HANDS-ON PRACTICE 3
SECURING FIND MODE

Peter Koletzke, Quovera
Duncan Mills, Oracle Corp.

The practice is taken in part from the *Oracle JDeveloper 10g for Forms and PL/SQL Developers* (Oracle Press, McGraw-Hill/Osborne). The method we use in this example is the view object's getViewCriteriaClause( ) method. The ADF BC framework calls this method when constructing a SQL WHERE clause to incorporate the values entered by a user in a form in Find mode. We can override the method in our view object implementation class and add some validation code to reduce the risk of a user entering unintended criteria values.

In the code example in this hands-on practice, we implement the following simple rules that attempt to balance user functionality with safety:

- **Reject query criteria that contain the name of one of the attributes** the view object contains.
- **Reject query criteria that contain SQL operations** such as "BETWEEN," ">=," or "=" for columns that are not dates or numbers. These operators are allowed for dates and numbers because a user might want to use such comparisons in a legitimate way.

These simple rules will defeat the most obvious attacks. However, they may also give rise to false results, depending on your data set and database column names. Consider this and tailor the code to suit your needs and satisfy your security requirements and testing. In addition, consider your data when thinking about other possible SQL injection possibilities.

> **Note**
>
> The code listings are available in text files in the sample code for this practice.

getViewCriteriaClause( ) Method

Enter the following method in your `AllEmployeesImpl` view object class. As before, the line numbers are for reference in this text and should not be included in the code. This code is split by explanations, but you can use the line numbers to ensure that you code the entire method.

```
01: public String getViewCriteriaClause(boolean forQuery)
02: {
03:   ViewCriteria viewCriteria = getViewCriteria();
04:   if (viewCriteria != null)
05:   {
06:     AttributeDef[] attrs = viewCriteria.getViewObject().getAttributeDefs();
07:     StringBuffer columnNameListBuff = new StringBuffer();
08:     for (AttributeDef attr: attrs)
09:     {
10:      columnNameListBuff.append(attr.getColumnName().toUpperCase());
11:      columnNameListBuff.append("|");
12:     }
13:     String columnNameList =
14:         columnNameListBuff.substring(0, columnNameListBuff.length() - 1);
```

Check for Criteria

Line 01 defines the method for getViewCriteriaClause(). The base framework class, ViewObjectImpl, contains this method and performs default ADF BC behavior when Query-By-Example (Find mode) functionality is required. We override this method in the ADF BC base framework class and call the superclass' corresponding method at the end of our method. The base class requires a boolean argument, and we pass that value back to the framework class without changing it at the end of the method.

Line 03 obtains the ViewCriteria object from the view object. This object contains values for all Find mode fields the user entered.

Line 04 bypasses the rest of the code if no criteria exist.

Obtain a List of Column Names

Line 06 uses the getAttributeDefs() view criteria method to gather a list of column names used by the view object's attributes. We need this to implement the rule to reject any query criteria that contain one of these column names.

Lines 07–12 convert the array of AttributeDef objects returned by line 06 into a single long string (columnNameListBuff, defined in line 07) containing column names, with each column name separated by a "|" (pipe symbol). This long string of column names will be used later for the pattern matching that searches for possible SQL injections.

> **Note**
>
> We use a StringBuffer instead of a String for the column name list, because, reassigning a String uses more memory than appending to a StringBuffer.

Lines 13–14 strip the unwanted extra "|" from the end of the column name string.

```
15:     ViewCriteriaRow vcr = (ViewCriteriaRow)viewCriteria.first();
16:     while (vcr != null)
17:     {
18:       for (AttributeDef attr : attrs)
19:       {
20:         int index = attr.getIndex();
21:         String criteria = (String)vcr.getAttribute(index);
```

Process Criteria Rows and Attributes in Each Row

The ViewCriteria object contains a row for each set of criteria field values. Therefore, one row will contain values for the EmployeeID, FirstName, LastName, Email, PhoneNumber, DepartmentId, JobId, and other AllEmployees attributes. The Find mode capability of ADF BC allows the user to create multiple rows of criteria by invoking an insert action on the View Object while in Find mode. Each of the query attributes would appear in each row of the ViewCriteria object. Although we have not coded the page to allow for multiple criteria rows, the code in Lines 15–21 accounts for adding this capability in the future.

When the query criteria are processed, all attribute values entered for the first row are assembled into a WHERE clause condition with an AND operator between each field-value set. The values entered in the second row would be assembled in the same way into a second condition, which is then joined to the first row's condition using the OR operator.

Line 15 retrieves the first criteria row into the object "vcr."

Line 16 starts a loop through each criteria row. The page we created for searching employees only contains one row of query criteria but the loop is created in case other rows are added in the future.

Line 18 starts a loop through all attributes in the query criteria row. In a typical Query-By-Example screen, the user enters criteria in several fields, so the code needs to examine each attribute.

Lines 20–21 retrieve the index number associated with the attribute and use the index number to extract the criteria value entered by the user for that attribute. Each column or attribute can be identified by index number or column name; since the column name can vary, the index number is more reliable.

```
22:         if (criteria != null)
23:         {
24:           boolean restricted = true;
25:           switch (attr.getSQLType())
26:           {
27:             case (Types.INTEGER):
28:             case (Types.NUMERIC):
29:             case (Types.DECIMAL):
30:             case (Types.DATE):
```

```
31:             case (Types.TIMESTAMP):
32:             {
33:               restricted = false;
34:             }
35:             break;
36:           }  // end of switch
37:           String newCriteria =
38:                 detectInjection(criteria,columnNameList,restricted);
39:           vcr.setAttribute(index, newCriteria);
40:         }  // end of if (criteria != null)
41:       }  // end of for (AttributeDef attr: attrs)
42:       vcr = (ViewCriteriaRow)viewCriteria.next();
43:     }  // end of while (vcr != null)
44:   }  // end of if (viewCriteria != null)
45:   return super.getViewCriteriaClause(forQuery);
46: }  // end of method
```

Test the Criteria

Line 22 determines if something has been entered in this attribute and, if so, processing continues; otherwise, the code continues with Line 41, which retrieves the next criteria row and passes control back to the top of the attribute loop.

Lines 24–36 retrieve the SQL datatype of the attribute and decide how restrictive the check should be. The mode is initially restrictive, as defined by the boolean variable restricted, but if the datatype of the attribute turns out to be a number or a date, restricted is set to "false," because our design rule states that we will allow operators in these fields in case the user wants to search a range of numbers or dates (using BETWEEN, >, <, and so on).

Lines 37–38 call another method, detectInjection(), that checks for possible injection. This method returns a String into the newCriteria variable. If the criteria entered by the user is deemed to be safe, the criteria is returned unchanged. If the criteria is suspect, then "null" will be returned instead; this code effectively removes the criteria from the WHERE clause. (See the sidebar "Why Suppress SQL Injection Errors?" for the reason.) The variable, restricted, is passed into this method (explained later).

Line 39 resets the criteria to the value passed back from detectInjection(). After all attributes are processed in this way, the attribute loop is exhausted and control passes to Line 42.

Line 42 retrieves the next criteria row. The query criteria row loop and nested attribute loop continues until no view criteria rows remain.

Line 45 calls back to the superclass to perform its normal processing using the now-corrected set of query criteria.

> **Note**
>
> You will need to import oracle.jbo.ViewCriteria, oracle.jbo .AttributeDef, oracle.jbo.ViewCriteriaRow, and java.sql.Types.

The detectInjection( ) Filter Method

The getViewCriteria() method is only half of the story. We also need a method that performs the hard work—detectInjection(). This method uses regular expressions to look for suspicious patterns in the supplied criteria. Regular expression is a powerful string manipulation feature that is available in most programming languages, including native PL/SQL (with Oracle 10*g*) and the Oracle Application Server PL/SQL Web Toolkit package OWA_PATTERN. A regular expression allows the code to check for matches against multiple patterns in one pass, which removes the need to use multiple conditions with different patterns.

The code for detectInjection() follows, with line numbers that would not appear in the code. Add this method to the view object Impl class, AllEmployeesImpl.

> **Note**
>
> Both methods would be reasonable candidates for refactoring to the framework buffer class.

```
01: protected String detectInjection(String criteria,
02:                                  String columnNames,
03:                                  boolean restrictive)
04: {
05:   boolean reject = false;
06:   String testPattern;
07:   if (restrictive)
08:   {
09:     testPattern = "^(>=|<=|<|>|<>|!=|=|BETWEEN|IN|LIKE|IS)";
10:   }
11:   else
12:   {
13:     StringBuffer constructLooseTestPattern =
14:                          new StringBuffer("(^(=|!=|LIKE))|(");
15:     constructLooseTestPattern.append(columnNames);
16:     constructLooseTestPattern.append(")");
17:     testPattern = constructLooseTestPattern.toString();
18:   }
19:   String testCriteria = criteria.trim().toUpperCase();
20:   if (testCriteria != null && testCriteria.length() > 0)
21:   {
22:     Pattern sqliPattern = Pattern.compile(testPattern);
23:     Matcher matcher = sqliPattern.matcher(testCriteria);
24:     if (matcher.find())
25:     {
26:       reject = true;
27:     }
28:   }
29:   return reject?null:criteria;
30: }
```

Lines 01 and 30 define the method block. Lines 01–03 declare the method signature with arguments of the criteria string to be tested, the list of column names for this view object, and the boolean flag indicating if the check needs to be more or less restrictive.

Lines 05–06 set up variables for the rejection flag and test pattern string.

Lines 07–18 create the appropriate regular expression-matching pattern based on whether the check is restrictive.

Line 09 loads the testPattern variable with operators that are not allowed for criteria values that are not number or date datatypes.

Lines 13–16 set up a StringBuffer, constructLooseTestPattern, containing some operators and the names of the columns represented by the view object.

Line 17 assigns the StringBuffer to the testPattern variable so it can be used in later code.

Line 19 removes any trailing and leading white spaces from the criteria string and converts the value to uppercase for ease of comparison.

Line 20 determines if there is anything to check now that white space has been removed. If so, it processes the values in Lines 22–27.

Line 22 invokes Java's regular expression engine to compile the expression pattern object, testPattern, just constructed.

Line 23 performs the pattern match between the compiled test pattern and the criteria that the user entered.

Lines 24–27 check if the criteria matched the filter pattern; if so, the reject flag is set to "true."

Line 29 returns "null," if the regular expression test found a matching string, or the original criteria string, if the expression test did not find a match.

> **Note**
>
> As always, you will need to import classes required for this code, such as java.util.regex.Pattern and java.util.regex.Matcher.

You can now test the code using the searchEmployee.jsp file. First try a SQL injection violation. Comment out the new getViewCriteriaClause() method before running the application (press CTRL-/ after selecting the method text). Run the application and, in the Search page, enter the following in the First Name field:

```
=employees.first_name and employees.salary > 10000
```

You will see a subset of the records—just those employees with a salary over 10,000. This is a violation of the intended design, which was to not allow users to determine the employee earnings. Even though the salary amount is not displayed, you could repeat the queries to hone in on the exact amount for a particular employee.

Close the browser and stop the server. Then uncomment the method in the view object code (pressing Ctrl-Z should reverse the comments) and re-run the application. Try that search clause again to see the effect of the filter. The filter will ignore the query value because it contains an attribute name (Salary), so you should see all rows. With this filter, the confidentiality of employees' salaries is secure. Close the browser and stop the server.