SQL Injection Detection and Prevention Techniques

Article *in* International Journal of Advancements in Computing Technology · August 2011

DOI: 10.4156/ijact.vol3.issue7.11

3 authors:

Atefeh Tajpour
Universiti Teknologi Malaysia
11 PUBLICATIONS **158** CITATIONS
SEE PROFILE

Suhaimi Ibrahim
Universiti Teknologi Malaysia
140 PUBLICATIONS **871** CITATIONS
SEE PROFILE

Maslin Masrom
Universiti Teknologi Malaysia
109 PUBLICATIONS **844** CITATIONS
SEE PROFILE

SQL Injection Detection and Prevention Techniques

Atefeh Tajpour, Suhaimi Ibrahim, Maslin Masrom
University Technology Malaysia, tajpour81sn@yahoo.com, suhaimiibrahim@utm.my, maslin@ic.utm.my

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 SQL injection attack types and also current techniques which can detect or prevent these attacks. Finally we evaluate these techniques.

Keywords: *SQL Injection Attacks, detection, prevention, evaluation, technique, web application security.*

1. INTRODUCTION

Web applications are often vulnerable for 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 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 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.

2. OVERVIEW OF SQL INJECTION ATTACK

2.1. 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 [31]. Meanwhile, large companies developing SQL-based database management systems rely heavily on hardware to ensure the desired performance [32].

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.

2.2. SQL Injection Attacks (SQLIAs) 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. Figure 1 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

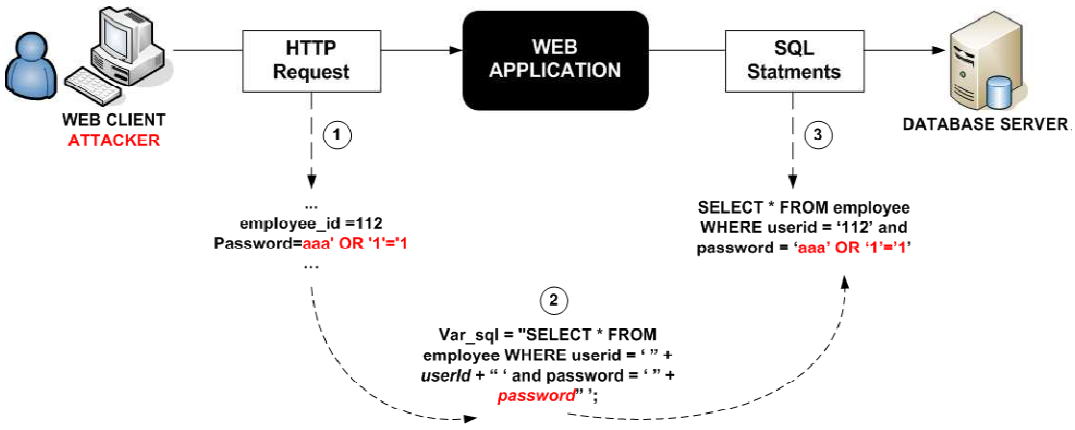

Figure 1: Example of a SQL Injection Attack

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.

2.3. CONSEQUENCE OF SQLIA

The results of SQLIA can be disastrous because a successful SQL injection can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administrative operations on the DB (such as shutdown the DBMS), recover the contents on the DBMS file system and execute commands (xp cmdshell) to the operating system. The main consequences of these vulnerabilities are attacks on [11]:
Authorization: Critical data that are stored in a vulnerable SQL database may be altered by a successful SQLIA, as authorization privilege.
Authentication: If there is no any proper control on username and password inside the authentication page, it may be possible to login to a system as a normal user without knowing the right username and/or password.
Confidentiality: Usually databases are consisting of sensitive data such as personal information, credit card numbers and/or social numbers. Therefore loss of confidentially is a big problem with SQL Injection vulnerability. Actually, theft of sensitive data is one of the most common intentions of attackers.
Integrity: By a successful SQLIA not only an attacker reads sensitive information, but also, it is possible to change or delete this private information.

3. CLASSIFICATION OF SQLIA

Depending on many factors, SQL Injection attacks can be divided into several groups. Here, there are two categories in which are possible to unite all the SQLIAs. This classification is according to the main publications related to the phenomena of SQL Injection [2].

3.1. By Intent

An important classification of SQLIA is related to the attacker's intent, or in other words, the goal of the attack [4].

- **Extracting data** - This category of attacks tries to extract data values from the back end database. Based on the type of web application, this information could be sensitive, for example, credit card numbers, social number, private data are highly valuable to the attacker. This kind of intent is the most common type of SQLIA.
- **Adding or modifying data** -The purpose of these attacks is to add or change data values within a database.
- **Performing database finger printing** – In this category of attack the malicious user wants to discover technical information on the database such as the type and version that a specific web application is using. It is noticeable that certain types of databases respond differently to different queries and attacks, and this information can be used to "fingerprint" the database. Once the intruder knows the type and the version of the database it is possible to organize a particular attack to that database.
- **Bypassing authentication** -By this attack, intruders try to bypass database and application authentication mechanisms. Once it has been over passed, such mechanisms could allow the intruder to assume the rights and privileges associated with another application user.
- **Identifying inject able parameters**- Its goal is to explore a web application to discover which parameters and user-input fields are vulnerable to SQLIA. By using an automated tool called a "vulnerabilities scanner" this intent can be identified.
- **Determining database schema** -The goal of this attack is to obtain all the database schema information (such as table names, column names, and column data types). This is very useful to an attacker to gather this information to extract data from the database successfully. Usually by exploiting specific tools such as penetration testers and vulnerabilities scanners this goal is achieved.
- **Evading detection** – in this category attackers try to avoid auditing and detection, including evading defensive coding practices and also many automated prevention techniques. These attack techniques try to bypass Intrusion Detection and Prevention or other security mechanisms which provide security for systems.
- **Performing denial of service** – in this category intruders make interrupt in system services by performing some instruction so the database of a web application shutdown, thus denying service happen. Attacks involving locking or dropping database tables also fall into this category.
- **Executing remote commands** -The goal of this action is to execute arbitrary commands on the database. These commands can be stored procedures or functions available to database users. This kind of attack is the most dangerous because it may allow the intruder to gain control of the whole system. For instance Microsoft's SQL Server supports a stored procedure xp - cmdshell that permits to arbitrary command execution, and if this attack runs then complete compromise of the server is unavoidable.

3.2. By Type

There are different methods of attacks. 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, 11] 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.

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.

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. 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 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.

4. PREVENTION OF SQL INJECTION ATTACK.

Researchers have proposed a wide range of techniques to address the problem of SQL injection. These techniques range from development best practices to fully automated frameworks for detecting and preventing SQLIAs [1].

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.

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.

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.

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.

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.

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.

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.

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.

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.

5. EVALUATION

In this section, the SQL injection detection or prevention techniques presented in section V would be compared. It is noticeable that this comparison is based on the evaluation which the authors of thechniques have done, not empirically experience, because most of techniques are not available. It is noticeable that Halfond, Viegas, and Orso [4] have done a complete evaluation for SQL injection detection techniques in 2006 but this paper covers more techniques after 2006.

5.1. 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.

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

$$\frac{\text{Number of techniques that can stop Tautology}}{\text{Total number of techniques}} \times 100 = \frac{14}{23} \times 100 = 61$$

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 Hardening (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	•	°	-	•	°	°	°	-	•	•	•	-	-	°	•	•	°	°	•	°	•	•	•

It is evident that only 26% of techniques can stop Stored Procedure. Nevertheless it is unfortunate that 39% of techniques cannot stop this type of attack and Alter Encodings is in the second grade with 48% of stopping and 17% of non stopping by detective or preventive techniques. It is interesting that almost steadily, 35% of attack types could be addressed by these techniques partially.

Table 2: Percentage of Techniques that Detect or Prevent SQL Injection Attacks

	Attack Type	Techniques that can stop all attacks of that type(•)	Techniques that address the attack type only partially(°)	Techniques that cannot stop attacks of that type(-)
1	**Taut**	**%61**	**%35**	**%4**
2	**Illegal/ Incorrect**	**%56**	**%35**	**%9**
3	**Piggy-back**	**%61**	**%35**	**%4**
4	**Union**	**%61**	**%35**	**%4**
5	**Stored Proc**	**%26**	**%35**	**%39**
6	**Infer**	**%61**	**%35**	**%4**
7	**Alter Encodings**	**%48**	**%35**	**%17**

5.2. 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 3.

Table 3 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 3: 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

ACKNOWLEDGMENT

This project is sponsored by the UTM-RU Grant under the Vot. No. 00H68. Thanks to UTM-RMC and other individuals who are directly or indirectly involved in this project.

6. 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 tool, then compare effectiveness, efficiency, stability, flexibility and performance of tools to show the strength and weakness of the tool.

REFERENCES

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

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

[3] Marco Cova, Davide Balzarotti. "Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications", International Symposium On Recent Advances in Intrusion Detection, Volume: 4637, Pages:63-86, 2007.

[4] William G.J. Halfond, Jeremy Viegas and Alessandro Orso, "A Classification of SQL Injection Attacks and Countermeasures", IEEE, 2006.

[5] Zhendong Su and Gary Wassermann, "The Essence of Command Injection Attacks in Web Applications", ACM SIGPLAN Notices, Volume: 41, Pages: 372-382, 2006.

[6] Fredrik Valeur, Darren Mutz, and Giovanni Vigna, “A Learning-Based Approach to the Detection of SQL Attacks”, Conference on Detection of Intrusions and Malware and Vulnerability Assessmen, Volume:3548, Pages:123-140, 2005.
[7] Prithvi 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] Konstantinos Kemalis and Theodoros Tzouramanis,”SQL-IDS: A Specification-based Approach for SQL Injection Detection Symposium on Applied Computing”, pp: 2153-2158, USA, ACM, 2008.
[9] Aske Simon Christensen, Anders Moller, and Michael.I Schwartzbach, “Precise Analysis of String Expressions”, Proceedings of the 10th international conference on Static analysis, ACM, Volume: 2694, Pages: 1-18, 2003.
[10] Gregory T. Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti G, “Using Parse Tree Validation to Prevent SQL Injection Attacks”, In International Workshop on Software Engineering and Middleware (SEM), 2005.
[11] Fabrizio Monticelli, PhD SQLPrevent thesis. University of British Columbia (UBC) Vancouver, Canada, 2008.
[12] Carl Gould, Zhendong Su, and Premkumar Devanbu, “JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications”, Proceedings of the 26th International Conference on Software Engineering , pp 697–698, 2004.
[13] Gary Wassermann, Carl Gould, Zhendong Su and Premkumar Devanbu, “Static Checking of Dynamically Generated Queries in Database Applications”, ACM Transactions on Software Engineering and Methodology, Volume: 16, Issue: 4, 2007.
[14] Michael Martin, Benjamin Livshits and Monica S. Lam, “Finding Application Errors and Security Flaws Using PQL: A Program Query Language”, ACM SIGPLAN Notices, Volume: 40, Issue: 10 Pages: 365-383, 2005.
[15] Yao Wen Huang, Fang Yu, Christian Hang, Chung Hung Tsai, D. T. Lee, Sy Yen Kuo, “Securing Web Application Code by Static Analysis and Runtime Protection”, 12th International World Wide Web Conference , 2004.
[16] William G.J. Halfond and Alessandro Orso, “AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks”, IEEE and ACM International Conference on Automated Software Engineering, USA, 2005.
[17] William G.J. Halfond and Alessandro 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] Yao Wen Huang, Fang Yu, Christian Hang, Chung Hung Tsai, D. T. Lee, Sy Yen Kuo, ”A Testing Framework for Web Application Security Assessment”, Journal of Computer Networks, Volume: 48 Issue: 5, Pages: 739-761, 2005.
[19] Tadeusz Pietraszek, Chris Vanden Berghe, “Defending against Injection Attacks through Context-Sensitive String evaluation”, International Symposium on Recent Advances in Intrusion Detection, Volume: 3858, Pages: 124-145, 2006.
[20] Anh Nguyen Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, David Evans, “Automatically Hardening Web Applications Using Precise Tainting”, Journal of Security and Privacy in the Age of Ubiquitous Computing, Volume: 181, Pages: 295-307, 2005.
[21] Vivek Haldar, Deepak Chandra and Michael Franz. Dynamic Taint Propagation for Java. In Proceedings 21st Annual Computer Security Applications Conference, Dec. 2005.
[22] Stephen W. Boyd, Angelos 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] Gary Wassermann, Zhendong 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] William R. Cook, Siddhartha Rai. Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), 2005.
[25] Russell A. McClure, Ingolf H. Krüger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. Proceedings of the 27th International Conference on Software Engineering (ICSE 05), pp 88–96, 2005.
[26] David Scott, Richard Sharp. Abstracting Application-level Web Security. IEEE Transactions on Knowledge and Data Engineering, Volume: 15, Issue: 4, Pages: 771-783, 2003.
[27] V.Benjamin Livshits, Monica S. Lam. Finding Security Errors in Java Programs with Static Analysis. ACM SIGPLAN Notices, Volume: 40, Issue: 10, Pages: 365-383, 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. pp 34-39: ACM, 2008.
[29] Roberto Navigili, Paola Velardi. Quantitative and Qualitative valuation of the OntoLearn Ontology Learning System. Proceedings of the 20th international conference on Computational Linguistics, ACM, 2004.
[30] Michael Howard and David LeBlanc. Writing Secure Code. Microsoft Press, Redmond, Washington (second edition), 2003.
[31] Bogdan Carstoiu, Dorin Carstoiu. Zatara, the Plug-in-able Eventually Consistent Distributed Database, Journal of AISS, Vol. 2, No. 3, pp. 56-67, 2010.
[32] Dorin Carstoiu, Elena Lepadatu, Mihai Gaspar, "Hbase - non SQL Database, Performances Evaluation", Journal of IJACT, Vol. 2, No. 5, pp. 42-52, 2010.