Web Application Security by SQL Injection DetectionTools

Atefeh Tajpour , Suhaimi Ibrahim, Mohammad Sharifi
Advanced Informatics School
University Technology Malaysia
Malaysia

***Abstract*— SQL injection is a type of attack which the attacker adds Structured Query Language code to a web form input box 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. Researchers have proposed different tools to detect and prevent this vulnerability. In this paper we present all SQL injection attack types and also current tools which can detect or prevent these attacks. Finally we evaluate these tools.**

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

1. 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 been 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. These problems motivate the need for a solution to the SQL injection problem.

Researchers have proposed some tools to help developers to compensate the shortcoming of the defensive coding [7, 10, 12]. The problem is that some current tools could not address all attack types or some of them need special deployment requirements.

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 evaluate SQL Injection detection or/and prevention tools against all types of SQL injection attacks and deployment requirements. Conclusion and future work is provided in section 6.

2. DEFINITION OF SQLIA

Most web applications today use a multi-tier design, usually with three tiers: a presentation, a processing and a data tier. The presentation tier is the HTTP web interface, the application tier implements the software functionality, and the data tier keeps data structured and answers to requests from the application tier [21]. Meanwhile, large companies developing SQL-based database management systems rely heavily on hardware to ensure the desired performance [22].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.

2.1 SQL injection attack process

SQLIA is a hacking technique which the attacker adds SQL statements through a web application's input fields or hidden parameters to access to resources. Lack of input validation in web applications causes hacker to be successful. For the following examples we will assume that a web application receives a HTTP request from a client as input and generates a SQL statement as output for the back end database server.

For example an administrator will be authenticated after typing: employee id=112 and password=admin. Figure1 describes a login by a malicious user exploiting SQL

Injection vulnerability [11]. Basically it is structured in three phases:

1. an attacker sends the malicious HTTP request to the web application
2. creates the SQL statement
3. submits the SQL statement to the back end database

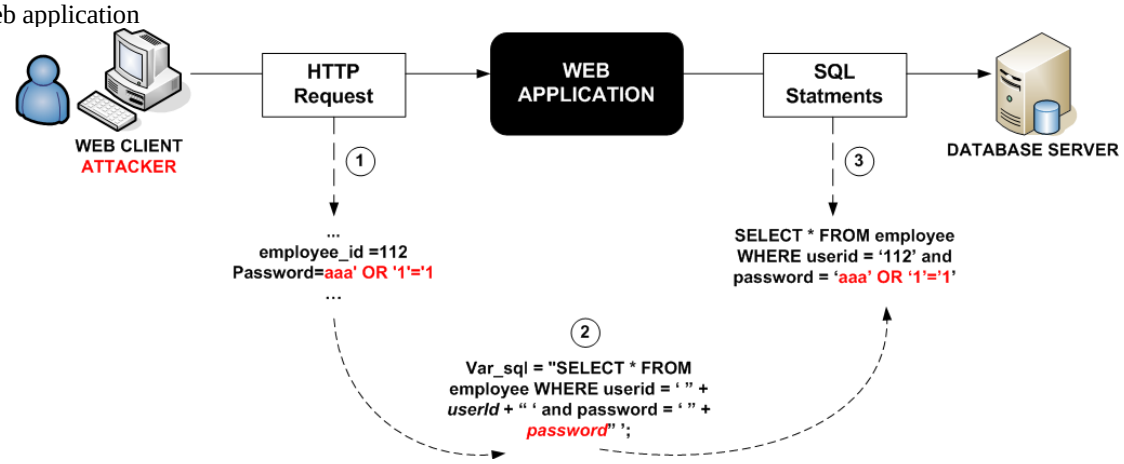

Figure 1: Example of a SQL Injection Attack

The above SQL statement is always true because of the Boolean tautology we appended (OR 1=1) so, we will access to the web application as an administrator without knowing the right password.

2.2 Main cause of SQL injection

Web application vulnerabilities are the main causes of any kind of attack [19]. In this section, vulnerabilities that might exist naturally in web applications and can be exploited by SQL injection attacks will be presented:

Invalidated input: This is almost the most common vulnerability on performing a SQLIA. There are some parameters in web application, are used in SQL queries. If there is no any checking for them so can be abused in SQL injection attacks. These parameters may contain SQL keywords, e.g. INSERT, UPDATE or SQL control characters such as quotation marks and semicolons.

Generous privileges: Normally in database the privileges are defined as the rules to state which database subject has access to which object and what operation are associated with user to be allowed to perform on the objects. Typical privileges include allowing execution of actions, e.g. SELECT, INSERT, UPDATE, DELETE, DROP, on certain objects. Web applications open database connections using the specific account for accessing the database. An attacker who bypasses authentication gains privileges equal to the accounts. The number of available attack methods and affected objects increases when more privileges are given to the account. the worst case happen If an account can connect to system that is associated with the system administrator because normally has all privileges.

Uncontrolled variable size: If variables allow storage of data be larger than expected consequently allow attackers to enter modified or faked SQL statements. Scripts that do not control variable length may even open the way for attacks, such as buffer overflow.

Error message: Error messages that are generated by the back-end database or other server-side programs may be returned to the client-side and presented in the web browser. These messages are not only useful during development for debugging purposes but also increase the risks to the application. Attackers can analyze these messages to gather information about database or script structure in order to construct their attack.

Variable Orphism: The variable should not accept any data type because attacker can exploit this feature and store malicious data inside that variable rather than is suppose to be. Such variables are either of weak type, e.g. variables in PHP, or are automatically converted from one type to another by the remote database.

Dynamic SQL: SQL queries dynamically built by scripts or programs into a query string. Typically, one or more scripts and programs contribute and finally by combining user input such as name and password, make the WHERE clauses of the query statement. The problem is that query building components can also receive SQL keywords and control characters. It means attacker can make a completely different query than what was intended.

Client-side only control: If input validation is implemented in client-side scripts only, then security functions of those scripts can be overridden using cross-site scripting. Therefore, attackers can bypass input validation and send invalidated input to the server-side.

Stored procedures: They are statements which are stored in DBs. The main problem with using these procedures is that an attacker may be able to execute them and damage database as well as the operating system and even other network components. Usually attackers know system stored procedures that come with different and almost easily can execute them.

Into Outfile support: Some of RDBMS benefit from the INTO OUTFILE clause. In this condition an attacker can manipulate SQL queries then they produce a text file containing query results. If attackers can later gain access to this file, they can abuse the same information, for example, bypass authentication.

Multiple statements: If the database supports UNION so, attacker has more chance because there are more attack methods for SQL injection. For instance, an additional INSERT statement could be added after a SELECT statement, causing two different queries to be executed. If this is performed in a login form, the attacker may add him or herself to the table of users.

Sub-selects: Supporting sub-selects is weakness for RDBMS when SQL injection is considered. For example, additional SELECT clauses can be inserted in WHERE clauses of the original SELECT clause. This weakness makes the web application more vulnerable, so they may be penetrated by malicious users easily.

3. 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 [4, 20] 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.

llegal/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 creditCardNumber,1 FROM CreditCarTable
We will have the following query:

SELECT Name, Phone FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1 FROM CreditCarTable

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.isAuthenticated
@userName varchar2, @pass varchar2, @pin int
AS
EXEC("SELECT accounts FROM users
WHERE login='" +@userName+ "' and pass='" +@password+
"' and pin=" +@pin);
GO
```

For authorized/unauthorized user the stored procedure returns true/false. As an SQLIA, intruder inp" " ' '

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

4. 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 tools not techniques.

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.

5. EVALUATION

In this section, the SQL injection detection or prevention tools presented in section IV would be compared. It is noticeable that this comparison is based on the evaluation which the authors of tools have done empirically. They used a testbed for their tool. In particular, they used a set of web applications and a set of inputs for those applications that included both legitimate inputs and SQLIAs.

5.1 Comparison of SQL Injection Detection/Prevention Tools Based on Attack Types

Proposed tools 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 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 the attack type only partially because of natural limitations of the underlying approach.

Table1: Comparison of 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]	Positie Tainting[1]	SQLPrevent[11]
1	Tautologies	●	°	°	°	●	●	●	°	●	°	°	●	●
2	Illegal/ Incorrect	●	°	°	°	●	●	●	°	●	°	°	●	●
3	Piggy-back	●	°	°	°	●	●	●	°	●	°	°	●	●
4	Union	●	°	°	°	●	●	●	°	●	°	°	●	●
5	Stored Proc	●	°	°	°	-	-	-	°	●	°	°	●	●
6	Infer	●	°	°	°	●	●	●	°	●	°	°	●	●
7	Alter Encodings	●	°	°	°	●	●	●	°	●	°	°	●	●

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.

5.2. Comparison of SQL Injection Detection/Prevention Tools Based on Deployment Requirement

Each tool with respect to the following criteria was evaluated: (1) Does the tool require developers to modify their code base? (2) What is the degree of automation of the detection aspect of the tool? (3) What is the d[illegible]

automation of the prevention aspect of the tool? (4) What infrastructure (not including the tool itself) is needed to successfully use the tool? The results of this classification are summarized in Table2.

Table2: Comparison of Tools Based on Deployment Requirements

No	Tool	Modify Code base	Detection	Prevention	Additional Infrastructure
1	AMNESIA [16]	No	Auto	Auto	None
2	IDS [6]	No	Auto	Generate report	IDS system-Training set
3	JDBC Checker [12]	No	Auto	Code suggestion	None
4	Securifly[14]	No	Auto	Auto	None
5	SQLCHECK [5]	Yes	Semi Auto	Auto	Key management
6	SQLGaurd [10]	Yes	Semi Auto	Auto	None
7	WAVES [18]	No	Auto	Generate report	None
8	WEBSSARY [15]	No	Auto	Semi auto	None
9	CANDID [7]	No	Auto	Auto	None
10	SQL_IDS [8]	No	Auto	N/A	None
11	Swaddler [3]	No	Auto	Auto	Training
12	Positive Tainting [1]	No	Auto	Auto	None
13	SQLPrevent [11]	No	Auto	Auto	None

Table2 determines the degree of automation of tool in detection or prevention of attacks. Actually automatically detection and prevention is ability of tool that provides user satisfaction. Also table shows that which tool needs to modify the source code of application. Moreover, additional infrastructure that is required for each tool that usually leads to inconvenience for users is illustrated.

5.3Comparison of Tools Based on Evaluation Parameters

The authors of proposed tools have evaluated their tools in common parameters: efficiency, effectiveness and performance, flexibility and stability. The results of this classification are summarized in Table 3. Definition of the measured parameters [11]:

Efficiency

- **False positive:** is a false alarm. It is when the tool incorrectly categorizes a benign request being as a malicious attack.
- **False negative**: occurs when a malicious attack is not recognized, so the tool lets it pass normally.

Effectiveness

- **Attacks Detection:** the percentage of real attacks, correctly detected.
- **Attacks Prevention**: the percentage of real attacks correctly blocked after being detected.

Flexibility

- **Different Types of SQLIAs**: the ability of the tool to detect/prevent different types of SQL Injection attacks such as those were presented in section II.

Performance

- **Detection Overhead:** is the time spent for a detection of a SQLIA once the tool is running.
- **Prevention Overhead:** is the time spent to detect and block (prevent) a SQLIA once the tool is running.

Stability

- **Environment Independence**
  - **Web Applications:** the possibility to test the tool on different types of web applications, such as open source/commercial, large/small.
  - **Databases:** testing on web applications that use different backend databases, such as open source (e.g. MySQL) commercial (e.g. Oracle).
  - **Programming Languages:** the ability of the tool to work on web applications written in different programming languages, such as J2EE, .NET, PHP and so On.
  - **Operating Systems**: the ability of the tool to run on different OS such as Windows and Linux.
  - **Application Servers:** the possibility to run the tool in a network using different type of Application Server such Tomcat.

Table 3: Comparison of Tools based on Evaluation Parameters

No	Tool	Efficiency	Effectiveness	Flexibility	Performance (ms)	Stability	
						Programming Language	Equipment for evaluation
1	AMNESIA [16]	0	100	All	Negligible	JAVA	N/A
2	IDS [6]	F.P=0.06 F.N=N/A	N/A	All /P	0.5	All	Server: 2 GHz Pentium 4 with 1 GB of RAM Linux 2.6.1. Apache web server (v2.0.52), the MySQL database (v4.1.8), and PHP-Nuke (v7.5).
3	JDBC Checker [12]	F.P=low F.N=N/A	N/A	All /p	Negligible	Java /JDBC	N/A
4	Securify [14]	F.P=N/A F.N=0	100	All	14.4	Java	Client: AMD Opteron 150 machine with 4GB of memory running Linux Server: 2 GHz AMD Athlon XP with 256MB of memory running Linux
5	SQLCHECK [5]	0	100	6	2	All	Linux kernel 2.4.27,2 GHz Pentium M processor and 1 GB of memory
6	SQLGaurd [10]	N/A	N/A	6	3	J2EE	The web server is a 733MHz windows 2000 machine, 256MB RAM
7	WAVES [18]	F.P=N/A F.N=2.6	N/A	All/P	N/A	ASP/PHP	Unix
8	WEBSSARY [15]	F.P=10.3	N/A	All	N/A	All	N/A
9	CANDID [7]	F.P=0 F.N=N/A	100	All	12	Java	Client: 2GHz Pentium processor and 2GB of RAM,Server: a Red Hat Enterprise GNU / Linux machine.
10	SQL_IDS [8]	0	100	All	5	All	Client: AMD Athlon 1GHz, with 256 MB RAM and Microsoft Windows 2000. Server : Apache Tomcat (ver. 5.5.23) and Microsoft SQL Server 2000
11	Swaddler [3]	F.P=low F.N=0	N/A	All /P	N/A	All	Client: 3.6GHz Pentium 4 with 2 GB of RAM running Linux 2.6.18. Server: Apache web server (version 2.2.4) and PHP version 5.2.1
12	Positive Tainting [1]	0	100	All	6	Java	Client: Pentium 4, 2.4Ghz, with 1GB memory. Server: a dual-processor Pentium D, 3.0Ghz, with 2GB of memory, running GNU/Linux 2.6
13	SQLPrevent [11]	F.P=0 F.N=0	100	All	3	All	Client: a 1.8 GHz Intel Pentium 4, 512 MB RAM, Windows XP SP2.Host: 350 Mhz Pentium II processor and 256 MB of memory, Windows 2003 SP2.

Based on the table4, different criteria such as efficiency, effectiveness, stability, flexibility and performance for choosing an appropriate tool could be considered. For example the table shows that which programming language could be supported by the specific tool. Also, by flexibility, types of SQL injection attack which are addressed by the tool could be identified. “All” means that the tool can stop all type of attack successfully and “All/p” means that the tool can stop all the attack type partially.

On the other hand, we believe that the value of some evaluation parameters such as efficiency, effectiveness and performance is depend on testbed that have been used by each author such as equipments, tools and scripts for attack so the value of these parameters may change in empirically evaluation in a common testbed.

6. Conclusion And Future Work

In this paper we presented 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.

In addition, the current tools were compared based on deployment requirement (modifying source code, additional infrastructure and automation of detection or prevention) and common evaluation parameters (efficiency, effectiveness, stability, flexibility and performance).

In our future work we will propose a framework for measuring effectiveness, efficiency, stability and performance of tools in common criteria to prove the strength and weakness of them.

Acknowledgements

This research is supported by the RU grant of University Technology Malaysia. Thanks to UTM-RMC, government of Malaysia and individuals who are directly or indirectly involved in this research.

REFERENCES

[1] 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.

[2] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, *CANDID:* Preventing SQL Injection Attacks using Dynamic Candidate Evaluation. *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, Alexandria, Virginia, USA.page:12-24.

[3] Marco Cova, Davide 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 Published: 2007.

[4] William G.J. Halfond, Jeremy Viegas and Alessandro 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. ACM SIGPLAN Notices. Volume: 41, pp: 372-382, 2006.

[6] F. Valeur, D. Mutz, and G. Vigna. A Learning-Based to the Detection of SQL Attacks. Detection of Intru

Malware, And Vulnerability Assessment, Proceedings, Volume: 3548, pp: 123-140, 2005.

[7] P. Bisht, P. Madhusudan. CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks. ACM Transactions on Information and System Security Volume: 13, Issue: 2, 2010.

[8] K. Kemalis and T.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. Moller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In Proc. 10th International Static Analysis Symposium, SAS '03, volume 2694, 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] Wassermann, G; Gould, C; Su, Z, et al. Static Checking of Dynamically Generated Queries in Database Applications. *ACM Transactions on Software Engineering and Methodology*.-- Volume: 16, Issue: 4, 2007.

[14] M. Martin, B. Livshits, and M. S. Lam. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. *ACM SIGPLAN Notices*, Volume: 40, Issue: 10, 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. A Testing Framework for Web Application Security Assessment. *Journal of Computer Networks*, Volume: 48 Issue: 5, Pp: 739-761, 2005.

[19] T. Pietraszek and C. V. Berghe. Defending against Injection Attacks through Context-Sensitive String evaluation. *Recent Advances in Intrusion Detection*, Volume: 3858, Pp: 124-145, 2006.

[20] Atefeh Tajpour, Suhaimi Ibrahim, Maslin Masrom, "*Evaluation of SQL Injection Detection and Prevention Techniques"*. International Journal of Advancements in Computing Technology, 2011, Korea.

[21] Bogdan Carstoiu, Dorin Carstoiu. Zatara, the Plug-in-able Eventually Consistent Distributed Database, Journal of AISS, Vol. 2, No. 3, pp. 56-67, 2010.

[22] Dorin Carstoiu, Elena Lepadatu, Mihai Gaspar, "Hbase - non SQL Database, Performances Evaluation", Journal of IJACT, Vol. 2, No. 5, pp. 42-52, 2010.

Atefeh Tajpour: She received her B.S. in Computer Engineering from Iran University of Science and Technology in 1995 and her M.S. in Information Security from University Technology Malaysia in 2010. Also she has more than 12 years experience in application programming and system analysis in Iran. She is currently working toward the PhD degree in computer science in University Technology Malaysia. Her interest is in web application security. She has published articles in IEEE, Computer Society and International Journal of Advancements in Computing Technology. She is also reviewer of IEEE international conference.

Suhaimi Ibrahim received the Bachelor in Computer Science (1986), Master in Computer Science (1990), and PhD in Computer Science (2006). He is an Associate Professor attached to Dept. of Software Engineering, Advanced Informatics School (AIS), Universiti Teknologi Malaysia International Campus, Kuala Lumpur. He currently holds the post of Deputy Dean of AIS. He is an ISTQB certified tester and being appointed a board member of the Malaysian Software Testing Board (MSTB). He has published many articles in international conferences and international journals such as the International Journal of Web Services Practices, Journal of Computer Science, International Journal of Computational Science, Journal of Systems and Software, and Journal of Information and Software Technology. His research interests include software testing, requirements engineering, Web services, software process improvement, mobile and trusted computing.

Mohammad Sharifi received a Master's degree in Software Engineering and a Doctorate in Information Systems (Information Technology management and Improvement) from the IAUN (Iran) and UTM (Malaysia) in 2006 and 2010 respectively. He has several years' experiences at different universities in Iran and Malaysia and the inventor of two inventions in Iran as well. He has also authored and co-authored a lot of reviewed scientific publications and distinguished reviewer of some IEEE international conferences. His main research interests are IT Service and Security Management, IT Governance, E-Health, SMEs and other related fields.