Towards Study of Taint Analysis for PL/SQL Stored Procedures

Shraddha Makwana
Computing Science
University of Alberta
Edmonton, Alberta, Canada
smakwana@ualberta.ca

Abstract

Procedural language like PL/SQL provides powerful capability to databases for performing high computations tasks using stored procedures. Performing static analysis on such PL/SQL stored procedures hence is needed to present developers with alerts by providing possible vulnerable taints in the system. Taint introduced in stored procedures can be a potential threat for the data residing, and hence it is important to discover such possible attacks beforehand. In this paper, I describe in general the approach of carrying out taint analysis, a type of data-flow analysis, on stored procedures by representing them into Intermediate Representation, and extending it to perform analysis for detecting possible data leaks for the system. To provide more accurate taints it performs more stringent sanitization checks. The work done is publically available at https://github.com/shr1911/program-analysis-PLSQL-CMPUT-500

Keywords: PL/SQL, Taint Analysis, Database Static Analysis, Data-flow Analysis, Work-list Algorithm, sanitization, DBMS_ASSERT, Control Flow Graph

1 Introduction

Performing Static Analysis on a given programming language gives the developer a capability of detecting the software vulnerabilities well in advance during their software development life cycle. Despite the fact that program analysis has been evolved for many high-level programming languages, there is still a need to improve static analysis for database programming languages. Such languages enable programmers to implement the decision-making calculus inside the stored procedures on the database server rather than keeping them on the application server. The idea of developing a database-driven application, comes from the need of reducing the time to send requests back-and-forth between application servers and database servers over the network. Hence, for such heavy computation which contains the core of the software demands stringent vulnerability checks for any possible attacks on stored procedures.

In this paper, I present a study of taint analysis on one such database language called PL/SQL (Procedural Language extensions to the Structured Query Language). Here, taint analysis provides an untrusted possible source method for a given piece of stored procedure as an input, along with the sink method where sensitive data can be leaked. The analysis here is performed considering a broader level of sanitization and asserts provided by PL/SQL library package, and hence reducing the false taint alerts for the given stored procedures.

To illustrate this study, I am considering the PL/SQL stored procedure program to depict it into the Intermediate Representation. This task has earlier been implemented by static analysis tools like SonarSource [6] and PMD [10], which remained my base to implement this step. After that, in order to extend this study towards performing taint analysis, I will traverse it's Intermediate Representation using a work-list algorithm for tracking taint propagation throughout.

This study contributes in the following terms:

- It shows the number of taints present in database code with correctly identified source and sinks methods to identify potentially vulnerable flow in database applications
- It demonstrates broader range of sanitization checks, which in turn tries to reduces the false positive taint detection
- The analysis helps preventing various attacks on PL/SQL stored procedure like SQL Injection, Denial-of-Service attack beforehand to developers
- It also evaluates the time efficiency to perform taint analysis based on various factors like number of taint present and line of code (LOC)

2 Background

While we perform taint detection, which is information-flow analysis, it is important to include various sanitization checks so that we don't detect the taint incorrectly. PL/SQL provides multiple ways of sanitization for it's stored procedure, like using bind variables and by using DBMS_ASSERT library package [5]. Bind variables helps stored procedure protection against SQL Injection type of attacks. Here, the content of the query remains unchanged, however the bind variable values will get changed depending on what we pass.

The DBMS_ASSERT package was introduced in Oracle as an critical patch update[2]. This package provides an

UofA'21, December 2021, Edmonton, AB, Canada
2021.

Figure 1. Real-world example

interface to validate properties of the input value by the range of multiple functions that can be used to sanitize user input. This helps procedures to protect against SQL injection in applications that don't use bind variables. The study performed in this paper focuses on sanitizations that used DBMS_ASSERT for their program rather than bind variables.

Following are some of the assertion supported by DBMS _ASSERT library which the following approach will consider:

SIMPLE_SQL_NAME checks if the input string conforms a simple SQL name.

QUALIFIED_SQL_NAME asserts the input conforms to the basic characteristics of a qualified SQL name. It can be made up of several simple SQL names representing the names of the schema, object and database links.

SCHEMA_NAME checks if the input string represents an existing schema name.

SQL_OBJECT_NAME checks if the input string represents an existing object.

ENQUOTE_NAME function encloses the input string within double quotes.

ENQUOTE_LITERAL function encloses the input string within single quotes, and checks that all other single quotes are present in adjacent pairs.

Some of the above types of sanitization will be evaluated and discussed in the later part of this paper, where I will demonstrate how identifying such taints can help us to protect from attacks like SQL-injection and Denial-of-Service attacks.

3 Motivating Example

The major motivation of performing taint analysis on PL/SQL Stored Procedures comes from one of the real-world experiences, wherein one of the financial banks had SQL injection attack on PL/SQL code. As shown in Figure 1, Due to this, the wrong decision was made to incorrectly to reveal non-public information to networks (like MasterCard, Visa) interchangeably. PL/SQL by oracle, which is a procedural language for SQL, plays a key role to write such mission-critical applications on Oracle Databases [8], and it needs stringent vulnerability checks.

Above mentioned SQL injection type attack on PL/SQL is illustrated as shown in Figure 2 code snippet. Here, as shown at line 1, every parameter coming with type *IN* will be considered as tainted since it is an entry point for stored

```
1  CREATE PROCEDURE vulnerable(pname IN VARCHAR2, vresult OUT VARCHAR2) AS
2      vsql VARCHAR2(4000);
3      test NUMBER(100);
4      vname VARCHAR2(1000);
5  BEGIN
6      test := 20;
7      IF (test > 50) THEN
8          vname := 'John';
9      ELSIF (test > 30) THEN
10         vname := 'Christina';
11     ELSE
12         vname := pname;
13     END IF;
14     vsql := 'SELECT description FROM products WHERE name=''' || vname || '''';
15     EXECUTE IMMEDIATE vsql INTO vresult;
16 END;
```

Figure 2. Code example for PL/SQL taints

procedure. In this example, *pname* is considered to be tainted as it can be potential vulnerable entry point from where attacker can inject malicious input. As we see, on line 12 and 14, *pname* is propagated to *vname* and then to *vsql* respectively. In turn, on line 15 the taint is sinked when Execute Immediate Statement is getting executed without necessary Assertion. Hence, in such scenario SQL injection attack can happen on PL/SQL stored procedure.

4 Overview of Approach

Taint Analysis upon PL/SQL Stored Procedure in this study, aims to check possible information flow paths in PL/SQL stored procedure executions, and correctly detecting the one which can be potentially influenced by an outside input. Alongside, approach here also provides broader range of sanitization checks, and hence also reducing the number of false positive taints in certain scenario. I now provide an overview of the approach before going further into the internal details of the same. The workflow in this approach consist of two consecutive phases: Control Flow Graph construction phase (CFG Phase) and Taint Analysis phase (TA Phase). Both of these phases gets executed sequentially upon the input PL/SQL stored procedure provided to it. Figure 3 demonstrates the same.

4.1 CFG Phase: Control-Flow Graph Phase

In order to implement the methodology of Taint Analysis over stored procedure, I try to represent input stored procedures into Control Flow Graphs (CFG). This phase has been performed using SonarSource [1] and PMD [10] as the baseline work, as they have implemented this step using directed graph which represents various path that a code can traverse during its execution. However, it lacks performing further stringent level of data-flow analysis on the same. During the implementation I've modified the structure of their work in certain aspects to also identify start of the stored procedure with CREATE statement where we might get input tainted argument. This approach uses an adjacency-list representation of the CFG which is suitable for such kind of sparse category of graph representation [3]. Each CFG vertex is called a node means a single statement in stored procedure and it contains information such as details of their neighbours, type of statements and node number (corresponding line

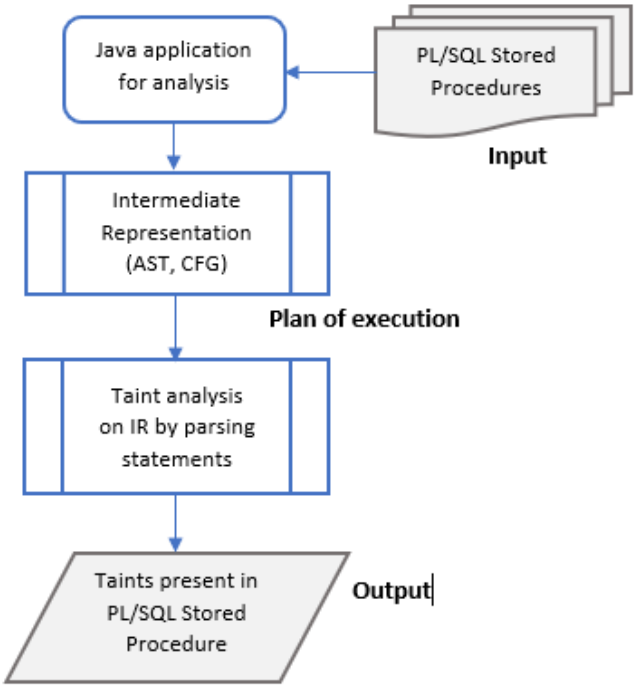

Figure 3. Overview Diagram

number in code). In order to identify the statement type this phase reuses already implemented PL/SQL pattern matching logic by above tools.

Once, the CFG is constructed it is stored in .dot file format which can be visualized using tools supporting this format like Gephi.

4.2 TA Phase: Taint Analyser Phase

This phase focuses on defining flow functions for taint analysis, which determines how the input code handles flow of data values during a program execution. These flow functions are usually to generate and kill facts in terms of taint as we traverse through CFG. Rules for the same are defined as follows:

• Source rules define statement where tainted data enter the system. This location can be the parameter or function arguments with type IN (means input arguments)

• Sink rules define statement that receives this tainted data and it ultimately leaks such information (for example, EXECUTE IMMEDIATE statement).

• Propagation rules define statement where tainted data is propagated further. In this case they are variable assignments. Here, taints are generated.

• Sanitization rules removes the taints if there is sanitization performed using DBMS_ASSERT.

The task of a taint analyser hence is to traverse through CFG by visiting each statement and applying above rules on the current node. Here, analyser implementation is done using a work-list algorithm [4] which follows the order of data flow as it's value changes over each statements. The analyser keeps the context of tainted variables during the graph traversal. This is represented as a map of tainted variables. Flow functions above trigger set operations and thus modify the data.

The analysis direction here will be forward and variables will be considered as data-flow facts. The flow functions for our taint analysis can kill or generate facts as follows:

Generating fact flow function: Taint facts are generated in 2 ways during start of the control-flow graph, when we get tainted input parameter at entry points. We also generate taints when we use assignment operator to propagate them towards the output.

Killing fact flow function: Whenever we see any DBMS_ASSERT sanitization the the taint is killed and hence we no more propagate the taint to it's output.

Hence, in this flow we determine possible paths which an attacker can misuse is called taint propagation.

5 Evaluation Results

I evaluate the methodology by performing experiments to address following three research questions:

1. Is the proposed method correctly identifying all taints present in terms of source and sink methods?
2. Does the tool check various types of sanitization supported by PL/SQL?
3. How efficient is the system in terms of time depending on the number of line of codes (LOC)?

Each of the above evaluation questions are discussed as follows along with their results and example.

5.1 Is the proposed method correctly identifying total taints in terms of source and sink methods?

The study reports number of taints present in the input stored procedure, and provides exact source and sink methods along with the information about that statement. Evaluation for this questions was performed with various mocked sample PL/SQL stored procedure with varying number of input and execute immediate statements. Number of taints gradually increases from one, two and then three to check if it provides correct taint results or not. Following is the results for the same. As we see, the results correctly identifies them along with the given time taken for them.

Num. of taints	Time for CFG	time for taint check	total time
one	29	7	36%
two	39	16	55%
three	169	26	195%

Table 1. Evaluating multiple number of taints (Times in milliseconds)

5.2 Does the tool check various types of sanitization supported by PL/SQL?

Here, the taint is only detected if there is no sanitization performed as per the DBMS_ASSERT subprogram library package. This was illustrated by various types of potential attack that can occur in PL/SQL stored procedure like SQL Injection and Denial-of-Service attack.

For example, SQL injection can be used to retrieve additional data when used against badly written dynamic SQL. Using DBMS_ASSERT.ENQUOTE_LITERAL function, we can protect this procedure as shown in Figure 4.

Also, some special types of SQL Injection attacks can be used in parallel with other techniques to achieve a specific goal which can cause failure to the system and can bring the whole system down, as an example Denial Of Service (DOS) attack. A malicious dynamic SQL can be used to perform such type of attack. Considering a function to keep track of the number of rows in a given table. Here, as shown in Figure 5, we are expected to send name of the table as an argument. This can be SQL injected to create a Cartesian join against the other table. Here, the impact of passing such argument which contains multiple table name can bring the whole system down. Consequently, this will add significant load to the system, and which in turn will lead to Denial-of Service attack. Hence, taint analyser should check sanitization like SQL_OBJECT_NAME procedure to verify if the value is a valid object name or not.

Therefore, referring to above given two example, the study here demonstrates the broader level of sanitization checks using DBMS_ASSERT utility.

5.3 How efficient is the system in terms of time depending on the number of line of codes (LOC)?

Finally, to answer question 3, I try to measure time elapsed to identify the taints for various sample stored procedure based on increasing number of lines of code.

Example 1	CFG time	Taint check	Total Time
1st run	33	5	38
2nd run	29	8	37
3rd run	30	6	36
4th run	32	5	37
5th run	29	6	35
6th run	29	6	35
7th run	28	6	34
Avg. time	30	6	36

Table 2. Evaluating on sample example with LOC 16

The Table 2 shown below is the time elapsed to calculate the taint for smaller stored procedure with approx. 16 LOC. Whereas, Table 3 shown below provides the time taken for taint analysis on larger stored procedure with approx. more than 100 LOC. Here, the same example has ran multiple times in order to get the precise time efficiency.

Example 1	CFG time	Taint check	Total Time
1st run	90	11	101
2nd run	109	10	119
3rd run	97	11	108
4th run	93	9	102
5th run	98	11	109
6th run	97	6	103
7th run	93	11	104
Avg. time	96.714	9.857	106.571

Table 3. Evaluating on sample example with LOC 101

6 Related work

SonarQube [6] which performs static analysis for over 25 languages. SonarSource, which is associated with SonarQube, measures conformity of Stored Procedures along with data flow analysis for PL/SQL. It has 188 rules for static analysis. It also gives recommendation for security hot spot like dynamic query generation for Stored Procedures of PL/SQL, which uses statements like EXECUTE IMMEDIATE. However, they check sanitization using bind variables and gives compliant solution if binding is missing, hence this can be still extended to improve more types of sanitizations provided by DBMS Assert package and giving exact number of taints in terms of source and sink. Since, sonar source does not make a sanitization check for DBMS_ASSERT it might identify Execute Immediate statement as a security hot spot which would be False Positive. Sonar source hence remained my baseline, for generating control flow graph and providing taint check considering sanitization level using the library package.

Another research done for PL/SQL Advisor [9] presents a static analysis-based tool, which automatically detects potential efficiency and code quality improvements on stored procedures written in PL/SQL. This tool tries to represent PL/SQL Stored Procedure into Control-Dependency Tree instead of Control Flow Graph, as it provides a hierarchical representation of the source code using the definition of control dependency between the statements of the source code, which they in turn use to provide analysis summary. Their evaluation was to measure run time, which was below 7 seconds average, along with the suitable warnings.

SQL Enlight tool [7], statistically analyses queries and identifies common mistakes. It also has a rule designer to implement custom analysis rules. SQL Enlight is developed for T-SQL and Microsoft SQL Server, along with that it also supports integration with SQL Server Management Studio and Visual Studio as an extension.

```
1  CREATE PROCEDURE vulnerable(pname IN VARCHAR2, vresult OUT VARCHAR2) AS
2      vsql VARCHAR2(4000);
3      test NUMBER(100);
4      vname VARCHAR2(1000);
5  BEGIN
6      test := 20;
7      IF (test > 50) THEN
8          vname := 'John';
9      ELSIF (test > 30) THEN
10         vname := 'Christina';
11     ELSE
12         vname := pname;
13     END IF;
14     vsql := 'SELECT description FROM products WHERE
       name=DBMS_ASSERT.ENQUOTE_LITERAL(''' || vname || ''')';
15     EXECUTE IMMEDIATE vsql INTO vresult;
16 END;
```

Figure 4. SQL-Injection type attack example

```
1  CREATE PROCEDURE get_tab_row_count(p_table_name IN VARCHAR2) AS
2    l_sql      VARCHAR2(32767);
3    l_count    NUMBER;
4  BEGIN
5    l_sql := 'SELECT COUNT(*) INTO :l_count FROM ' ||
6             DBMS_ASSERT.SQL_OBJECT_NAME(p_table_name);
7
8    EXECUTE IMMEDIATE l_sql INTO l_count;
9
10   DBMS_OUTPUT.put_line('l_count = ' || l_count);
11 END;
12 /
```

Figure 5. Denial-of-Service Ttype of attack

PMD Source Code Analyzer [10] is another open-source project which checks code conformity by using Abstract Syntax Parser on top of PL/SQL codebase. Also, they used continuous evaluation processes over the years to major trends of their product with the factor of analysis warning density change per year.

Overall, the work done for Procedural SQL program analysis majorly lacks providing exact number of taints in terms of source and sink along with wider range of sanitization and assertion checks, and hardly few have open-sourced the information about internal design of their tools.

7 Conclusion

In this paper, I presented the study upon taint analyses of PL/SQL stored procedures, which can help to maintain database-driven application which might deal with millions of transactions in a day. Hence, it demands of determining correct number of source and sink for the given input stored procedures.

I've used Sonar Source and PMD tool as an inspiration to create the architecture baseline of CFG creation, and after that I am using work-list algorithm to parse each statement and evaluated based on the various types of possible attacks which might occur for the PL/SQL stored procedures.

The future scope of this work is to improve upon parsing using more efficient algorithms. More types of taints

apart from SQL Injection can be implemented. Furthermore, scope of studying program analysis on PL/SQL is vast for topics like Control Graphs or IFDS, as its research is still at a premature level for PL/SQL. Along with that nested procedures, loop constructs are some of the limitations of the given study and this work can also be extended for other PL/SQL optimizations where we use voluminous merge and union query.

References

[1] 2008-2021. SonarSource tool for PL/SQL. (2008-2021). https://www.sonarsource.com/plsql/

[2] 2016. Database PL/SQL Packages and Types Reference. (2016). https://docs.oracle.com/database/121/ARPLS/d_assert.htm#ARPLS231

[3] Leiserson Charles E. Rivest Ronald L. Stein C. Cormen, Thomas H. 2009. Introduction to algorithms. Cambridge, Massachusetts. *MIT Press, ISBN 02-625-3305-7* (2009).

[4] Sanyal A. Karkare B. Khedker, Uday P. 2009. Data Flow Analysis: Theory and Practice. *Boca Raton: ISBN 978-0-8493-2880-0.* (2009).

[5] Karunya Rathan K.Priya, A.Sivasangari. 2017. International Journal of Pure and Applied Mathematics, Vol. 117. 161–165.

[6] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. 2019. Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In *2019 IEEE.*

[7] Biruk Muse, Mohammad Masudur Rahman, Csaba Nagy, Anthony Cleve, Foutse Khomh, and Giuliano Antoniol. 2020. On the Prevalence, Impact, and Evolution of SQL Code Smells in Data-Intensive Systems.

[8] Csaba Nagy, Rudolf Ferenc, and Tibor Bakota. 2011. A true story of refactoring a large Oracle PL/SQL banking system.

[9] Dimas Nascimento, Carlos Pires, and Tiago Massoni. 2013. PL/SQL Advisor: a Static Analysis-based Tool to Suggest Improvements for Stored Procedures. In *Anais do IX Simpósio Brasileiro de Sistemas de Informação.*

[10] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. 2020. A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in Apache open source projects. *Empirical Software Engineering* 25, 6 (Sep 2020), 5137–5192.