Basics of Oracle/PHP Bind Variables

WHAT is SQL Injection?

SQL injection is when someone tries to "inject" additional SQL clauses into a **dynamic** SQL statement -- one built at run time, especially one built based on user input.

You can try this out in `sqlplus`, with the SQL*Plus operator `&` you can use for interactive input in `sqlplus`:

```
SQL> select empl_last_name
     from empl
     where job_title = &job_ttl;

Enter value for job_ttl: 'Manager'

EMPL_LAST_NAME
---------------
Jones
Blake
Raimi
```

This is clearly intended for listing the names of employees with a particular job title.

But what if the user enters more than just a job title? For example: (recall that / re-runs the latest SQL statement):

```
SQL> /

Enter value for job_ttl: 'Manager' or 1=1
```

...now ALL of the employee's names are selected, because the injected **`or`** clause, with an always-true operand, makes the **`where`** clause always **`true`**, so that all the rows of the specified table are selected.

A **`union`** operator can open the door to obtaining even data about different columns, even different tables:

```
SQL> /

Enter value for job_ttl: 'Manager' union select table_name from
user_tables
```

...now ALL of the table names in the current database that are readable by this Oracle account are selected along with the names of employees with job title of `'Manager'`!

You can get names of columns of other tables, also -- as just one example:

```
SQL> /

Enter value for job_ttl: 'Manager' union select column_name from
user_tab_columns where table_name = 'DEPT'
```

That's from `sqlplus` -- but imagine a form that has a select/drop-down with job titles, and a PHP responding to that form that dynamically concatenates the job title the user selects to complete a `select` statement. If a rogue user builds their own form instead, that adds a **`union`** clause like one of those above, they could use SQL injection from that form, also.

One weapon against SQL injection is to use **OCI bind variables** *instead* of concatenation to complete a dynamic SQL select statement.

WHEN should you use bind variables?

You should use these **whenever possible** as an **alternative** to **concatenation** when building a dynamic SQL statement, one that is built by the PHP (as opposed to being hard-coded), ESPECIALLY when it is being built based on user-provided information.

(You will also use them for the arguments in PL/SQL stored procedures and functions.)

Note that they may **not** be used to replace ***column*** or ***table names***.

HOW do you write a bind variable in a PHP string containing a SQL statement?

You may have **multiple** bind variables in a PHP string containing a SQL statement.

You get to choose their names, making sure those names begin with a colon (**`:`**).

For example, this query string contains two bind variables, **`:chosen_dept`** and **`:chosen_mgr :`**

```
$empl_query_string = "select salary, hiredate
                      from   empl
                      where  dept_num = :chosen_dept
                             and mgr = :chosen_mgr";
```

Notice that no special quoting is necessary -- when the value is bound later, the system will quote the bound value as needed!

WHEN and HOW do you bind a value to a bind variable?

You need to do this:

- **AFTER** you create a statement object for this query using **`oci_parse`**,
- and **BEFORE** you execute that statement object using **`oci_execute`**.

You do it using an **oci_bind_by_name** statement for each bind variable.

When a bind variable is used for input purposes -- as is the case for a SQL statement -- **oci_bind_by_name** expects **THREE** arguments:

- the **statement object** for the statement containing bind variables
- the **bind variable** to have a value bound to it, **written in quotes**
- an expression giving the **desired value** to bind to that bind variable in the next execution of that statement object.

For example, if you have successfully connected to Oracle using **oci_connect**, and assigned the returned connection object to a variable named **$conn**, and you also have:

```
$desired_dept = strip_tags($_POST["dept_wanted"]);
$desired_mgr = strip_tags($_POST["mgr_chosen"]);

$empl_query_string = "select salary, hiredate
                      from   empl
                      where  dept_num = :chosen_dept
                             and mgr = :chosen_mgr";

$empl_query_stmt = oci_parse($conn, $empl_query_string);
```

...then you can bind values to that query's bind variables using:

```
oci_bind_by_name($empl_query_stmt, ":chosen_dept", $desired_dept);
oci_bind_by_name($empl_query_stmt, ":chosen_mgr", $desired_mgr);
```

And ***now*** you can execute the query using **oci_execute:**

```
oci_execute($empl_query_stmt, OCI_DEFAULT);
```

...and proceed as usual.

FUN FACT: if you'd like to run this query more than once with different values...

...just call **oci_bind_by_name** AGAIN with the next desired value for a bind variable, and then call **oci_execute** AGAIN -- you can reuse the statement, and it turns out this kind of reuse is quite efficient!

One final comment on advantages of bind variables

From https://www.php.net/oci_bind_by_name:

* "Binding is important for **Oracle database performance** and also as **a way to avoid SQL Injection** security issues.

* Binding allows the database to **reuse** the statement context and **caches** from previous executions of the statement, even if another user or process originally executed it.
* Binding **reduces SQL Injection concerns** because the data associated with a bind variable **is never treated as part of the SQL statement**. It **does not need quoting or escaping**.
* PHP variables that have been bound can be changed and the statement re-executed without needing to re-parse the statement or re-bind."