SQL Injection Detection and Prevention Tools Assessment

Atefeh Tajpour
CASE Center University Technology Malaysia Kuala Lumpur, Malaysia
tajpour81sn@yahoo.com

Maslin Masrom
CASE Center University Technology Malaysia Kuala Lumpur, Malaysia
maslin@ic.utm.my

Mohammad Zaman Heydari
IT& Management Dep UCSI University Kuala Lumpur, Malaysia
mz-heydari@yahoo.com

Suhaimi Ibrahim
CASE Center UTM University Kuala Lumpur, Malaysia
suhaimi@case.utm.my

Abstract—**SQL Injection Attacks (SQLIAs) is one of the most serious threats to the security of database driven applications. In fact, it allows an attacker to gain control over the database of an application and consequently, an attacker may be able to alter data. Many surveys have addressed this problem. Also some researchers have proposed different approaches to detect and prevent this vulnerability but they are not successful completely. Moreover, some of these approaches have not implemented yet and users would be confused in choosing an appropriate tool. In this paper we present all SQL injection attack types and also different tools which can detect or prevent these attacks. Finally we assessed addressing all SQL injection attacks type among current tools.**

Keyword: SQL Injection Attacks, detection, prevention, tool,assessment.

I. INTRODUCTION

In recent years, most of our daily tasks are depend on database driven web applications because of increasing activity, such as banking, booking and shopping. For performing activities such as ordering the merchandize or paying the bills, information must be trustable to these web applications and their underlying databases but unfortunately there is no any guarantee for confidentiality and integrity of this information. Web applications are often vulnerable to attacks, which can give attackers easily access to the application's underlying database.

The Structural Query Language Injection (SQLI) attack occurs when an attacker changes the logic, semantics or syntax of a SQL query by inserting new SQL keywords or operators. SQL Injection Attack is a class of code injection attacks that happens when there is no input validation. In fact, attackers can shape their illegitimate input as parts of final query string which operate by databases. Financial web applications or secret information systems could be the victims of this vulnerability because attackers by abusing this vulnerability can threat their authority, integrity and confidentiality. So, developers addressed some defensive coding practices to eliminate this vulnerability but they are not sufficient. SQLIAs can also escape traditional tools such as firewalls and Intrusion Detection Systems (IDSs) because they performed through ports used for regular web traffic which usually are open in firewalls. On the other hand, most IDSs focus on the network and IP layers whereas SQLIAs work at application layer. Researchers have proposed a range of techniques and tools to help developers to compensate the shortcoming of the defensive coding [14, 15, 16]. The problem is that some current techniques and tools are impractical in reality because they could not address all attack types or have not been implemented yet.

The paper is organized as follows. In section 2 we define SQL Injection attack. In section3 we present different SQLI attack types. In section 4 we review current tools against SQLI. In section 5 we assessed addressing all SQL injection attacks type among current SQL injection detection or prevention tools. Conclusion and future work are provided in section 6.

II. DEFINITION OF SQLIA

SQL injection is a type of attack which the attacker adds Structured Query Language code to input box of a web form to gain access or make changes to data. SQL injection vulnerability allows an attacker to flow commands directly to a web application's underlying database and destroy functionality or confidentiality.

III. SQL INJECTION ATTACK TYPES

There are different methods of attacks that depending on the goal of attacker are performed together or sequentially. For a successful SQLIA the attacker should append a syntactically correct command to the original SQL query. Now the following classification of SQLIAs in accordance to [4, 11] will be presented.

Tautologies: This type of attack injects SQL tokens to the conditional query statement to be evaluated always true. This type of attack used to bypass authentication control and access to data by exploiting vulnerable input field which use WHERE clause.

"SELECT * FROM employee WHERE userid = '112' and password ='aaa' OR '1'='1'"

As the tautology statement (1=1) has been added to the query statement so it is always true.

978-1-4244-5540-9/10/$26.00 ©2010 IEEE

Illegal/Logically Incorrect Queries: when a query is rejected , an error message is returned from the database including useful debugging information. This error messages help attacker to find vulnerable parameters in the application and consequently database of the application. In fact attacker injects junk input or SQL tokens in query to produce syntax error, type mismatches, or logical errors by purpose. In this example attacker makes a type mismatch error by injecting the following text into the *pin* input field:

1) Original URL:
http://www.arch.polimi.it/eventi/?id_nav=8864
2) SQL Injection:
http://www.arch.polimi.it/eventi/?id_nav=8864'
3) Error message showed:
SELECT name FROM Employee WHERE id =8864\'

From the message error we can find out name of table and fields: name; Employee; id. By the gained information attacker can organize more strict attacks.

Union Query*:* By this technique, attackers join injected query to the safe query by the word UNION and then can get data about other tables from the application.

Suppose for our examples that the query executed from the server is the following:

SELECT Name*, Phone FROM Users WHERE Id=$id*
By injecting the following Id value:
$id=1 UNION ALL SELECT credit Card Number,1 FROM Credit CarTable
We will have the following query:
SELECT Name, Phone FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1 FROM Credit Car Table

which will join the result of the original query with all the credit card users.

Piggy-backed Queries: In this type of attack, intruders exploit database by the query delimiter, such as ";", to append extra query to the original query. With a successful attack database receives and execute a multiple distinct queries. Normally the first query is legitimate query, whereas following queries could be illegitimate. So attacker can inject any SQL command to the database. In the following example, attacker inject " 0; *drop table user* " into the *pin* input field instead of logical value. Then the application would produce the query:

SELECT info FROM *users WHERE login='doe' AND pin=0; drop table users*

Because of ";" character, database accepts both queries and executes them. The second query is illegitimate and can drop *users* table from the database. It is noticeable that some databases do not need special separation character in multiple distinct queries, so for detecting this type of attack, scanning for a special character is not impressive solution.

Stored Procedure: Stored procedure is a part of database that programmer could set an extra abstraction layer on the database. As stored procedure could be coded by programmer, so, this part is as inject able as web application forms. Depend on specific stored procedure on the database there are different ways to attack. In the following example, attacker exploits parameterized stored procedure.

CREATE PROCEDURE *DBO .is Authenticated*
@user Name varchar2, @pass varchar2, @pin int
AS
EXEC("SELECT accounts FROM users
WHERE login='" +@user Name+ "' and pass='" +@password+
"' and pin=" +@pin);
GO

For authorized/unauthorized user the stored procedure returns true/false. As an SQLIA, intruder input " ' ; SHUTDOWN; - -" for username or password. Then the stored procedure generates the following query:

SELECT accounts FROM users WHERE login='doe' AND pass=' '; SHUTDOWN; -- AND pin=

After that, this type of attack works as piggy-back attack. The first original query is executed and consequently the second query which is illegitimate is executed and causes database shut down. So, it is considerable that stored procedures are as vulnerable as web application code.

Inference: By this type of attack, intruders change the behaviour of a database or application.There are two well-known attack techniques that are based on inference: blind-injection and timing attacks.

Blind Injection: Sometimes developers hide the error details which help attackers to compromise the database. In this situation attacker face to a generic page provided by developer, instead of an error message. So the SQLIA would be more difficult but not impossible. An attacker can still steal data by asking a series of True/False questions through SQL statements. Consider two possible injections into the login field:

SELECT accounts FROM users WHERE login='doe' and 1=0 -- AND pass= AND pin=0
SELECT accounts FROM users WHERE login='doe' and 1=1 -- AND pass= AND pin=0

If the application is secured, both queries would be unsuccessful, because of input validation. But if there is no input validation, the attacker can try the chance. First the attacker submit the first query and receives an error message because of "*1=0*". So the attacker does not understand the error is for input validation or for logical error in query. Then the attacker submits the second query which always true. If there is no login error message, then the attacker finds the login field vulnerable to injection.

Timing Attacks: A timing attack lets an attacker gather information from a database by observing timing delays in the database's responses. This technique by using if-then statement cause the SQL engine to execute a long running query or a time delay statement depending on the logic injected. This attack is similar to blind injection and attacker can then measure the time the page takes to load to determine if the injected statement is true. This technique uses an if-then statement for injecting queries. WAITFOR is a keyword along the branches, which causes the database to delay its response by a specified time.

For example, in the following query:
declare @s varchar(8000) select @s = db_name() if (ascii(substring(@s, 1, 1)) & (power(2, 0))) > 0 waitfor delay '0:0:5'

Database will pause for five seconds if the first bit of the first byte of the name of the current database is 1. Then code is then injected to generate a delay in response time when the condition is true. Also, attacker can ask a series of other questions about this character. As these examples show, the information is extracted from the database using a vulnerable parameter.

Alternate Encodings: In this technique, attackers modify the injection query by using alternate encoding, such as hexadecimal, ASCII, and Unicode. Because by this way they can escape from developer's filter which scan input queries for special known "bad character". For example attacker use *char* (44) instead of single quote that is a bad character. This technique with join to other attack techniques could be strong, because it can target different layers in the application so developers need to be familiar to all of them to provide an effective defensive coding to prevent the alternate encoding attacks. By this technique, different attacks could be hidden in alternate encodings successfully.

In the following example the *pin* field is injected with this string: *"0; exec (0x73587574 64 5f77 6e),"* and the result query is:
SELECT accounts FROM users WHERE login=" AND pin=0; exec (char(0x73687574646f776e))

This example use the *char* () function and ASCII hexadecimal encoding. The *char* () function takes hexadecimal encoding of character(s) and returns the actual character(s). The stream of numbers in the second part of the injection is the ASCII hexadecimal encoding of the attack string. This encoded string is translated into the *shutdown* command by database when it is executed.

IV. SQL INJECTION DETECTION AND PREVENTION TOOLS

Although developers deploy defensive coding or OS hardening but they are not enough to stop SQLIAs to web applications so researchers have proposed some of tools to assist developers. It is noticeable that there are more approaches that have not implemented as a tool yet. This paper emphasizes on tool not technique.

Huang and colleagues [18] propose WAVES, a black-box technique for testing web applications for SQL injection vulnerabilities. The tool identify all points a web application that can be used to inject SQLIAs. It builds attacks that target these points and monitors the application how response to the attacks by utilize machine learning.

JDBC-Checker [12, 13] was not developed with the intent of detecting and preventing general SQLIAs, but can be used to prevent attacks that take advantage of type mismatches in a dynamically-generated query string. As most of the SQLIAs consist of syntactically and type correct queries so this technique would not catch more general forms of these attacks.

CANDID [2, 7] modifies web applications written in Java through a program transformation. This tool dynamically mines the programmer-intended query structure on any input and detects attacks by comparing it against the structure of the actual query issued. CANDID's natural and simple approach turns out to be very powerful for detection of SQL injection attacks.

In SQL Guard [10] and SQL Check [5] queries are checked at runtime based on a model which is expressed as a grammar that only accepts legal queries. SQL Guard examines the structure of the query before and after the addition of user-input based on the model. In SQL Check, the model is specified independently by the developer. Both approaches use a secret key to delimit user input during parsing by the runtime checker, so security of the approach is dependent on attackers not being able to discover the key. In two approaches developer should to modify code to use a special intermediate library or manually insert special markers into the code where user input is added to a dynamically generated query.

AMNESIA combines static analysis and runtime monitoring [16, 17]. In static phase, it builds models of the different types of queries which an application can legally generate at each point of access to the database. Queries are intercepted before they are sent to the database and are checked against the statically built models, in dynamic phase. Queries that violate the model are prevented from accessing to the database. The primary limitation of this tool is that its success is dependent on the accuracy of its static analysis for building query models.

WebSSARI [15] use static analysis to check taint flows against preconditions for sensitive functions. It works based on sanitized input that has passed through a predefined set of filters. The limitation of approach is adequate preconditions for sensitive functions cannot be accurately expressed so some filters may be omitted.

SecuriFly [14] is another tool that was implemented for java. Despite of other tool, chases string instead of character for taint information. SecurityFly tries to sanitize query strings that have been generated using tainted input but unfortunately injection in numeric fields cannot stop by this approach. Difficulty of identifying all sources of user input is the main limitation of this approach.

Positive tainting [1] not only focuses on positive tainting rather than negative tainting but also it is automated and does need developer intervention. Moreover this approach benefits from syntax-aware evaluation, which gives developers a mechanism to regulate the usage of string data based not only on its source, but also on its syntactical role in a query string.

IDS [6] use an Intrusion Detection System (IDS) to detect SQLIAs, based on a machine learning technique. The technique builds models of the typical queries and then at runtime, queries that do not match the model would be identified as attack. This tool detects attacks successfully but

it depends on training seriously. Else, many false positives and false negatives would be generated.

Another approach in this category is SQL-IDS [8] which focus on writing specifications for the web application that describe the intended structure of SQL statements that are produced by the application, and in automatically monitoring the execution of these SQL statements for violations with respect to these specifications.

SQLPrevent [11] is consists of an HTTP request interceptor. The original data flow is modified when SQLPrevent is deployed into a web server. The HTTP requests are saved into the current thread-local storage. Then, SQL interceptor intercepts the SQL statements that are made by web application and pass them to the SQLIA detector module. Consequently, HTTP request from thread-local storage is fetched and examined to determine whether it contains an SQLIA. The malicious SQL statement would be prevented to be sent to database, if it is suspicious to SQLIA.

Swaddler [3], analyzes the internal state of a web application. It works based on both single and multiple variables and shows an impressive way against complex attacks to web applications. First the approach describes the normal values for the application's state variables in critical points of the application's components. Then, during the detection phase, it monitors the application's execution to identify abnormal states.

V. ASSESSING THE SQL INJECTION ATTACK TYPES AMOUNG THE CURRENT TOOLS

In this section, the SQL injection detection or prevention tools presented in section IV and their ability to address SQLI attack types are considered. It is noticeable that this comparison is based on the articles not empirically experience.

Tables 1 summarize the results of this comparison. The symbol "•" is used for tool that can successfully stop all attacks of that type. The symbol "-" is used for tool that is not able to stop attacks of that type. The symbol "°" refers to tool that stop the attack type only partially because of natural limitations of the underlying approach.

As the table shows the stored procedure is a critical attack which is difficult for some tools to stop it. It is consisting of queries that can execute on the database. However, most of tools consider only the queries that generate within application. So, this type of attack make serious problem for some tools.

TABLE I. COMPARISON OF SQLI DETECTION/PREVENTION TOOLS WITH RESPECT TO ATTACK TYPES

	Attack \ Tool	SQL IDS[8]	Swaddler[3]	IDS[6]	CANDID[7]	AMNESIA[16]	SQL Check[5]	SQL Guard[10]	JDBC-Checker[12]	WebSSARI[15]	SecuriFLY[14]	WAVES[18]	Positive Tainting[1]	SQLPrevent[11]
1	Tautologies	•	°	°	°	•	•	•	°	•	°	°	•	•
2	Illegal/ Incorrect	•	°	°	°	•	•	•	°	•	°	°	•	•
3	Piggy-back	•	°	°	°	•	•	•	°	•	°	°	•	•
4	Union	•	°	°	°	•	•	•	°	•	°	°	•	•
5	Stored Proc	•	°	°	°	-	-	-	°	•	°	°	•	•
6	Infer	•	°	°	°	•	•	•	°	•	°	°	•	•
7	Alter Encodings	•	°	°	°	•	•	•	°	•	°	°	•	•

Table 2, illustrates the addressing percentage of SQL Injection attacks among SQL Injection detection or prevention tools. The percentage of tools that stop Tautology is calculated by this formula:

$$\frac{\text{Number of tools that can stop Tautology}}{\text{Total number of tools}} \times 100 = \frac{7}{13} \times 100 = 54$$

TABLE II. PERCENTAGE OF TOOLS THAT DETECT OR PREVENT SQL INJECTION ATTACKS

	Attack Type	Tools that can stop all attacks of that type(•)	Tools that address the attack type only partially(°)	Tools that cannot stop attacks of that type(-)
1	Tautologies	54%	46%	-
2	Illegal/ Incorrect	54%	46%	-
3	Piggy-back	54%	46%	-
4	Union	54%	46%	-
5	Stored Proc	31%	46%	23%
6	Infer	54%	46%	-
7	Alter Encodings	54%	46%	-

It is significant that almost all types of attack have steadily reaction from the tools except Stored Procedure which cannot be stopped by some tools. On the other hand, only 54% of current tools can stop Tautologies, Illegal/Incorrect, Piggy back, Union, Inference and Alter Encoding completely and 46% of current tool can stop these attacks partially. Actually there is not disappointment for tools to address these types of attacks. It means the tools can address somewhat of these types of attack. Whereas, stored procedure has a different result in this table. It is unfortunate that only 31% of tools could stop this type of attack successfully. Moreover, it is evident that 23% of tools cannot stop Stored Procedure at all.

VI. CONCLUSION AND FUTURE WORK

In this paper, we assessed detecting and preventing SQLIAs among current SQL Injection detection and/or prevention tools. We first identified the various types of SQLIAs. Then we investigated SQL injection detection and prevention tools. After that we compared these tools in terms of their ability to stop SQLIA. Regarding the results some current tools' ability should be improved for stopping SQLI attacks.

In our future work we will compare SQL Injection detection or prevention tools based on deployment requirements.

REFERENCES

[1] William GJ. Halfond, Alessandro Orso, Using Positive Tainting and Syntax Aware Evaluation to Counter SQL Injection Attacks, 14th ACM SIGSOFT international symposium on Foundations of software engineering 2006, ACM. pp: 175 – 185.

[2] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations, 2007, Alexandria, Virginia, USA, ACM.

[3] Marco Cova, Davide Balzarotti. Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID), (Queensland, Australia), September 5-7, 2007, pp. 63-86.

[4] W. G. Halfond, J. Viegas and A. Orso, "A Classification of SQL Injection Attacks and Countermeasures," College of Computing Georgia Institute of Technology IEEE, 2006.

[5] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In The 33rd Annual Symposium on Principles of Programming Languages (POPL 2006), Jan. 2006.

[6] F. Valeur, D. Mutz, and G. Vigna. A Learning-Based Approach to the Detection of SQL Attacks. In Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), Vienna, Austria, July 2005.

[7] Prithvi Bisht, P. Madhusudan. CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks. Proceedings of the 14th ACM Conference on Computer and Communications Security. 2007. USA: ACM, pp 1–38.

[8] Konstantinos Kemalis and Theodoros Tzouramanis. SQL-IDS: A Specification-based Approach for SQL Injection Detection Symposium on Applied Computing. 2008, pp: 2153-2158 , Fortaleza, Ceara, Brazil. New York, NY, USA: ACM.

[9] A. S. Christensen, A. Molle r, and M. I. Schwartzbach. Precise Analysis of String Expressions. In Proc. 10th International Static Analysis Symposium, SAS '03, volume 2694 of LNCS, pp 1-18. Springer-Verlag, June 2003.

[10] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using Parse Tree Validation to Prevent SQL Injection Attacks. In International Workshop on Software Engineering and Middleware (SEM), 2005.

[11] F.Monticelli., PhD SQLPrevent thesis. University of British Columbia (UBC) Vancouver, Canada.2008.

[12] C. Gould, Z. Su, and P. Devanbu. JDBC Checker:A Static Analysis Tool for SQL/JDBC Applications. In Proceedings of the 26th International Conference on Software Engineering (ICSE 04) Formal Demos, pp 697–698, 2004.

[13] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In Proceedings of the 26th International Conference on Software Engineering (ICSE 04), pp645–654, 2004.

[14] M. Martin, B. Livshits, and M. S. Lam. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In Proceedings of the 20th Annual ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA 2005), pp 365–383, 2005.

[15] Y. 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 Proceedings of the 12th International World Wide Web Conference (WWW 04), May 2004.

[16] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005), Long Beach, CA, USA, Nov 2005.

[17] W. G. Halfond and A. Orso. Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks. In Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005), pp 22–28, St. Louis, MO, USA, May 2005.

[18] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In Proceedings of the 11th International World Wide Web Conference (WWW 03), May 2003.