AUTOMATIC DETECTION OF VULNERABILITY IN JAVA PACKAGES IN ORACLE

By Ivan Kyselgof
Advisor: Amichai Shulman
Sep. 2008
There are several types of vulnerabilities:

- SQL-injection
- Buffer overflow
- Unauthorized Access
- Denial of Service
- Cursor injection
SQL injection: what is it?

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. It is in fact an instance of a more general class of vulnerabilities that can occur whenever one programming or scripting language is embedded inside another. (en.wikipedia.org)
More than 400 SQL-injection vulnerabilities were officially declared in 2003-2007.

It's easy to exploit (there is many hacker tools which allows to user automatically search for SQL-injection vulnerability in given Oracle application without understand how does it works).

Till now are available plural not eliminated vulnerabilities.

In order to complete task of previous project.
Called “Automatic detection of vulnerability in wrapped packages in Oracle” deals with analyzing code of Oracle SQL packages in order to find SQL-injection cases.

Consist of 2 parts:
- Unwrapper
- Analyzer
Before I start work on my project I try to analyze the previous work in order to understand if we on right direction.

I take known SQL-injection vulnerable package DBMS_METADATA which was declared as vulnerable on 18 Apr 2005 and fixed on April's 2005 patch from clear Oracle 10.0g, unwrap it and analyze it using the tool.
Previous project (Cont.)

I obtain the following result:

```
Line 823 (procedure DO_CALLOUT): Vulnerability found - opening cursor with unsafe parameter QUERY
Line 2548 (procedure DO_OPEN): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter STMT
Line 2554 (procedure DO_OPEN): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter STMT
Line 2765 (procedure SET_FILTER): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter STMT
Line 2831 (procedure SET_FILTER): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter STMT
Line 2892 (procedure SET_FILTER): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter STMT
Line 2953 (procedure SET_COUNT): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter STMT
Line 3004 (procedure SET_XMLFORMAT): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter STMT
Line 3116 (procedure SET_PARSE_ITEM): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter STMT
Line 3201 (procedure ADD_TRANSFORM): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter STMT
Line 3283 (procedure SET_TRANSFORM_PARAM): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter STMT
Line 3355 (procedure SET_TRANSFORM_PARAM): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter STMT
Line 3421 (procedure SET_TRANSFORM_PARAM): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter STMT
Line 3490 (procedure SET_REMAP_PARAM): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter STMT
Line 5472 (procedure GET_PREPOST_TABLE_ACT): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter STMT
```

i.e. it succeed to find 15 SQL-injection cases
Previous project (Cont.)

After this I update my Oracle and analyze it again. Now the tool find only 2 vulnerabilities

it's mean that it works!!!
My work

Since it works on SQL packages I decide complete SQL-injection analyzing and build analyzer for procedures inside Oracle Java packages. The reasons is:

- It will help to build in future tool for check all Oracle function for vulnerabilities to SQL-injection.
- There are many SQL-injection vulnerabilities in Oracle Java packages.
How it works?

1. Find JAR file containing some package
2. Decompile file to obtain java plain text
3. Process java file for find available vulnerabilities
How it works?(Cont.)

First we need to decompile JAR file to obtain Java plain text.

In my work I use Jad (http://www.kpdus.com/jad.html) decompiler, but it’s not really matter what decompiler you use since it possible to convert JAR file into Java plain text.

After we have java files we can test them with tool.
How it works?(Cont.)

My tool can work in 2 modes:

- **Search mode** – given java file searches for all appearances of possible SQL-injection and builds tree of function calls (in order to serve check mode).

- **Check mode** – given function calls tree and specific function name checks if specified function arguments may be source of SQL-injection.
Tools I use in my project

I use some helpful tools during writing my project:

- Jad - the fast Java Decompiler
- AntLR parser generator
- Java.g – java language definitions for AntLR by Terence Parr
- Eclipse - an open development platform
- JFrameBuilder – tool for fast GUI building
Let's begin!

To make using process more comfortable I build small GUI. Main window looks like this:
Example

For example I take SubscriptionHandle.java from CDC package:
Example (Cont.)

After applying my tool on it I receive 4 alerts:

Input:
Input path of java file to proceed and then choose function name from list:
C:\Project Tests\SubscriptionHandle.jad

Output:
In changeSetAdvEnabled:
In line 108: Unsafe call: (OracleResultSet)stmt.executeQuery(sqltext)
In createSubscription:
In line 138: Unsafe call: stmt.executeQuery(sqltext)
In querySubscriptionHandle:
In line 199: Unsafe call: (OracleResultSet)stmt.executeQuery(sqltext)
In validateChangeSet:
In line 258: Unsafe call: (OracleResultSet)stmt.executeQuery(sqltext)
Examination the function source code give us the following:

```java
public static NUMBER querySubscriptionHandle(String subscriptionName)
    throws SQLException
{
    Connection pconn = DefaultContext.getDefaultContext().getConnection();
    Statement stmt = pconn.createStatement();
    OracleResultSet orset = null;
    NUMBER subscriptionHandle = NUMBER.ZERO();
    String unquoted_subscriptionName;
    if (subscriptionName.startsWith("\\"))
        unquoted_subscriptionName = subscriptionName.substring("\\".length(), subscriptionName.length() - "\\".length());
    else
        unquoted_subscriptionName = subscriptionName.toUpperCase();
    try
    {
        String sqltext = "SELECT HANDLE FROM SYS.CDC_SUBSCRIBERS$ WHERE SUBSCRIPTION_NAME = '" + unquoted_subscriptionName + '"';
        orset = (OracleResultSet) stmt.executeQuery(sqltext);
    }
```

i.e. execution performed with String derived from user String. The other 3 alerts is same.
Next we need to test the second part. After performing first part function list becomes available:

I insert functions to the list with their formal parameters because we can have 2 functions with the same name and different formal parameters one of them is safe and another one – no.
Now I check one of the functions in the list:

It's safe as expected. When we look in the body of the function we see that it calls to unsafe function `createSubscription()`, but no formal parameters is transferred, that's why `createSubscriptionHandle()` is safe.
myUnsafe() will be unsafe because it calls to vulnerable function querySubscriptionHandle() directly with user input, on another hand mySafe() will be safe one because it calls to myUnsafe() with constant input, making this call safe.
Example (Cont.)

Let's see if it really works:

**Input:**
Input path of java file to proceed and then choose function name from list:

C:\Project Tests\SubscriptionHandle.jad  
Browse  Proceed

myUnsafe(String s)

**Output:**
Call to myUnsafe(String s): Unsafe
Example (Cont.)

**Input:**

Input path of java file to proceed and then choose function name from list:

- C:\Project Tests\SubscriptionHandle.jad

**Output:**

Call to mySafe(String s): Safe
As we see the tool is really works, but it is still have things to do:

- Check if vulnerable code is reachable with potentially dangerous input.
- Build more general tool that can trace dependences on whole-package level and not only on single-file level.
- Buffer overflow checking tool will be pretty to add to.