Input-based Analysis Approach to Prevent SQL Injection Attacks

Angshuman Jana
Indian Institute of Information Technology
Guwahati, India
angshuman@iiitg.ac.in

Priyam Bordoloi
Indian Institute of Information Technology
Guwahati, India
prmbordoloi@gmail.com

Dipendu Maity
Indian Institute of Information Technology
Guwahati, India
dipendu@iiitg.ac.in

***Abstract*—SQL injection attack is one of the serious security threat for a database application. It permits attackers to get unauthorized access to the database by inserting malicious SQL code into the database application through user input parameters. In this paper, we propose input-based analysis approach to detect and prevent SQL Injection Attacks (SQLIA), as an alternative to the existing solutions. This technique has two part (*i*) input categorization and (*ii*) input verifier. We provide a brief discussion of the proposal *w.r.t* the literature on security and time cost point of view.**

***Index Terms*—SQL Injection Attacks, Input categorization, Input Filtering**

I. Introduction

The SQL injection attack (SQLIA) is arrived when the malicious SQL code is injected into the database program through input parameters which are later submitted to a back-end database server for the execution [1], [2]. An SQLIA can hamper the database in different ways like unauthorized manipulation of the database, obtaining of confidential data and etc. The number of SQLIA increased rapidly in recent years, a report [3] shows SQLIA accounted for 51 percent of all web application attacks in the second quarter of 2017. The direct insertion of user input into parameters which are concatenated into SQL statement and executed is the main happening behind SQL injection. For instance, examine the following PHP code [4]:

```
//Connect to the database
$con = mysql connect("localhost", "uname", "pass");
//Dynamically generating SQL query with user input
$dq="SELECT * FROM product WHERE price < '$ GET["val"]' "."ORDER BY pdesc";
//Executing the query against the database
$result = mysql query($dq);
```

The URL "http://www.hunting.com/product.php?val=1000" displays the cost of all product which are less than $1000. When inserting the malicious user input 1000'OR'1'= '1 and the corresponding URL http://www.hunting.com/product .php?val= '1000' OR '1'='1', the dynamically constructed SQL "SELECT * FROM product WHERE price < '100' OR '1'= '1' ORDER BY pdesc" gives out all information of product as the WHERE clause results to *true* always. This is called tautology-based SQLIA, various other forms of attacks exist with various attacker intents *e.g.* Union Query, Piggy-Backed Query, Stored Procedures, etc. [1], [2], [4]–[6]. The attacker intents is to find out the several way like identifying injectable parameters, determining database schema, performing denial of service, executing remote commands, etc. to perform the various type of SQLIA. However, the main cause of this attack is the effect of direct involvement of code into parameters which are concatenated with SQL statements and will execute. When Web applications unable to correctly sanitize the user inputs, then there is a possibility to the attacker to alter the generation of underlying SQL commends, thus leading to a massive loss situation. Therefore, require an effective and efficient technique to detect and prevent SQLIA.

Over the decades, lot of research works have been done and huge number of detection and prevention techniques are proposed for the SQL injection problems like AMNESIA – a model-based approach [7], Taint-based approach [8], Intrusion Detection technique [9], Static Code Checker [10], Instruction Set Randomization [11], Defensive Coding Practices [12], etc. All the existing approaches either suffer form generating large number of false positive alarm or unable to prevent many types of SQLIA e.g. Illegal/Logically Incorrect Query, Alternate Encodings and etc. As per our correct knowledge, any existing approach can not guarantee a complete safety of the database programs.

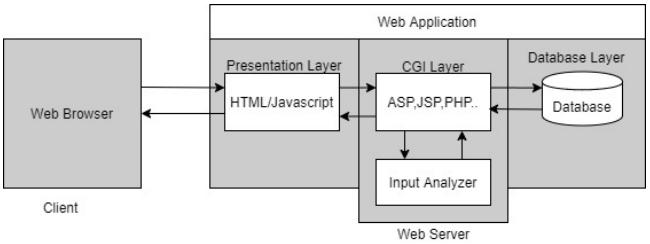

Fig. 1: Proposed Web Application Architecture

In this paper, we propose a input-analysis based approach to automatically detect and prevent SQLIA, as an alternative solutions. The objective of this approach is to identify the malicious user inputs which are mainly consist of either some kind of special symbols or key-

978-1-7281-7366-5/20/$31.00 ©2020 IEEE

words or combination of the both. This propose model serves as a powerful basis to develop a input checker to automatically detect and prevent the SQLIA. The system architecture is shown in Figure 1. We also provide a comparative study *w.r.t.* literature on security guaranty and time cost point of view.

II. Related Works

In [13], [14] the authors proposed a combined static analysis and automated reasoning method to prevent SQLIA. This method is efficient for detecting SQLIA, but other SQLIA without tautology cannot be detected. The proposed approach in [15] is used to detect and prevent SQL injection vulnerabilities. However, it is very much inefficient technique because it fails to consider inputs from all the sources. Note that in several applications operators, SQL-keywords may be used to express names (e.g. O'Brian), normal text entry, formulas and it generate high rate of false alarm. CANDID [16] is the code transformation-based technique which mainly constructs the coder-intended query structure. For that it executes the code on the set of candidate inputs which are self-evidently non-attacking. However, some cases it may generate a false positive alarm. Another two techniques SQL DOM [9] and Safe Query [17] are published to encapsulate the SQL statements and ensure the safety and reliability of the databases access. However, these approaches have prime restriction that they require experts to experience and use the new coding paradigm. The proposed approach in [8] is based on input flow analysis and input validation analysis to build a white-box and generated test input data to locate SQL Injection vulnerabilities. However, the vulnerabilities must be manually fixed by the developers found in the web application. In [11], [18], the authors introduced instruction set randomization technique and SQLrand approach respectively. It takes random input values into the runtime SQL statements of a web application and checks for mutability in order to detect SQL injection attacks. But this is fails to protect the database applications form several types of SQL injection attacks e.g. Illegal/Logically Incorrect Query, Alternate Encodings etc. On the other hand this also suffers from the infrastructure overhead. A dynamic tool DIGLOSSIA [19] is proposed which process dual parsing to compare the shadow query with the actual program generated query and it verifies whether a query issued by a application does not present any injected string. The tool fails to process all possible user inputs.

III. Problem Formalization

Formalization of a dynamic application program is:

- In the run tine environment, it accepts user inputs as the strings.
- It generates SQL codes using concatenation operation between SQL constructs and user input strings.
- Malicious user input, after performing the concatenation operations which may consider as the part of query control constructs and it leads to SQLIA.

Let Σ be an input alphabet. A database program $\mathbb{D}_p$: $(\Sigma^* \times \Sigma^* \times \cdots \times \Sigma^*) \rightarrow \wp(\Sigma^*)$ is defined as the mapping between the user inputs Σ and the query strings of Σ. Consider the set of SQL sub-strings $\mathbb{S}$= { c_1, c_2, … c_n } and a set of input strings $\mathbb{K}$ = { k_1, k_2, … k_m }, $\mathbb{D}_p$ constructs a query string (performing the concatenation operation) $\mathbb{Q} = q_1 + q_2 + \ldots + q_l$, where

$$\forall l : q_l = \begin{cases} \mathrm{K} & \textit{where} \quad \mathrm{K} \in \{k_1, k_2, \ldots, k_m\} \\ \mathrm{S} & \textit{where} \quad \mathrm{S} \in \{c_1, c_2, \ldots, c_n\} \end{cases}$$

Consider the SQL string $\mathbb{S}_l$ which may be represented into data- and control-part such as: $\mathbb{S}_l = \langle \{D_1, D_2, \ldots, D_i\}, \quad \{A_1, A_2, \ldots, A_j\} \rangle$. SQLIA occurs iff $\{A_1, A_2, \ldots, A_j\} \cap \{k_1, k_2, \ldots, k_m\} \neq \phi$.

IV. Proposed Framework

We propose a unique model as an alternatives to the existing ones. Our model has two phases, first one is input categorization and second one is input verifier that is design based on input categorization. This model is able to automatically detect and prevent SQL injection attacks.

A. Input Categorization

Due to the presence of malicious user input in dynamically generate SQL query, the query execution may lead to SQLIA. Our main objective is to detect such malicious user input and protect the concatenate operation during query generation. To achieve this, at first phase we define the user input domain within the four categories (Keywords, Special Chars, Alphabets, Numbers) as follows:

Keywords	OR, UNION, SELECT, DROP, SHUTDOWN …
SpecialCharacters	/, , $, (,), ", //, ?, +, !, @, =, ' …
Alphabets	[a - z, A - Z]
Numbers	[0 - 9]

All possible input string can be generated based from either four individual category or combination between two or more categories.

Observe that, among all possible input string some of input is always *safe* (not malicious) and use of these type of input in a SQL query statement never the cause of the SQLIA. On the other hand, some other input set exist which may lead to *unsafe* (vulnerable) and these may be caused of the SQLIA. Therefore, for only these type of inputs (rather all possible inputs), we design the input verifier to identify the actual malicious inputs. This way we protect the database application program from any kind of SQLIA.

Let $\mathbb{K}$ denote a set of keywords, $\mathbb{A}$ denote a set of alphabets in the upper case and lower case, $\mathbb{S}$ denote a set of special characters and $\mathbb{N}$ denote a set of numbers form 0 to 9. Let $\mathbb{M}$ denote a set of strings in the domain

$\mathbb{I} = \mathbb{K} \cup \mathbb{A} \cup \mathbb{S} \cup \mathbb{N}$. That is, $\mathbb{M}$ is the set of all possible words of elements from the union set $\mathbb{K} \cup \mathbb{A} \cup \mathbb{S} \cup \mathbb{N}$. We say a string is *safe* in $\mathbb{M}$ if the domain of string is $\mathbb{N} \cup \mathbb{A}$, otherwise may *unsafe*.

An algorithm to perform the user input categorization. We design an algorithm to generate the user input into the four categories classes. The algorithm `IntCat` in Algorithm 1 takes an user input and decompose it based on the categories classes. This result highlight the nature (*safe*/ may *unsafe*) input.

Algorithm 1: `IntCat`

Input: User Input String $\mathbb{I}$.
Output: Identify safe and unsafe

```
int f=0, c=0;
String str = readInput(); ;  // Accept user input
int l= strlen(str); ;  // Compute length of input
for int i =0 to l-1 do
    if str[i] ∈ [A - Z, a - z] then
        f = f+1 ;
    if str[i] ∈ [0 - 9] then
        c = c+1 ;
    if substr(str) ∈ {Keywords} then
        ... do nothing ...
    else
        ... do nothing ...
    if f=l || c=l || (f+c) =l then
        Safe;
    else
        May unsafe;
End
```

B. *Input Verifier*

Now, the input verifier identify the actual malicious input from the set of possibly unsafe input (highlighted by the Algorithm 1). We introduce *rs* based code checking to protect the database applications from SQLIA. The execution result of a SQL query Q store in *rs* variable and compared with the value of user input string. If the comparison result is *true* then the result of Q can be outsourced. Otherwise the input string treated as malicious input and discord the execution result of Q. Now, we present a code for user input verification at application-level:

```
int fl = 0 ;
while (rs.next) {
      if (!Slog.eq(rs.getStr(1)) | | !Spass.eq(rs.getStr(2))) {
            fl = 1 ;
            break ;      }   }
if (fl == 1)
      print (SQL Injection attack);
```

Observe that in Figure 2, we show the possible input categorization and execution of SQL queries statements *w.r.t* safe inputs and protect the database application from malicious input.

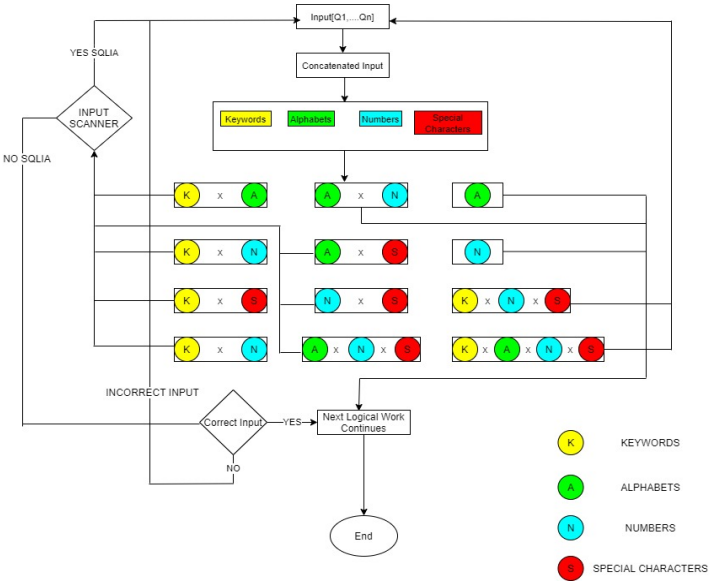

Fig. 2: System architecture of the propose model.

V. Implementation and Experimental Results

In this section, we evaluate the implementation results on the benchmark codes that are open-source web applications developed by JSP language [20]. We consider all possible type of user input which are mostly salf generated as well as collected from different sources. For this experiment, we set up the system with Intel i5 processor, 2.30GHz clock speed, 64-bit Windows 10 platform with 4GB RAM. The experimental results is depicted in the Table I.

VI. Discussion and Conclusion

Now-a-days, many businesses and organizations use web applications to provide services to users. Web applications depend on the back-end database to supply with correct data. However, data stored in databases are often targets of attackers. The SQL injection is a pre-dominant technique that attackers use to compromise databases. SQLIAs have been proposed over the past and several countermeasures against its have been proposed and implemented by researchers. In this paper, we provide a input-based analysis model to automatically detect and prevent the SQLIA. We provide an experimental result on the set benchmark programs. In Table II, we provide a comparative study *w.r.t*. literature on security guaranty and time complexity point of view. This propose model serves as a powerful basis to develop a input checker to automatically detect and prevent the SQLIA. We are now in progress to build a tool based on the proposals.

Database Applications	LOC	No. of SQL	NO.of Attributes	No. of Input	Detect & Prevent
Events(EventNew.jsp)	344	6	5	58	Y
Ledger(LedgerRecord.jsp)	436	9	8	60	Y
Portal(EditOfficer.jsp)	300	7	4	54	Y
Portal(EditMembers.jsp)	362	10	5	55	Y
EmplDir(DepsRecord.jsp)	285	4	3	59	Y
EmplDir(EmpsRecord.jsp)	435	9	7	71	Y
Bookstore(EditorialsRecord.jsp)	294	6	3	70	Y
Bookstore(BookMaint.jsp)	357	6	5	51	Y
BugTrack(PrjectMaint.jsp)	307	7	4	55	Y
BugTrack(EmployeeMaint.jsp)	316	6	5	60	Y
BugTrack(BugRecord.jsp)	336	6	4	69	Y

TABLE I: Results on benchmark programs

Tool	Detect	Prevent	User input from all sources	False positive	False negative	Modification of code	Time cost
Static Tainting [18]	√	x	√	x	√	x	linear
Security Gateway [21]	x	√	x	x	x	x	linear
WebSSARI [15]	√	x	√	x	x	x	linear
SQLrand [11]	√	√	x	√	√	√	linear
SQL DOM [17]	-	√	-	x	√	√	linear
IDS [9]	√	x	√	√	√	x	linear
JDBC-Checker [10]	√	x	x	x	x	x	linear
AMNESIA [7]	√	√	√	√	√	x	exponential
CANDID [16]	√	√	√	√	√	x	linear
CIAOs [22]	√	√	√	x	x	x	linear
DIGLOSSIA [19]	√	√	x	√	x	x	linear
Input-based analysis	√	√	√	x	x	√	linear

TABLE II: Comparative study of our approach w.r.t. existing approaches.

References

[1] W. G. Halfond, J. Viegas, and A. Orso, "A Classification of SQL-Injection Attacks and Countermeasures," in *Proc. of the IEEE International Symposium on Secure SE*. IEEE, 2006.

[2] B. K. Ahuja, A. Jana, A. Swarnkar, and R. Halder, "On preventing sql injection attacks," in *Advanced Computing and Systems for Security*. Springer, 2016, pp. 49–64.

[3] H.-C. Huang, Z.-K. Zhang, H.-W. Cheng, and S. W. Shieh, "Web application security: Threats, countermeasures, and pitfalls," *Computer*, vol. 50, no. 6, pp. 81–85, 2017.

[4] J. Clarke, *SQL Injection Attacks and Defense*, 1st ed. Syngress Publishing, 2009.

[5] R. Halder, A. Jana, and A. Cortesi, "Data leakage analysis of the hibernate query language on a propositional formulae domain," in *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXIII*. Springer, 2016, pp. 23–44.

[6] K. K. Mandal, A. Jana, and V. Agarwal, "A new approach of text steganography based on mathematical model of number system," in *2014 International Conference on Circuits, Power and Computing Technologies [ICCPCT-2014]*. IEEE, 2014, pp. 1737–1741.

[7] W. G. J. Halfond and A. Orso, "Amnesia: Analysis and monitoring for neutralizing sql-injection attacks," in *Proc. of the 20th IEEE/ACM ASE*. ACM, 2005, pp. 174–183.

[8] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *Security and Privacy in the Age of Ubiquitous Computing, IFIP TC11 20th International Conference on SEC, 2005*, pp. 295–308.

[9] F. Valeur, D. Mutz, and G. Vigna, "A learning-based approach to the detection of sql attacks," in *Proc. of the 2nd International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer-Verlag, 2005, pp. 123–140.

[10] C. Gould, Z. Su, and P. Devanbu, "Jdbc checker: A static analysis tool for sql/jdbc applications," in *Proc. of the 26th ICSE*. IEEE Computer Society, 2004, pp. 697–698.

[11] S. W. Boyd and A. D. Keromytis, "Sqlrand: Preventing sql injection attacks," in *In Proc. of the 2nd ACNS Conference*, 2004, pp. 292–302.

[12] J. Lin, J. Chen, and C. Liu, "An automatic mechanism for adjusting validation function," in *22nd AINA, 2008, Okinawa, Japan*. IEEE Computer Society, pp. 602–607.

[13] W. G. J. Halfond and A. Orso, "Combining static analysis and runtime monitoring to counter sql-injection attacks," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–7, July 2005.

[14] A. Jana, R. Halder, A. Kalahasti, S. Ganni, and A. Cortesi, "Extending abstract interpretation to dependency analysis of database applications," *IEEE Transactions on Software Engineering*, 2018.

[15] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *Proc. of the 13th International Conference on WWW*. ACM, 2004, pp. 40–52.

[16] P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "Candid: Dynamic candidate evaluations for automatic prevention of sql injection attacks," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 2, pp. 14:1–14:39, 2010.

[17] R. A. Mcclure and I. H. Krüger, "Sql dom: compile time checking of dynamic sql statements," in *in ICSE'05: Proc. of the 27th ICSE*. ACM, 2005, pp. 88–96.

[18] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proc. of the 14th Conference on USENIX Security Symposium - Volume 14*. USENIX Association, 2005, pp. 18–18.

[19] S. Son, K. S. McKinley, and V. Shmatikov, "Diglossia: detecting code injection attacks with precision and efficiency," in *Proc. of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1181–1192.

[20] "Gotocode," http://www.gotocode. com, [Online; accessed 20-Dec-2015], (now archived at: https://github.com/angshumanjana/GotoCode).

[21] D. Scott and R. Sharp, "Abstracting application-level web security," in *Proc. of the 11th International Conference on WWW*. ACM, 2002, pp. 396–407.

[22] D. Ray and J. Ligatti, "Defining code-injection attacks," in *Proc. of the 39th POPL*. ACM, 2012, pp. 179–190.