A Literature Review and Comparative Analyses on SQL Injection: Vulnerabilities, Attacks and their Prevention and Detection Techniques

Bojken Shehu [1] and Aleksander Xhuvani [2]

[1] Computer Engineering Department, Faculty of Information Technology
Polytechnic University of Tirana
Tirana, Albania

[2] Computer Engineering Department, Faculty of Information Technology
Polytechnic University of Tirana
Tirana, Albania

Abstract

SQL injection is a technique that exploits a security vulnerability occurring in the database layer of an application. The attack takes advantage of poor input validation in code and website administration. It allows attackers to obtain unauthorized access to the back-and database to change the intended application generated SQL queries. Researchers have proposed various solutions to address SQL injection problems. However, many of them have limitations and often cannot address all kind of injection problems. What's more, new types of SQL injection attacks have arisen over the years. To better counter these attacks, identifying and understanding existing techniques are very important. In this research we present all SQL injection attack types and also different techniques and tools which can detect or prevent these attacks.

Keywords: *SQL injection attacks, Web application, prevention, detection.*

1. Introduction

Internet is a widespread information infrastructure. Unaware of the security and privacy, the internet is becoming a repository of information. Information and data is the most important business asset in today's environment and achieving an appropriate level of Information Security. Applications are vulnerable to a variety of new security threats. One of the most threads to web application is SQL injection attack. According to Open Web Application Security Project (OWASP) [1] top 10 threats for web application security in 2013, injection attacks stands first. For example, financial fraud, online banking, theft of private data, shopping and cyber terrorism. Web applications that are susceptible to SQL injection may allow an attacker to gain complete access to their essential databases. To implement security guidelines inside or outside the database the access of the sensitive databases should be monitored. Detection or prevention of SQL injection attacks is a topic of active research in the industry and academia. To achieve those purposes, automatic tools and security system have been implemented, but none of them are complete or accurate enough to guarantee an absolute level of security on web application.

2. SQL injection background

SQL injections is one of the many web attack mechanism used by hackers to steal data from organizations. If it happens against the information systems of a hospital, the private information [2] of the patients may be leaked out which could threaten their reputation or may be a case of defamation. These attacks not only make the attacker to breach the security and steal the entire content of the database but also, to make arbitrary changes to both the database schema and the contents.

2.1 What is SQL injection?

Most web applications today use a multi-tier design, usually with three tiers: a) a presentation tier (front end). This is the topmost level of the application. This tier displays information related to such services as browsing merchandise, purchasing, and shopping cart contents. It communicates with other tiers by outputting results to the browser/client tier and all other tiers in the network. b) Application tier (Middle tier). This tier implements the software functionality by performing detailed processing, and c) the data tier (Backend). This tier consists of database servers, keeps data structured and answers to request from the application tiers. Three-tier is a client-server architecture in which the user interface, functional

process logic, data storage and access are developed and maintained as independent modules, most often on separate platforms. 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 web applications underlying database and destroy functionality or confidentiality.

2.2 Types of vulnerabilities

In this section, we present the most common security vulnerabilities found in web programming languages [3] (see Table 1).

Table 1: Types of vulnerabilities

Vulnerability Types	***Description***
Type I	Input validation is an attempt to verify or filter any input for malicious behavior. Insufficient input validation will allow code to be executed without proper verification of its intention. Attacker taking advantages of insufficient input validation can utilize malicious code to conduct attacks.
Type II	Lack of clear distinction between data types accepted as input in the programming language used for the web application development.
Type III	Delay of operations analysis till the runtime phase where the current variables are considered rather than source code expressions.

2.2 Types of SQL injection attacks

The SQL injection attacks can be performed using various techniques. Some of them are specified as follows:

Tautologies: *The main goal* of tautology-based attack is to inject code in conditional statements so that they are always evaluated as true. Using tautologies, the attacker wishes to either bypass user authentication or insert inject-able parameters or extract data from the database. A typical SQL tautology has the form, where the comparison expression uses one or more relational operators to compare operands and generate an always true condition. Bypassing authentication page and fetching data is the most common example of this kind of attack. In this type of injection, the attacker exploits an inject-able field contained in the WHERE clause of query. He transforms this conditional query into a tautology and hence causes all the rows in the database table targeted by the query to be returned. For example, SELECT * FROM user WHERE id='1' or '1=1'-'AND password='1234'; "or 1=1" is the most commonly known tautology.

Logically incorrect query attacks: *The main goal* of the Illegal/Logically Incorrect Queries based SQL Attacks is to gather the information about the back end database of the Web Application. 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 error by purpose. In this example attacker makes a type mismatch error by injecting the following text into the input field: 1) Original URL:*http://www.toolsmarket-al.com/veglat/?id_nav=2234* 2) SQL Injection: *http://www.toolsmarket-al/veglat/?id_nav=2234'* 3) *Error message showed: SELECT name FROM Employee WHERE id=2234\'. 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. The Illegal/Logically Incorrect Queries based SQL attack is considered as the basis step for all the other techniques.

Union Query: *The main goal* of the Union Query is to trick the database to return the results from a table different to the one intended. 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. This technique is mainly used to bypass authentication and extract data. *For example 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 sys Table. We will have the following query: SELECT Name, Phone FROM Users WHERE Id=1 UNION ALL SELECT credit card Number, 1 FROM Credit sys Table.* This will join the result of the original query with all the credit card users.

Stored Procedures: *The main goal* of the Stored Procedures SQL attack is to perform privilege escalation and try to execute the SQL procedures. SQL injection attacks of this type try to execute the SQL procedures. 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 [4], attacker exploits parameterized stored procedure. *CREATE PROCEDURE DBO. is Authenticated @user Name varchar2, @pass varchar2, @pin int ASEXEC("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 SQL injection attack, intruder input "'; SHUTDOWN; - -"for username or password. Then the stored procedure generates the following query: SELECT accounts FROM users WHERE login='boni' 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.

Piggy-Backed Queries: *The main goal* of the Piggy-Backed Query is to execute remote commands or add or modify data. In this attack type, an attacker tries to inject additional queries along with the original query, which are said to "piggy-back" onto the original query. As a result, the database receives multiple SQL queries for execution. Vulnerability of this kind of attack is dependent of the kind of database [5]. *For example, if the attacker inputs [';drop table users--] into the password field, the application generates the query: SELECT Login_ID FROM users_ID WHERE login_ID='john' and password=''; DROP TABLE users-' AND ID=2345* After executing the first query, the database encounters the query delimiters (;) and execute the second query. The result of executing second query would result into dropping the table users, which would likely destroy valuable information.

Inference: *The main goal* of the inference is to change the behavior 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: *For example, SELECT accounts FROM users WHERE id= '1111' 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 submits 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, declare @ 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: *The main goal* of the Alternate Encodings is to avoid being identified by secure defensive coding and automated prevention mechanisms. Hence it helps the attackers to evade detection. It is usually combined with other attack techniques. 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". 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 5f177 6e), " and the result query is: SELECT accounts FROM users WHERE login=" AND pin=0; exec (char (0x73687574646j776e))* 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.

3. Related Work

In order to detect and prevent SQL Injection attacks, filtering and other detection methods are being researched. This section explains the related work.

Black Box Testing Huang and colleagues [6] propose WAVES, a black-box technique for testing Web applications for SQL injection vulnerabilities. The technique uses a Web crawler to identify all points in a Web application that can be used to inject SQLIAs. It then builds attacks that target such points based on a specified list of patterns and attack techniques. WAVES then monitors the application's response to the attacks and uses machine learning techniques to improve its attack methodology. This technique improves over most penetration-testing techniques by using machine learning approaches to guide its testing. However, like all black-box and penetration testing techniques, it cannot provide guarantees of completeness.

WebSSARI [7] 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 [8] is tool that is 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.

Dynamic Analysis: This approach is also known as post-generated approach. Post-generated technique are useful for analysis of dynamic or runtime SQL query, generated with user input data by a web application. Detection techniques under this post-generated category executes before posting a query to the database server [13].

Code Checkers are based on static analysis of web application that can reduce SQL injection vulnerabilities and detect type errors. For instance, JDBC-Checker [9] is a tool used to code check for statically validating the type rightness of dynamically-generated SQL queries. However, researchers have also developed particular packages that can be applied to make SQL query statement safe [10].

Combined Static and Dynamic Analysis: AMNESIA [11] is technique that combines dynamic and static for preventing and detecting web application vulnerabilities at the runtime. AMNESIA uses static analysis to generate different type of query statements. In the dynamic phase AMNESIA interprets all queries before they are sent to the database and validates each query against the statically built models. AMNESIA stops all queries before they are sent to the database and validates each query statement against the AMNESIA models. However, the

primary limitation in AMNESIA according to article [12] is that the technique is dependent on the accuracy of its static analysis for building query models for successful prevention of SQL injection.

In **SQL Guard** [15] and **SQL Check** [14] 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.

SQL-IDS, a specification based approach to detect malicious intrusions. Authors in article [16] suggest using a novel specification-based methodology for the detection of exploitations of SQL injection vulnerabilities. The proposed query-specific detection allowed the system to perform focused analysis at negligible computational overhead without producing false positive or false negatives.

SQLrand, [17] is proposed by Boyd and Keromytis. For the implementation, they use a proof of concept proxy server in between the Web server (client) and SQL server; they de-randomized queries received from the client and sent the request to the server. This de-randomization framework has 2 main advantages: portability and security. The proposed scheme has a good performance: 6.5 ms is the maximum latency overhead imposed on every query.

SQLIA Prevention Using Stored Procedures – Stored procedures are subroutines in the database which the applications can make call to [18]. The prevention in these stored procedures is implemented by a combination of static analysis and runtime analysis. The static analysis used for commands identification is achieved through stored procedure parser and the runtime analysis by using a SQL Checker for input identification.

Ruse et al.'s Approach, [19], propose atechnique that uses automatic test case generation to detect SQL Injection Vulnerabilities. The main idea behind this framework is based on creating a specific model that deals with SQL queries automatically. Adding to that, the approach identifies the relationship (dependency) between sub-queries. Based on the results, the methodology is shown to be able to specifically identify the causal set and obtain 85% and 69% reduction respectively while experimenting on few sample examples.

SAFELI, [20] proposes a Static Analysis Framework in order to detect SQL Injection Vulnerabilities. SAFELI framework aims at identifying the SQL Injection attacks during the compile-time. This static analysis tool has two main advantages. Firstly, it does a White-box Static Analysis and secondly, it uses a Hybrid-Constraint Solver. For the White-box Static Analysis, the proposed approach considers the byte-code and deals mainly with strings. For the Hybrid-Constraint Solver, the method implements an efficient string analysis tool which is able to deal with Boolean, integer and string variables.

Ali et al.'s Scheme, [21] adopts the hash value approach to further improve the user authentication mechanism. They use the user name and password hash values SQLIPA (SQL Injection Protector for Authentication) prototype was developed in order to test the framework. The user name and password hash values are created and calculated at runtime for the first time the particular user account is created.

Thomas et al.'s Scheme, [22] suggest an automated prepared statement generation algorithm to remove SQL Injection Vulnerabilities. They implement their research work using four open source projects namely: (i) Net-trust, (ii) ITrust, (iii) WebGoat, and (iv) Roller. Based on the experimental results, their prepared statement code was able to successfully replace 94% of the SQLIVs in four open source projects.

Dynamic Candidate Evaluations Approach, [23], Bisht et al. propose CANDID. It is a Dynamic Candidate Evaluations method for automatic prevention of SQL Injection attacks. This framework dynamically extracts the query structures from every SQL query location which are intended by the developer (programmer). Hence, it solves the issue of manually modifying the application to create the prepared statements.

Haixia and Zhihong's Database Security Testing Scheme, [24] propose a secure database testing design for Web applications. They suggest a few things; firstly, detection of potential input points of SQL Injection; secondly, generation of test cases automatically, then finally finding the database vulnerability by running the test cases to make a simulation attack to an application. The proposed methodology is shown to be efficient as it was able to detect the input points of SQL Injection

exactly and on time as the authors expected. However, after analyzing the scheme, we find that the approach is not a complete solution but rather it needs additional improvements in two main aspects: the detection capability and the development of the attack rule library.

Swaddler, [25] 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.

DIWeDa approach, [26] propose IDS (Intrusion Detection Systems) for the backend databases. They use DIWeDa, a prototype which acts at the session level rather than the SQL statement or transaction stage, to detect the intrusions in Web applications. The proposed framework is efficient and could identify SQL injections and business logic violations too.

Manual Approaches, [27] MeiJunjin highlights the use of manual approaches in order to prevent SQLI input manipulation flaws. In manual approaches, defensive programming and code review are applied. In defensive programming: an input filter is implemented to disallow users to input malicious keywords or characters. This is achieved by using white lists or black lists. As regards to the code review [29], it is a low cost mechanism in detecting bugs; however, it requires deep knowledge on SQLIAs.

Automated Approaches, [28] Besides using manual approaches, MeiJunjin also highlights the use of automated approaches. The author notes that the two main schemes are: Static analysis FindBugs and Web vulnerability scanning. Static analysis FindBugs approach detects bugs on SQLIAs, gives warning when an SQL query is made of variable. However, for the Web vulnerability scanning, it uses software agents to crawl, scans Web applications, and detects the vulnerabilities by observing their behavior to the attacks.

Removing SQL query attribute values, [30] Authors proposed an approach to detect SQL injection attacks is based on static and dynamic analysis. This method removes the attribute values of SQL queries at runtime (dynamic method) and compares them with the SQL queries analyzed in advance (static method) to detect the SQL injection. When run the application each dynamical generated query is compared or performs XOR operation with fixed query if it results zero then that particular query allowed to the database and if it not results to zero then that query reported as abnormal query stop sending to database.

SQL DOM Scheme, [31], authors closely consider the existing flaws while accessing relational databases from the OOP (Object-Oriented Programming) Languages point of view. They mainly focus on identifying the obstacles in the interaction with the database via CLIs (Call Level Interfaces). SQL DOM object model is the proposed solution to tackle these issues through building a secure environment for communication.

SQL Prevent, [32] is consists of an HTTP request interceptor. The original data flow is modified when SQL Prevent 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.

Shin et al.'s approach suggests SQLUnitGen, a Static-analysis-based tool that automate testing for identifying input manipulation vulnerabilities*:* [33]. The authors apply SQLUnitGen tool which is compared with FindBugs, a static analysis tool. The proposed mechanism is shown to be efficient as regard to the fact that false positive was completely absent in the experiments.

Positive tainting, [34] 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.

4. Comparative Analyses

In this section, the SQL injection detection and prevention techniques presented in section III would be compared. It is noticeable that this comparison is based on the articles not empirically experience.

4.1 Comparison of SQL Injection Detection Techniques With Respect to Attack Types

Detection techniques are techniques that detect attacks mostly at runtime. Table 1 shows a chart of the schemes and their detection capabilities against various SQL injections attacks and summarize the results of this comparison. The symbol √ is used for techniques that can successfully detect all attacks of that type. The symbol × is used for techniques that is not able detect all attacks of that type. The symbol □ refers to techniques that detect the attack type only partially because of natural limitations of underlying approach.

Table 1: Comparison of SQL injection detection techniques with respect to attack types

Attacks	SAFELI[20]	SQL-IDS[16]	Swaddler[25]	SQL Prevent[32]	SQLrand[17]	SQLIPA[21]	AMNESIA[11]	Automated Approaches[28]	CANDID[23]	DIWeDa[26]	Tautology Checker[35]	Removing SQL query[30]	SQLCheck[14]	SQL Guard[15]
Tautologies	×	√	□	√	√	√	√	√	□	×	√	√	√	√
Piggy-backed	√	√	□	√	√	×	√	√	□	×	×	√	√	√
Illegal/ Incorrect	√	√	□	√	√	×	√	√	□	×	×	√	√	√
Union	√	√	□	√	√	×	√	√	□	×	×	√	√	√
Stored Procedure	√	√	□	√	×	×	×	×	□	×	×	√	×	×
Inference	√	√	□	√	√	×	√	√	□	√	×	√	√	√
Alternate Encodings	√	√	□	√	√	×	√	×	□	×	×	√	√	√

Table 2 illustrates the addressing percentage of SQL injection attacks among SQL injection detection techniques. The percentage of techniques that detect tautology is calculated by this formula (1):

$$\frac{\text{Number of techniques that can detect tautology}}{\text{Total number of techniques}} * 100 = \frac{10}{14} * 100 = 72 \quad (1)$$

Table 2: Comparison of SQL injection detection techniques with respect to attack types

Attack Types	Techniques that can detect all attacks of that type (√)	Techniques that can detect the attacks only partially (□)	Techniques that is not able to detect attacks of that type (×)
Tautologies	72%	14%	14%
Piggy-backed	64%	14%	22%
Illegal/ Incorrect	64%	14%	22%
Union	64%	14%	22%
Stored Procedure	29%	14%	57%
Inference	72%	14%	14%
Alternate Encodings	57%	14%	29%

4.2 Comparison of SQL Injection Prevention Techniques With Respect to Attack Types

Prevention techniques are techniques that statistically identify vulnerabilities in the code. Table 3 shows a chart of the schemes and their prevention capabilities against various SQL injections attacks and summarize the results of this comparison.

Table 3: Comparison of SQL injection preventiontechniques with respect to attack types

Attacks	JDBC Checker[9]	Positive Tainting[34]	SecuriFly[8]	Security Gateway[36]	SQLDOM[31]	WAVES[6]	WebSSARI[7]
Tautologies	□	√	□	□	√	□	√
Piggy-backed	□	√	□	□	√	□	√
Illegal/ Incorrect	□	√	□	□	√	□	√
Union	□	√	□	□	√	□	√
Stored Procedure	□	√	□	□	×	□	√
Inference	□	√	□	□	√	□	√
Alternate Encodings	□	√	□	□	√	□	√

Table 4 illustrates the addressing percentage of SQL injection attacks among SQL injection prevention

techniques. The percentage of techniques that detect tautology is calculated by this formula:

$$\frac{\text{Number of techniques that can prevent tautology}}{\text{Total number of techniques}} * 100 = \frac{3}{7} * 100 = 43 \quad (2)$$

Table 4: Comparison of SQL injection prevention techniques with respect to attack types

Attack Types	Techniques that can prevent all attacks of that type (√)	Techniques that can prevent the attacks only partially (□)	Techniques that is not able to prevent attacks of that type (×)
Tautologies	43%	57%	0%
Piggy-backed	43%	57%	0%
Illegal/ Incorrect	43%	57%	0%
Union	43%	57%	0%
Stored Procedure	29%	57%	11%
Inference	43%	57%	0%
Alternate Encodings	43%	57%	0%

4.3 Comparison of detection or prevention techniques based on deployment and evaluation criteria.

The result of this classification are summarized in table 5.

Table 5: Comparison of detection or prevention techniques based on deployment and evaluation criteria.

Techniques	Detection time	Detection Location	Code Modify	Additional infrastructure
JDBC Checker [9]	Coding time	Server side application	No	N/A
Positive Taintitng [34]	Run time	Server side application	No	N/A
SecuriFly [8]	Run time	Server side application	No	N/A
Security Gateway [36]	Run time	Server side proxy	No	Required
SQL DOM [31]	Compile time	Server side application	Yes	Required
WAVES [6]	Testing time	Client side application	No	N/A
WebSSARI [7]	Run time	Server side application	No	Required
SQL Guard [15]	Run time	Server side application	Yes	Required
SQL Check [14]	Run time	Server side proxy	Yes	Required
Removing SQL query [30]	Run time	Server side	No	N/A
Tautology Checker [35]	Run time	Server side application	No	N/A
DIWeDa [26]	Run time	Server side application	No	N/A
CANDID [23]	Run time	Server side application	Yes	Required
Automated Approaches [28]	Run time	Server side application	No	N/A
AMNESIA [11]	Run time	Server side application	No	N/A
SQLIPA [21]	Run time	Server side application	No	N/A
SQLrand [17]	Run time	Server side Proxy	Yes	N/A
SQL Prevent [32]	Run time	Server side application	No	N/a
Swaddler [25]	Run time	Server side application	No	Required
SQL-IDS [16]	Run time	Server side proxy	No	N/A
SAFELI [20]	Compile time	Server side application	No	N?A

5. Conclusions

In this paper, we have presented a survey report on various types of SQL injection attacks, vulnerabilities, and detection and prevention techniques. We assessed SQL injection attacks among current SQL Injection detection and prevention techniques. To perform this evaluation, we first identified the various types of SQL injection attacks. Then we investigated SQL injection detection and prevention techniques. After that we compared these techniques in terms of their deployment and evaluation criteria. Different authors have presented their work at deferent levels of detail, extracting uniform data from such a diverse range of papers was very tedious task .Our future work will be to extend our research in terms of their evaluation criteria.

References

[1] The Open Web Application Security Project,"OWASP TOP Project", https://www.owasp.org/SQL_Injection.

[2] Sh. Bojken, A. Shqiponja, A. Marin, and Xh. Aleksander, "Protection of Personal Data in Information Systems", International Journal of Computer Science, Vol. 10, No. 2, July 2013, ISSN (Online): 1694-0784.

[3] N. Seixas, J. Fonseca, M. Vieira, and H. Madeira."Looking at Web Security Vulnerabilities from the Programming Language Perspective: A Field of Study", ISSRE 2009 pp.129-135.

[4] P. Y. Yane, M.S. Chaudhari, "SQLIA: Detection And Prevention Techniques: A Survay", IOSR Journal of Computer Enginerering (IOSR-JCE), 2013, Vol. 2, pp.56-60.

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

[6] Y. Huang, S. Huang, T. Lin, and C. Tsai. Sivilotti."Web Application Security Assessment by Fault Injection and Behavior Monitoring", In Proceedings of the 11th International Word Wide Web Conference, May 2003.

[7] Y. Huang, F. Yu, C. Yang, C. H. Tsai, D. T. Lee, and S. Y. Ku. "Securing Web Application Code by Static Analysis and Runtime Protection", In Proceedings of the 12th International Word Wide Web Conference, May 2004.

[8] M. Martin, B. Livshits, and M. S. Lam "Finding Application Errors and Security Flaws Using PQL: A Program Query Language", ACM Notices, Volume 40, Issue:10 pages, 2005.

[9] C. Gould, Z. Su, and P. Devanbu, "JDBC checker: A static analysis tool for SQL/JDBC applications," 2004, pp. 697-698.

[10] R. A. McClure and I. H. Krüger, "SQL DOM: compile time checking of dynamic SQL statements," 2005, pp. 88-96

[11]W. G. J. Halfond and A. Orso, "Preventing SQL injection attacks using AMNESIA," presented at the Proceedings of the 28th international conference on Software engineering, Shanghai, China, 2006.

[12]IndraniBalasundaram and Ramaraj, "An Approach to Detect and Prevent SQL Injection Attacks in Database Using Web Service," International Journal of Computer Science and Network Security, vol. 11, pp. 197-205, 2011.

[13] B.I.A. Barry and H.A. Chan, "Syntax, and Semantics-Base Signature Database f or Hybrid Intrusion Detection Systems," Security and Comm. Networks, vol. 2, no. 6, pp. 457-475, 2009.

[14] 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.

[15] 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.

[16]K. Kemalis, and T. Tzouramanis. "SQL-IDS: A Specification-based Approach for SQL Injection Detection", SAC, 2008, Brazil, ACM, pp.2153-2158.

[17]S. W. Boyd and A. D. Keromytis. "SQLrand: Preventing SQL Injection Attacks", In Proceedings of the 2nd Applied Cryptography and Network Security Conference, pages 292–302, June 2004.

[18]K. Amirtahmasebi, S. R. Jalalinia, S. Khadem, "A survey of SQLinjection defense mechanisms," Proc. Of ICITST 2009, pp.1-8, 2009.

[19] M. Ruse, T. Sarkar and S. Basu. "Analysis & Detection of SQL Injection Vulnerabilities via Automatic Test Case Generation of Programs", 10th Annual International Symposium on Applications and the Internet pp. 31 – 37, 2010.

[20] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. "A StaticAnalysis Framework for Detecting SQL Injection Vulnerabilities",COMPSAC 2007, pp.87-96, July 2007.

[21]Shaukat Ali, Azhar Rauf, Huma Javed "SQLIPA:An authentication mechanism Against SQL Injection"

[22]S. Thomas, L. Williams, and T. Xie, "On automated prepared statement generation to remove SQL injection vulnerabilities". Information and Software Technology, 2009.

[23] P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan. "CANDID:Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks". ACM Trans. Inf. Syst. Secur, 2010.

[24] Haixia, Y. and Zhihong, N., " A database security testing scheme of web application". Proc. of 4th International Conference on Computer Science & Education 2009 (ICCSE '09),July 2009.

[25]M. Cova, D. Balzarotti. "Swaddler:An Approach for the Anomaly-based Detection of State Violations in Web Applications", Recent Advances in Intrusion Detection, Proceedings, volume: 4637 Pages: 63-86 ,2007.

[26] A. Roichman, E. Gudes, "DIWeDa - Detecting Intrusions in WebDatabases". In: Atluri, V. (ed.) DAS 2008. LNCS, vol. 5094, pp. 313–329. Springer, Heidelberg (2008).

[27]M. Junjin, "An Approach for SQL InjectionVulnerability Detection", Sixth International Conferenceon Information Technology: New Generations ITNG, pp. 1411-1414, 2009.

[28] Mei Junjin, "An Approach for SQL Injection Vulnerability Detection," Proceedings. of the 6th Int. Conf. on Information Technology: New Generations, Las Vegas, Nevada, pp. 1411-1414, Apr. 2009.

[29]R. A. Baker, "Code Reviews Enhance Software Quality". In Proceedings of the 19th international conference on Software engineering ICSE, Boston, MA, USA, 1997.

[30] I. Lee , S. Jeong, S. Yeoc, J. Moond, "A novel method for SQL injection attack detection based on removing SQL query attribute", Journal Of mathematical and computer modeling, Elsevier 2011.

[31]McClure, and I.H. Kruger, "SQL DOM: compile time checking of dynamic SQL statements," Software Engineering, ICSE 2005, Proceedings. 27th International Conference on, pp. 88- 96, 2005.

[32]P. Grazie, "SQL Prevent thesis", University of British Columbia (UBC) Vancouver, Canada,2008.

[33] Y. Shin, L. Williams and T. Xie, "SQLUnitGen: Test Case Generation for SQL Injection Detection," North Carolina

StateUniv., Raleigh Technical report, NCSU CSC TR 2006-21, 2006.

[34] G. William, J. 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.

[35]G. Wassermann, 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, pp. 70–78, 2004.

[36] D. Scott and R. Sharp, "Abstracting Application-level Web Security", in Proceedings of the 11th International Conference on the World Wide Web, pages 396–407, 2002.

Bojken Shehu. He is a pedagogue in Polytechnic University of Tirana, Faculty of Information Technology, in Computer Engineering Department. In 2007 he has finished the Bachelor Thesis in Saint Petersburg State Polytechnic University, Russia and in 2010 he has finished the Master Thesis in Bauman Moscow State Technical University, Russia and now he is a PhD student in Polytechnic University of Tirana, Albania.

Aleksander Xhuvani. He is a pedagogue in Polytechnic University of Tirana, Faculty of Information Technology, in Computer Engineering Department. He has finished the PhD study at Bordeaux in France. At 2004 he is graduated as Prof. Dr.