Evaluation of SQL Injection Detection and Prevention Techniques

Conference Paper · July 2010

DOI: 10.1109/CICSyN.2010.55 · Source: DBLP

Atefeh Tajpour
Universiti Teknologi Malaysia

2010 Second International Conference on Computational Intelligence, Communication Systems and Networks

Evaluation of SQL Injection Detection and Prevention Techniques

Atefeh Tajpour
Centre for Advanced Software Engineering (CASE)
University Technology Malaysia
Kuala Lumpur, Malaysia
tajpour81sn@yahoo.com

Mohammad JorJor zade Shooshtari
Centre for Advanced Software Engineering (CASE)
University Technology Malaysia
Kuala Lumpur, Malaysia
mohammadjorg@yahoo.com

***Abstract*—Database driven web application are threaten by SQL Injection Attacks (SQLIAs) because this type of attack can compromise confidentiality and integrity of information in databases. Actually, an attacker intrudes to the web application database and consequently, access to data. For stopping this type of attack different approaches have been proposed by researchers but they are not enough because usually they have limitations. Indeed, some of these approaches have not implemented yet and also most of implemented approaches cannot stop all type of attacks. In this paper all type of SQL injection attack and also different approaches which can detect or prevent them are presented. Finally we evaluate these approaches against all types of SQL injection attacks and deployment requirements.**

Keyword: SQL Injection Attacks, detection, prevention, evaluation, technique.

I. INTRODUCTION

Web applications are often vulnerable to attacks, which can give attackers easily access to the application's underlying database. SQL injection attack occurs when a malicious user, through specifically crafted input, causes a web application to generate and send a query that functions differently than the programmer intended.

SQL Injection Attacks (SQLIAs) have known as one of the most common threats to the security of database-driven applications. So there is not enough assurance for confidentiality and integrity of this information. SQLIA is a class of code injection attacks that take advantage of lack of user 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.

For preventing the SQLIAs defensive coding has offered as a solution but it is very difficult. Not only developers try to put some controls in their source code but also attackers continue to bring some new ways to bypass these controls. Hence it is difficult to keep developers up to date, according the last and the best defensive coding practices. On the other hand, implementing of best practice of defensive coding is very difficult and need to special skills and also erring. These problems motivate the need for a solution to the SQL injection problem.

Researchers have proposed some techniques to help developers to compensate the shortcoming of the defensive coding [7, 10, 12]. The problem is that some current techniques could not address all attack types or some of them need special deployment requirements. Also some of them have not 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 approaches against SQLI. In section 5 we evaluate SQL Injection detection or/and prevention techniques against all types of SQL injection attacks and deployment requirements. Conclusion and future work are provided in section 6.

II. OVERVIEW OF SQL INJECTION ATTACK

A. 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.

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 the Halfond, Viegas, and Orso researches [4] will be presented.

Tautologies: This type of attack injects SQL tokens to the conditional query statement to be evaluated always true.

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.

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.

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.

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.

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.

• **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.

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.

III. SQL INJECTION DETECTION AND PREVENTION TECHNIQUES

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 techniques to assist developers.

Static

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.

Wassermann and Su propose Taoutology Cheker [23] that uses static analysis to stop tautology attack. The important limitation of this technique is that its scope is limited to tautology and cannot detect or prevent other types of attacks.

Xiang Fu and Kai Qian [28] proposed the design of a static analysis framework, called SAFELI for identifying SQLIA vulnerabilities at compile time. SAFELI statically monitor the MSIL (Microsoft Symbolic intermediate language) byte code of an ASP.NET Web application, using symbolic execution. SAFELI can analyze the source code and will be able to identify delicate vulnerabilities that cannot be discovered by black-box vulnerability scanners. The main drawback of this technique is that this approach can discover the SQL injection attacks only on Microsoft based product.

compare points code
→ static & dynamic
→ language of webapp
→ which prevention solve which attack

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.

hybrid

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.

hybrid

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.

Livshits and Lam [27] use static analysis techniques to detect vulnerabilities in software. Java Static Tainting uses information flow techniques to detect when tainted input has been used to make a SQLIA. The primary limitation of this approach is that it can detect only known patterns of SQLIAs and it can generate a relatively high amount of false positives because it uses a conservative analysis.

hybrid ??

Java Dynamic Tainting [21] and SecuriFly [14] is another tool that was implemented for java. Despite of other tool, chase string instead of character for taint information and try 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.

hybrid ??

Two similar approaches by Nguyen-Tuong [20] and Pietraszek [19], modify a PHP interpreter to track precise per-character taint information. A context sensitive analysis is used to detect and reject queries if certain types of SQL tokens has been constructed by illegitimate input. Limitation of these two approaches is that they require rewriting code.

Two approaches, SQL DOM [25] and Safe Query Objects [24], use database queries encapsulation for trustable access to databases. They use a type-checked API which cause query building process be systematic. Consequently by API they apply coding best practices such as input filtering and strict user input type checking. The drawback of the approaches is that developer should learn new programming paradigm or query-development process.

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.

dynamic

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.

hybrid

A proxy filtering system that intensifies input validation rules on the data flowing to a Web application is called Security Gateway [26]. In this technique for transferring parameters from webpage to application server, developers should use Security Policy Descriptor Language (SPDL). So developer should know which data should be filtered and also what patterns should apply to the data.

static

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.

dynamic

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.

hybrid

In SQLrand [22] instead of normal SQL keywords developers create queries using randomized instructions. In this approach a proxy filter intercepts queries to the database and de-randomizes the keywords. By using the randomized instruction set, attacker's injected code could not have been constructed. As it uses a secret key to modify instructions, security of the approach is dependent on attacker ability to seize the key. It requires the integration of a proxy for the database in the system same as developer training.

IV. Evaluation

In this section, the SQL injection detection or prevention techniques presented in Section IV would be compared. In fact the attack types which each technique is able to address are considered as well as deployment requirements.

A. *Comparison of SQL Injection Detection/Prevention Techniques Based on Attack Types*

Proposed techniques were compared to assess whether it was capable of addressing the different attack types presented in Section III. 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 technique that can successfully stop all attacks of that type. The symbol "-" is used for technique that is not able to stop attacks of that type. The symbol "◦" refers to technique that stop the attack type only partially because of natural limitations of the underlying approach.

As the table shows the Stored Procedure and Alternate Encoding are critical attacks which are difficult for some techniques to stop them. Stored Procedure is consisting of queries that can execute on the database. However, most of techniques consider only the queries that generate within application. So, this type of attack make serious problem for some techniques.

B. *Comparison of SQL Injection Detection/Prevention techniques Based on Deployment Requirement*

Each technique with respect to the following criteria was evaluated: (1) Does the technique require developers to modify their code base? (2) What is the degree of automation of the detection aspect of the technique? (3) What is the degree of automation of the prevention aspect of the technique? (4) What infrastructure (not including the technique itself) is needed to successfully use the technique? The results of this classification are summarized in Table 2.

Table2 determines the degree of automation of technique in detection or prevention of attacks and also it could be cleared that which technique needs to modify the source code of application. Moreover, additional infrastructure that is required for each technique is illustrated.

Table 1 Comparison of SQLI Detection/Prevention Techniques with Respect to Attack Types

	Techniques / Attack Types	Detective													Preventive									
		SQL_IDS(8)	Swaddler(3)	Web application Hardennig (20)	SAFELI (28)	IDS (6)	JAVA Dynamic Tainting (21)	CANDID (7)	CSSE (19)	AMNESIA (16)	SQL Check (5)	SQL Guard (10)	SQL rand (22)	Tautology checker (23)	JDBC-Checker (12)	WebSSARI (15)	Safe QUERY (24)	GateWay (26)	SecuriFLY (14)	SQL DOM (25)	WAVES (18)	Java Static Tainting (27)	SQLPrevent (11)	Positie Tainting (1)
1	Taut	•	◦	•	-	◦	◦	◦	•	•	•	•	•	•	◦	•	•	◦	◦	•	◦	•	•	•
2	Illegal/ Incorrect	•	◦	•	•	◦	◦	◦	•	•	•	•	-	-	◦	•	•	◦	◦	•	◦	•	•	•
3	Piggy-back	•	◦	•	•	◦	◦	◦	•	•	•	•	•	-	◦	•	•	◦	◦	•	◦	•	•	•
4	Union	•	◦	•	•	◦	◦	◦	•	•	•	•	•	-	◦	•	•	◦	◦	•	◦	•	•	•
5	Stored Proc	•	◦	-	•	◦	◦	◦	-	-	-	-	-	-	◦	•	-	◦	◦	-	◦	•	•	•
6	Infer	•	◦	•	•	◦	◦	◦	•	•	•	•	•	-	◦	•	•	◦	◦	•	◦	•	•	•
7	Alter Encodings	•	◦	-	•	◦	◦	◦	-	•	•	•	-	-	◦	•	•	◦	◦	•	◦	•	•	•

Table 2 Comparison of Techniques Based on Deployment Requirements

Technique		Modify Code Base	Detection	Prevention	Additional Infrastructure
1	**Positive Tainting [1]**	No	Auto	Auto	None
2	**SQLPrevent [11]**	No	Auto	Auto	None
3	**Java Static Tainting [27]**	No	Automated	Code Suggestions	None
4	**WAVES [18]**	No	Automated	Generate Report	None
5	**SQLDOM [25]**	Yes	N/A	Automated	Developer Training
6	**SecuriFly [14]**	No	Automated	Automated	None
7	**Gateway [26]**	No	Manual Specification	Automated	Proxy Filter
8	**Safe Query Objects [24]**	Yes	N/A	Automated	Developer Training
9	**WebSSARI [15]**	No	Automated	Semi-Automated	None
10	**JDBC-Checker [12]**	No	Automated	Code Suggestions	None
11	**Tautology-checker [23]**	No	Automated	Code Suggestions	None
12	**SQLrand [22]**	Yes	Automated	Automated	Proxy, Developer Training, Key Management
13	**SQLGuard [10]**	Yes	Semi-Automated	Automated	None
14	**SQLCheck [5]**	Yes	Semi-Automated	Automated	Key Management
15	**AMNESIA[16]**	No	Automated	Automated	None
16	**CSSE [19]**	No	Automated	Automated	Custom PHP Interpreter
17	**CANDID [7]**	No	Automated	Automated	None
18	**Java Dynamic Tainting [21]**	No	Automated	Automated	None
19	**IDS [6]**	No	Automated	Generate Report	IDS System-Training Set
20	**SAFELI[28]**	No	Semi-Automated	N/A	None
21	**Web App Hardening [20]**	No	Automated	Automated	Custom PHP Interpreter
22	**Swaddler [3]**	No	Auto	Auto	Training
23	**SQL_IDS [8]**	No	Auto	N/A	None

V. CONCLUSION AND FUTUREWORK

In this paper we first identified the various types of SQLIAs. Then we investigated on SQL injection detection and prevention techniques. After that we compared these techniques in terms of their ability to stop SQLIA. Regarding the results, some current techniques' ability should be improved for stopping SQLI attacks. Moreover, we compared these approaches in deployment requirements that lead to inconvenience for users.

In our future work we separate techniques which have been implemented as tools then compare effectiveness, efficiency, stability, flexibility and performance of tools to show the strength and weakness of the tool.

VI. REFERENCES

[1] W. G. Halfond, A. 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. M0ller, 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] P.Grazie., 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)*.

[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.

[19] T. Pietraszek and C. V. Berghe. Defending Against Injection Attacks through Context-Sensitive String Evaluation. In *Proceedings of Recent Advances in Intrusion Detection (RAID2005)*, 2005.

[20] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting Information. In *Twentieth IFIP International Information Security Conference (SEC 2005)*, May 2005.

[21] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Proceedings 21st Annual Computer Security Applications Conference*, Dec. 2005..

[22] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pp 292–302. June 2004.

[23] G. Wassermann and Z. Su. An Analysis Framework for Security in Web Applications. In Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems *(SAVCBS 2004)*, pp 70–78.

[24] W. R. Cook and S. Rai. Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, 2005.

[25] R. McClure and I. Kr¨uger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In Proceedings of the 27th International Conference on Software Engineering (ICSE 05), pp 88–96, 2005.

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

[27] V. B. Livshits and M. S. Lam. Finding Security Errors in Java Programs with Static Analysis. In *Proceedings of the 14th Usenix Security Symposium*, pp 271–286, Aug. 2005.

[28] Xiang Fu , Kai Qian. SAFELI–SQL Injection Scanner Using Symbolic Execution. *Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications.* (2008). pp 34-39: ACM.