# Prevention of SQL Injection Attacks by Using Service Oriented Authentication Technique

Indrani Balasundram and E. Ramaraj

*Abstract*—**Web applications have become steadily increased in daily routines activities and continue to integrate them. On-line reservations, paying bills and on-line shopping expect these web applications to be secure and reliable; the fear of SQL–Injection Attacks has become increasingly frequent and serious. SQL Injection Attacks (SQLIAs) are one of the topmost threats for web application security. Using SQL Injection attackers can leak confidential information; such as credit card numbers from web applications and even corrupt the database. This paper presents a new technique to protect Web applications against SQL injection Attacks. SQL Injection Attacks are a class of attacks that many of these systems are highly vulnerable to, and there is no known foolproof defense against such attacks. The new innovative technique "Service -Oriented Authentication" is to prevent SQL–Injection Attacks in database the deployment of this technique is by appending first level Service has the functionality of Tame-card detection and Prevention. The Second level Service has the functionality of Authentication Checker also dataset (the temporary storage of database) of application scripts additionally allowing seamless integration with currently-deployed systems.**

*Index Terms*—**Database security, world-wide application security, SQL–injection attacks, runtime monitoring.**

## I. Introduction

Attackers have developed a wide array of sophisticated attack techniques that can be used to exploit SQL injection vulnerabilities. In SQL injection attack, an attacker attempts to exploit vulnerabilities in custom Web applications by entering SQL code in an entry field such as a login. If successful, such an attack can give the attacker access to data on the database used by the application and the ability to run malicious code on the Web site. Malicious attacks occur when developers combine hard-coded strings with user provided input to create dynamic queries. Intuitively, if user input is not properly validated, attackers may be able to change the developer's intended SQL command by inserting new SQL keywords or operators through specially crafted input strings [1]. Many enterprise applications deal with sensitive data due to the extraordinary growth of World Wide Web. SQL-Injection Attacks constitute an important class of attack against web applications. Web applications that are vulnerable to SQL attacks user inputs the attacker's embeds commands and gets executed . The attackers directly access the database underlying an application and leak or alter confidential information and execute malicious code. The resulting security violations can include identity theft, loss of confidential information, and ultimately fraud. In some cases, attackers even use an SQL Injection vulnerability to take control and corrupt the system that hosts the Web application. The increasing number of web applications falling prey to these attacks is alarmingly high prevention of SQLIA's is a major challenge. It is difficult to implement and enforce a rigorous defensive coding discipline. Many solutions based on defensive coding address only a subset of the possible attacks. The evaluation of "Service Oriented Authentication Technique" does not have any code modification as well as automation of detection and prevention. In fact; SQLIA's have been included in list of top 10 threats to web applications. U.S. industry regulations such as the Sarbanes-Oxley Act pertaining to information security, try to enforce strict security compliance by application vendors.

### A. *Example of SQL Injection Attack*

#### 1) *Tautology*

A website uses this source (figure: 1), which would be vulnerable to SQLIA. For example, if a user enters"' OR 1=1--" and"", instead of User1 ="indra" and Pass1 = "rani", the resulting query is: SELECT * from FROM data_master WHERE user1='' OR 1=1 --' AND pass1='' The database interprets everything after the WHERE token as a conditional statement, and the inclusion of the "OR 1=1" clause turns this conditional into a tautology. As a result, the database returns the records for all users in the database. An attacker could insert a wide range of SQL commands via this exploit, including commands to modify or destroy database tables.

#### 2) *Union Query*

By using UNION: SELECT query the attacker can retrieve information from a specified table. The result of this attack is that the database returns a dataset that is the union of the results of the original first query and the results of the injected second query. "' UNION SELECT pass1 from data_master where user1 ='indra'- -'into the login field, which produces the following query: SELECT pass1 FROM data_master WHERE user1=' ' UNION SELECT pass1 from data_master where user1='indra'- –' AND pass1='' Assuming that there is no login equal to "", the original first query returns the null set, whereas the second query returns data from the "data_master" table. In this case, the database would return column "pass1" for account "100001". Database takes the results of these two queries, unions them, and returns them to the application. In many applications, the effect of this operation is that the value for "pass1" is displayed along with the account information.

Manuscript received March 9, 2013; revised May 29, 2013.

Indrani Balasundaram and E. Ramaraj are with the Madurai Kamaraj University, Madurai, India (e-mail: indrani.phd@ gmail.com, eramaraj@rediffmail.com).

## II. Realted Work

There are some existing techniques that can be used to detect and prevent input manipulation vulnerabilities

### A. Defense Mechanism

Halfond *et al.* [2] presented an extensive review for the different types of SQL Injection Attacks known up to date. This paper presented the techniques of various attack and shows how the attacks gain information from web application. This work shows the intent of the attacker and Injection mechanism of SQL Injection. This review classified and analyzed existing detection and prevention techniques against SQL Injection Attacks. They provide strengths and weaknesses for each technique in addressing the entire range of SQL Injection Attacks.

Eliminating SQL Injection Attacks Ke Wei [3] shows that static analysis is processed by using SQL Graph representation using Finite State Machine (FSM).

Halfond *et al*. [4] proposed a new highly automated approach for dynamic detection and prevention of SQLIA' s. Intuitively, this work is by identifying "trusted" strings in an application and allowing only these trusted strings to be used to create the semantically relevant parts of a SQL query such as keywords or operators. Dynamic tainting is a mechanism used to implement this approach. The advantage of this approach is highly automated and requires minimal intervention does not require additional infrastructure and can be automatically deployed.

Attacks occur when developers combine hard-coded strings with user-provided input to generate dynamic queries. In AMNESIA[5]built SQL-query models by Non-deterministic Finite State Automation (NDFA) in which the transition labels consist of SQL tokens (SQL keywords and operators), delimiters and place holders for string values.. Intuitively, if user input is not properly validated, the attackers may be able to change the developer's intended SQL command by inserting new SQL keywords or operators through specially crafted input strings.

### B. Prepare Statement

Stephen Thomas *et al.* [6] proposed an automated method for removing SQL Injection Vulnerabilities from JAVA code by converting plain text SQL statements into prepared statements. Prepared statements restrict the way that input can affect the execution of the statement. An automated solution allows developers to remove SQL Injection Vulnerabilities by replacing the vulnerable code with generated secure code. The prepared statement separates the values in a query from the structure of SQL. The programmer defines a skeleton of an SQL query and then fills in the holes of the skeleton at runtime. The prepared statement makes it harder to inject SQL queries because the SQL structure cannot be changed. To use the prepared statement, the web application has to modify entirely; all the legacy web application must be re-written to reduce the possibility of SQL injections

The Software poses a particularly difficult problem because of the cost and complexity of reworking on the existing code. Many techniques [7] rely on complex static analysis in order to find potential vulnerabilities in the code. A recent penetration testing study of more than 250 Web applications concluded that at least 92% of Web applications are vulnerable to some form of malicious intruders.

### C. Instruction – Set Randomization

SQL rand [8] provides a framework that allows developer to create SQL Queries using randomized keywords instead of the normal SQL keywords. A proxy between the web application and the database intercepts SQL queries and de-randomizes the keywords. The SQL keywords injected by an attacker would not have been constructed by the randomized keywords, and thus the injected commands would result in a syntactically incorrect query. Since SQL rand uses a secret key to modify keywords, its security relies on attackers not being able to discover this key. SQL rand requires the application developer to rewrite code.

V. B. Livshits [9] proposed a static analysis approach based on a scalable and precise point to analysis. In this system, user-provided specifications of vulnerabilities are automatically translated into the static analyzers. This approach finds all vulnerabilities matching a specification in the statically analyzed code. Context sensitivity, combined with improved object naming, proved instrumental in keeping the number of false positives low.

### D. Proxy Filter

Scott and Sharp [10] use a proxy to filter input and output data streams for a web application although this technique can be effective against SQLIA; it requires developers to correctly specify filtering rules for each application input. This step of the process is prone to human error and leaves the application vulnerable if the developer has not adequately identified all injection points and correctly expressed the filtering rules. Like defensive coding practices this techniques cannot provide guarantees of completeness and accuracy.

## III. Proposed Technique

This approach addresses SQLIA's with runtime monitoring. The details of this technique is described in subsequent section: (1) the source code contains enough information to infer models of the expected, legitimate SQL queries generated by the application, and (2)By injecting additional SQL statements into a query, would violate the model**.** Proposed technique (Fig. 1) monitors dynamically generated queries with Tame-card detector model and Authentication Checker model at runtime and check them for compliance. If the Data Comparison violates the model then it represents potential SQLIA's and prevented from executing on the database. For each application, when the login page is redirected to our checking page, it was to detect and prevent attacks without stopping legitimate accesses. Moreover, this technique proved to be an efficient, imposing only a low overhead on the Web applications. This technique consists of two levels of filtration to prevent SQLIA'S. First level is Tame-card Detector and the second level is Authentication Checker. The steps are summarized and then describe them in more detail in subsequent sections.

### A. Tame-card Detector

Tame-card detector is a first level filtration method at

Service1 in (Fig. 1). It detects the Wildcard characters or Meta characters from the User input and prevents the malicious attacks

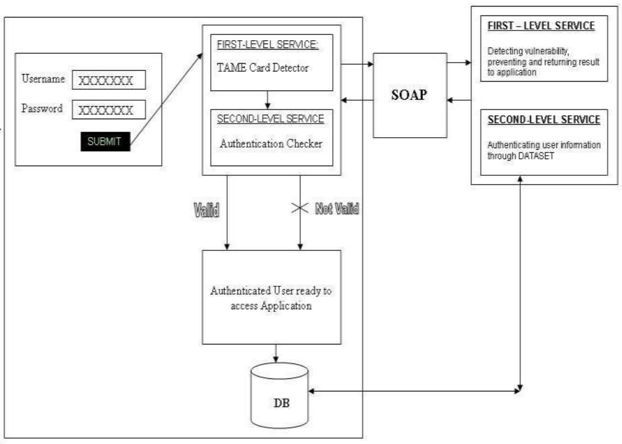


Fig. 1. Proposed architecture of (service- oriented authentication technique)

### B. Authentication Checker

Authentication Checker validates user input from Data-Set where the Sensitive data's are stored from the Database at second level filtration method in Service. The user input fields compare with the data existed in Dataset if it is identical then the legitimate user is allowed to proceed.

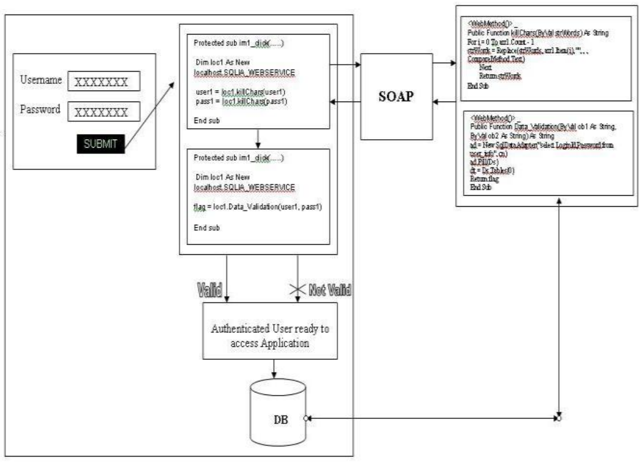
### C. Identify Hotspot

```
Protected sub btnsubmit( )
cn.Open()
cmd = New SqlCommand("select * from data_master where User1='" & t1.Text &"' and Pass1='" & t2.Text & "'", cn)
rd = cmd.ExecuteReader
If rd.Read Then
Response.Redirect("valid_user. aspx")
Else
Response.Redirect("Login_err.aspx")
End If
cn.Close()
cmd.Dispose()
rd.Close()
End sub
```

Fig. 2. Example VB code in .NET application

Scan the application code to identify hotspots points in the application code that issue SQL queries to the underlying database. It performs a simple scanning of the application code to identify hotspots. For example .NET in Fig. 2, the set of hotspots would contain a single element: the statement at line 4. (In .NET based applications, interactions with the database occur through calls to specific methods in the System.Data.Sqlclient namespace, 1 such as Sqlcommand- . ExecuteReader (String))

The injection process works by prematurely terminating a text string and appending a new command. Because the inserted command may have additional strings appended to it before it is executed, the malefactor terminates the injected string with a comment mark "--". Subsequent text is ignored at execution time. The proposed technique contributes a Tame-card Detector, to validate the user input fields to detect the wild card character (patch file) and prevent the wild -card attacker. Transact-SQL statements will be prohibited directly from user input. For each hotspot, build a wildcard model, to check any Wild-card strings or characters append SQL tokens (SQL keywords and operators), delimiters, or string tokens to the legitimate command.



```
Protected sub im1_click(.....)

 Dim loc1 As New
localhost.SQLIA_WEBSERVICE

 user1 = loc1.killChars(user1)
 pass1 = loc1.killChars(pass1)

End sub
```

```
Protected sub im1_click(.....)

 Dim loc1 As New
localhost.SQLIA_WEBSERVICE

flag = loc1.Data_Validation(user1, pass1)

End sub
```

```
<WebMethod()> _
Public Function killChars(ByVal strWords) As String
For i = 0 To arr1.Count - 1
strWork = Replace(strWords, arr1.Item(i), "", , ,
CompareMethod.Text)
    Next
    Return strWork
End Sub
```

```
<WebMethod()> _
Public Function Data_Validation(ByVal ob1 As String,
ByVal ob2 As String) As String
ad = New SqlDataAdapter("select LoginId,Password from
user_info", cn)
ad.Fill(Ds)
dt = Ds.Tables(0)
Return flag
End Sub
```

Fig. 3. Functions generated in tame-card detector, Web service and dataset

### D. Comparison of Data at Runtime Monitoring

In Fig. 1: At runtime, If the user input is given to a web application, input is received from the client side at first level Service and passed through protocol SOAP (Simple Object Access Protocol) to Tame-card Detector which is located in web service such that Web services are typically application programming interfaces (API) or web APIs that can be accessed over a network, such as the Internet, and executed on a remote system hosting the requested services. If there is any Meta character concatenated with user input then the Tame-card Detector check the dynamically generated queries to prevent the malicious user. If the pattern matching is identical with the patch file then it is termed as a legitimate user, through the SOAP protocol the validated data is sent back to the client side at Service1. From first level Service the validated User input is sent to second level of Service in client side, through the SOAP protocol to Authentication Checker which is located in web service. Authentication Checker at second level Service also consists of Dataset which is a collection of data usually presented in tabular form. Each column represents a particular variable. Each row corresponds to a given member of the data set in question its values for each of the variables, such as height and weight of an object or values of random numbers. Each value is known as a datum. The data set may comprise data for one or more members, corresponding to the number of rows. This is used to store the sensitive data from the valid database so the validated data is compared with the sensitive data's stored in the dataset. The validations occur in dataset with out affecting directly to the database. The user input and sensitive data are identical then legitimate data will be back to second level Service from the web service level to client side and allowed to access the web application and hence the invalid input will be discarded. Here there is no Query validation occurs only the data will be validated in Dataset. If the script builds an SQL query by concatenating hard-coded strings together with a string entered by the user, As long as injected SQL code is syntactically correct, tampering cannot be detected programmatically. String concatenation is the primary point of entry for script injection Therefore; all user inputs are carefully validated with Authentication Checker (Second filtration method). If the user input and Sensitive data's are identical then executes constructed SQL commands in the Application server. Fig. 3 shows that the application architecture level of Service Oriented Authentication Technique.

## IV. Evaluation

The proposed technique is deployed and tried few trial runs on the web server

### A. *SQLIA Prevention Accuracy*

Both the protected and unprotected web Applications are tested using different types of SQLIA's; namely use of Tautologies, Union, Piggy-Backed Queries, Inserting additional SQL statements, Second-order SQL injection and various other SQLIA s. Table I shows that the proposed technique prevented all types of SQLIA s in all cases. The proposed technique is thus a secure and robust solution to defend against SQLIA's.

TABLE I: SQLIA'S PREVENTION ACCURACY

| SQLINJECTION TYPES | UN PROTECTED | PROTECTED |
|---|---|---|
| 1. TAUTOLOGIES | NOT PREVENTED | PREVENTED |
| 2.PIGGY BACKED QUERIES | NOT PREVENTED | PREVENTED |
| 3. ADDITIONAL SQL - STATEMENT | NOT PREVENTED | PREVENTED |
| 4. SECOND - ORDER | NOT PREVENTED | PREVENTED |
| 5. UNION | NOT PREVENTED | PREVENTED |

### B. *Execution Time at Runtime Validation*

The runtime validation incurs some overhead in terms of execution time at both the Service Oriented Authentication Technique and SQL-Query based Validation Technique. Taken a sample website E-Transaction measured the extra computation time at the query validation, this delay has been amplified in the graph (Fig. 4 and Fig. 5) to distinguish between the Time delays using bar chart shows that the data validation in dataset performs better than query validation. In Query validation(Fig. 5) the user input is generated as a query in script engine then it gets parsed in to separate tokens then the user input is compared with the statistical generated data if it is malicious generates error reporting. Service Oriented Authentication technique (Fig. 4) states that user input is generated as a query in script engine then it gets parsed in to separate tokens, and send through the protocol SOAP to Tame-card Detector, then the validated user input is sequentially send the user input field to Authentication Checker through the protocol SOAP then the user input is compared with the sensitive data, which is temporarily stored in dataset. If it is malicious data, it will be prevented otherwise the legitimate data is allowed to access the Web application. Existing techniques directly allows accessing the database in database server after the Query validation. Current technique does not allow directly to access database in database server because here, the Service Oriented Authentication Technique using dataset the sensitive data's from database and compare with the user input if user data is legitimate then it allows accessing the database in database server.

## V. Discussion

Proposed technique was able to correctly identify the malicious user as SQLIA's, while allowing all legitimate queries to be performed and no false positives and no false negatives are generated. The results of our study may be related to the specific subjects considered and may not generalize to other web applications. To minimize this risk, Service Oriented Technique uses a set of web applications and an extensive set of realistic attacks. Therefore, although this strategy helps to eliminate certain attacks such as SQL Injection, it will not work against attacks such as cross-site scripting, where sanitization requires escaping a different set of characters according to HTML character entity references.

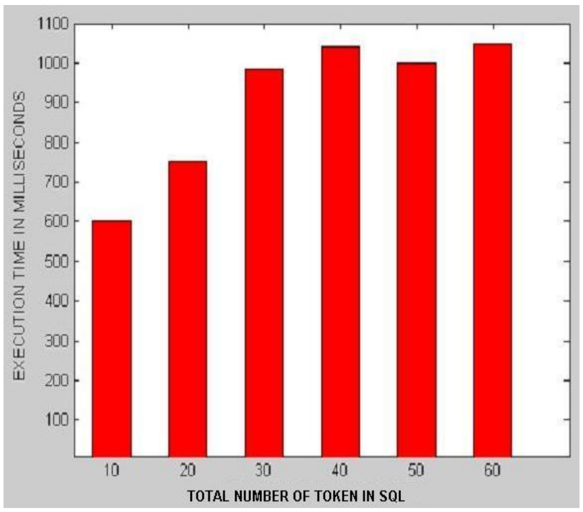


Fig. 4. Execution time based on dataset

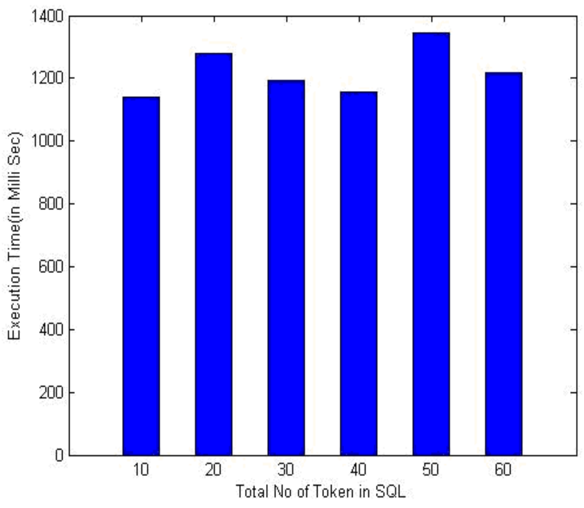


Fig. 5. Execution time based on SQL queries

## VI. Conclusion

The SQL - Injection Attacks are extremely dangerous in comparison to other types of Web-based attacks, because the end result is data manipulation. SQL injection holes can be easy to exploit, a novel technique against SQLIA's. The web-application code implicitly contains a policy that allows distinguishing legitimate and malicious queries. This technique is used to detect and prevent the SQLI flaw (wild characters & exploiting SQL commands) in Tame-card Detector and prevent the Tame-card attacker Transact-SQL statements will be prohibited directly from user input. Service Oriented Authentication technique checks the user input with valid database which is stored separately in dataset and do not affect database directly then the validated user input field is allowed to access the web application as well as used to improve the performance of the server side validation This proposed technique was able to correctly identify the attacks that we performed on the applications without blocking legitimate accesses to the database (i.e., the technique produced neither false positives nor false negatives). These results show that our technique represents a promising approach to countering SQLIA's and motivate further work in this direction.

## References

[1] Top Ten Most Critical Web Application Vulnerabilities. OWASP Foundation. (2005). [Online]. Available: http://www.owasp.org/documentation/ topten.html

[2] W. G. H. fond, "A Classification of SQL-Injection Attacks and Countermeasures," in *Proc. of the Intern. Symposium on Secure Software Engineering*, pp. 1-11, Mar. 2006.

[3] M. Muthuprasanna, "Eliminating SQL Injection Attacks - A Transparent Defense Mechanism," *IEEE International Workshop on, 8th IEEE Inter. Symposium on Web Site Evolution (WSE'06)*, pp. 22-32, 2006.

[4] W. G. J. H. Fond, "WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation,*" IEEE Trans. on Software Engineering*, vol. 34, Issue. 1, pp. 65-81, 2006.

[5] W. G. J. Halfond, "AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks," in *International Proc. 20th IEEE/ACM international Conference on Automated software engineering (ASE '05)*, pp. 174-183, New York, NY, USA, 2005.

[6] S. Thomas, "Using Automated Generation to secure SQL statements," *3rd International Workshop on software engineering for secure systems*, pp. 9, 2007.

[7] G. Wassermann and Z. Su, "A static Analysis Framework for Security in Web Applications," in *Proc. of the FSE Workshop on Specification and Verification of Component-Based Systems* , pp. 70–78, 2004.

[8] S. Boyd, "SQLrand: Preventing SQL - Injection Attacks," in *proc. of the applied cryptography and network security (ANCS)*, pp. 292-304, 2004.

[9] V. B. Livshits, "Finding Security Errors in Java Programs with Static Analysis," in *Proc. of the 14th Use nix Security Symposium*, pp. 271–286, 2005.

[10] D. Scott and R. Sharp, "Abstracting Application-level Web Security," in *Proc. of the 11th International Conference on the World Wide Web (WWW 2002*), pp. 396–407, 2002.

**B. Indrani** received the B.Sc. degree in Computer Science, in 2002; the M.Sc. degree in Computer Science and Information Technology, in 2004. She had completed M.Phil. in Computer Science. She worked as a research assistant in Smart and Secure Environment Lab under IIT, Madras. Her current research interests include Database Security

**E. Ramaraj** is presently working as a Technology Advisor, Madurai Kamaraj University, Madurai, Tamilnadu, India on lien from Director, computer centre at Alagappa university, Karaikudi. He has 22 years teaching experience and 8 years research experience. He has presented research papers in more than 50 national and international conferences and published more than 55 papers in national and international journals. His research areas include Data mining, software engineering, database and network security.