ps* must be declared at schema level. [Code_32](#) shows the SQL*Plus script that creates it.

```
-- Code_32
create type Strings_t is table of varchar2(30)
/
```

It is not uncommon for programmers who don’t know about the *table operator* to satisfy the functionality requirement by building the text of the SQL statement at run time. At best, this implementation is cumbersome and inefficient; and at worst, it will be vulnerable to SQL injection.

76. The use of the *table operator* is explained in the section *Manipulating Individual Collection Elements with SQL* in the *Object-Relational Developer’s Guide*. The *PL/SQL Language Reference* mentions it, with no explanation, only in the *PL/SQL Language Elements* section.

Query by example form

This scenario complements the one discussed in the section “*Definition of dynamic SQL syntax template*” on page 8⁷⁷. There, the Functional Specification document required a customizable report format for a fixed restriction condition (primary key identity). In this section, the Functional Specification document requires a fixed report format but a customizable restriction condition. This leads to the need for a design which, if it is to avoid replacing value placeholders in the dynamic SQL syntax template with SQL literals, needs to bind to placeholders, the number of which, and the datatypes of what they stand for, are unknown until run time.

Neither the requirement to handle a *select* list whose composition is not known until run time nor the requirement to handle an *order by* clause whose composition is not known until run time leads to design choices that impact the discussion of exposure to the risk of SQL injection. Therefore, to keep the code illustrations as short as possible, this section shows only how to code when the *where clause* is not known at compile time.

This paper needs to do only two things: to show how to build a SQL statement as an instance of a dynamic SQL syntax template but using only static text; and to show how to implement the binding that this implies⁷⁸.

consider the procedure *x.p()* as shown in [Code_33](#).

```
-- Code_33
package x is
  type t1 is Record(
    Val varchar2(4000),
    Exact boolean := false);
  type t2 is varray(20) of t1;
  procedure p(Cols in t2);
end x;
```

Its purpose is to list the values of the primary key, PK, for rows that satisfy the customizable *where clause*. Just as with [Code_6](#) on page 8, the author of code that will call *x.p()* knows the purpose of *My Table*⁷⁹, and the names and significance of all its columns. All this, and especially the names, is fixed in the Design Specification document. In particular, the programmer knows the order in which the columns are listed in its external documentation. The *nth* element of the *in* formal parameter *Columns* determines if, and how, the *nth* column of *My Table* is to be used in the *where clause*. When *Val* is not null, the *nth* column is to be included; and, in this case, *Exact* specifies whether an equality comparison or a *like* comparison is to be used.

77. The code that implements this scenario is shown in [Code_6](#) on page 8, [Code_7](#) on page 9, and [Code_8](#) on page 10.

78. The need to compose the *order by* clause at run time impacts only the composition of the SQL statement; it has no impact on binding nor on how the results are fetched. The need to compose the *select* list at run time, while it has no impact on binding, does potentially affect how the results are fetched. The approach shown in [Code_7](#) on page 9 sidesteps this by concatenating all the required columns into a single *text* item. Sometimes, though, each *select* list item must be fetched into its own variable, and the datatypes of the columns have to be taken into account. The *DBMS_Sql* API is well able to support this; but the techniques have no consequence for the subject of this paper.

79. The exotic name *My Table* was chosen deliberately to make a teaching point in this example.

The implementation of *p()* in the body of package *x* is shown in [Code_34](#). The *Where_Clause()* inner function composes the *where clause*.

```
-- Code_34
procedure p(Cols in t2) is
  type t3 is varray(200) of "My Table".PK%type;
  Results t3;

  type cn is varray(20) of varchar2(30);
  Col_Names constant cn := cn('c1', 'c2', 'c3', 'c4');

  function Where_Clause return varchar2;

  Stmt constant varchar2(32767) :=
    'select PK from "My Table"'
    || Where_Clause()
    || ' order by PK';

  function Where_Clause return varchar2 is ... end Where_Clause;

begin
  DBMS_Output.Put_Line(Stmt);

  declare
    nc integer := DBMS_Sql.Open_Cursor(Security_Level=>2);
    rc Sys_Refcursor;
    Dummy number;
  begin
    DBMS_Sql.Parse(nc, Stmt, DBMS_Sql.Native);

    for j in 1..Cols.Count() loop
      if Cols(j).Val is not null then
        DBMS_Sql.Bind_Variable(nc, ':b'||j, Cols(j).Val);
      end if;
    end loop;

    Dummy := DBMS_Sql.Execute(nc);

    rc := DBMS_Sql.To_Refcursor(nc);
    fetch rc bulk collect into Results;
    for j in 1..Results.Count() loop
      DBMS_Output.Put_Line(Results(j));
    end loop;
    close rc;
  end;
end p;
```

Notice that, following this paper's recommendation in [Rule_1](#) on [page 9](#), *Stmt* is declared using the *constant* keyword and that this means that it must be initialized as part of the declaration. Notice too that the assignment to *Stmt* starts like this:

```
Stmt constant varchar2(32767) :=
  'select PK from "My Table"' || ...
```

The exotic SQL name *My Table* is enclosed with double quote characters but its safety is *not* ensured with *DBMS_Assert.Simple_Sql_Name()*. This is entirely safe. The reason is that it occurs within a PL/SQL static *varchar2* expression and therefore its spelling is established with certainty by the PL/SQL source text⁸⁰. The same logic applies to the list ('c1', 'c2', 'c3', 'c4') that initializes the *Col_Names* static *constant*. The names of the columns are given in the Design Specification document; and *x.p()* is specified to work against *My Table* with the columns it has. Because the columns all have common SQL names, there is no need to surround each with double quote characters⁸¹.

80. It would not be harmful to use *DBMS_Assert.Simple_Sql_Name()* in the initialization of *Stmt*. However, because this is manifestly unnecessary, to do so might confuse subsequent maintainers of the code.

The key feature in this example, with respect to the binding challenge, is that *DBMS_Sql.Bind_Variable()*, as a procedure, can be invoked in a loop as many times as emerges at run time to be necessary and with actual arguments whose values are known only then. The *execute immediate* statement and the *open rc for* statement, because the elements to be bound are written in the source code of the PL/SQL unit, support only a binding requirement that is fixed at compile time.

The design capitalizes on the fact that the *select* list is known at compile time. It uses *DBMS_Sql.To_Refcursor()*⁸² to move from the regime of the *DBMS_Sql* API to that of native dynamic SQL to fetch the results in a single *fetch... bulk collect* statement. Notice that the call to *DBMS_Sql.Open_Cursor()* uses an actual value 2 for the formal *Security_Level*⁸³. Using this ensures that every operation attempted using a cursor opened this way must be done by the same user that opened it with the same, or greater, set of enabled roles⁸⁴.

For completeness, the implementation of the *Where-Clause()* function is shown in [Code_35](#).

```
-- Code_35
function Where-Clause return varchar2 is
  Clause varchar2(32767);
  Seen_One boolean := false;
begin
  for j in 1..Cols.Count() loop
    if Cols(j).Val is not null then
      Clause :=
        Clause
        || case Seen_One
           when true then ' and '
           else           ' where '
        end
        || Col_Names(j)
        || case Cols(j).Exact
           when true then ' = :b'||j
           else           ' like ''%''||:b'||j||''||''%''
        end;
      Seen_One := true;
    end if;
  end loop;
  return Clause;
end Where-Clause;
```

Notice that the design is very similar to that for the *Column_List()* function shown in [Code_8](#) on [page 10](#) and that elements from which *Clause* is composed, here too, are all PL/SQL static *varchar2* expressions.

Again as with the *Column_List()* function, the set of the more than one million distinct SQL statements that *x.p()* might execute is too numerous to allow each

-
81. Writing the list as ('C1', 'C2', 'C3' 'C4') would have no consequence for correctness of for immunity to SQL injection. (Curly double quote characters are used to write this because the straight double quote character that source code demands is, in the font used, visually indistinguishable from two straight single quote characters.)
 82. This function, and its counterpart *DBMS_Sql.To_Cursor_Number()*, were introduced in Oracle Database 11g.
 83. The new overload for *DBMS_Sql.Open_Cursor()*, that has the *Security_Level* formal, was introduced in Oracle Database 11g.
 84. This eliminates the a rather different kind of vulnerability named by [David Litchfield](#) as *Cursor Injection* and described his article available on the internet here: www.databasesecurity.com/dbsec/cursor-injection.pdf.

one to be inspected. Rather, the programmer (and auditor) must reason what the possible members are — and from that deduce what the possible SQL syntax templates are. In this example, there are very many distinct SQL syntax templates. [Template_9](#) shows some.

```
-- Template_9
select PK from "My Table"
where c1 like '%||:b1||%' order by PK

select PK from "My Table"
where c1 = :b1 order by PK

select PK from "My Table"
where c1 like '%||:b1||%' and c3 = :b3 and c4 = :b4 order by PK
```

Because these templates use no value placeholders or simple SQL name placeholders, there is no need for dynamic text and, tautologically, no need to ensure its safety. This approach is self-evidently not vulnerable to SQL injection. Should, in a bungled maintenance cycle, the name of the table or any of its columns be changed without correspondingly updating the implementation of $x.p()$, then it would simply fail at run time with an ordinary semantic error.

Callback

The requirement is that a compiled PL/SQL unit must, without recompiling it, be somehow instructed to call a subprogram whose identity isn't known until run time. Though the identity is unknown, the signature of formal parameters and their meanings is known⁸⁵. This scenario is relatively common in ISV applications that allow the customer to supply code to perform a generically characterized operation in a customer-specific fashion.

The challenge can be illustrated by specifying this simple shape for the to-be-called subprogram:

```
function Callback(Input in integer) return integer
```

The “obvious” design, shown in [Code_36](#), uses dynamic SQL.

```
-- Code_36
procedure p(Name in varchar2) is
...
  Safe_Name constant varchar2(32767) :=
    Sys.DBMS_Assert.Simple_Sql_Name(Name);
  Stmt constant varchar2(32767) :=
    'begin :x := '||Safe_Name||'(:n); end;';
begin
  execute immediate Stmt using out x, in n;
```

Here, p is the unit that must not be recompiled. The actual for $P.Name$ would be obtained at run-time by, for example, selecting from a schema-level table⁸⁶.

The approach does follow the rules that this paper sets out and is proof against SQL injection. However, as we shall see, a better approach is available; it takes

85. This is analogous to the scenario where a table's identity is unknown until run time but the name and purpose of each column is.

86. The ISV's instructions would explain that the customer should create a callback function with the required shape and with any name. To simplify this example, we assume that this must be done in a specified schema. Then the customer should insert a row into a specified table to state what name he used for the callback.

advantage of dynamic polymorphism — sometimes known as virtual dispatch⁸⁷. This is one of the cornerstones of object-oriented programming and it goes hand-in-hand with subtyping. The to-be-called subprogram is described first as the *not final* member method of an abstract datatype. [Code_37](#) shows this.

```
-- Code_37
type Hook authid Current_User is object(
  Dummy number,
  not instantiable member function Callback(
    self in Hook,
    Input in integer ) return integer
)
not final
not instantiable
```

Other syntactic elements are needed in the definition of the *type*.

- A *type* has at least one data attribute. It is declared as “*Dummy number*” to emphasize that it has no significance for the implementation.
- The element *not final* allows a *type* to be created *under Hook*.
- The *Callback()* method is declared as *not instantiable* because no implementation will be provided at this level in the *type* hierarchy. *Hook* will be the supertype of a subtype which will describe the actual implementation. Because it has a *not instantiable* method, *Hook* itself must also be declared as *not instantiable*.
- Every member method of a *type* has the implicit first formal *self* which by default is of *in out* mode. It may alternatively be declared as *in*, but not as *out*. *In out* is the default because member methods are typically used to set data attributes. *In out* formals are passed by value unless the *nocopy* hint is used. For this particular use case, where the attribute *Dummy* is never referenced, the best practice is to establish *self* explicitly as *in*; formals with mode *in* are passed by reference.
- The qualifier *authid current_user* — to establish the supertype with invoker’s rights — is used in accordance with probable best practice. The choice between invoker’s rights and definer’s rights would be taken in a real project in the light of the overall requirements.

87. The complete listing, from which the code in this section is extracted, is shown in *Appendix E: Self-contained code to illustrate implementing callback using dynamic polymorphism*, on [page 61](#).

As mentioned, the implementation of the to-be-called subprogram is described in a subtype of *Hook* :

```
-- Code_38
type My_Implementation under Hook(
  overriding member function Callback(
    ...
    input in integer) return integer
)

type body My_Implementation is
  overriding member function Callback(
    ...
    input in integer) return integer
is
begin
  ...
  return ...;
end Callback;
end;
```

The element *overriding member* establishes the method as the actual implementation for the virtual declaration in the supertype.

The magic of the above construct is that a variable of the supertype *Hook* — say, *Obj* — can be given a value at run-time which is an instance of the subtype *My_Implementation*; and an invocation of what is apparently the supertype's virtual method — *Obj.Callback()* — will in that case denote the overriding match declared in the subtype. This gives you what we could loosely call a dynamic case statement whose legs can be determined after having compiled the statement, and without recompiling it, simply by creating new compilation units. It's as if you could make *p()*, in [Code_39](#), acquire an new leg to invoke *Callback3()* in its *case* expression by merely by establishing *Callback3()* in a new compilation unit — without needing to recompile *p()*.

```
-- Code_39
function Callback1(input in integer) return integer is
begin return input*2; end Callback1;

function Callback2(input in integer) return integer is
begin return input*2; end Callback2;

procedure P(Which pls_integer) is
  ...
begin
  x := case Which
        when 1 then Callback1(n)
        when 2 then Callback2(n)
        ...
      end;
  ...
end;
```

[Code_40](#) shows how to implement intent of [Code_36](#) using dynamic polymorphism.

```
-- Code_40
procedure p(Obj in Hook) is
  ...
begin
  ...
  x := Obj.Callback(n);
  ...
end;
```

The actual for *p.Obj* would be obtained at run-time by, for example, selecting from a schema-level table. This time, instead of using a *varchar2* column to hold the name of the implementation, the column would have datatype *Hook* and would be populated with an instance of the subtype *My_Implementation*.

The dynamic SQL approach, shown in [Code_36](#), uses a static SQL syntax template and safely replaces a simple SQL name placeholder with a simple SQL name. It would take the auditor some effort to prove that this code was safe. The approach that uses dynamic polymorphism, shown in [Code_40](#), doesn't use SQL at all and so is self-evidently safe.

As a bonus, the approach that uses dynamic polymorphism has two additional advantages.

- Tests show that the dynamic polymorphism approach is on the order of 10 times faster than the dynamic SQL approach when, as is typical, the body of the callback executes relatively quickly compared to the cost of the PL/SQL to SQL to PL/SQL context switch.
- When the dynamic polymorphism approach is used, *Callback()*'s formal parameters may have any PL/SQL datatype but when the dynamic SQL approach is used, these datatypes are limited to only those that SQL understands.

ANALYSIS AND HARDENING OF EXTANT CODE

Through Oracle Database 11g, this must be a mainly manual task. *Ad hoc* methods must be used to find those top level PL/SQL blocks that do dynamic SQL.

It is easy to find all of the sites where a procedural API is used⁸⁸. PL/Scope⁸⁹ can be used to identify each site with no risk of omission or of false positives. However, there is no corresponding method to find the sites that issue native dynamic SQL⁹⁰. There is no alternative except to search the source code manually. Any PL/SQL unit with such a sites will contain either both words *execute* and *immediate* or both words *open* and *for*. Therefore, a mechanical search in *User_Source*, seen from the viewpoint of the relevant users, might be able to prune away some units.

Once the interesting units have been identified then it remains, for each relevant top level PL/SQL block, to inspect the source code manually to determine if the rules that this paper has advocated have been followed.

When violations are found, they will be where a SQL statement is executed using a run-time-created SQL statement text composed with dynamic text that has not been made safe according to the principles that this paper has described⁹¹.

In a fortunate case, the auditor will be able to establish quickly that a static SQL syntax template was intended but that dynamic text is being used unsafely. Here, the addition of calls to *DBMS_Assert.Simple_Sql_Name()*, *DBMS_Assert.Enquote_Literal()*, or *To_Char(x,f,n)* just before the PL/SQL statement that executes the run-time-created SQL statement text will eliminate the vulnerability.

In a less fortunate case, where the design relies on the assumption that SQL keywords and operators will be supplied as dynamic text, the solution can only be radical re-design. But, at least, the vulnerability will have been identified.

88. The relevant subprograms are listed in *Appendix D: Additional Oracle-supplied subprograms that implement dynamic SQL* on page 58.

89. PL/Scope was introduced by Oracle Database 11g. It is described in the chapter *Using PL/Scope* in the Oracle Database Advanced Application Developer's Guide.

90. Enhancement request 6913337 asks to provide a way to do this.

91. See especially the section "*Ensuring the safety of a SQL literal or a simple SQL name*" on page 19 and the section "*Formal sufficient prescription for guaranteed safety*" on page 33.

CONCLUSION

This paper has carefully defined, and named, some notions⁹² that are essential to support a proper discussion of SQL injection and its avoidance⁹³:

- *common SQL name* and *exotic SQL name*
- *compile-time-fixed SQL statement text* and *run-time-created SQL statement text*
- *value placeholder*, *simple SQL name placeholder*
- *SQL syntax template*, *static SQL syntax template*, and *dynamic SQL syntax template*
- *static text*, *dynamic text*, *safe dynamic text*, and *safe SQL statement text*
- *top level PL/SQL block*

And it has relied on careful use of some established notions and terminology:

- *regular placeholder*, *simple SQL name*, *qualified SQL name*
- *embedded SQL*, *native dynamic SQL*, and the *DBMS_Sql API*
- *PL/SQL numeric value*, *PL/SQL datetime value*, and *PL/SQL text value*
- *SQL numeric literal*, *SQL datetime literal*, and *SQL text literal*
- *PL/SQL static varchar2 expression* and *PL/SQL static varchar2 constant*

This allowed us to establish a simple, brief definition of the paper's topic, from its asserted perspective: SQL injection occurs at a particular call site in a PL/SQL subprogram when a SQL statement is executed whose SQL syntax template differs from what the subprogram's author intended. It also allowed us to see that SQL injection is possible only when run-time-created SQL statement text is executed.

This gave the basis for understanding how most cost-effectively to ensure that PL/SQL code is not vulnerable to SQL injection.

- Aim to use execute only compile-time-fixed SQL statement text using, as appropriate, either embedded SQL, *execute immediate*, or *open for* with a PL/SQL static *varchar2* expression.
- Only after explaining carefully in the Design Specification document why compile-time-fixed SQL statement text is insufficient to meet the requirements in the Functional Specification document, aim to execute run-time-created SQL statement text that is entirely static text.
- Only after explaining carefully why neither compile-time-fixed SQL statement text nor run-time-created SQL statement text that is entirely static text is sufficient, use run-time-created SQL statement text that is safe SQL statement text.
- When composing this safe SQL statement text
 - ensure the safety of dynamic text that replaces a value placeholder in the SQL syntax template with *DBMS_Assert.Enqueue_Literal()* or with

92. See *Appendix B: Definitions of new terms of art introduced by this paper* on [page 51](#).

93. This seems never to have been done before.

To_Char(x, f, n). For this design, explain explicitly why binding to a regular placeholder is insufficient. Use the former for a SQL *text* literal or a SQL *datetime* literal. Additionally, when composing a SQL *datetime* literal, use *To_Char()* with a format model appropriate for the required precision and encode *To_Date()* with the same model into the safe SQL statement text. Use *To_Char(x, f, n)* for a SQL *numeric* literal

- ensure the safety of dynamic text that replaces a simple SQL name placeholder in the SQL syntax template with *DBMS_Assert.Simple_Sql_Name()*.

Clearly, then, the expense of implementing a design that is proof against SQL injection increases with each successively more sophisticated approach. However, this expense must never be used to justify taking shortcuts in the implementation. Only by honoring the principles that this paper has set out can you be sure that you are safe from the threat of SQL injection.

Mark Fallon, mark.l.fallon@oracle.com
Bryn Llewellyn, bryn.llewellyn@oracle.com
Howard Smith, howard.smith@oracle.com
Oracle Headquarters
26-October-2008

APPENDIX A: CHANGE HISTORY

19-September-2008

- First published version.

21-September-2008

- Completing the section “*Appendix D: Additional Oracle-supplied subprograms that implement dynamic SQL*” on [page 58](#). Some minor corrections in response to review by colleagues.

26-October-2008

Changed the title from *How to write injection-proof PL/SQL* to *How to write SQL injection proof PL/SQL*. to increase the chances of an internet search for “SQL injection” finding this whitepaper.

All occurrences of *Google* and its inflexions replaced with brand-free references to internet searching.

Several minor typos corrected.

APPENDIX B: DEFINITIONS OF NEW TERMS OF ART INTRODUCED BY THIS PAPER

This appendix collects the definitions of the new terms of art that this paper has introduced.

common SQL name

A *common SQL name* starts with an upper case alphabetic character in the range *A..Z* and then has only upper case alphanumeric characters in the range *A..Z*, or underscore, # or \$. A common SQL name doesn't need to be surrounded by double quote characters in a SQL statement; and if it is not so surrounded, the case with which it is written doesn't matter. It may, however, be so surrounded. If it is, then because the SQL parser preserves its case, it must be written in all upper case.

See *Example 2: user-supplied table name* on [page 14](#).

exotic SQL name

An *exotic SQL name* is one that violates the rules for a common SQL name and that *must*, therefore, be surrounded by double quote characters in a SQL statement.

See *Example 2: user-supplied table name* on [page 14](#).

compile-time-fixed SQL statement text

Compile-time-fixed SQL statement text is the text of a SQL statement that cannot change at run time and that can be confidently determined by reading the source code of the PL/SQL unit that executes the SQL statement in question. For dynamic SQL, it is the text of a SQL statement that is a PL/SQL static *varchar2* expression. The value of a PL/SQL static *varchar2* expression cannot change at run time and could be pre-computed at compile time. Because the PL/SQL compiler composes the text of the SQL statement that implements a PL/SQL embedded SQL statement, we regard this, too, as compile-time-fixed SQL statement text.

See *Distinguishing between compile-time-fixed SQL statement text and run-time-created SQL statement text* on [page 6](#).

run-time-created SQL statement text

Run-time-created SQL statement text is the text of a SQL statement that is *not* compile-time-fixed SQL statement text.

See *Distinguishing between compile-time-fixed SQL statement text and run-time-created SQL statement text* on [page 6](#).

SQL syntax template

A *SQL syntax template* looks something like a regular SQL statement but the notion belongs in the domain of discourse of the Design Specification document. Here is an example:

```
select &&1 from &&2 where &&3 = &4
```

It is a documentation device to prescribe the set of regular SQL statements that are instances of the template. Instantiation, by a PL/SQL program, is achieved

by using regular SQL elements in place of the elements that start with `&&` or `&` in the template. All the other non- whitespace elements in the template must be reproduced faithfully in the instantiation as a regular SQL statement. This includes a SQL hint — but excludes ordinary comments, which can be treated as whitespace.

See *Introducing a new notion: SQL syntax template* on [page 4](#).

value placeholder

A *value placeholder* is the element in a SQL syntax template that starts with `&`. It stands for *either* a well-formed SQL literal *or* a regular placeholder in a regular SQL statement.

See *Introducing a new notion: SQL syntax template* on [page 4](#).

simple SQL name placeholder

A *simple SQL name placeholder* is the element in a SQL syntax template that starts with `&&`. It stands for a simple SQL name in a regular SQL statement.

See *Introducing a new notion: SQL syntax template* on [page 4](#).

static SQL syntax template

A *static SQL syntax template* is a SQL syntax template that can be written down explicitly in the Design Specification document. The implementation that instantiates and executes a static SQL syntax template can always be written like this, and *should* be:

```
<<Some_Inner_Block>>declare
  Stmt_constant varchar2(32767) :=
    'select c1 from '
    || Sys.DBMS_Assert.Simple_Sql_Name(Dynamic_Name)
    || ' where PK = :b for update wait '
    || Sys.DBMS_Assert.Enquote_Literal(Dynamic_Value);
begin
  execute immediate Stmt...
  ...
end Some_Inner_Block;
```

The assignment is a single concatenation of one or several PL/SQL static *varchar2* expressions with one or several invocations of any of

- `DBMS_Assert.Enquote_Literal()`, or
- `To_Char(x f, 'NLS_Numeric_Chars = ",,")`, or
- `DBMS_Assert.Simple_Sql_Name()`.

A static SQL syntax template can be used, for example, when it would have been possible to use embedded SQL were it not for the fact that the name(s) of the *from* list items are not known until run time.

By definition, compile-time-fixed SQL statement text conforms to a static SQL syntax template.

See *Definition of static SQL syntax template* on [page 7](#).

dynamic SQL syntax template

A *dynamic SQL syntax template* is one of a large set of such templates that are designed to prescribe the regular SQL statements for execution at a particular

call site, where the set is too large to allow each to be written down but where, nevertheless, the set can be described with certainty. The description could be written in the Design Specification document using, for example, regular expression syntax.

A set of dynamic SQL syntax templates is needed, for example, to implement a query by example interface of the kind illustrated in the section “*Query by example form*” on [page 40](#). A dynamic SQL syntax template needs to use neither simple SQL name placeholders nor value placeholders in order to be useful — and its safety is easier to determine in a code review when it does not. In such a use, every regular SQL statement that conforms to the set of dynamic SQL syntax templates for a particular call site will be composed using only PL/SQL static *varchar2* expressions.

Run-time-created SQL statement text might conform to a static SQL syntax template or might conform to a dynamic SQL syntax template.

See *Definition of dynamic SQL syntax template* on [page 8](#).

static text

Static text is either

- a PL/SQL static *varchar2* expression as defined in the PL/SQL Language Reference book, or
- an expression formed by an arbitrary concatenation of static text items, or
- the value of a local variable that has been visibly assigned with static text. (A local variable is one that is declared within the present top level PL/SQL block.)

This definition is intentionally recursive. The concatenation may be controlled by tests whose outcome is not known until run time. But it must be self-evident, following trivial human inspection, that every possible concatenation will result only in static text whose ultimate source is only PL/SQL static *varchar2* expressions.

See *Static text* on [page 33](#).

dynamic text

Dynamic text is any text that is not static text. Obvious examples are formal parameters, variables declared at top level in a package without the *constant* keyword, and ordinary variables that are assigned by executing a SQL statement or as the actual argument for the *Buffer* formal parameter to *Utl_File.Get_Line()*.

See *Dynamic text* on [page 34](#).

safe dynamic text

Safe dynamic text is the output of either

- *DBMS_Assert.Enquote_Literal()*, or
- *To_Char(x, f, 'NLS_Numeric_Characters = ",,")* where *x* is a variable of a numeric datatype and *f* is an the explicit format model 'TM', or
- *DBMS_Assert.Simple_Sql_Name()*.

See *Safe dynamic text* on [page 34](#).

safe SQL statement text

Safe SQL statement text is an arbitrary concatenation of static text and safe dynamic text. Local variables may be used for intermediate results or for the final result.

See *Safe SQL statement text* on [page 35](#).

top level PL/SQL block

A top level PL/SQL block is one of the following: a schema-level function; a schema-level procedure; a function or procedure defined at top level within a package body or a type body; a package's initialization block; or the implementation of a trigger. Thus a variable that is declared at top level in a package or package body is, by definition, not a local variable.

See *Static text* on [page 33](#).

APPENDIX C: SUMMARY OF SQL INJECTION PREVENTION RULES

This appendix collects the rules for immunization against SQL injection that have been advocated in this paper.

When composing a SQL statement programmatically, help the code reviewer by declaring variables used for intermediate results as constant.

- ✓ *Rule_1:* When it is necessary to compose a SQL statement programmatically, the code usually needs, or at least benefits from the use of, variables for intermediate results. Aim to declare these as constant, assigning the values in the declarations. This sometimes requires the use of nested block-statements or forward-declared functions. This technique makes code review easier because the reader can be sure that the value of a variable cannot change between its initial assignment and its use. ([page 9](#))

Understand what is meant by the term SQL syntax template and the difference between a static SQL syntax template and a dynamic SQL syntax template.

- ✓ *Rule_2:* Understand what is meant by the term SQL syntax template. Apply this understanding to the design of any code that constructs run-time-created SQL statement text. Understand the difference between a static SQL syntax template and a dynamic SQL syntax template. ([page 11](#))

Understand that SQL injection is the execution of a SQL statement with an unintended SQL syntax template and that the risk can occur only when run-time-created SQL statement text is executed using dynamic SQL.

- ✓ *Rule_3:* Understand how to define the term SQL injection as the execution of a SQL statement with an unintended SQL syntax template. Know that only run-time-created SQL statement text that, therefore, must be executed using dynamic SQL is potentially vulnerable to SQL injection. ([page 11](#))

Expose the database to clients only via a PL/SQL API.

- ✓ *Rule_4:* Expose the database to clients only via a PL/SQL API. Carefully control privileges so that the client has no direct access to the application's objects of other kinds — especially tables and views. ([page 28](#))

Insist that the Design Specification document for ordinary application code defends any proposal to execute SQL using any method other than Embedded SQL.

- ✓ *Rule_5:* When the Design Specification document for ordinary application code proposes to use anything other than embedded SQL, insist on examining the rationale very carefully when the document is reviewed. The design may well be defensible. But this defense must be made explicitly. ([page 29](#))

Use compile-time-fixed SQL statement text unless you cannot. Use Embedded SQL or execute immediate with a PL/SQL static varchar2 expression.

- ✓ *Rule_6:* Use compile-time-fixed SQL statement text unless you cannot. Use embedded SQL when the SQL statement is of a kind that this supports. Otherwise, use execute immediate with single PL/SQL constant argument composed using only PL/SQL static varchar2 expressions. When you conclude that you cannot, review the Functional Specification document and the Design Specification document carefully with colleagues and then explain in the latter exactly why compile-time-fixed SQL statement text cannot be used. ([page 30](#))

Insist on a justification for a design that proposes to replace a value placeholder in a SQL syntax template.

- ✓ *Rule_7:* A Design Specification document that proposes to replace a value placeholder in a SQL syntax template (whether this is a static SQL syntax template or a dynamic SQL syntax template) should be regarded with extreme suspicion. It must present a convincing argument for this approach before it is signed off. ([page 31](#))

Insist on a justification for a design that proposes to replace a simple SQL name placeholder in a SQL syntax template.

- ✓ *Rule_8:* A Design Specification document that proposes to replace a simple SQL name placeholder in a SQL syntax template (whether this is a static SQL syntax template or a dynamic SQL syntax template) should be regarded with some suspicion. It must present a convincing argument for this approach before it is signed off. ([page 32](#))

Don't confuse the need to use a dynamic SQL syntax template with the need to replace value placeholders in the template with SQL literals.

- ✓ *Rule_9:* Don't confuse the requirement to bind to a set of placeholders whose composition emerges first at run time, which is fully supported by the DBMS_Sql API, with a requirement to use directly encoded literals in pursuit of optimal query execution performance. ([page 33](#))

Dynamic SQL may execute only a concatenation of static text and safe dynamic text. Safe dynamic text is the output of one of exactly three Oracle-supplied functions: Simple_Sql_Name(), Enquote_Literal(), and To_Char(x f, n).

- ✓ *Rule_10:* When a SQL statement represented by a PL/SQL text expression is executed using one of PL/SQL's APIs for dynamic SQL, then the expression must be safe SQL statement text. Safe SQL statement text is a concatenation of static text and safe dynamic text. Static text is composed only of PL/SQL static varchar2 expressions. Dynamic text is anything that isn't static text. Safe dynamic text is the output of one of exactly three Oracle-supplied functions: DBMS_Assert.Simple_Sql_Name(), DBMS_Assert.Enquote_Literal(), and To_Char(x f, n). ([page 35](#))

Use `Simple_Sql_Name()`, to ensure the safety of a SQL name. Use `Enquote_Literal()` to ensure the safety of a string or datetime literal. Use `To_Char(x f, n)` to ensure the safety of a numeric literal.

Keep the code that invokes `Simple_Sql_Name()`, `Enquote_Literal()`, or `To_Char(x f, n)` very close to the code that executes the SQL statement that these calls make safe.

Ensure the safety of the run-time-created SQL statement text in the same way no matter to which of the several Oracle-supplied APIs for executing dynamic SQL you submit it.

Don't use `DBMS_UTILITY.Exec_DDL_Statement()` in new code.

- ✓ *Rule_11*: The safe dynamic text produced by `DBMS_Assert.Enquote_Literal()` or by `To_Char(x f, n)` should be used only at a spot within the SQL statement where a SQL literal is intended; and the safe dynamic text produced by `DBMS_Assert.Simple_Sql_Name()` should be used only at a spot within the SQL statement where a simple SQL name is intended. ([page 35](#))

- ✓ *Rule_12*: Make the job of the auditor easy by establishing the safety of run-time-created SQL statement text in the code that immediately precedes the PL/SQL statement that executes it. ([page 37](#))

- ✓ *Rule_13*: Learn the full list of Oracle-supplied APIs that are designed to execute run-time-created SQL statement text. Ensure the safety such text by using exactly the same rules as you use to ensure the safety of text that is used with native dynamic SQL and the `DBMS_Sql` API. The approach is identical whether the text represents a complete SQL statement or a component of one. ([page 58](#))

- ✓ *Rule_14*: There is never a good reason to use `DBMS_UTILITY.Exec_DDL_Statement()` in new code. Avoid it. ([page 59](#))

APPENDIX D: ADDITIONAL ORACLE-SUPPLIED SUBPROGRAMS THAT IMPLEMENT DYNAMIC SQL

PL/SQL programmers are used to thinking that there are just two ways to execute a SQL statement that is presented as a PL/SQL *text* value: native dynamic SQL and the *DBMS_Sql* API. This is not the case. Others exist, and they are described in this appendix.

Some are designed to accept and execute a complete SQL statement but insist that it must be of a certain kind (for example, a DDL statement of any kind or a DDL statement to create a PL/SQL unit). Others are designed to accept a component of a SQL statement (for example, a *where clause*) and concatenate this with other components that they compose programatically to compose a SQL statement and then execute it.

All these APIs share the property that they were designed specifically on the assumption that it is the caller's responsibility to ensure immunity to SQL injection.

Rule_13

Ensure the safety of the run-time-created SQL statement text in the same way no matter to which of the several Oracle-supplied APIs for executing dynamic SQL you submit it.

Learn the full list of Oracle-supplied APIs that are designed to execute run-time-created SQL statement text. Ensure the safety such text by using exactly the same rules as you use to ensure the safety of text that is used with native dynamic SQL and the *DBMS_Sql* API. The approach is identical whether the text represents a complete SQL statement or a component of one.

DBMS_Utility.Exec_DDL_Statement()

Code_41 shows the important part of the implementation of *DBMS_Utility.Exec_DDL_Statement()*.

```
-- Code_41
Cur := DBMS_Sql.Open_Cursor();
DBMS_Sql.Parse(Cur, Stmt, DBMS_Sql.Native);
DBMS_Sql.Close_Cursor(Cur);
```

It turns out that when *Stmt* is a DDL statement⁹⁴, then this code is sufficient to execute it⁹⁵. But for other kinds of statement that are not supported by embedded SQL, like for, for example, *alter session*, it is necessary to call *DBMS_Sql.Execute()*. This fact has the consequence that *DBMS_Utility.Exec_DDL_Statement()* will quietly ignore anything but a DDL statement.

It's hard to imagine a Design Specification document that would prescribe this as the intended behavior. Therefore, in new code, you should use *execute immediate* to execute a DDL statement. And you should use native dynamic SQL or the *DBMS_Sql* API to execute other kinds of statement when embedded SQL cannot be used. The purpose of the calling code is known; and the SQL statement text will have been composed specifically for that purpose.

94. The list of the kinds of statement that are classified as DDL statements is given in the SQL Language Reference book.

95. This fact is documented in the PL/SQL Packages and Types Reference book.

Don't use
DBMS_UTILITY.Exec_DDL_Statement()
 in new code.

Rule_14

There is never a good reason to use *DBMS_UTILITY.Exec_DDL_Statement()* in new code. Avoid it.

DBMS_DDL.Create_Wrapped()

The purpose of this procedure is to create a PL/SQL unit and to store the code in the catalog using the obfuscated representation. The input is the plain text of the appropriate *create or replace* statement. If it was not required to obfuscate the source, you would simply use *execute immediate*.

Use cases do arise where it is appropriate generate a PL/SQL unit programmatically. One example is given by code that implements post-installation steps when ISV code is installed at a customer site. The logic of particular subprograms needs to reflect specific properties of the installation environment. Notice that *Rule_10* on [page 35](#) and *Rule_11* on [page 35](#) limit the freedom with which PL/SQL can be programmatically composed. The huge bulk of such source must be composed as static text; only identifiers and *text* literals within the source may derive from dynamic text. Huge caution must be used with respect to *text* literals⁹⁶, even when, as this paper requires, safe dynamic text is used. For example, it would clearly be unsafe to allow safe dynamic text as the argument for *execute immediate*. Probably the only sensible use of anything other than static text is for externally visible names, like the name of the unit itself or of the subprograms it exposes.

DBMS_HS_Passthrough

The pass-through facility provides a mechanism for developers to execute a SQL statement on a non-Oracle system without being interpreted by Oracle Database.

DBMS_HS_Passthrough.Execute_Immediate().

This function runs a SQL statement immediately. Any valid SQL statement except a *select* statement will be run immediately. Internally, the SQL statement is run using the *passthrough SQL* protocol sequence of *Open_Cursor()*, *Parse()*, *Execute_Non_Query()*, *Close_Cursor()*. The SQL statement cannot contain placeholders.

DBMS_HS_Passthrough.Parse()

This function parses the provided SQL statement at the non-Oracle system. The execution of the statement occurs at the subsequent *DBMS_HS_Passthrough.Execute_Non_Query()* or in the case of a *select* statement *DBMS_HS_Passthrough.Fetch()*. The SQL statement cannot contain placeholders.

OWA_Util

The *OWA_Util* package provides, among other things, the ability to generate HTML pages or page fragments using the results of a *select* statement presented as a PL/SQL *text* value.

96. While composing the source code of a PL/SQL unit using safe *numeric* and *datetime* literals isn't intrinsically risky, it's very hard to see how a sensible Design Specification document could call for this.

OWA_Util.Bind_Variables()

This function is intended to prepare a *select* statement for subsequent use by other *OWA_Util* subprograms. However, it is nothing other than a wrapper for calls to *DBMS_Sql.Open_Cursor()*, *DBMS_Sql.Parse()*, and a series of calls to *DBMS_Sql.Bind_Variable()*. It returns an ordinary *DBMS_Sql* numeric cursor.

OWA_Util.Bind_Variables() has 25 pairs of formal parameters with names like *bv17Name*, *bv17Value*, all with datatype *varchar2*, and with default value *null* so they can be omitted according to purpose. Each *not null* pair causes a corresponding call to *DBMS_Sql.Bind_Variable()*.

OWA_Util.ListPrint()

This procedure generates an HTML form user-interface element (a selection list) from the output of a *select* statement. It has two versions: the first takes a complete *select* statement as a PL/SQL *text* value; and the second takes a *DBMS_Sql* numeric cursor prepared, for example, by a preceding call to *OWA_Util.Bind_Variables()*.

OWA_Util.TablePrint()

This function generates an HTML fragment to represent the contents of a database table. It does not take full *select* statement; rather, the caller supplies the table's name, a comma separated list of the required columns, and specifies the *where clause* and *order by clause*. The *where clause* cannot contain placeholders.

APPENDIX E: SELF-CONTAINED CODE TO ILLUSTRATE IMPLEMENTING CALLBACK USING DYNAMIC POLYMORPHISM.

This code can be run as any ordinary user. To save space, it is written as if none of the objects it creates already exist.

To prove that the implementation that uses dynamic polymorphism gives the same results as the naïve implementation with *execute immediate*, the test harness — procedure *p()* at the end — uses each method in turn to compute a checksum over many invocations with different arguments. You can easily add some timing code — use *DBMS_UTILITY.GET_CPU_TIME()* — and greatly increase the range of input values over which the checksum is computed.

```
create function Callback(Input in integer) return integer is
begin
    return Input*Input;
end Callback;
/
```

```
create type Hook authid Current_User is object(
    Dummy number,
    not instantiable member function Callback(
        self in Hook,
        Input in integer ) return integer
    )
not final
not instantiable
/
```

```
create type My_Implementation under Hook(
    overriding member function Callback(
        self in My_Implementation,
        Input in integer) return integer
    )
/
```

```
create type body My_Implementation is
    overriding member function Callback(
        self in My_Implementation,
        Input in integer) return integer
    is
    begin
        return Input*Input;
    end Callback;
end;
/
```

```
create package Which_Callback is
    function The_Callback return Hook;
end Which_Callback;
/
```

```
create package body Which_Callback is
    -- Implements some logic to chose an instance of
    -- the appropriate subtype of Hook
    function The_Callback return Hook is
        Which constant Hook := My_Implementation();
    begin
        return Which;
    end;
end Which_Callback;
/
```

```

create procedure p(Name in varchar2, Obj in Hook) is
  No_Of_Repeats    constant pls_integer := 100;
  Checksum         integer;
  n                integer;
  x                integer;

  Safe_Name constant varchar2(32767) :=
    Sys.DBMS_Assert.Simple_Sql_Name(Name);
  Stmt constant varchar2(32767) :=
    'begin :x := '||Safe_Name||'(:n); end;';

  procedure Check_It is
  begin
    if(Checksum is null or Checksum <> 338350) then
      Raise Application_Error(-20000,
        'Callback: unexpected Checksum: ' || Checksum);
    end if;
  end Check_It;
begin
  Checksum := 0;
  n := 0;
  for r in 1..No_Of_Repeats loop
    n := n + 1;
    execute immediate Stmt using out x, in n;
    Checksum := Checksum + x;
  end loop;
  Check_It();

  Checksum := 0;
  n := 0;
  for r in 1..No_Of_Repeats loop
    n := n + 1;
    Checksum := Checksum + Obj.Callback(n);
  end loop;
  Check_It();
end p;
/

begin p('Callback', Which_Callback.The_Callback()); end;
/

```



How to write SQL injection-proof PL/SQL

October 2008

Mark Fallon, Software Development Director, Oracle Headquarters

Bryn Llewellyn, PL/SQL Product Manager, Oracle Headquarters

Howard Smith, Director, Global Product Security, Oracle UK

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Copyright © 2008, Oracle. All rights reserved.

This document is provided for information purposes only and the contents hereof are subject to change without notice.

This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle, JD Edwards, PeopleSoft, and Retek are registered trademarks