A Detail Survey on Various Aspects of SQLIA

Sudhakar Choudhary
Student
SISTech-E
Bhopal, MP, India

Arvind Kumar Jain
Assistant Professor
SISTech-E
Bhopal, MP, India

Anil Kumar
M.Tech
IIIT, Allahabad
UP, India

ABSTRACT
While using internet for proposing online services is increasing every day, security threats in the web also increased dramatically. One of the most serious and dangerous web application vulnerabilities is SQL injection. SQL injection attack took place by inserting a portion of malicious SQL query through a non-validated input from the user into the legitimate query statement. Consequently database management system will execute these commands and it leads to SQL injection. A successful SQL injection attack interfere Confidentiality, Integrity and availability of information in the database. Based on the statistical researches this type of attack had a high impact on business. Finding the proper solution to stop or mitigate the SQL injection is necessary. To address this problem security researchers introduce different techniques to develop secure codes, prevent SQL injection attacks and detect them. In this paper the authors present a comprehensive review of different types of SQL injection and various aspects related to SQL injection attacks. Such a structural classification would further help other researchers to choose the right technique for the further studies.

Keywords
Web Application Vulnerability, SQL Injection Types, SQL Injection.

1. INTRODUCTION
Information is the most important business asset today and achieving an appropriate level of information security can be viewed as essential requirement. SQL Injection Attacks (SQLIAs) are one of the topmost threats for web application security and SQL injections are one of the most serious vulnerability types. They are easy to detect and exploit; that is why SQLIAs are frequently employed by malicious users for different reasons, e.g. financial fraud, theft confidential data, deface website, sabotage, espionage, cyber terrorism, or simply for fun. Structured Query Language injection is a code injection technique that used to attack database driven web application. In this attack the attacker inserts a portion of SQL statement via not sanitized user input parameters into the original SQL query and passes them to database server. Based on Open Web Application Security Project (OWASP) studies, SQL injection has the first position in the top 10 list of web application vulnerabilities [2]. The targets of these attacks are not only limited to the web application but they also can hits desktop applications which their databases are powered by SQL. The amount of financial losses in result of SQL Injection was enormous, therefore finding a solution to stop SQL Injection attacks is necessary. Attackers may insert the malicious query via a web form or directly by appending the malicious query to the end of the URL in the address bar of browser.

In a more unusual way of attack, attacker might try to inject the malicious variable through HTTP headers. For instance when the web application have a module that record the statistic related to the users activities such as users IP address, browser type and language. Basically these data will fetch from the HTTP header which comes from the user browser and it will be stored inside the database for further analysis or drawing charts. Changing the HTTP headers is very simple by using specific programs which are designed for this goal or headers add-ons in browsers.

There must be some rules that one should be incorporated in every website to make it secure from SQL injections. Many Web applications can be exploited because the user input is being processed in an unsafe manner. All the data provided by a user must be treated as untrustworthy. One of the key requirements for a Web application's security is the proper user input handling, which is not always an easy task. To propose the classification the inputs based on probability and use of character as a vulnerability that helps to identify in SQL detection process. Proper neutralization of such special characters used in an SQL command to avoid the SQL injection.

2. CONSEQUENCES AND ATTACK INTENTIONS
With SQL injection, cyber criminals can take complete remote control of the database, with the consequences that they can become able to manipulate the database to do anything they want to, including:

1. Shut down or Delete a database.
2. Upload or Download files.
3. Through reverse lookup, gather IP addresses and attack those computers with an injection attack.
4. Corrupting, deleting or changing files and interact with the OS, reading and writing files.
5. Online shoplifting e.g. changing the price of a product or service, so that the cost is negligible or free.
6. Insert a bogus name and credit card in to a system to scam it at a later date.

When a threat agent utilizes a crafted malicious SQL input to launch an attack, the attack intention is the goal that the threat agent tries to achieve once the attack has been successfully executed. Some of intensions are:

Identifying inject-able parameters, Performing database, finger-printing, Bypassing Authentication, Determining database schema, Adding or modifying data, Extracting data and Executing remote commands, Evading detection and Performing denial of services.

3. INPUT VALIDATION BASED VULNERABILITY
The most prominent class of input validation errors are SQL injections. SQL injections are the classes of vulnerabilities in which an attacker causes the web application server to

produce HTML documents and database queries, respectively, that the application programmer did not intend. In that sense, SQL injections are integrity violations in which low-integrity data is used in a high-integrity channel; that is, the browser or the database executes code from an un-trusted user, but does so with the permissions of the application server.

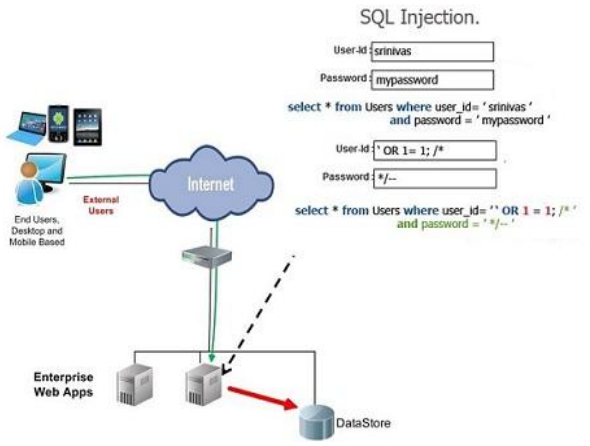

Figure 2.1: How malicious inputs is injected

Figure 2.1 shows how a SQL injection attack occurs. In that figure, it can be seen that malicious code is injected in input textbox. In place of *User-id* attacker places *'OR 1=1--'* and *Password* field left blank.

To highlight how ubiquitous web applications have become and how prevalent their problems are, Figure-2.2 shows, for each year from 2008 to 2016, the percentage of newly reported security vulnerabilities in eight vulnerability classes: Dos, Code Execution, XSS, SQL injection, buffer overflows, Gain Information, Number of Exploits and directory traversals. These were the eight most reported vulnerabilities during these years. All of these except buffer overflows are specific to web applications. Note that SQL injections are consistently at or near the top: 9-10% of the reported vulnerabilities during these years. Some web security analysts speculate that because web applications are highly accessible and databases often hold valuable information, the percentage of SQL injection attacks being executed is significantly higher than the percentage of reported vulnerabilities would suggest.

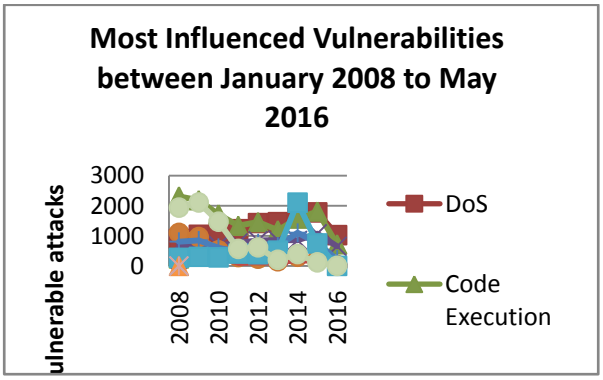

Figure 2.2: Most Inflenced Vulnerabilities

A report says [4], in 2008, SQL Injection was on its peak with 14-16%. In 2008, SQL injection replaced cross-site scripting as the predominant Web application vulnerability. In fact, the overall increase of 2008 Web application vulnerabilities can be attributed to a huge spike in SQL injection vulnerabilities, which was up a staggering 134 percent from 2007.

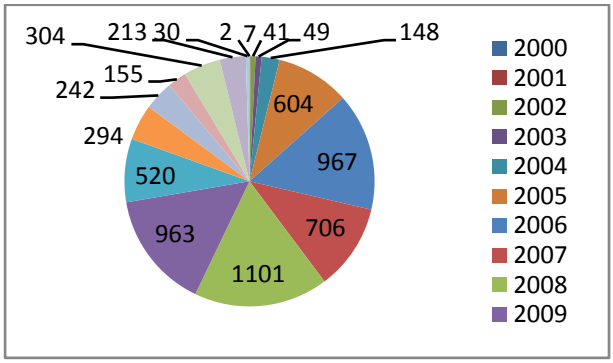

Figure 2.3: Number Of SQL Injection Attacks

Figure 2.3 shows the number of attacks of SQL injection from January 2000 to May 2016, and one can see the increment of number of attacks in 2008.

4. TYPES OF SQL INJECTION

In this section, the authors discussed about the various types of SQL Injection Attacks. Among various types, some are frequently used by the attackers. Hence, in this section, the authors present an in-depth look at some of the most common SQL Injection Attacks. The authors explain each of these major attacks with simple examples. SQL injection attacks are classified under seven main categories.

4.1 Tautology

SQL injection codes are injected into one or more conditional statements so that they always evaluate to true. The most common usages are to bypass authentication pages and extract data. In this type of injection, an attacker exploits an inject-table field that is used in a query's WHERE clause. Transforming the conditional condition into a tautology causes all of the rows in the database table targeted by the query to be returned. Typically, the attack is successful when the code either displays all of the returned records or performs some action if at least one record is returned.

*SELECT * FROM Employee WHERE EmpName= 'or1=1--' AND EmpPwd= ''*

In the above example, the code injected in the condition *OR1=1* transforms the entire WHERE clause into a tautology. The database uses the conditional as the basis for evaluating each row and deciding which ones to return to the application. Because the conditional is a tautology, the query evaluates to true for each row in the table and returns all of them. After injecting code into a particular field, legitimate code that follows are nullified through usage of *end of line comments*.

4.2 Logically Incorrect Query

This attack lets attacker gather important information about the type and structure of the back-end database of a Web application. The attack is considered as an information gathering step for other attacks. The vulnerability leveraged by this attack is that the default error page returned by application servers is often overly descriptive. In fact, an error messages is generated can often reveal vulnerable parameters to an attacker. Additional error information, originally intended to help programmers to debug their applications. When performing this attack, an attacker tries to inject statements that cause a syntax, type conversion, or logical error into the database. Syntax errors can be used to identify inject-able parameters. Type errors can be used to deduce the data types of certain columns or to extract data. Logical errors often reveal the names of the tables and columns that caused

the error.

http://www.wShop.com/Items/Items.aspx?ItemId=123 UNION SELECT TOP 1 COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME='admin'--

The injected query extracts the 1st column name of *adminlogin* table from the *INFORMATION_SCHEMA* database. The query then converts the table name into an integer but this is not a legal type conversion, the database throws an error. For Microsoft SQL Server, the error would be:

Microsoft OLEDB Provider for ODBC Drivers error '80040e07' [Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value "AdminId' to a column of data type int./index.asp, line 5.

There are two useful pieces of information in this message. First, the attacker can see that the database is an SQL Server database, as the error message explicitly states this fact. Second, the error message reveals the value of the string that caused the type conversion to occur. In this case, this value is also the name of the first column name of *admin* table in the database: *AdminId*. A similar strategy can be used to systematically extract the name and type of each column in the database.

4.3 Union Query

In union-query attacks, an attacker exploits a vulnerable parameter to change the data set returned for a given query. With this technique, an attacker can trick the application into returning data from a table different from the one that was intended by the developer. Because the attackers completely control the second injected query, they can use that query to retrieve information from a specified table. The result of this attack is that the database returns a dataset that is the union of the results of the original first query and the results of the injected second query.

*SELECT * FROM Employee WHERE EmpName= '' UNION SELECT password from Customer where CustName= 'abc'-- AND CustPwd= ''*

Assuming that there is no login equal to '', the original first query returns the null set, whereas the second query returns data from the *Customer* table. In this case, the database would return column *password* for *name 'abc'*. The database takes the results of these two queries, unions them, and returns them to the application. In many applications, the effect of this operation is that the value for *CustPwd* is displayed along with the user information.

4.4 Piggy-backed Query

In this attack type, an attacker tries to inject additional queries into the original query. This query is different from others because, in this case, attackers are not trying to modify the original intended query; instead, they are trying to include new and distinct queries that *piggy-backed* to the original query. As a result, the database receives multiple SQL queries. The first is the intended query which is executed as normal; the subsequent ones are the injected queries, which are executed in addition to the first. This type of attack can be extremely harmful. If successful, attackers can insert virtually any type of SQL command, including stored procedures. Vulnerability to this type of attack is often dependent on having a database configuration that allows multiple statements to be contained in a single string.

*SELECT * FROM Employee WHERE EmpName= 'doe' AND EmpPwd= ''; drop table Employee -- '*

After completing the first query, the database would recognize the query delimiter *";"* and execute the injected second query. The result of executing the second query would be to drop table *users*.

4.5 System Stored Procedure

SQLIAs of this type try to execute stored procedures present in the database. Today, most database vendors develop databases with a standard set of stored procedures that extend the functionality of the database and allow for interaction with the operating system. Therefore, once an attacker determines which backend database is in use, SQLIAs can be crafted to execute stored procedures provided by that specific database, including procedures that interact with the operating system.

CREATE PROCEDURE DBO.isAuthenticated @EmpName varchar2, @EmpPwd varchar2, @EmpPin int AS
*EXEC ("SELECT * FROM Employee WHERE EName = '" +@ EmpName + "' and EPwd = '" +@ EmpPwd + "' and EPin = '" +@ EmpPin "'");*
GO

This example demonstrates how a parameterized stored procedure can be exploited via an SQLIA. In the example, it is assume that the query string constructed in this example has been replaced by a call to the stored procedure defined above. The stored procedure returns a true/false value to indicate whether the user is authenticated or not. To launch an SQLIA, the attacker simply injects *'; SHUTDOWN;* -- into either the login or pass fields. This injection causes the stored procedure to generate the following query:

*SELECT * FROM Employee WHERE EName = 'doe' AND EPwd = ''; SHUTDOWN;* --

At this point, this attack works like a piggy-back attack. The first query is executed normally, and then the second, malicious query is executed, which results in a database shut down. This example shows that stored procedures can be vulnerable to the same range of attacks as traditional application code.

4.6 Inference

In this attack, the query is modified to recast it in the form of an action that is executed based on the answer to a true/false question about data values in the database. In this type of injection, attackers are generally trying to attack a site that has been secured enough so that, when an injection has succeeded, there is no useful feedback via database error messages. Since database error messages are unavailable to provide the attacker with feedback, attackers must use a different method of obtaining a response from the database. In this situation, the attacker injects commands into the site and then observes how the function of the website changes. By carefully noting when the site behaves the same and when its behavior changes, the attacker can deduce not only whether certain parameters are vulnerable, but also additional information about the values in the database. There are two well known attack techniques that are based on inference. They allow an attacker to extract data from a database and detect vulnerable parameters.

(1) *Blind Injection:* In this technique, the information must be inferred from the behavior of the page by asking the server

true/false questions. If the injected statement evaluates to true, the site continues to function normally. If the statement evaluates to false, although there is no descriptive error message, the page differs significantly from the normally-functioning page.

(2) *Timing Attacks:* A timing attack allows an attacker to gain information from a database by observing timing delays in the response of the database. This attack is very similar to blind injection, but uses a different method of inference. To perform a timing attack, attackers structure their injected query in the form of an if-then statement, whose branch predicate corresponds to an unknown about the contents of the database. Along one of the branches, the attacker uses a SQL construct that takes a known amount of time to execute, (e.g. the WAITFOR keyword, which causes the database to delay its response by a specified time). By measuring the increase or decrease in response time of the database, the attacker can infer which branch was taken in his injection and therefore the answer to the injected question.

Example*:* Using the code, the authors illustrate two ways in which Inference based attacks can be used. The first of these is identifying inject-able parameters using blind injection. Consider two possible injections into the login field. The first being

[legalUser' and 1=0 - -] and the second,
[legalUser' and 1=1 - -].

These injections result in the following two queries:

*SELECT * FROM Employee WHERE EmpName= 'abc' and 1=0 --' AND EmpPwd = ''*

*SELECT * FROM Employee WHERE EmpName ='abc' and 1=1 --' AND EmpPwd =''*

In the first scenario, it is a secure application, and the input for login is validated correctly. In this case, both injections would return login error messages, and the attacker would know that the login parameter is not vulnerable. In the second scenario, the query is an insecure application and the login parameter is vulnerable to injection. The attacker submits the first injection and, because it always evaluates to false, the application returns a login error message. At this point however, the attacker does not know if this is because the application validated the input correctly and blocked the attack attempt or because the attack itself caused the login error. The attacker then submits the second query, which always evaluates to true. If in this case there is no login error message, then the attacker knows that the attack went through and that the login parameter is vulnerable to injection.

The second way inference based attacks can be used is to perform data extraction. Here example illustrates how to use timing based inference attack to extract a table name from the database. In this attack, the following is injected into the login parameter:

['abc' and ASCII (SUBSTRING ((select top 1 name from sysobjects), 1, 1)) > X WAITFOR 5 –'].

This produces the following query:

*SELECT * FROM Employee WHERE EmpName = 'abc' and ASCII (SUBSTRING ((select top 1 name from sysobjects), 1, 1)) > X WAITFOR 5 --' AND EmpPwd = ''*

In this attack the SUBSTRING function is used to extract the first character of the first table's name. Using a binary search strategy, the attacker can then ask a series of questions about this character. In this case, the attacker is asking if the ASCII value of the character is greater-than or less-than or equal-to the value of X. If the value is greater, the attacker knows this by observing an additional 5 second delay in the response of the database. The attacker can then use a binary search by varying the value of X to identify the value of the first character.

4.7 Alternate Encodings

This attack type is used in conjunction with other attacks. In other words, alternate encodings do not provide any unique way to attack an application. These evasion techniques are often necessary because a common defensive coding practice is to scan for certain known "bad characters," such as single quotes and comment. To evade this defense, attackers have employed alternate methods of encoding their attack strings (e.g., using hexadecimal, ASCII, and Unicode character encoding). Different layers in an application have different ways of handling alternate encodings. The application may scan for certain types of escape characters that represent alternate encodings in its language domain. Another layer (e.g., the database) may use different escape characters or even completely different ways of encoding.

For example, a database could use the expression char (120) to represent an alternately-encoded character "x", but char (120) has no special meaning in the application language's context. An effective code-based defense against alternate encodings is difficult to implement in practice because it requires developers to consider of all of the possible encodings that could affect a given query string as it passes through the different application layers. Therefore, attackers have been very successful in using alternate encodings to conceal their attack strings.

Example: Because every type of attack could be represented using an alternate encoding, here the authors simply provide an example of how esoteric an alternatively-encoded attack could appear. In this attack, the following text is injected into the login field:

abc'; exec (0x73687574646f776e)--.

The resulting query generated by the application is:

*SELECT * FROM Employee WHERE EmpName= 'abc'; exec (char (0x73687574646f776e)) -- AND EmpPwd= ''*

This example makes use of the char () function and of ASCII hexadecimal encoding. The char () function takes as a parameter an integer or hexadecimal encoding of a character and returns an instance of that character. The stream of numbers in the second part of the injection is the ASCII hexadecimal encoding of the string *"SHUTDOWN"*. Therefore, when the query is interpreted by the database, it would result in the execution, by the database, of the *SHUTDOWN* command.

5. EVATION TECHNIQUES

Evasion techniques are techniques that employed in an attack to avoid detection by signature-based detection system. In the context of SQL injection detection, a signature is the pattern of known attack strings. SQL injection attack occurs when input string changes the intended syntactical structure of SQL statement. Signature-based detection systems build a database of attack signatures, and then examine input strings against the signature database at runtime in detection of attacks. Evasion techniques obscure input strings, making look different but yielding the same results when executed by a database server.

5.1 Sophisticated Matches

One of the most famous signatures used by such mechanisms is some sort of variant of *OR 1=1* attack. Sophisticated matches evasion technique uses alternative expression of *"OR 1=1".*

OR "Unusual"="Unusual",
OR "Simple"="Sim"+"ple",
"OR 2>1"
All have the same effect as *"OR 1=1".*

5.2 Hex Encodings

This technique uses hexadecimal encoding to represent a string. For example, the string *SELECT* can be represented by the hexadecimal number 0x73656c656374, which most likely will not be detected by a signature protection mechanism. See the example given below which shows the content of c:\boot.ini

SELECT LOAD_FILE (0x633A5C626F6F742E696E69)

5.3 Char Encodings

This technique uses build-in CHAR function to represent a character, which make it very difficult for detection system to build a signature that match it.

"SELECT" can be represented as
char (73) + char (65) + "LECT"

5.4 In-Line Comment

This technique complicates input strings by inserting in-line comments between SQL keywords. One can escape detection from signatures that expect white space between SQL keywords.

*/**/UNION/**/SELECT/**/*

5.5 Remove White Space

This technique complicates input strings by dropping white space between SQL keyword and string or number literals.

OR "Simple"="Simple" works exactly the same way as
OR"Simple" = "Simple",

But has no spaces in it, make it capable of evading any spaces based signature.

5.6 Break Words

In MySQL, the in-line comments would not work as space. The in-line comments can be used in MySQL to break words in the middle,

*UN/**/ION/**/ SE/**/LECT/**/* is evaluated as
UNION SELECT.

6. COUNTERMEASURES

There are a number of ways a programmer/system administrator can prevent or counter attacks made on their systems. Although these techniques remain the best way to prevent SQL injection vulnerabilities, but their application is problematic in practice. These techniques are prone to human error and are not as rigorously and completely applied as automated techniques. While most developers do make an effort to code safely, it is extremely difficult to apply defensive coding practices rigorously and correctly to all sources of input. In fact, many of the SQL injection vulnerabilities discovered in real applications are due to human errors: developers forgot to add checks or did not perform adequate input validation.

6.1 Parameterized Query

Parameterized query is parameterized database access API provided by development platform such as PrepareStatement in Java or SQLParameter in .NET. Instead of composing SQL by concatenating string, each parameter in a SQL query is declared using place holder and input is provided separately.

6.2 Least Privilege

The account that an application uses to access the database should have only the minimum permissions necessary to access the objects that it needs to use. Use a different database account for a task that requires a different level of privilege.

6.3 Customized Error Message

Threat agents may gain access to knowledge through overly informative error messages, yet completely removing error messages makes debugging a difficult task. Customized error messages hinder the reconnaissance progress of threat agents, particularly in deducing specific details such as inject-able parameters, etc.

6.4 System Stored Procedure Reduction

Once a threat agent gains knowledge of which back-end server is used, he/she has knowledge of an entire set of system stored procedures that are available. By limiting the system stored procedures one can execute on a server, especially the processes that are not used, one can reduce or even eliminate vulnerabilities that may arise from these stored procedures.

6.5 SQL Keyword Escaping

Escape specific SQL keyword or delimiter in the input string like semicolon, double dash, single quote etc.

6.6 Input Variable Length Checking

By checking for input variable length, malicious code strings beyond certain length limits will not be applicable. Even if the length limitation is long enough to fit a few additional queries, the inability to input an infinitely long string disables the threat agent from employing evasion techniques such as encoding, and consequently, allows signature based detection mechanisms to intercept simple attacks.

7. CONCLUSION AND FUTURE WORK

Though many approaches and frameworks have been identified and implemented in many interactive Web applications, security still remains a major issue. It would be difficult to give a clear verdict which scheme or approach is the best as each one has some proven benefits for specific types of settings or systems. SQL Injection prevails as one of the top-5 vulnerabilities and threat to online businesses targeting the backend databases. In this paper, the authors have reviewed the most popular existing SQL Injections related issues.

As a future work, the authors would like to develop a countermeasure that can efficiently tackle the innovative SQL Injection attacks and fix as much vulnerability as possible. Hackers are in reality very innovative and as the time is passing by, new attacks are being launched that may need new ways of thinking about the solutions currently have at our hands. A strong countermeasure can remove or at least block all the available vulnerabilities in a system and thus it could protect it against various types of attacks that take advantage of the vulnerabilities.

8. REFERENCES

[1] Wikipedia, "information security" http://en.wikipedia.org/wiki/Information_security

[2] (OWASP), "O.W.A.S.P. Top 10 Vulnerabilities."; Available from:https://www.owasp.org/index.php/Top_10 2013

[3] Wikipedia, "web application", http://en.wikipedia.org/wiki/Web_application

[4] https://www.cvedetails.com/vulnerabilities-by-types.php

[5] Wikipedia, "SQL injection" http://en.wikipedia.org/wiki/SQL_injection

[6] secerno.com," SQL Injection Attack: A Security Threat",http://www.secerno.com/?pg=SQL-Injection#2

[7] IBM, IBM Internet Security SystemsX-Force 2008 Trend & Risk Report, Jan 2009,http://www935.ibm.com/services/us/iss/xforce/trendreports/xforce-2008-annual-report.pdf

[8] W. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures. *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE)*, 2006.

[9] Stephen Thomas, Laurie Williams. Using Automated Fix Generation to Secure SQL Statements. Third International Workshop on Software Engineering for Secure Systems (SESS'07), pages 9-9,May 2007.

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

[11] F. Valeur, D. Mutz, and G. Vigna. A Learning-Based Approach to the Detection of SQL Attacks. *In Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, pages 123–140, 2005.

[12] S. Boyd and A. Keromytis. SQLrand: Preventing SQL injection attacks. In Proceedings of the Applied Cryptography and Network Security (ACNS), pages 292–304, 2004.

[13] T. Pietraszek and C. Vanden Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. *In Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 124–145, 2005.

[14] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. *Annual Symposium on Principles of Programming Languages (POPL)*, pages 372–382, 2006.

[15] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "Using Parse Tree Validation to Prevent SQL Injection Attacks," 5th International Workshop on Software Engineering and Middleware, pages 106-113, 2005

[16] W. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQLInjection Attacks. *In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 174–183, 2005.

[17] William G.J. Halfond and Alessandro Orso. Preventing SQL Injection Attacks Using AMNESIA. *Proceedings of the 28th international conference on Software engineering*. Pages 795-798, May 2006

[18] William G. J. Halfond, Alessandro Orso. Combining Static Analysis & Runtime Monitoring to Counter SQL-Injection Attacks. *SIGSOFT Software Engineering Notes Volume 30 Issue 4*. July 2005.