LETTER

Detection of SQL Injection Vulnerability in Embedded SQL

Young-Su JANG(a), Member

SUMMARY Embedded SQL inserts SQL statements into the host programming language and executes them at program run time. SQL injection is a known attack technique; however, detection techniques are not introduced in embedded SQL. This paper introduces a technique based on candidate code generation that can detect SQL injection vulnerability in the C/C++ host programming language.

key words: SQL injection vulnerability, embedded SQL, candidate code generation

1. Introduction

According to the Open Web Application Security Project (OWASP) Top Ten 2017, SQL injection is one of the major types of attack[1]. The causes of SQL injection vulnerabilities are well-known. One problem is the insufficient validation of input sources. Without validating the input source correctly, an attacker could insert a new SQL keyword through a specially crafted input string to change the developer’s intended SQL query. Due to the fact that the input from potential malicious users can be presumed to be arbitrary values, the program should properly validate this input. Therefore, the application should be able to check the input values that violate the specification.

Embedded SQL is used in the host programming language (e.g., C/C++, Java) where execution statements are used. In particular, embedded SQL in the C/C++ host programming language uses host variables (e.g., input or output variables) to pass data and status information. Therefore, when an attacker injects malicious SQL statements into the input host variables, they might be able to access the data of the database or extract private information. Sanitization is a type of input validation method that is performed to remove malicious elements from the input provided by the user. In addition, sanitization is performed before external input parameters are used in SQL statements.

[2]–[5] and [6] specify a regular expression or a built-in validation function in input sources. These studies presented pattern-filtering information (or, equivalently, regular expressions) that can detect a malicious input string pattern that may cause vulnerability. However, the proposed approaches only detect or prevent some SQL injection vulnerabilities [7], [8].

In this paper, we propose a novel approach to be able to defend against SQL injection vulnerability in embedded SQL. We expanded the candidate code generation technique introduced in [9] with the proposed technique. The candidate variables are transformed to verify SQL Injection Attack (SQLIA) in embedded SQL. Our approach is based on source code transformation to extract subtree matching (See Sect. 2.3). The verify_process() compares the real variables and candidate variables and verifies variable derivation tree for sanitization (See Sect. 2.4). If the derivation tree structure of the real variable and candidate variable is different, we can see that a SQLIA attack has occurred. Figure 1 shows an example of the SQL command text values and Fig. 2 shows an example of a ruleset for detecting SQLIA in SQL statement. The paper makes the following contributions:

(1) An expanded candidate code generation technique that describes the sanitization process by investigating how embedded SQL reacts.
(2) Attempts to verify the accuracy of the sanitization process and identify SQL injection vulnerabilities.

This work advances the detection of SQL injection vulnerabilities because this task can represent detailed system be-

---

Manuscript received August 1, 2019.
Manuscript revised November 29, 2019.
Manuscript publicized February 13, 2020.

†The author is with Korea Polytechnics, Anseong, Korea.
a) E-mail: jyskkh@naver.com
DOI: 10.1587/transinf.2019EDL8143

Copyright © 2020 The Institute of Electronics, Information and Communication Engineers
2. Host Variable Sanitization

2.1 Typical Example

We illustrate the SQLIA detection process by considering the sample program given in Fig. 3. This is a simple DELETE statement that removes rows from a books table. In lines 4 and 5, the dynamic SQL statement contains the placeholders \( v1 \) and \( v2 \) (host variables of book_title and book_author, respectively) as substitutes. In lines 9 and 11, the \( \text{gets()} \) function accepts input strings. In lines 12 and 13, the SQL statement is EXECUTEd, and \( \text{input host variables} \) in the USING clause replace corresponding placeholders in the PREPAREd dynamic SQL statement.

2.2 SQL Injection Definition

Let \( P \) be a program with input values \( I = \{I_1, \ldots, I_m, I_{s1}, \ldots, I_{sj}\} \). An input valuation is a \( v \) that contains each \( I_m \) to some integer and each \( I_{sj} \) to some string. For any input valuation \( v \), the program \( P \) takes a unique control path \( \text{Exec}_v \) [10]. We consider only the finite state.

Let us define two queries, \( q \) and \( q' \). If the query structures of \( q \) and \( q' \) are equivalent, the derivation trees of \( q \) and \( q' \) are isotypic. Therefore, if we know another input valuation \( v' \) that exercised the same query structure as \( v \), then we can evaluate the query structures of \( v \) and \( v' \) and check whether they are the same.

However, the problem of finding a corresponding valid valuation for every input valuation \( v \) that exercises the same path as \( v \) is a challenging problem. Consequently, we define \( \text{Fval}(v) \) [9], [10] as the valuation \( v_\pi \) that maps every string variable. We note that \( v_\pi \) is a benign and safe candidate variable corresponding to \( v \). The candidate variable generation is carried out in \( \text{VDLs} \) tool [9], in which the data type is specified to correspond to a specific C/C++ data type. In addition, if \( v \) is initialized by the user input or program, \( v_\pi \) is also initialized to a benign applicant value that has the same length as \( v \). Our definition of SQL injection is based on the following rules:

**Rule 1**: The query structure of input valuation \( v \) is determined by the input values.

**Rule 2**: If the input valuation generates a different query structure than that in the same control path, the input valuation has vulnerabilities.

2.3 Source Code Transformation

The natural approach to verify embedded SQL is transforming the source code [11]. This is accomplished in prototypes based on exact subtree matching using the Code Transformation Tool (CTT) [12].

Figure 4 shows the transformed example program presented in Fig. 3. The program contains the strings

| Line | Code |
|------|------|
| 1.   | ...  |
| 2.   | char delete_stmt[100], book_title[25], book_author[10]; |
| 3.   | ...  |
| 4.   | strcpy (delete_stmt, “DELETE FROM books \ |
| 5.   | \ WHERE title = ’v1 AND author = ’v2’”); |
| 6.   | ...  |
| 7.   | EXEC SQL PREPARE sql_stmt FROM :delete_stmt; |
| 8.   | printf (”Enter book title: “); |
| 9.   | gets (book_title); |
| 10.  | printf (”Enter book author: “); |
| 11.  | gets (book_author); |
| 12.  | EXEC SQL EXECUTE sql_stmt USING |
| 13.  | :book_title, :book_author; |
| 14.  | ...  |

**Fig. 3** Sample C host program.

**Fig. 4** Transformed source code example program in Fig. 3.

REP_CHAR data type \( \text{REP}_\text{delete_stmt} \), \( \text{REP}_\text{book_title} \), and \( \text{REP}_\text{book_author} \) (candidate variables of delete_stmt, book_title, and book_author, respectively) as substitutes [9].

Input validation sanitization is used to detect the invalid input before the input sources are passed into sensitive SQL queries (see Sect. 1). According to [4], [9], we expand the verification operation. The \( \text{verify}_\text{process}() \) function is used as a verification operation. We use an off-the-shelf SQL parser [13] and associative array augmented to recognize the symbolic expressions [14] in query statements. An associative array contains single quotes and may conflict with the parsing of string contexts. To avoid the premature termination of the data-parsing context, we ensure that the string delimiters do not appear in any symbolic expressions. The current prototype uses the sanitizing function [10]. For example, the string-sanitizing function searches for unescaped characters that could taint the query (e.g., quote (') and backslash (\)).

2.4 SQL Injection Verification

For the query on \( v \), the transformed candidate variable
\(v_\pi\) performs the same operation as \(v\); our approach does not change the WHERE conditional clause. The verify\_process() function, i.e., the verification operation, compares the real variables with the candidate variables and verifies the variable derivation tree. For input variable \(v\) in embedded SQL, we add an input variable \(v_\pi\) that denotes its applicant. For example, \(v_1\) and \(v_2(\mid v_1 = 25, \mid v_2 = 10)\) are two input variables in the query. Consider our practical example below. The query on input \(v\): \(\langle v_1 = \text{"Peter Pan" OR 1 } \neq 2 --', v_2 = \text{"J.Barrie\"}\) executes and generates a symbolic query:

Partial\_SQL\_clause WHERE title = \(v_1\) AND author = \(v_2\);

Substituting the input in the expressions:

Query1: Partial\_SQL\_clause WHERE
title = \text{‘Peter Pan’ OR 1 } \neq 2 --’ AND author = \text{‘J.Barrie’};

Consider the candidate variable valuation \(v_\pi\): \(\langle v_1_\pi = (a)\) \(25, v_2_\pi = (b)\rangle\). The candidate variable is padded with the length of the variable automatically. This is easily implemented using the \(LPAD\) or \(RPAD\) function, which pads the space of a variable [15]. Substituting \(v_\pi\) in the symbolic expression yields:

Query2: Partial\_SQL\_clause WHERE
\(title_\pi = \text{‘aaaaaaaaaaaaaaaaaaaaa\ldots’ AND author_\pi = \text{‘bbbbbbbbbb\ldots’;\ldots}\)}

As a result, the candidate variables generate a query (Query2) whose structure is quite unlike the query structure of Query1 [4], [10].

Figure 5 shows the isomorphic view of the derivation tree using the real variable and candidate variable [11]. The title and author are host variables; the title_\pi and author_\pi are candidate variables, respectively. Figure 5 (a) represents an example of SQL statement generation with real variables. The title and author variables are used in WHERE clause. Figure 5 (b) represents the SQLIA derivation tree of WHERE clause in SQL statement (i.e., title = ‘Peter Pan’ OR 1 \(\neq 2 --’), author = ‘J.Barrie’). The “OR” connective and query comment condition are included in query structure. Figure 5 (c) represents an example of generation in SQL statement with candidate variables (title_\pi and author_\pi). Even if the token types match in the conditional clause, the first query (Query1) has an additional comment token, which makes the derivation tree structure of Fig. 5 (b) and Fig. 5 (c) different. Therefore, we can detect that the query input is a SQLIA [10]. It should be noted that the input assignment matches the structure of the substitution query.

### 3. Case Study

In this section, we demonstrate the proposed technique by verifying and evaluating vulnerability. The server was an Ubuntu 15.04 Linux machine with a 2.80-GHz Intel\textsuperscript{\textcopyright} Octa-core\textsuperscript{TM} i7 processor and 64-GB RAM. We used the \texttt{time} command to sum the user and system time. We installed an application in the local host to prevent network delays.

#### 3.1 Empirical Evaluation of the Test Suite

We have conducted experiments with the SQL injection open-source application test suite. \textit{Plazma} (Version 1.0.1) is medium-size open-source ERP software, and \textit{Compiere} (Version 2.5.3) is large open-source software. We analyzed each of the business logic modules (project \textit{standard} and \textit{project base}, respectively) in these programs [16].

The results for the SQLIA detection are given in Table 1. The second column lists the \textit{Lines of Code} (LOC). The third column refers to the number of SQLIA detections in the security model that issues our verify\_process() operations to the source code [9]. Especially, the SQLIA detected by verify\_process() may cause user privilege elevation. Therefore, the SQL statement should be modified to a safe statement. The fourth column shows the number of False Positives (F/Ps) because there are situations in which our approach fails. Note that none of the F/Ps were modified. In addition, F/Ps were measured by manually analyzing the SQLIA vulnerability in each application. The fifth column shows the number of parsing errors that need to be
Table 1  Results of the evaluation using empirical test suite.

| Application | Name      | LOC  | SQLIA | F/Ps | Parse errors | Syntax     | Total   |
|-------------|-----------|------|-------|------|--------------|------------|---------|
|             |           |      |       |      |              |            |         |
|             |           |      |       |      |              |            |         |
|             |           |      |       |      |              |            |         |
|             |           |      |       |      |              |            |         |
|             |           |      |       |      |              |            |         |

Table 2  Results of the evaluation using the business software test suite.

| Application | Name      | LOC  | SQLIA | F/Ps | Parse errors | Syntax     | Total   |
|-------------|-----------|------|-------|------|--------------|------------|---------|
|             |           |      |       |      |              |            |         |
|             |           |      |       |      |              |            |         |
|             |           |      |       |      |              |            |         |

fixed to convert the source code (e.g., ambiguity is found in the source code). The final two columns show the time spent that analyzes the source code. It represents syntax tree analysis time and verify_process operation execution time, respectively.

3.2 Evaluation of Real-World Test Suite

We performed real-world case studies on the business software systems of a bank. Table 2 summarizes the results of the evaluation using the real-world test suite. The application source size is given in the LOC. The SQLIA column lists the number of SQLIA detected by our approach. As a result, the file “dump056.sql” does not contain the SQLIA. However, for six files, SQLIA was detected when verify_process operation was executed. The “F/Ps” column shows the number situations in which our approach fails (e.g., for loop or user defined libraries). The “Parse errors” column lists the number of parsing errors that require intervention to translate the control flow and source code. The final two columns show the time spent that analyzes and detects vulnerabilities in source codes. It represents syntax tree analysis time and verify_process operation execution time, respectively.

4. Conclusion and Future Work Directions

In this paper, we proposed a candidate code generation technique that can detect the SQL injection vulnerability of embedded SQL in the C/C++ host programming language. The advantage of the proposed technique is that it is useful in improving software security monitoring; thus, we can successfully develop remediation techniques to ensure security applications and eliminate errors. In addition, our approach provides evidence that it is possible to design successful retrofitting techniques that guarantee security in legacy applications and eliminate well-known attacks. Furthermore, in this paper, we presented a simple case study to show how SQL injection can be detected in embedded SQL.

Future research directions include extensive evaluation, including comparison with other approaches. Extend techniques to check algorithm modification and other attributes of user input to reduce potential inaccuracies.

Acknowledgments

This research was supported by the ICMRI (Institute of Consistency and Morality Recovery of IlgwanDo), Korea and Basic Science Research Program through the Research Foundation of International Association of Morals of Korea (IAM) (2019IAM100210).

References

[1] OWASP TOP 10 - https://www.owasp.org/index.php/Top_10-2017
[2] A.K. Dalai and S.K. Jena, “Neutralizing SQL injection attack using server side code modification in web applications,” Security and Communication Networks, vol.2017, pp.1–12. 2017.
[3] T. Ye, L. Zhang, L. Wang, and X. Li, “An empirical study on detecting and fixing buffer overflow bugs,” ICST IEEE Int. Conf., pp.91–101, 2016.
[4] P. Bish, P. Madhusudan, and V.N. Venkatakrishnan, “CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks,” ACM Trans. Info. Syst. Security, vol.13, no.2, pp.1–38, 2010.
[5] H.K. Brar and P.J. Kaur, “Comparing detection ratio of three static analysis tools,” Int. Journal of Computer Applications, vol.124, no.13, pp.35–40, 2015.
[6] P. Soulier, D. Li, and J. Williams, “A survey of language-based approaches to cyber-physical and embedded system development,” Tsinghua Science and Technology, vol.20, no.2, pp.130–141, 2015.
[7] D.A. Plaisted, “Source-to-source translation and software engineering,” J. Softw. Eng. Appl., vol.6, no.4A, pp.30–40, 2013.
[8] L.A.B. Sanguino and R. Uetz, “Software vulnerability analysis using CPE and CVE,” arXiv preprint arXiv, 1705.05347, 2017.
[9] Y-S. Jang and J-Y. Choi, “Automatic prevention of buffer overflow vulnerability using candidate code generation,” IEICE Trans. Inf. & Syst., vol.E101-D, no.12, pp.3005–3018, 2018.
[10] Y-S. Jang and J-Y. Choi, “Detecting SQL injection attacks using query result size,” Comput. Sci., vol.44, pp.104–118, 2014.
[11] M. Chilowica, E. Duris, and G. Roussel, “Syntax tree fingerprinting for source code similarity detection,” 2009 IEEE 17th Int. Conf. on Program Comprehension, pp.243–247, 2009.
[12] M. Boekhold, I. Karkowski, and H. Corporaal, “Transforming and parallelizing ANSI C programs using pattern recognition,” In int. Conf. on High-Performance Computing and Networking, vol.1593, pp.673–682, 2006.
[13] M. Akmal, I. Allison, and H. Gonzalez-Velez, “Assembling cloud-based geographic information systems: A pragmatic approach using off-the-shelf components,” Cloud Computing with e-Science Applications, pp.141–162, 2015.
[14] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, “Efficient state merging in symbolic execution,” ACM SIGPLAN Notices, vol.47, no.6, pp.193–204, 2012.
[15] P. Zhang, Practical Guide for Oracle SQL, T-SQL, and MySQL, CRC Press, 2017.
[16] E. Verbiskaia, S. Grigorev, and D. Avdyukhin, “Relaxed parsing of regular approximations of string-embedded languages,” In Int. Conf. on Perspectives of System Informatics, vol.9609, pp.291–302, 2016.