AUTOMATIC DETECTION OF VULNERABILITIES IN WRAPPED PACKAGES IN ORACLE

Submitters:

Yaron Gur-Arieh
Nikita Zubrilov
Ilya Kolchinsky

Introduction

I - Short introduction to the problem our project deals with

"One of the key threats to the security of Oracle database servers are bugs in the default PL/SQL packages, triggers, and types that are shipped with the database." (taken from "The Oracle Hackers Handbook/by David Litchfield").
The bugs Litchfield mentioned are indeed lead to a major security concern in Oracle, they make Oracle susceptible to SQL injection, which can allow a low-privilege user to gain full control over the database with DBA privileges, meaning the database is now completely compromised.
The problem of this kind of vulnerability is not merely that it “built into” Oracle, but that it is often very hard to detect and neutralize vulnerabilities, such as with buffer overflows and unsafe parameter handling by Execute Immediate.
(NOTE: we will properly explain vulnerabilities later, in the next chapter in "II- Different Vulnerabilities Detection Approaches").
The problem is even more complicated because packages can be wrapped, in order to properly understand wrapped packages, consider the following scenario:
There is a new tax reform that has come into effect, and the information on a database needs to be updated to properly account for the current monetary situation, in the case of an accounting firm this is a crucial undertaking that is likely to be very time consuming and must be completed as quickly as possible.
Luckily a programming company that specializes in Oracle applications has already created a program that can automatically update the information on the database, so that it complies with the new tax reform. This programming company sells their product to the accounting firm, and thus solve the accounting firm's problem.

So where does wrapping packages fit in this scenario?
Wrapping a package causes the code to be obfuscated, that is to say made unreadable by programmers, so that the code cannot be readily stolen, or looked through by hackers for weaknesses to exploit, thus protecting the intellectual property of the programming company.
However wrapped packages can still be used by others such as the accounting firm, despite the code being unreadable, so the programming company has guarded their secrets and still enabled the accounting firm to use their product.

Why does this make the problem more complicated?
It is important to understand that wrapping code is a superficial action, it only obfuscates the code, and does not alter the code in any fundamental way.
This means that if the unwrapped code is vulnerable to certain attacks, the wrapped code is just as vulnerable to the exact same attacks that the unwrapped code was vulnerable to.
So while wrapping packages protects the programming company, it presents a problem to the accounting firm, because there is a product they want to use, but they have no easy way of verifying if it is safe for them to use this product because the code is wrapped.
In short, wrapping packages obfuscates the package's code, thus enabling companies to sell products without having their code stolen.

However, wrapped code is vulnerable to the exact same exploits that unwrapped code is vulnerable too. Thus anyone who uses a wrapped package made by another company may be putting their databases at risk, and they cannot easily verify if the code is safe to use.

II - Our Objective

The objective of our project was to research and create:
A Tool(under the following conditions)
-Which enables people to be certain that their databases are secure, even when using wrapped code provided by other people
-That can help people be sure that their code is safe to be used by others
-Which lets people stay safe even when new exploits are discovered
-That provides the industry with an automated way to test their code, granting them a tool to help safeguard themselves and others

Historical Background and Detection Approaches

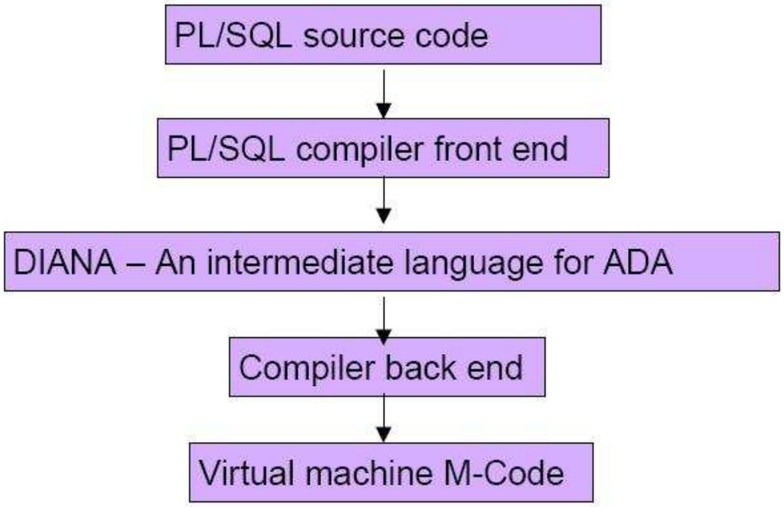
I – unwrapping techniques

In order to achieve our goal, first we needed to unwrap Oracle packages somehow. Thus, we had to build(or find) a unwrapper. In this chapter we'll discuss two main unwrapping techniques that can be used in our purposes.
We read many internet material concerning Oracle packages unwrapping and found Pete Finnigan and David Litchfield to be the most reliable authors who work on this subject.

1. Pete Finnigan way (understanding source code by reading byte-code):

9i and lower versions of Oracle:

PL/SQL language compilation structure


PL/SQL source code
PL/SQL compiler front end
DIANA – An intermediate language for ADA
Compiler back end
Virtual machine M-Code

Before 9i:
In old versions of Oracle we can almost easily deduce the original source code of PL/SQL package from the wrapped code produced. The symbol table is visible, so we can understand the purpose of the procedures (because we see variable names in symbol table), we can find out encryption algorithms that used. We also can modify the source code working straight with the wrapped code, thus we can plant trojans, make SQL injections etc…

9i:
Almost the same information of the source code we can get in 9i analyzing DIANA code.

10g:
Finnigan claims that unwrapping is almost the same as for 9i version, though some it is more difficult now because new wrap mechanism is provided(we'll

discuss it later), the symbol table is no longer visible, used base64 encryption, but IDL$ (DIANA) tables still contain DIANA m-code.

2. **David Litchfield (unwrap, then work with plain PL/SQL text):**

Versions 9i and earlier:
Litchfield uses the same technique as Finnigan.

10g:
In 10g the clear-text PL/SQL is encrypted in the following way. The text is first compressed using the Lempel-Ziv algorithm, and a SHA1 hash is generated from the compressed data. This hash is then copied to a buffer and the compressed data concatenated to the end. Then the value of each byte in the buffer is used as an index into a character substitution table. This table is probably considered an Oracle trade secret, so it is hidden somewhere. Litchfield claims that it is easy to find in the binary. The resulting cipher text is then base64 encoded. Unwrapping the encrypted code is the reverse. First, it is base64 decoded. Then, each byte is resubstituted with a second corresponding substitution table. Last, the text is decompressed, leaving the clear text of the PL/SQL.

II – Different Vulnerabilities Detection Approaches

This chapter is dedicated to SQL injections, types of injections, ways to defend your code from injections, injections specific to Oracle, solutions available in the web (commercial or free).

1. **Potential vulnerabilities(SQL injections in Oracle)**

- SQL manipulation (adding OR,AND,MINUS,UNION,.. to an existing statement to change the set of results)
- Code Injection (INSERTs,UPDATEs and DELETEs...)
- invocation of a malicious function defined by an attacker with privileges of the writer of an attacked function
- invocation of an existing function vulnerable to SQL injection and defined using AUTHID CURRENT_USER
- attacker defines a function with the PRAGMA AUTONOMOUS_TRANSACTION compiler directive and AUTHID CURRENT_USER keyword that executes SQL statements the attacker wants with elevated privileges. Then he injects this function using a known SQL injection. This may allow attacker to become DBA.
- stored procedure that is defined without AUTHID CURRENT_USER will execute with privileges of the owner
- anonymous PL/SQL block (a PL/SQL block that has a BEGIN and an END that can be used to execute multiple SQL statements). There is no

limitation in what the attacker can do. Allows to execute SELECTs, DML and DDL

- Single PL/SQL statement: Doesn’t have a BEGIN and an END.
  The attacker cannot insert “;” to inject more SQL commands.
- exploiting a buffer overflow

2. Solution approach

Let’s discuss the practices to avoid SQL injection, so the way to detect the existing vulnerabilities will be to point the places, where these practices are violated. This apparently will bring us to detect some false vulnerabilities, so we’ll need more complex techniques to reduce the number of false detections. We’ll talk about “smart” implementations later.

Practices against SQL injections:

- using bind variables - the best known solution. Will also improve application performance. Cannot be used for table names or column names. Cannot be used for the procedure or function name.
- avoiding dynamic SQL statements.
  if absolutely necessary, use bind variables.
- Defining functions using AUTHID CURRENT_USER. Vulnerability in such function cannot be exploited directly, however it can be used to exploit another vulnerability in function defined without AUTHID CURRENT_USER.
- avoiding concatenation in SQL statements.
  If necessary then check the input for malicious code. Using numeric values in concatenation. This way strings cannot be passed in to add SQL
- validating input strings very carefully
- check for special characters such as quote symbols, semicolons,'--', stored words like 'union', 'or', 'delete', words equal to fields of a table currently in use etc.

3. Solutions available

a. Commercial solutions:
   There are no real commercial solutions that specifically detect SQL injection attempts against an Oracle database. There are a reasonable number of firewall products that incorporate an Oracle proxy and a few IDS tools that claim to support Oracle. A number of companies are at present seriously looking into the design and development of a true application IDS for Oracle, and perhaps these tools will detect SQL injection. At present most of the commercial tools to be used properly would need rules and signatures to be defined for the specific Oracle cases.

b. Free solutions:
   There are a number of free SQL injection detection tools in the web, but one can’t prove that they do their work, moreover, most of such a tools try

to solve common problems, and do not focus specifically on Oracle. So, most likely they miss or overestimate great percent of the vulnerabilities.

c. Known solutions list(this just the top of the list):

Tool name	Free/Commercial	Oracle specific
sqlmap	Free	No
SQL Injection Brute-forcer	Free	No
SQLBrute	Free	No
BobCat	Free	No
NGSS SQL Injector	Free	No
SQLiX	Free	No
SQL Power Injector	Free	No
WebCohort SecureSphere	Commercial	Yes
appDetective	Commercial	No
OScanner	Free	Yes

4. **Companies that deal with database security**
   - Microsoft with Columbia University– with ShieldGen, a system for automatically generating a data patch or a vulnerability signature for an unknown vulnerability, given a zero-day attack instance.
   - Acunetix - Web Vulnerability Scanner
   - Imperva - SecureSphere
   - Elanize KG - Maui Security Scanner
   - N-Stalker - Web Application Security Scanner
   - …

5. **Academy efforts**

   We searched for the existing solutions or at least academic researches on the theme of automatic detection or just any kind of detection of SQL injections and buffer overflows, but surprisingly(or not) didn't find anything worthwhile, only heuristics and tips for implementation. Thus, we can deduce, that our work is a kind of novelty, and we are among the first people who try to solve the problem of automatic detection of database vulnerabilities (mostly SQL injections) inside academy.

III – What we have done to achieve the goal

1. **Possible solution implementations comparison**

   As we saw earlier we have two main approaches to automatic detection of database vulnerabilities. The first one is deducing package source code from the byte-code, the second is firstly unwrapping Oracle packages and then finding vulnerabilities in plain PL/SQL text. In this chapter we'll discuss advantages and disadvantages of both methods.

- **Byte-code approach** (by Pete Finnigan)

  Positive aspects:
  This approach can help to understand Oracle underlining flaws, than can't be seen watching a source code. This means flaws in translation into intermediate language(DIANA) etc… It can be used in the same way independently of Oracle specific version, because we just need to look at the byte-code and try to understand things from it (this does not mean, that byte-code of each version looks alike).
  With this approach we can detect buffer overflows.

  Negative aspects:
  Does not guarantee to that it is really understandable, looks different in different Oracle versions. Does not point to mistakes it does not understand, but we know about, i.e. we must search for each vulnerability inside byte-code, even if we know the exact vulnerable PL/SQL function, procedure, block or statement. Thus, the method is hard to extend (knowing a new vulnerability does not help us with byte-code).

- **Plain text approach** (PL/SQL analysis)

  Positive aspects:
  The work can be divided into two parts: 1. unwrapping, 2. PL/SQL analysis. That means, we can choose any unwrapping function independently, so the method becomes easily adoptable to different kinds of architectures. PL/SQL part is extendable, because each new known vulnerability can be easily understood from PL/SQL code, what means, that we can just add new rule for the PL/SQL parser. Marks vulnerabilities in the source code, enabling easy correction. More programmer-friendly – no need to dig deep into byte-code, the only work is with the plain PL/SQL text.

  Negative aspects:
  The unwrapping part is quite nontrivial. Though David Litchfield claims, that it is not hard to find symbol table inside binaries, it is not easy work to do, the search inside the binaries may be very complicated, plus we cannot exactly know where to search, we only can have assumptions. In common, the unwrapping is Oracle version-dependant, so for each Oracle version we need to build a new unwrapper. The method can cause many "false alarms", i.e. there can be very complicated cases, where, for example, some variable received from user is bubbled through many procedures and functions, so it is difficult to know is this variable properly checked. (We'll talk about this subject later in implementation section).
  Limited buffer overflow detection.

2. <u>Why did we chose "plain text" approach</u>

We had several reasons to choose this approach. First of all, we had better documentation here, i.e. hundreds of articles on PL/SQL in web, David Litchfield book vs. roughly speaking, one article of Pete Finnigan about DIANA, which was written not for version 10g, but mainly for 9i. Second reason is that it is easier to see work results and verify correctness of our efforts, because PL/SQL is more readable than byte-code and also we can know what exactly we want to achieve, we can sort vulnerabilities simply by PL/SQL grammar and Oracle known flaws.

IV – resources we needed for our work

1. The resources

- The Oracle unwrapper – in order to work on detecting vulnerabilities in PL/SQL code we needed the source code of Oracle packages. And to get the source code we needed to unwrap the existing Oracle packages.
- PL/SQL language specification – without knowing PL/SQL deeply it is impossible to understand the problems arisen and also impossible to plan the ways to overcome the problems.
- Known vulnerabilities types - i.e. to build the detection tool we needed to know how the vulnerabilities look, what are the possible attacks, what are the possible solutions, how we can produce a secure PL/SQL code. Knowing common attack types is also can be listed in this section, because only after understanding the attack, one can understand the way to defend against it.
- Bison parser – to build parser to Pl/SQL

2. Problems we faced in implementation

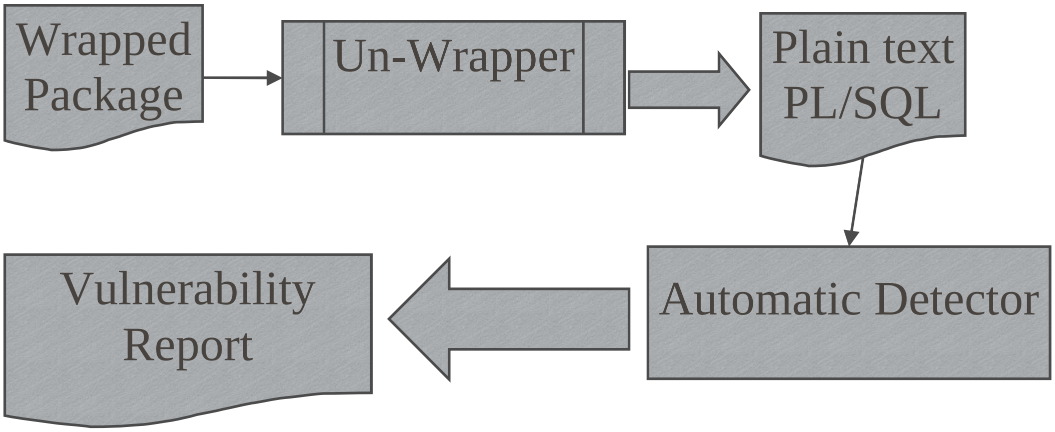
- Unwrapping Oracle packages: we tried to act according David Litchfield book “The Oracle Hacker’s Handbook”, i.e. we created simple packages, applied wrap utility of Oracle’s, base64 decoded the wrapped code, and then the troubles came: for a long time we couldn’t find the substitution table – it turned up, that the search in binaries is not the easiest mission (especially, when you don’t know what exactly you are searching for). The technique we used to search is: we took wrap.exe program of Oracle’s and tried to find something resembling to a symbol table inside the DLLs that are used by this application, because it sounds logically, that wrapper program will sometime use the substitution table. But this way didn’t bring any results.
  So it was nothing to talk about the last step – decompression to get clear text of the PL/SQL.
  So, we started to think about alternative ways to solve the problem of unwrapping, simple Google search didn’t returned positive results, as can be understood, but when we made search in Russian net,

surprisingly, we found a file called unwrap10.exe, with which help we finally succeeded to unwrap Oracle packages.
All antiviruses claim that the file contains a Trojan inside, we don’t know, maybe ☺

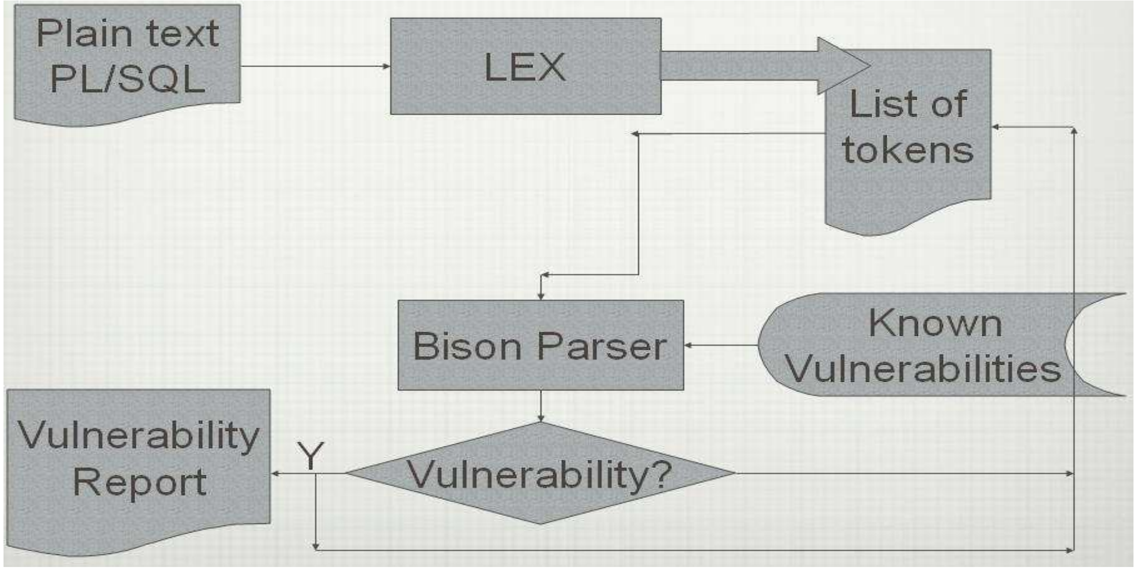
Our Implementation

I - How our way works

In general, we built PL/SQL parser that detects all types of Oracle vulnerabilities we know. The way of its work can be best described by the following drawing:



In our case the wrapped package is Oracle package with .plb extension, unwrapper – unwrap10.exe we found in the web.
Next we’ll talk about our automatic detector, the way it works and reports it produces.
The automatic detector can be described by this drawing:



II – Automatic detector

1. Common features

Given a PL\SQL source file as an input, our automatic detector tries to locate suspicious commands. Those having a non-zero probability of being vulnerable to SQL injection attack as described earlier (in section II of the previous chapter).
The detector performs only those operations or calculations which are really needed, i.e. nothing is done while facing non-vulnerable command.
The automatic detector can be easily extended. Only basic knowledge of Lex/Bison is required in order to add new rules for vulnerability detection.
Additional advantage is that, the detector silently ignores the unidentified code, so it is fault tolerant and will never fail in the middle (always completes vulnerabilities search).

2. Vulnerabilities detection

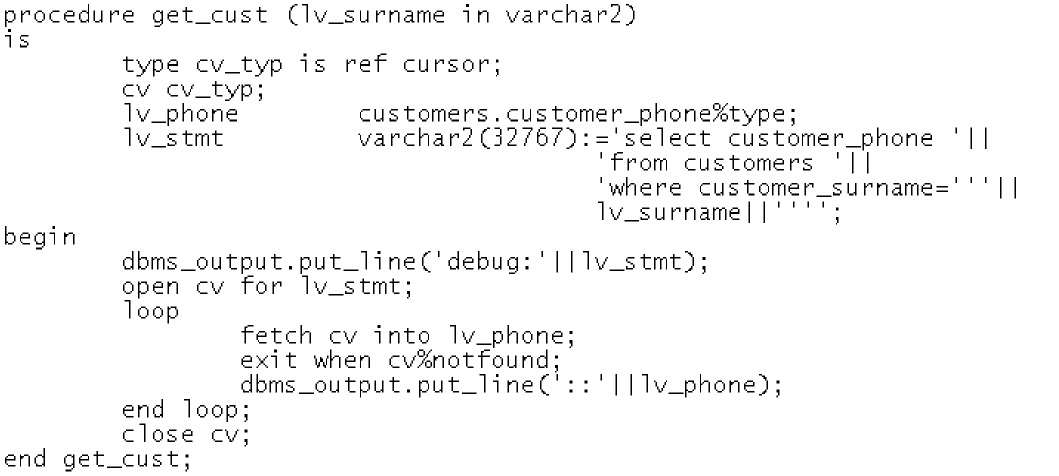
- Maximize the number of real vulnerabilities detected
    - The automatic detector recognizes all types of mechanisms for dynamic SQL execution, that are EXECUTE IMMEDIATE, dynamic cursors, DBMS_SQL package procedures and functions.
    - The detector makes no assumption about the content of function parameters. It treats each variable as a potential source of danger. For each suspicious input variable, calculates the set of internal variables whose values are influenced by this one, this way enabling detection of malicious input propagation.
    - Treats AUTHID CURRENT_USER as carefully as AUTHID DEFINER.
      We know that the real problem is the functions that were created with AUTHID DEFINER, because function with vulnerability defined for example by DBA, makes an attacker DBA.
      Though, a vulnerability in a function defined as AUTHID CURRENT_USER cannot be exploited directly, researches show that it can be used for attack on another function(we talked about this kind of vulnerabilities in section II of the previous chapter).

- Minimize the number of “false alarms”
    - The automatic detector recognizes the usage of bind variables that are considered to be the best currently known solution to the problem of SQL injections, so it doesn’t produce a warning, when encountering bind variables.
    - Our detector recognizes the usage of DBMS_ASSERT functions. DBMS_ASSERT is Oracle library that provides an interface for input strings validation. This is very reliable library, so when a string was checked by a function of this library, it won’t cause warning.

- The automatic detector performs basic data flow analysis to determine whether a given variable is dangerous in a given program point, i.e. if a formerly malicious string was substituted by harmless content is considered safe.
- The detector checks only input variables of vulnerable types, i.e. strings. For example it is OK to concatenate an input numeric value to a dynamic SQL statement even without knowing its exact value.

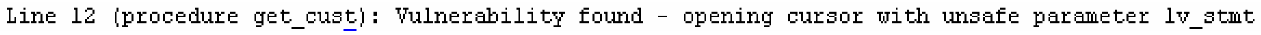
II – Examples

- **A simple example(**taken from "SQL injection and Oracle" by Pete Finnigan**)**



```
procedure get_cust (lv_surname in varchar2)
is
        type cv_typ is ref cursor;
        cv cv_typ;
        lv_phone        customers.customer_phone%type;
        lv_stmt         varchar2(32767):='select customer_phone '||
                                        'from customers '||
                                        'where customer_surname='''||
                                        lv_surname||'''';
begin
        dbms_output.put_line('debug:'||lv_stmt);
        open cv for lv_stmt;
        loop
                fetch cv into lv_phone;
                exit when cv%notfound;
                dbms_output.put_line('::'||lv_phone);
        end loop;
        close cv;
end get_cust;
```

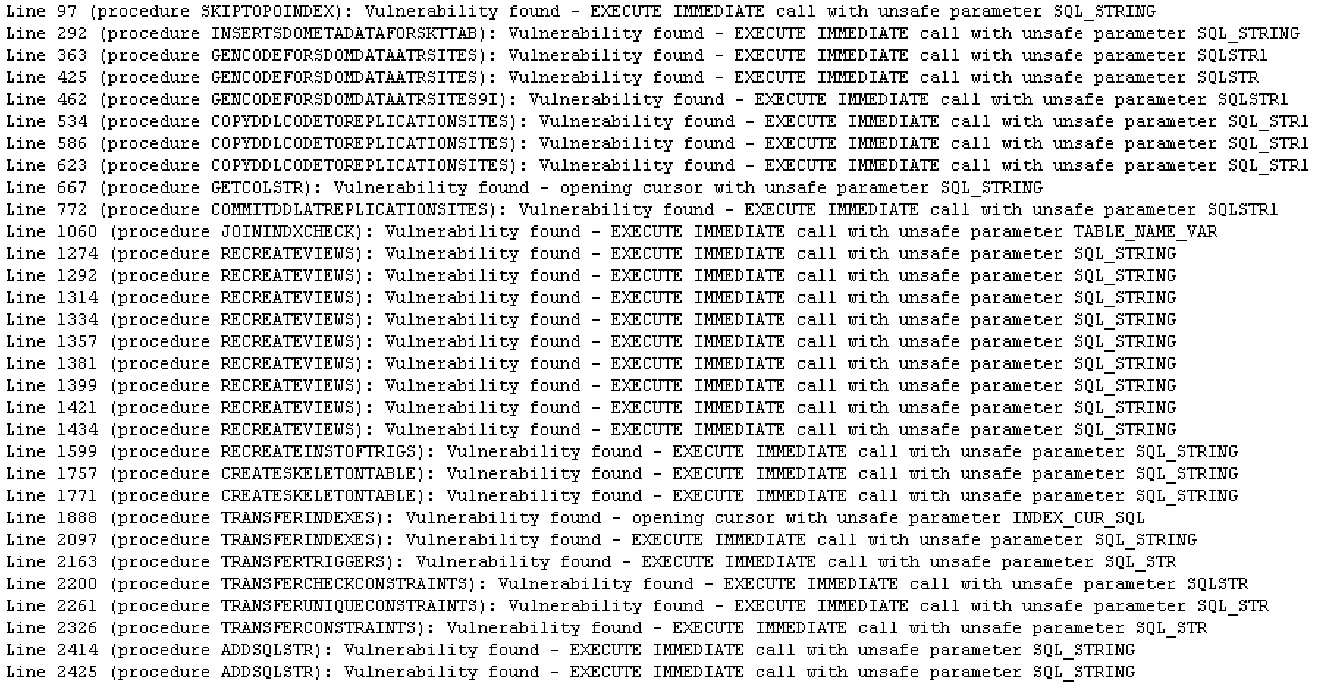
An output of our detector for this input is:



```
Line 12 (procedure get_cust): Vulnerability found - opening cursor with unsafe parameter lv_stmt
```

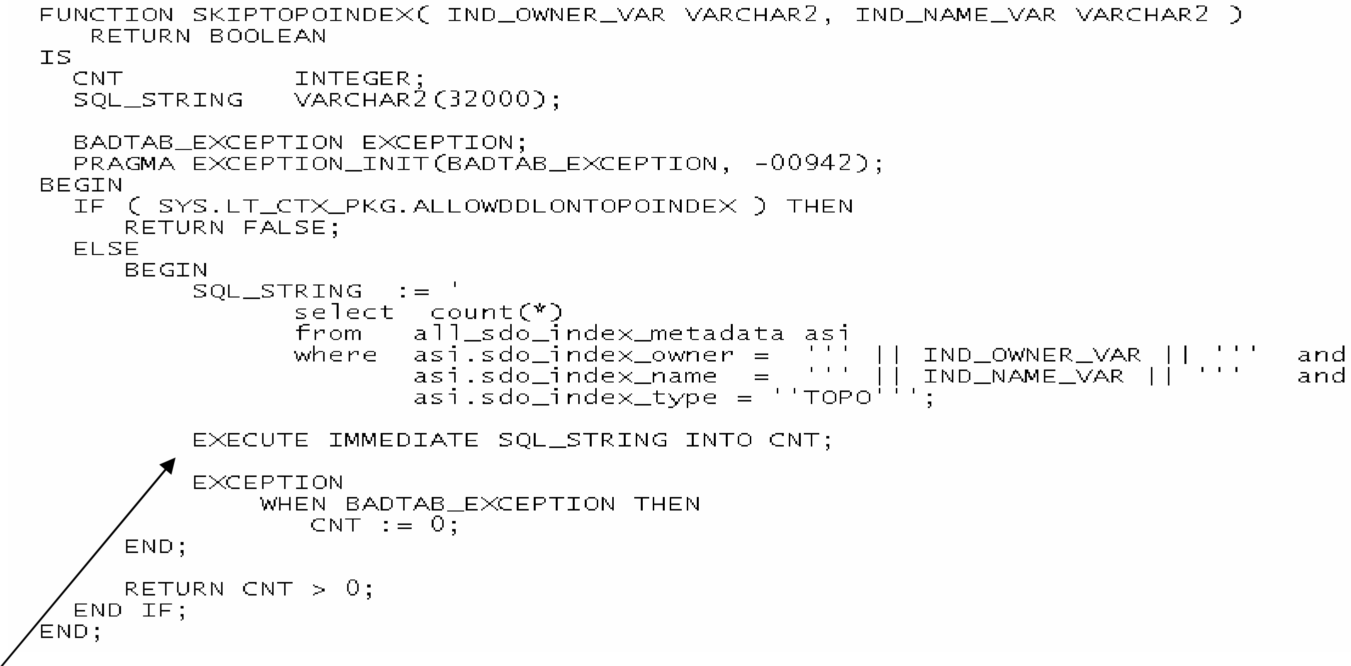
And we can easily be convinced, that lv_stmt is indeed unsafe variable (so it is forbidden to open cursor with it), because it is a product of concatenation of a constant string and the string lv_surname, received from the user and never checked for safety.

- **A more complicated example -** a package **owm_ddl_pkg**
  Size of an unwrapped package body is over 252 KB (6320 lines of code). A part of detector's output(report) for this package:



```
Line 97 (procedure SKIPTOPOINDEX): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
Line 292 (procedure INSERTSDOMETADATAFORSKTTAB): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
Line 363 (procedure GENCODEFORSDOMDATAATRSITES): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQLSTR1
Line 425 (procedure GENCODEFORSDOMDATAATRSITES): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQLSTR
Line 462 (procedure GENCODEFORSDOMDATAATRSITES9I): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQLSTR1
Line 534 (procedure COPYDDLCODETOREPLICATIONSITES): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STR1
Line 586 (procedure COPYDDLCODETOREPLICATIONSITES): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STR1
Line 623 (procedure COPYDDLCODETOREPLICATIONSITES): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STR1
Line 667 (procedure GETCOLSTR): Vulnerability found - opening cursor with unsafe parameter SQL_STRING
Line 772 (procedure COMMITDDLATREPLICATIONSITES): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQLSTR1
Line 1060 (procedure JOININDXCHECK): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter TABLE_NAME_VAR
Line 1274 (procedure RECREATEVIEWS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
Line 1292 (procedure RECREATEVIEWS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
Line 1314 (procedure RECREATEVIEWS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
Line 1334 (procedure RECREATEVIEWS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
Line 1357 (procedure RECREATEVIEWS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
Line 1381 (procedure RECREATEVIEWS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
Line 1399 (procedure RECREATEVIEWS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
Line 1421 (procedure RECREATEVIEWS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
Line 1434 (procedure RECREATEVIEWS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
Line 1599 (procedure RECREATEINSTOFTRIGS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
Line 1757 (procedure CREATESKELETONTABLE): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
Line 1771 (procedure CREATESKELETONTABLE): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
Line 1888 (procedure TRANSFERINDEXES): Vulnerability found - opening cursor with unsafe parameter INDEX_CUR_SQL
Line 2097 (procedure TRANSFERINDEXES): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
Line 2163 (procedure TRANSFERTRIGGERS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STR
Line 2200 (procedure TRANSFERCHECKCONSTRAINTS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQLSTR
Line 2261 (procedure TRANSFERUNIQUECONSTRAINTS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STR
Line 2326 (procedure TRANSFERCONSTRAINTS): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STR
Line 2414 (procedure ADDSQLSTR): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
Line 2425 (procedure ADDSQLSTR): Vulnerability found - EXECUTE IMMEDIATE call with unsafe parameter SQL_STRING
```

We went over all this list of vulnerabilities inside the package code, here is an example of the vulnerability from the line 97 (first in list):



```
FUNCTION SKIPTOPOINDEX( IND_OWNER_VAR VARCHAR2, IND_NAME_VAR VARCHAR2 )
   RETURN BOOLEAN
IS
  CNT            INTEGER;
  SQL_STRING     VARCHAR2(32000);

  BADTAB_EXCEPTION EXCEPTION;
  PRAGMA EXCEPTION_INIT(BADTAB_EXCEPTION, -00942);
BEGIN
  IF ( SYS.LT_CTX_PKG.ALLOWDDLONTOPOINDEX ) THEN
     RETURN FALSE;
  ELSE
     BEGIN
         SQL_STRING  := '
                select  count(*)
                from   all_sdo_index_metadata asi
                where  asi.sdo_index_owner =  ''' || IND_OWNER_VAR || '''  and
                       asi.sdo_index_name  =  ''' || IND_NAME_VAR || '''   and
                       asi.sdo_index_type = ''TOPO''';

         EXECUTE IMMEDIATE SQL_STRING INTO CNT;

         EXCEPTION
             WHEN BADTAB_EXCEPTION THEN
                 CNT := 0;
     END;

     RETURN CNT > 0;
  END IF;
END;
```

This is the line 97 of the package. As we can see, execution of SQL_STRING is indeed a vulnerability, because, one of SQL_STRING components is, for example, user-string IND_OWNER_CHAR that is never checked.

Future work

There is still much work left to be done. The possible extensions of the application can be the following:

- More advanced data flow analysis to detect rare complicated cases.
- Defining several classes of risks, i.e. not all vulnerabilities are of the same level of potential risk, some of vulnerabilities are harder to exploit, etc…
- Providing more information about a found vulnerability, for instance, which input variable was unchecked, what is an exact path from function entry point to the vulnerable statement (may be unreachable) etc…
- Providing quick tips for a user describing what changes should be applied to the code in order to eliminate the vulnerability found.
- At a later stage it is possible to automatically replace vulnerable code with safe one (only in limited cases!).
- Buffer overflow analysis – start with known vulnerabilities: 3 Oracle database functions – bfilename, TZ_OFFSET, TO_TIMESTAMP_TZ that reside in STANDARD database package, and step by step build a database of all functions that make usage of these known vulnerable functions, thus vulnerable to buffer overflow too.

Our Work References And Sources

- unwrap10.exe : un-wrapper utility by mysterious Russian hacker
- David Litchfield – Wiley's "The Oracle Hacker's Handbook" and "The Database Hacker's Handbook"
- http://www.ngssoftware.com/papers/DBMS_ASSERT.pdf
- Pete Finnigan – "How to unwrap PL/SQL", **"**SQL injection and Oracle**"**
- Esteban Martinez Fayo – "Advanced SQL injection in Oracle databases"
- http://www.securityfocus.com/infocus/1714
- http://www.security-hacks.com/2007/05/18/top-15-free-sql-injection-scanners
- Various PL/SQL internet books, tutorials, etc..