Securing web applications from SQLIA using progressive detector

N. Gunaseeli*
Department of Computer Applications,
K.L.N College of Engineering,
Pottapalayam, Sivagangai District, Tamil Nadu, India
Email: parkavithai.mdu@gmail.com
*Corresponding author

D. Jeya Mala
Thiagarajar College of Engineering,
Madurai, Madurai District,
Tamil Nadu, India
Email: djeyamala@gmail.com

Abstract: Web applications are distributed applications that are stored on a server and delivered to users over the internet. Many people enjoy its utilisation due to its benefits such as global access, cost saving, faster delivery of products and opportunities to manage the business from anywhere in the world. The users of web applications expect that the web applications should be secure and reliable, while doing their online transactions. But, modern web applications face threats like cross-site scripting, cookie poisoning, buffer overflow etc. Among the attacks of web applications, SQLIA is one of the most significant of such threats. SQL injection attack is a type of injection attack (SQLIA), in which SQL commands are injected as data-plane input in order to affect the execution of predefined SQL commands. SQL injection is a code injection technique that exploits the security vulnerability occurring in the database layer of an application. It allows attackers to get unauthorised access to the back-end database consisting of confidential user information. The paper presents a novel approach and it prevents all types of SQLIAs with minimum time frame. The proposed system aims at preventing all types of SQLIA by quick identification of malicious query at run time. This novel approach has been framed with the progressive detector (PD) tool and carried out on real time web applications. The experimental results clearly show that the Progressive Detector prevents all kinds of SQLIAs with minimum time frames.

Keywords: software engineering; software security; database security; SQL injection; run time monitoring; vulnerabilities; web applications.

Reference to this paper should be made as follows: Gunaseeli, N. and Mala, D.J. (2017) ‘Securing web applications from SQLIA using progressive detector’, Int. J. Information Systems and Change Management, Vol. 9, No. 1, pp.44–69.

Biographical notes: N. Gunaseeli is an Associate Professor of Department of Computer Applications at the K.L.N College of Engineering in India. She has 16 years teaching experience and six years research experience. Her research areas include software testing database and network security.
D. Jeya Mala is an Associate Professor of Department of Computer Applications at the Thiagarajar College of Engineering in India. She has 17 years teaching experience and ten years research experience. She has presented research papers in more than 30 national and international conferences and published more than 30 papers in national and international journals. She is the Principal Investigator of a project sponsored by the University Grant Commission, New Delhi, India. Her research areas include software testing and database security.

1 Introduction

The utilisation of web applications has increased the landscape of information retrieval delivery in various arenas like financial data, social security numbers, intellectual property and national security data. Web applications must handle the information securely while maintaining efficiency and availability. Unfortunately, web applications are often vulnerable to attacks like cross-site scripting, cookie poisoning, buffer overflow etc. Among these attacks, structured query language injection attacks (SQLIA) is ranked first in the open web application security project (OWASP). With SQL injections; the cyber-attackers can get complete control of the database. Consequently, they can gain the private information. The attackers can are to manipulate the database and it leads to security violations such as fraud, loss of confidential and identity theft (Agarwal et al., 2015; Fayo, 2005; Fonseca et al., 2014; Guimaraes, 2009; Lee et al., 2012).

SQL injection is a kind of vulnerability in which a hacker can send crafted SQL command to web application. If the crafted SQL query works as intended by the hacker, then they can perform operations like create, update, read, modify or delete the data stored in the database of the web application.

SQLIA has been classified into five basic classes (Al-Nabulsi et al., 2014; Dakwala and Lavingia, 2016; Halfond et al., 2008; Shahriar and Zulkernine, 2014; Sunkari and Rao, 2016), with respect to the attacker’s target and the vulnerabilities in web-applications. A classification of SQLIA and the attacker’s achievements are illustrated in Table 1.

The outcome of SQL injection attacks can be assorted as follows:

1. Loss of privacy is a noteworthy issue with SQL injection assaults since SQL databases for the most part hold touchy and basic data. It could be seen by unapproved clients as a result of fruitful SQL infusion assault.

2. Successful SQL infusion assault permits outside source to make unapproved changes, for example, changing or notwithstanding erasing data from target databases.

3. Poorly composed SQL inquiries don’t appropriately approve client names and passwords, which permit unauthenticated element or assailant to associate with the influenced database or application as a verified client, without learning the secret key or even client name.

4. Successful exploitation of SQL injection vulnerability allows the attacker to change authorisation information and gains elevated privileges, if the authorisation
information is stored in the affected database. In real time scenario, it is difficult to detect the SQL injection prior to its impact. In most number of scenarios, unauthorised activity is performed by the attacker through valid user credentials such as malicious modification of existing SQL queries of web applications that are accessing critical sections of the affected databases.

**Table 1** Classification of SQLIA

<table>
<thead>
<tr>
<th>SQL injection types</th>
<th>Attack query example</th>
<th>Attackers achievement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bypass authentication</td>
<td>&quot;select balance from account where username='&quot; or 1=1 --&quot; and password='&quot;;</td>
<td>Two dashes, comment the remaining text. Expression l=1 is always true. User will be logged in with privileges of the first user stored in the database.</td>
</tr>
<tr>
<td>Injected union and union ALL query</td>
<td>select bin from sqlia_table_1 where username='apple' and password='&quot; or 1=1 union select null from sqlia_table_1'</td>
<td>The actual runtime query returns null data. However, the injected query generates data from the database.</td>
</tr>
<tr>
<td>Piggybacked queries</td>
<td>Query= &quot;select account balance from sqlia_table_1 where username='prakash' and password='&quot;; drop table user&quot;;</td>
<td>The database server executes the second injected query. Thus, a harmful operation may also be performed on the database with such injected query(s). The attackers easily access the data from database by using the alternate codlings like ASCII, hexadecimal, etc.</td>
</tr>
<tr>
<td>Alternate encodings</td>
<td>select bin from sqlia_table_1 where username='apple123 or 1 = 1' and password='&quot; or 1=1;SHUTDOWN’ SELECT acct FROM users login=' _AND pin=convert(int, (select top 1 name from sysobjects where xtype='u'))</td>
<td>1 Perform SQLIA, the attackers need some prior knowledge of database schema, which is often unknown. 2 Malformed queries allow for overcoming this problem by taking the advantage of overly descriptive error messages that are generated by the database.</td>
</tr>
</tbody>
</table>

The researchers have proposed a variety of techniques to prevent SQL injection vulnerabilities (George and Jacob, 2016; Buehrer et al., 2005; Tajpour et al., 2012; Natraj and Subramani, 2012; Huang et al., 2004; Seixas et al., 2014; Kumar, 2014). For example, one kind of solutions captures the query at runtime and compares it with the existing query structure. This approach helps to prevent all types of queries but the query structure must be known at compile time itself. Some other solutions are based on static analysis which detects only subset of SQL injection vulnerabilities. Although the researchers have proposed extensive varieties of techniques to prevent SQL injection vulnerabilities, none of the approach has concentrated on minimising the time frame...
while fixing all kinds of SQL injection vulnerabilities. It is a lack of efficiency in most of the research solutions.

Therefore, the proposed approach aims to prevent all kinds of SQL injections with minimum time frame at run time. It is based on the novel idea of progressive detection. The proposed system scans the query in a progressive fashion and blocks the query as soon as the SQL vulnerability is found. Thus, it prevents the SQL injection vulnerability with minimum time frame at run time.

This paper has been organised as follows: Section 2 provides a literature survey of the recent researches related to SQLIA. The proposed system based on progressive detection is presented in Section 3. The experimental results are discussed in Section 4. This section consists of six web applications that are well known to vulnerable to SQLIA. The conclusion is included in Section 5.

2 Literature survey

Different techniques have been proposed by researchers to treat the problem of SQLIA. Considering SQLIA as one of the main issues in database security, there has been keen research on detection and prevention mechanisms against this attack (Buehrer et al., 2005; George and Jacob, 2016; Halfond et al., 2008; Kermalis and Tzouramanis, 2008; Natrajan and Subramani, 2012; Tajpour et al., 2012; Sharma et al., 2016).

2.1 Static analysis approach

The approaches based on static analysis are helpful in debugging the application development. Since, it fixes the bugs of the program easily, the developer can make patches for the application by modifying the source code (Kermalis and Tzouramanis, 2008; Seixas et al., 2014). However, it is a tedious and time-consuming process to patch a deployed system. In addition, static analysis often fails to detect all kinds of vulnerabilities. Therefore, it brings high false negative rate. Moreover, static analysis cannot capture the exact structure of an intended query because, the full structure is known at runtime (Buehrer et al., 2005).

2.2 Dynamic tainting

The approach based on dynamic tainting (Halfond et al., 2008) prevents all kinds of SQLIA by tokenising the query into keywords, operators and literals. However, the approach expects the developers to specify the valid database objects of a query before testing and deployment. If the valid objects are not included in the list properly, it generates false positives.

2.3 Parse tree validation approach

The technique based on parse tree validation compares the parse tree of the input query statement with original query statement at run time (Buehrer et al., 2005). The approach stops its execution when there is a mismatch between the input query and the original query. The drawbacks of the approach are:
it is suitable only for static query
additional overhead is imposed on computation.

2.4 Web vulnerability scanning
This technique scans the web vulnerabilities using software agents. It performs its execution similar to black box testing. The drawbacks of the approach are:
1 there is knowledge about the internal architecture of the applications
2 generates more number of false positives (Fonseca et al., 2007, 2014; Roy et al., 2011).

3 Proposed system

3.1 Overview of the proposed system
Securing web applications from SQLIA using progressive detector aspires not only to prevent of SQL injection but also aims at its efficiency in fixing the query as earlier as possible. It scans the important tokens of a query like table, attributes, etc in a progressive fashion and blocks the input query as soon as the SQLIA is detected. Thus, it reduces the time frame in fixing the malicious query and proves its efficiency at run time. Unlike the existing dynamic technique, the proposed approach does not expect developers to specify the valid database objects of a query before testing and deployment. The valid database objects are updated at runtime based on rule sets of the proposed approach. Thus, the proposed approach does not produce any false positives. Moreover, the proposed approach is very suitable for data warehouse applications. Because, many data warehouse applications interact with data that are generated periodically. For example, the data warehouse applications generate new tables during every quarter. Since, the proposed approach updates all valid database objects at run-time; it helps to prevent all kinds of SQLIA without yielding false positives.

3.2 System architecture
The architecture of the system proposed work is shown in Figure 1, which consists of seven major components namely, rule sets, query tokeniser, dynamic generator, extractor, library, token validator and controller. Moreover, this system has been implemented as a two phase system for monitoring and capturing the attacks. In phase one, the preparation of rule sets and updation of library are done to scan the query fragments at run time. This preparation is done before the deployment of the system. In phase two, the intentions of the queries are identified by using token validator based on the rule sets by comparing the current token with the appropriate rule set. If any mismatch is identified by the token validator, the controller immediately stops scanning the query and declares the query as malicious.
3.2.1 Rule sets

The rule set plays a crucial role in fixing the malicious query as earlier as possible. It specifies the order of checking the input query fragments and the preparation of rule set is done before the deployment of the system.

The economical order of checking the input query is prepared based on a survey prepared from different sources (Borade and Despande, 2013; Fayo, 2005). The survey shows that the attackers often have the targets to know the tables of a database to extract data correctly from a database. Because, knowing the table names helps them to gain information stored inside the tables in the server in order to get access to the valuable data stored in it. After getting the table names, the attacker tries to find the column names of tables in a database. These basic reasons have stimulated to scan the important query tokens in a particular economical order. The order of scanning the query fragment is framed as rule sets and created in the form of an xml file. The controller triggers the rule set in a step-by-step fashion, until all the rules are applied on the input query. The stored rule sets are stored in an XML file with predefined structure. The XML format is chosen because, it is more structured than the flat files, more flexible than matrices, simpler and consumes less storage than databases.
This rule set clearly gives the highest priority for scanning the tables of the input query. The next priority goes to scanning the column names of the query. Third priority concentrates on tautologies conditions. The last rule set checks whether any black list contains in the input query.

The representation of the rule_set in XML file is

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?><Priority>
    <rank_1><validate>tables</validate></rank_1>
    <rank_2><validate>attributes</validate></rank_2>
    <rank_3><validate>tataulogy</validate></rank_3>
    <rank_4><validate>black_list</validate></rank_4>
    <rank_5><validate>datatype</validate></rank_5>
</Priority>
```

3.2.2 Query tokeniser

This component converts the input query into useful tokens. Query Tokens are splitted on the basis of spaces between them. This component generates productive tokens at run time.

This query tokenisation process is executed in the following four essential steps:

Step 1  Process the input query

Consider the following example input query:

`Select * from table where u_id = 'kar' or 1 = 1.`

Step 2  Break the input query into useful tokens

Tokenisation gives a set of tokens. It tokenises the query string by using string tokeniser and each token is stored in an array of string called input_query_array.

Table 2  The list of tokens parsed by string tokeniser

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>T8</th>
<th>T9</th>
<th>T10</th>
<th>T11</th>
<th>T12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select</td>
<td>*</td>
<td>From</td>
<td>Table</td>
<td>Where</td>
<td>U_id =</td>
<td>‘kar’</td>
<td>or</td>
<td>1 = 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.2.3 Dynamic generator

To validate each query fragments of the input query at run time, the token validator needs a set valid database objects. These relevant valid data objects are generated by a
component called dynamic generator. The dynamic generator yields a sequence of valid
data objects at run time depending on the value specified by the controller. If the
controller passes the value 1 to dynamic generator, then it fetches the first element of the
rule_set XML file. Based on the element content, it generates valid user defined table
names at run time and stores it in an XML file called table_xml file. This table_xml file is
used to check out the valid query tokens related to the tables in an input query. If the
controller passes the value 2, then the dynamic generator fetches the second element of
the rule set, generates the valid attributes at run time and stores it in an attributes_xml
file. The dynamic generator generates attributes related to tables stored in table_xmlfile.

3.2.4 Extractor

To validate the important query fragments, it is necessary to extract the relevant token
from the input query. This process is done by the extractor, until all the tokens have gone
for validation. The extractor extracts iteratively until the all important query fragments
are validated. Anyway, the extractor stops the extraction, if the token validator finds any
kind of SQLIA. The extraction of query fragment is done by parsing the input query
using DOM parser. The extraction of query fragment is done in step-by-step procedure
based on the rule sets.

When the controller passes value 1, then the extractor extracts the table names used in
the query. When it increases its value by 1, then the extractor extracts attributes from the
input query. When the controller passes the value 3, then the extractor extracts three types
of tokens:

1. where tokens
2. group by string
3. having string.

The extracted tokens are stored in an array one by one in tautogies_array. The array is
used by token validator for checking tautology conditions. These extractions are shown in
the following table.

Table 3 Various kinds of tokens

<table>
<thead>
<tr>
<th>Controller value</th>
<th>Name of the token</th>
<th>Start token</th>
<th>End token</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Relation tokens</td>
<td>From</td>
<td>Where, group by , end of query</td>
</tr>
<tr>
<td>2.</td>
<td>Attribute token</td>
<td>Select</td>
<td>From</td>
</tr>
<tr>
<td>3.</td>
<td>Where token</td>
<td>Where</td>
<td>Group by , order by , end of query</td>
</tr>
<tr>
<td>3.</td>
<td>Group by string</td>
<td>Group by</td>
<td>Order by, having , end of query</td>
</tr>
<tr>
<td>3.</td>
<td>Having string</td>
<td>Having</td>
<td>Order by , end of query</td>
</tr>
</tbody>
</table>

3.2.5 Library

The progressive detector gives completeness, since it scans the input query with black
list. The ignorance of the black list may lead to false positive (Halfond et al., 2008).
Therefore, the developer takes the responsibility of collecting and updating the black list
from various sources. The collected black lists are stored in black list XML file. By
using this black_list(XML) file, the progressive detector creates a deployment library and
it can be deployed like any other web application. The following shows the representation of the black list stored in library.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?><black_list>
  <list>table</list>
  <list><script</list>
  <list>*</list>
  <list>@@</list>
  <list>@</list>
  <list>sys</list>
  <list>syscolumns</list>
  <list>begin</list>
  <list>cast</list>
  <list>create</list>
  <list>cursor</list>
  <list>declare</list>
  <list>delete</list>
  <list>update</list>
  <list>drop</list>
  <list>execute</list>
  <list>execute</list>
  <list>fetch</list>
  <list>insert</list>
  <list>open</list>
</black_list>
```

3.2.6 Token validator

This component performs indispensable verification of incoming tokens to assess their validities. This is done in the following steps:

Input:
Tokens of the Query under test, XML file containing data objects generated by generator and Library.

Output: returns flag=0 to controller, if the query is malicious, returns flag=1 to controller, if the query is safe.

Begin
If rule = 1 and flag =1 then // check for valid tables Total= total count of relation tokens.
Count=0
For each user defined tables in table_xml_file
For each relation token in relation_token_array
If relation token = user defined table then Count = count +1
End if
Securing web applications from SQLIA using progressive detector

End if
If count = total then
    Return flag=1 to controller else
    return flag=0 to controller
End if
If rule = 2 and flag=1 then
    // check for valid attributes and data types
    Total= total count of attribute tokens.
    Count=0
    For each attributes in attribute_xml_file
        For each attribute token in attribute_token_array
            If attribute token = valid attribute then
                If data type of attribute token = valid attribute then
                    Count = count +1
                End if
            End if
        End for
    End for
End if
If count = total then
    Return flag=1 to controller else
    return flag=0 to controller
End if
If rule =3 and flag=1 then
    // check for tautology
    Total= total count of tokens in tautology array.
    count=0
    For each attributes in attribute_xml_file
        For each token in tautology array
            If token in tautology array= valid attribute then
                count = count + 1
            End if
        End for
    End for
End if
If count = total then
    Return flag=1 to controller else
    return flag=0 to controller
End if
    If rule = 4 and flag=1 then // check for black list Total= total
count of tokens in input query array.
count=0
For each attributes in black_list_xml_file
    For each token in input_query_array
        If token in input query array= black_list then
            Return flag=0 to controller
            Break;
        End if
    End for
End for
Return flag=1
End if
End

3.2.7 Controller
This component takes the responsibility of checking SQLIA. It is a special iterator which
works in a repetitive mode for fixing the malicious query. It controls the following
components:
1 extractor
2 rule sets
3 token validator
4 library.
The controller validates the input query in the following the essential steps:
Begin
Flag=1,Rule =1
For each rule set from 1 to n If rule = 1 and flag=1 then // check for valid tables
    Step 1: The controller passes the rule to dynamic generator and to extractor.
    Step 2: Generates table_xml file containing
currently used user defined tables. Step 3: The extractor extracts
relation tokens from input query.
    Step 4: Call token validator (table_xml_file, extracted relation token,flag=1)
    Step 5: The relation tokens are compared against the table_xml file by the token
validator. If token_validator returns flag=0 then
        Declare the query is malicious and stop scanning the input query
Else
    rule = rule + 1;
End if
Securing web applications from SQLIA using progressive detector

If rule = 2 and flag=1 then
// check for valid attributes and its data type
   Step 1: The controller passes the rule to dynamic generator and to extractor.
   Step 2: Generates attribute_xml file containing attributes belong to tables specified in
          table_xml file
   Step 3: The extractor extracts attribute tokens from input query.
   Step 4: Call token validator (attribute_xml_file, extracted attribute tokens and its datatype,
          flag=1) Step 5:
          If token_validator returns flag=0 then Declare the query is malicious and stop
          scanning the input query
          Else
          rule = rule + 1;
          End if
If rule =3 and flag=1 then
// check for tautology
   Step 1: Use the attributes_xml file containing attributes belong to tables specified in
          table_xml file
   Step 2: extracts three types of tokens 1) where tokens 2) Group by string 3) Having in the
          input
          query and store it in a tatologyarray
   Step 3: Call token validator (attribute_xml_file, tatautology_array, flag=1)
   Step 4: Tetautology_array is compared with element content of the attribute_xml file by
          the
          token validator.
If token_validator returns flag=0 then
   Declare the query is malicious and stop scanning the input query
Else
   rule = rule + 1;
End if
If rule =4 and flag=1 then
// check for black list
   Step 1: Pass the rule to library and fetch the black_list xml file.
   Step 2: The extract all the tokens from array
   Step 3: Call token validator (black_list_xml_file, array with all tokens, flag=1 )
   Step 4: The token validator compare with element content of the
          black_list_xml file with each one of the token.
   If token_validator returns flag=0 then
      Declare the query is malicious and stop scanning the input query
   End if
End for
If
flag=1
    Declare the query is safe
    Fetch relevant data from
    database
    End if
End

4 Experimental setup

The experimental setup consists of 6 web applications that are known to be vulnerable to SQLIA. Five of the applications are commercial applications that are obtained from Kashipara (http://wwwkashipara.com/): campus management system, online examination system, library management system, online airline reservation management system. One the web application has been developed by a student. The name of the web application is: college management. The tool has been assessed using the open source security teaching Java web application web goat, which is developed by OWASP. All the applications have been deployed on glassfish server with MySQL as database. These applications are executed in the Netbeans IDE.

The space of real attacks is unlimited. A repository is built by collecting SQL injection attacks from white papers, technical reports, web advisories, hacker on-line communities, web sites and mailing lists (Avireddy et al., 2012; Dharam and Shiva, 2012; Elia et al., 2012; Fayo, 2005; IBM internet security systems, 2009; Valeur et al., 2005).

4.1 Result and discussion

Various kinds of case studies are carried out for ascertaining the efficiency of the proposed approach. The proposed system is evaluated in three kinds of aspects. They are:

1. whether the proposed system prevents all kinds of SQLIAs vulnerabilities.
2. Whether the proposed system prevents the SQLIAs vulnerabilities with minimum time frame
3. whether the proposed system generates false positives.

4.2 Evaluation of the SQLIA prevention rate

To evaluate the first aspect, all the web applications are tested using various kinds of SQLIA vulnerabilities like union query, tautologies, piggy-backed queries, malicious queries and alternate encodings. For the evaluation of SQLIA prevention rate, the parameters mentioned in the second column of Table 3 are applied. The outcome of the
evaluation is displayed in Table 4. The second column of the table reports the total number of attacks faced by the various web applications. The third column of the table reports the total number of attacks prevented by the proposed system. The result shows that the proposed system is able to prevent all the intrusions injected by the intruders and is able to achieve 100% prevention rate. The outcome of the evaluation clearly concludes that the proposed tool is an effective technique for preventing all kinds of SQLIA’s. Thus, the proposed approach is a secure and robust solution to protect against all kinds of SQLIA’s.

Table 4 Shows the prevention rate of proposed system

<table>
<thead>
<tr>
<th>Web application</th>
<th>No. of attacks</th>
<th>No. of prevention</th>
<th>Prevention rate (in %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>College management</td>
<td>96</td>
<td>96</td>
<td>100</td>
</tr>
<tr>
<td>Web goat</td>
<td>52</td>
<td>52</td>
<td>100</td>
</tr>
<tr>
<td>Online examination system</td>
<td>19</td>
<td>19</td>
<td>100</td>
</tr>
<tr>
<td>Campus management system</td>
<td>23</td>
<td>23</td>
<td>100</td>
</tr>
<tr>
<td>Library management system</td>
<td>82</td>
<td>82</td>
<td>100</td>
</tr>
<tr>
<td>Online airline reservation</td>
<td>34</td>
<td>34</td>
<td>100</td>
</tr>
<tr>
<td>Management system</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.3 Evaluation of Query validation and execution time measurement

To address the execution time measurement, all the web applications have been executed using progressive detector tool. The tool gets input query from the user through login page.

The input query is validated after separating into tokens using appropriate rule set and token validator of progressive tool. If the query is found as malicious, then the tool immediately stops validating the input query and stops accessing the web applications. Otherwise, the user can access the web application. When the tool completes its validation, the execution time is reported and stored in a database. The execution time of each query is tabulated in Table 5. The results of the evaluation are very promising and encouraging. The tabulated results clearly state that the proposed system can prevent all kinds of SQLIA vulnerabilities with minimum time frame. The report expresses the comparison of time frame between the proposed system and the existing system in milliseconds. It is the outcome result of the benchmark problem ‘web goat’.

The efficiency of the proposed system can be viewed through Table 6. It summarises the performance of each of the web application. It reports the total execution time taken by each web application in milliseconds. The results of the proposed study clearly demonstrate that the proposed system can handle all kinds of attacks with minimum time frame.

Based on the obtained results a graph has been drawn for the total time taken by the existing system and the proposed system. Figure 2 provides another view of the timing measurements by using a bar chart.
<table>
<thead>
<tr>
<th>Query</th>
<th>Existing system</th>
<th>Proposed system</th>
<th>Username</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select bin from sqlia_table_1 where username=&quot;or 1=1--' and password='gh'</td>
<td>1051</td>
<td>171</td>
<td>_ or 1=1--'</td>
<td>gh</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username=&quot;union select from sql' and password=&quot;or 1=1--'</td>
<td>130</td>
<td>75</td>
<td>'union select from sql'</td>
<td>&quot;or 1=1--'</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username=&quot;or 2=2 or select column_name from information_schema.columns' and password=&quot;or 1=1--'</td>
<td>154</td>
<td>116</td>
<td>'or 2=2 or select column_name from information_schema.columns'</td>
<td>&quot; or 1=1--'</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username=&quot;or (select 1 from dual where database() like '%j%') and password=&quot;or 1=1--'</td>
<td>1235</td>
<td>51</td>
<td>'or (select 1 from dual where database() like '%j%') and &quot; or 1=1--'</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username=&quot;or (select count(*) from information_schema.tables' and password=&quot;or 1=1--'</td>
<td>188</td>
<td>53</td>
<td>'or (select count(*) from information_schema.tables'</td>
<td>&quot; or 1=1--'</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username=&quot;or 2=2 or select column_name from information_schema.columns' and password=&quot;or 1=1--'</td>
<td>927</td>
<td>15</td>
<td>'or 2=2 or select column_name from information_schema.columns'</td>
<td>&quot; or 1=1--'</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username=&quot;convert (int, (select top 1 from sysobjects where xtype= 'u'))&quot; and password=&quot;or 1=1--'</td>
<td>2624</td>
<td>1177</td>
<td>Select bin from sqlia_table_1 where username=&quot;convert (int, (select top 1 from sysobjects where xtype= 'u'))&quot; and password=&quot;or 1=1--'</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username=&quot;ASCII(SUBSTRING((select top 1 name from sysobjects ),1,1 )) &gt; X wait for 10 and pin=</td>
<td>1236</td>
<td>115</td>
<td>ASCII(SUBSTRING((select Top 1 name from sysobject s ),1,1 ))&gt; X wait for 10 and pin=</td>
<td></td>
</tr>
<tr>
<td>Query</td>
<td>Existing system</td>
<td>Proposed system</td>
<td>Username</td>
<td>Password</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>----------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username=''' or '''=''' and password=''' or '''=''='''</td>
<td>2074</td>
<td>931</td>
<td>&quot;or&quot;=&quot;=&quot;</td>
<td>&quot;or&quot;=&quot;=&quot;</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username=''' or '=' and password='''</td>
<td>143</td>
<td>111</td>
<td>&quot;or&quot;='='</td>
<td>or &quot;=&quot;</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='Apple' and password='orange' union select username from sqlia_table_2</td>
<td>597</td>
<td>318</td>
<td>Apple</td>
<td>&quot;or&quot;='=' or &quot;=&quot;</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='Apple' and password='orange' union select username from sqlia_table_1</td>
<td>1622</td>
<td>47</td>
<td>apple</td>
<td>&quot;or&quot;='=' or &quot;=&quot;</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='Apple' and password='orange' union select username from sqlia_table_1</td>
<td>167</td>
<td>16</td>
<td>apple</td>
<td>Select username from sqlia_table_1</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='Apple' or 5=5 or 2=2 and password='orange' union select username from sqlia_table_1</td>
<td>125</td>
<td>15</td>
<td>Apple' or 5=5 or 2=2='orange' union select username from sqlia_table_1</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='Apple' or 5=5 or 2=2 and password='orange' union select username from sqlia_table_1</td>
<td>141</td>
<td>15</td>
<td>'or' 150=51 or SELECT COLUMN_NAME FROM INFORMATION_SCHEMA. COLOUMNS 'and password='orange'</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='Apple' or 5=5 or 2=2 and password='orange' union select username from sqlia_table_1</td>
<td>104</td>
<td>55</td>
<td>'OR EXISTS (SELECT 1 FROM dual WHERE database() LIKE '_%a%' AND='=' and password _'</td>
<td></td>
</tr>
<tr>
<td>Query</td>
<td>Existing system</td>
<td>Proposed system</td>
<td>Username</td>
<td>Password</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple' and password = ASCII(SUBSTRING((select top 1 name from sysobjects),1,1)) &gt; X wait for 10 -- and pin =</td>
<td>913</td>
<td>124</td>
<td>apple</td>
<td>ASCII(SUBSTRING((select top 1 name from sysobjects),1,1)) &gt; X wait for 10 -- and pin =</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple' and password = ' or 1=1 unions select bin from sqlia_table_1</td>
<td>1446</td>
<td>32</td>
<td>apple</td>
<td>' or 1=1 union select bin from sqlia_table_1</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='Apple' and password = ' or 1=1 union select null from sqlia_table_1</td>
<td>157</td>
<td>15</td>
<td>Apple</td>
<td>' or 1=1 union select null from sqlia_table_1</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple123' and password=' or 1=1 union select null from sqlia_table_1</td>
<td>117</td>
<td>16</td>
<td>apple 123</td>
<td>' or 1=1 union select null from sqlia_table_1</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple123' or 1=1' and password = ' or 1=1 SHUTDOWN'</td>
<td>110</td>
<td>94</td>
<td>apple 123</td>
<td>Select bin from sqlia_table_1 where username='apple123 or 1=1' and password = ' or 1=1 SHUTDOWN'</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple' or 1=1' and password = ' order by 1 --</td>
<td>893</td>
<td>95</td>
<td>apple ' or 1=1</td>
<td>Select bin from sqlia_table_1 where username='apple' or 1=1' and password = ' order by 1 --</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple' and password = (SELECT COUNT(*) FROM INFORMATION_SCHEMA.TABLES ' 126 and password=' or 1=1</td>
<td>126</td>
<td>47</td>
<td>Apple ' or 1=1</td>
<td>OR (SELECT COUNT(*) FROM INFORMATION_SCHEMA.TABLES ' 126 and password=' or 1=1</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple' and password = 'orange; exec ('select from login ')</td>
<td>126</td>
<td>62</td>
<td>apple'</td>
<td>Orange'; exec ('select from login ')</td>
</tr>
</tbody>
</table>
Table 5: Comparison of time frame between the proposed system and existing system (continued)

<table>
<thead>
<tr>
<th>Query</th>
<th>Existing system</th>
<th>Proposed system</th>
<th>Username</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select from sqlia_table_1 where username='apple' and password='''</td>
<td>202</td>
<td>130</td>
<td>apple</td>
<td>execute((0x736875746f 776e) - - and password='''</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple' 1=0 -- and password=''</td>
<td>314</td>
<td>160</td>
<td>apple' 1=0 --</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple' 1=1 and password=''</td>
<td>258</td>
<td>158</td>
<td>apple' 1=1</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple' UNION SELECT password from sqlia_table_1 where username='apple' -- and password=''</td>
<td>163</td>
<td>12</td>
<td>apple' UNION SELECT password from sqlia_table_1 where username='apple' --</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple drop table users -- and password=''</td>
<td>140</td>
<td>109</td>
<td>Apple'; drop table users--</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='or 1=1;/<em>' and password='</em>/--'</td>
<td>103</td>
<td>101</td>
<td>'0r1=1;/*</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='or 1=1/<em>' and password='</em>/--'</td>
<td>214</td>
<td>126</td>
<td>' or 1=1 union delete from sqlia_table_1 where username = 'apple'</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='Apple' union delete from sqlia_table_1 where username='apple and password = 'orange'</td>
<td>157</td>
<td>142</td>
<td>Apple'; union delete from sqlia_table_1 where username = 'apple'</td>
<td></td>
</tr>
</tbody>
</table>

(continued)
<table>
<thead>
<tr>
<th>Query</th>
<th>Existing system</th>
<th>Proposed system</th>
<th>Username</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select bin from sqlia_table_1 where username = 'apple' union insert into sqlia_table_1 values ('11', 'oo', 1000') and password = 'orange'</td>
<td>111</td>
<td>95</td>
<td>apple' union insert into sqlia_table_1 values ('11', 'oo', 1000')</td>
<td>Orange</td>
</tr>
<tr>
<td>Select from bin sqlia_table_1 username = 'union select bin from sqlia_table_1 --' and password = ''</td>
<td>2414</td>
<td>14</td>
<td>'union select from sqlia_table_1 --'</td>
<td></td>
</tr>
<tr>
<td>Select from bin sqlia_table_1 username = 'union select from password sqlia_table_1 --' and password = ''</td>
<td>151</td>
<td>10</td>
<td>'union select from password sqlia_table_1 --'</td>
<td></td>
</tr>
<tr>
<td>Select from bin sqlia_table_1 username = 'union select 1 from sqlia_table_1 --' and password = ''</td>
<td>131</td>
<td>10</td>
<td>'union select 1 from sqlia_table_1 --'</td>
<td></td>
</tr>
<tr>
<td>Select from bin sqlia_table_1 where username = 'apple' 'union select version () from information_schema.columns' and password = ''</td>
<td>241</td>
<td>40</td>
<td>apple' union select version () from information_schema.columns</td>
<td></td>
</tr>
<tr>
<td>Select from bin sqlia_table_1 where username = 'apple' 'union select 1, group_concat(schema_name), 3,4,5,6 from information_schema.columns' and password = ''</td>
<td>121</td>
<td>10</td>
<td>'apple' 'union select 1, group_concat(schema_name), 3,4,5,6 from information_schema.columns'</td>
<td></td>
</tr>
<tr>
<td>Select from bin sqlia_table_1 where username = 'apple' 'union select 1, group_concat(table_name), 3,4,5,6 from information_schema.tables -- where table_schema= database()' and password = ''</td>
<td>151</td>
<td>12</td>
<td>'apple' 'union select 1, group_concat(table_name), 3,4,5,6 from information_schema.tables -- where table_schema= database()'</td>
<td></td>
</tr>
</tbody>
</table>
Table 5

Comparison of time frame between the proposed system and existing system

<table>
<thead>
<tr>
<th>Query</th>
<th>Existing system</th>
<th>Proposed system</th>
<th>Username</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select from bin sqlia_table_1 where username='apple'; DELETE FROM customers WHERE 1 or username ='' and password =''</td>
<td>251</td>
<td>90</td>
<td>Apple' DELETE FROM customers WHERE 1 or username =''</td>
<td></td>
</tr>
<tr>
<td>Select from bin sqlia_table_1 where username='' OR 1' and password =''</td>
<td>427</td>
<td>90</td>
<td>'OR 1'</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='' OR 1' and password =''</td>
<td>121</td>
<td>110</td>
<td>''; DELETE FROM customers WHERE 1 or username =''</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple'; UPDATE sqlia_table_1 SET password ='jack' WHERE username ='' AND password =''</td>
<td>412</td>
<td>241</td>
<td>'apple'; UPDATE sqlia_table_1 SET password ='jack' WHERE username =''</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple union select null from sqlia_table_1 and password =''</td>
<td>121</td>
<td>10</td>
<td>'apple union select null from sqlia_table_1</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple' DROP TABLE users; SELECT * FROM sqlia_table_1 WHERE 't'='t' and password =''</td>
<td>800</td>
<td>10</td>
<td>Select bin from sqlia_table_1 where username='apple' DROP TABLE users; SELECT * FROM sqlia_table_1 WHERE 't'='t'</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple' union SELECT 1/0 FROM sqlia_table_1 WHERE username ='' AND password =''</td>
<td>201</td>
<td>10</td>
<td>'apple' union SELECT 1/0 FROM sqlia_table_1 WHERE username =''</td>
<td></td>
</tr>
</tbody>
</table>

Securing web applications from SQLIA using progressive detector
<table>
<thead>
<tr>
<th>Query</th>
<th>Existing system</th>
<th>Proposed system</th>
<th>Username</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select bin from sqlia_table_1 where username='anything' or 'x'='x' and password='--'</td>
<td>231</td>
<td>101</td>
<td>'anything' or 'x'='x' --'</td>
<td>'--'</td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple' and password is null; --' and password=''</td>
<td>151</td>
<td>101</td>
<td>'apple' and password is null</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple' AND 1=(SELECT COUNT(*) FROM tabname);--</td>
<td>95</td>
<td>62</td>
<td>'apple' AND 1=(SELECT COUNT(*) FROM tabname);--</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple' AND sqlia_table_1.Email is NULL;--' password=''</td>
<td>251</td>
<td>126</td>
<td>'apple' AND sqlia_table_1.Email is NULL;--' password=''</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple' OR username LIKE '%ppl%'; and password=''</td>
<td>281</td>
<td>126</td>
<td>'apple' OR username LIKE '%ppl%' and password=''</td>
<td></td>
</tr>
<tr>
<td>Select bin from sqlia_table_1 where username='apple' AND password = orange ' and password=''</td>
<td>168</td>
<td>111</td>
<td>Select bin from sqlia_table_1 where username='apple' AND password = orange ' and password=''</td>
<td></td>
</tr>
</tbody>
</table>
Table 6 The total time taken between existing system and the proposed system

<table>
<thead>
<tr>
<th>Web application</th>
<th>No. of attacks</th>
<th>Total time taken by existing system (ms)</th>
<th>Total time taken by proposed system (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web goat</td>
<td>52</td>
<td>24787</td>
<td>6083</td>
</tr>
<tr>
<td>College management</td>
<td>96</td>
<td>52162</td>
<td>17832</td>
</tr>
<tr>
<td>Online examination system</td>
<td>19</td>
<td>6719</td>
<td>1684</td>
</tr>
<tr>
<td>Campus management system</td>
<td>23</td>
<td>7201</td>
<td>1847</td>
</tr>
<tr>
<td>Library management System</td>
<td>82</td>
<td>45256</td>
<td>11987</td>
</tr>
<tr>
<td>Online airline reservation</td>
<td>34</td>
<td>9217</td>
<td>2602</td>
</tr>
<tr>
<td>Management system</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2 Time difference between existing and proposed system (see online version for colours)

It presents time difference between the existing and the proposed system.

Figure 3 The malicious input from a user (see online version for colours)
Figure 3 shows the login page where the user tries to give malicious input to the web application. It shows that a query containing tautology attack is inserted into the enter query field. Figure 4 shows the validation process of PD tool and its outcome. As soon as the tool detects SQLIA vulnerabilities, it immediately prevents the malicious query and displays the message ‘The page could not be displayed’.

**Figure 4** The response, when a malicious URL is given (see online version for colours)

<table>
<thead>
<tr>
<th>Web application</th>
<th>No. of legitimate inputs</th>
<th>False positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web goat</td>
<td>92</td>
<td>0</td>
</tr>
<tr>
<td>College management</td>
<td>125</td>
<td>0</td>
</tr>
<tr>
<td>Online examination</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>Campus management</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>Library management</td>
<td>72</td>
<td>0</td>
</tr>
<tr>
<td>Online airline</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>Management system</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5 Conclusions

This paper presents a new technique for preventing SQLIAs based on the novel concept of progressive detector. It is a technique which visualises a new dimension to fix the various kinds of SQLIA vulnerabilities with minimum time frame. To fix the malicious query, it does not take all the query tokens. Since it scans the query in a progressive fashion, it is performance has increased 60% of execution time than other detectors. The proposed system contributes the following:

1. It scans the important tokens of the input query instead of checking ‘where’ clause queries.
2. There is no need to store the query structure before deployment.
3. Since the proposed system scans the token of query one by one, it avoids unnecessary waste of system resources.
4. It can be used where new tables are created at runtime often.

Also, it can be used where SQL statement may be known only at runtime. The proposed system has been implemented in the progressive detector tool that requires only minimal deployment requirements. The developed tool has been validated in six real time web application environments. The results prevailed clearly indicate that the proposed approach succeeds fully prevents all sort of SQLIA vulnerabilities with minimum time frame without generating any false positives.

References


Securing web applications from SQLIA using progressive detector

Website