International Journal of Applied Engineering Research ISSN 0973-4562 Volume 13, Number 11 (2018) pp. 9874-9880

A Secure Coding Approach For Prevention of SQL Injection Attacks

Bhawana Gautam[1], Jyotiraditya Tripathi[2] and Dr. Satwinder Singh[3]

[1]M.Tech. Student, [2]M.Tech. Student, [3]Assistant Professor

[1,2,3]Department of Computer Science & Technology, Central University of Punjab, Bathinda, Punjab, India.

Abstract:-

SQL injection attack is considered as one of the most dangerous threat in the world of web application security. This attack occurs when an attacker/hacker succeeds to insert the malicious code in any particular web application and these injected codes successfully interacts with database query. The injection of such type of infected queries can be done by attackers through URLs or from web forms. The impact of SQL injection attacks can be very severe if target web application contains very sensitive information in its database such as credit/debit card details, national security policy details or some other information in terms of confidentiality. In this research paper, a secure coding approach is proposed that can be used by web developers and security professionals to secure their application against such type of attacks at the time of coding. To check the accuracy and efficiency of proposed approach, several real time PHP based web applications have been tested and a comparison analysis is done among previous prevention techniques and proposed technique.

Keywords: SQLi, Tautology, web security, dbms, php

INTRODUCTION

Generally people say that SQL injection is not new for them and they know about this terminology but fact is totally different from this and they have only heard about it or experienced only through trivial examples. SQL injection is one of the most devastating vulnerabilities that's impact on business can be very severe as it can expose all sensitive information which are stored in an application's database it may also include handy information such as usernames, passwords, names, addresses, phone numbers, credit & debit card details.

Hence, SQL injection is very severe vulnerability that results when application developer gives ability to hacker/attacker that they can influence Structured Query Language (SQL) queries which an application passes to a back-end database. Basically it is the hacking technique which attempts to pass SQL statements through a web application for execution by the backend database. If input data is not sanitized properly, web applications may be victim SQL Injection attacks that allow hackers to view and extract sensitive information from the database and/or even wipe it out.

Database applications typically call for user input for data to add to the database. A SQL injection attack occurs when user modifies a SQL statement through a user input field. Therefore, SQL injection attacks can occur when SQL statements are dynamically created using user input. The threat occurs when users enter malicious code that 'tricks' the database into executing unintended commands. If we have to stop SQL injection attack then we have to use input validation for every necessary field. Input validations can be used to prevent SQL injection attack.

UNDERSTANDING SQL INJECTION

Now Web applications are more sophisticated and complex in terms of modules and functions. Web application is now everywhere and in every prospects of life and working, it ranges from dynamic Internet and intranet portals, like e-commerce sites, social networking sites etc, to HTTP-delivered enterprise applications like document management systems and ERP applications and CMS based applications. The availability of such systems and the type and sensitivity level of the data that they handle and process are increasing the critical situation of almost all major businesses, It is not just happening to those applications that have online e-commerce stores. Web applications, their server and their infrastructure and environments in which they are being operated are very diverse in terms of technologies and consist of significant amount of modified and customized codes. The rich-feature design and the capability of collating, processing, and disseminating the information over the Internet or from within an intranet makes web applications very popular target among the attacker community. Also, we know that the network security market is matured now and there are very possibilities to breach information systems by using network-based vulnerabilities, so that hackers are concentrating and switching their focus on application to compromise the security systems.

Basically SQL injection is type of attack where attacker injects or appends the SQL code into application/user input parameters that are later passed to a back-end SQL server for parsing and execution to extract or manipulate the data in unauthorized manner. Any procedure that is able to construct SQL statements can be vulnerable potentially because of the diverse nature of SQL in coding options. The primary form of SQL injection consists of direct insertion of code into parameters that are concatenated with SQL commands and executed. A less direct attack tries to inject some malicious code into strings that are stored in a table or destined to store as metadata. SQL injection attack happens when attacker succeeds to concatenate the stored strings into a dynamic SQL commands and the malicious code is executed. If some crucial parameters are not sanitized properly means these parameters are passed to dynamically created SQL statements then it is very strong possibility that attacker can alter the construction of SQL

statements in back-end. When an attacker is able to modify an SQL statement, the statement will execute with the same rights as the application user; when using the SQL server to execute commands that interact with the operating system, the process will run with the same permissions as the component that executed the command (e.g. database server, application server, or Web server), which is often highly privileged.

To illustrate this, let's return to the previous example of a simple online retail store. In this example, we tried an attempt to view all products that are in the store and having lesser cost than $100, by using the following URL:

http://172.158.56.72/products.php?val=100

Now, to test the possibility of such attacks an attempt is done to inject arbitrary SQL commands by appending them to the input parameter *Val*. It can be done by appending the string *'OR '1'='1* to the URL:

http:/172.158.56.72/products.php?val=100' OR '1'='1

This time, because parameters are not properly sanitized, this SQL statement executes successfully and returns all of the products in the database regardless of their price. This is because you have altered the logic of the query. This happens because the appended statement results in the *OR* operand of the query always returning *true*, that is, 1 will always be equal to 1. Here is the query that was built and executed:

SELECT *

FROM ProductsTbl

WHERE Price < '100.00' OR '1' = '1'

ORDER BY ProductDescription;

There are many ways to exploit SQL injection vulnerabilities to achieve a myriad of goals; the success of the attack is usually highly dependent on the underlying database and interconnected systems that are under attack. Sometimes it can take a great deal of skill and perseverance to exploit a vulnerability to its full potential.

The preceding simple example demonstrates how an attacker can manipulate a dynamically created SQL statement that is formed from input that has not been validated or encoded to perform actions that the developer of an application did not foresee or intend. The example, however, perhaps does not illustrate the effectiveness of such vulnerability; after all, we only used the vector to view all of the products in the database, and we could have legitimately done that by using the application's functionality as it was intended to be used in the first place.

DIFFERENT TYPE OF SQLI PREVENTION TECHNIQUES

Black Box Testing. Huang and his colleagues proposed WAVES, this is a black-box technique for testing Web applications regarding SQL injection vulnerabilities. Basically this technique uses a Web crawler that can identify all points in a Web application from where SQLIAs can be performed. It then builds attacks points that points are based on a specified list of patterns and attack techniques. WAVES then monitors the application's response when attack happens and after this it uses machine learning techniques to improve and to secure its attack methodology. To guide its testing, this technique improves over most penetration-testing techniques by using machine learning approaches However, like all black-box and penetration testing techniques, it cannot provide guarantees fully that it can prevent and detect all type of SQLIAs.

Combined Static and Dynamic Analysis: Essentially AMNESIA is a model-based method that includes static investigation and runtime observing. In its static stage, AMNESIA utilizes static investigation to assemble models of the diverse kinds of queries an application can lawfully create at each purpose of access to the database. In its dynamic stage, AMNESIA catches all queries previously they are sent to the database and checks each query against the statically build models. Queries that violate the model are distinguished as SQLIAs and kept from executing on the database. In their assessment, the authors have demonstrated that this system performs well against SQLIAs. The essential constraint of this strategy is that its prosperity is subject to the exactness of its static examination for building query models. Certain kinds of code obscurity or query advancement procedures could make this progression less exact and result in both false positives and false negatives.

Likewise, two late related methodologies, SQLGuard and SQLCheck additionally check queries at runtime to check whether they comply with a model of expected queries. In these methodologies, the model is communicated as a syntax that exclusive acknowledges legitimate queries. In SQLGuard, the model is derived at runtime by inspecting the structure of the query when the expansion of client input. In SQLCheck, the model is determined freely by the designer. Both methodologies utilize a mystery key to delimit client contribution amid parsing by the runtime checker, so security of the approach is subject to assailants not having the capacity to find the key. Also, the use of these two approaches requires the developer to either rewrite code to use a special intermediate library or manually insert special markers into the code.

Taint Based Approaches: WebSSARI distinguishes input-validation related errors utilizing information flow analysis. In this approach, static investigation is utilized to check taint flows against preconditions for sensitive functions. The analysis recognizes the focuses in which preconditions have not been met and can recommend filters and sanitization functions that can be consequently added to the application to fulfill these preconditions. The WebSSARI framework works by considering as sanitized input that has gone through a predefined set of channels. In their assessment, the authors could recognize security vulnerabilities in a scope of existing applications. The essential downsides of this method are that it accepts that satisfactory preconditions for sensitive functions can be precisely communicated utilizing their typing system and that having input going through specific types of filters is adequate to consider of it as not tainted.

For some kinds of functions and applications, this supposition is excessively solid. Livshits and Lam utilize static analysis techniques to recognize vulnerabilities in programming. The fundamental approach is to utilize information flow techniques to detect when tainted input has been used to construct a SQL query. These queries are then hailed as SQLIA vulnerabilities. The authors exhibit the practicality of their strategy by utilizing this way to deal with discovers security vulnerabilities in a benchmark suite. The essential constraint of this approach is that it can identify just known examples of SQLIAs and, on the grounds that it utilizes a moderate investigation and has restricted help for untainting activities, can produce a generally high measure of false positives. A few dynamic tainted analysis approaches have been proposed. Two comparative methodologies by Nguyen-Tuong and associates and Pietraszek and Berghe change a PHP mediator to track exact per-character taint information. The procedures utilize a setting delicate investigation to recognize and dismiss queries if untrusted input has been utilized to make certain kinds of SQL tokens. A typical disadvantage of these two methodologies is that they expect alterations to the runtime condition, which influences compactness. A strategy by Haldar and partners and SecuriFly actualize a comparative approach for Java. In any case, these procedures don't utilize the context sensitive analysis utilized by the other two methodologies and track tainted input on a for every string premise (rather than per character). SecuriFly additionally endeavors to sanitize query strings that have been created utilizing tainted b inputs. In any case, this sanitization approach does not help if injection is performed into numeric fields. In general, dynamic taint-based techniques have shown a lot of promise in their ability to detect and prevent SQLIAs. The essential disadvantage of these methodologies is that recognizing all sources of tainted user input contribution to exceedingly measured Web applications and precisely engendering taint informationis regularly a troublesome assignment..

New Query Development Paradigms: Two late methodologies, SQL DOM and Safe Query Objects, utilize encapsulation of database queries to give a protected and solid approach to get to databases. These strategies offer a compelling method to stay away from the SQLIA issue by changing the query building process from an unregulated one that utilizations string link to a methodical one that uses a sort checked API. Inside their API, they can methodically apply coding best practices, for example, input separating and thorough write checking of client input. By changing the improvement paradigm in which SQL queries are made, these procedures kill the coding hones that make most SQLIAs conceivable. Albeit powerful, these methods have the downside that they expect developers to learn and utilize another programming paradigm or query-development process. Moreover, in light of the fact that they center around utilizing another advancement procedure, they don't give any kind of assurance or enhanced security for existing legacy systems.

Technique	Tautology	Illegal/Inc.	Piggy-back	Union	Stored Procedure	Inference	Alt. Encoding
JDBC-Checker	o	o	o	o	X	X	X
JAVA-Static Tainting	•	•	•	•	•	•	•
Safe-Query Ob	•	•	•	•	X	o	•
Security Gateway	o	o	o	o	o	o	o
SQL DOM	•	•	•	•	X	•	o
WAVES	o	o	o	o	o	o	o
WebSSARI	•	•	•	•	o	o	•

Comparison of prevention-focused techniques with respect to attack types

• Denotes that technique can prevent attack successfully

o Denotes that technique can address attack type but cannot guarantee of completeness

X Denotes that technique is not able to address attack type

PROPOSED APPROACH

There are so many techniques for prevention of SQL injection attack but each has some shortcomings that give ability to hacker or attacker to break the security of application. Among several techniques, main focus of research will be on:

- Input and URL validation
- Data Sanitization
- Prepared Statement (or PDO) for query execution
- Query and session tokenization

1. Input and URL validation:- This is the most common web application security loop hole. It means failure of validating user inputs before using it. This weakness results major vulnerabilities in web applications, such as cross site scripting, SQL injection. Hence to prevent this, "All Input is Evil" make this rule number one. Validation of user inputs includes checking the length, format, range, and allowable characters of user input. For example, if your user form has a text field for mobile number, then you need to validate this field that it will only accept numbers not characters. Hence to avoid SQLIA, all user input must be properly validated and sanitized before interaction with database query.

2. Data Sanitization:-The best suited way to protect against SQL Injection is to sanitize all input data before placing it with a SQL query. Sanitization is a method to modify user input which ensures that it is valid (such as single quotes and white spaces). Basically it is the act of filtering of any character that we don't want from the data. Let us take an example of username/password, the username field should only contain alphanumeric characters. Importantly, username values (and password values, for that matter) should not contain apostrophes. Sanitization of user input always ensures that inputs are containing only the valid characters. Use of regular expressions to strip unwanted characters outside of the predefined "legal" characters is preferable.

Regular expression is the most powerful tool for string parsing and pattern matching. It is not sufficient to apply data sanitization to data from form fields it must be applied on hidden fields, disabled fields and cookies as well. It is fully wrong assumption that client side validation works correctly, because hackers can circumvent this, and they can also edit cookie so data sanitization is very important to prevent SQLIA.

3. Prepared Statement (or PDO) for query execution:- A prepared statement is a feature which is used for execution of the same (or similar) SQL statements repeatedly in efficient manner. PHP is one of the most accessible and widely used programming languages however it also leads insecure codes. PDO (PHP data Object) is a database abstraction layer which allows developer to work with many different types of databases very quickly and securely. A Database Abstraction Layer is able to hide the interaction with the database. It provides all functionality through an API. All this basically means is, you talk to the database through a separate layer that handles all the complexity and processing. Prepared statements basically work like this:-

1. Prepare: First of all, SQL statement template is created and sent to the database. Certain values are left unspecified, these unspecified values are called parameters (labeled "?"). Example: INSERT INTO Registration VALUES(?, ?, ?)
2. Now database parses, compiles, and performs query optimization on the SQL statement template, and stores the result without executing it.
3. Execute: At a later time, when needed the application binds the values to the parameters, and database executes the statement. The application may execute the statement as many times as it wants with different values

   If we compare prepared statements with sql statements that are being executed directly then three main advantages of prepared statement are:-

- Prepared statements reduce parsing time because the preparation on the query is done only once.
- Bounded parameters actually minimize bandwidth to the server because you need to send only the parameters each time. Whole query is not needed to send every time.
- Prepared statements are very useful against SQL injections, because parameter values, which are transmitted later using a different protocol, need not be correctly escaped. If the original statement template is not derived from external input, SQL injection cannot occur.

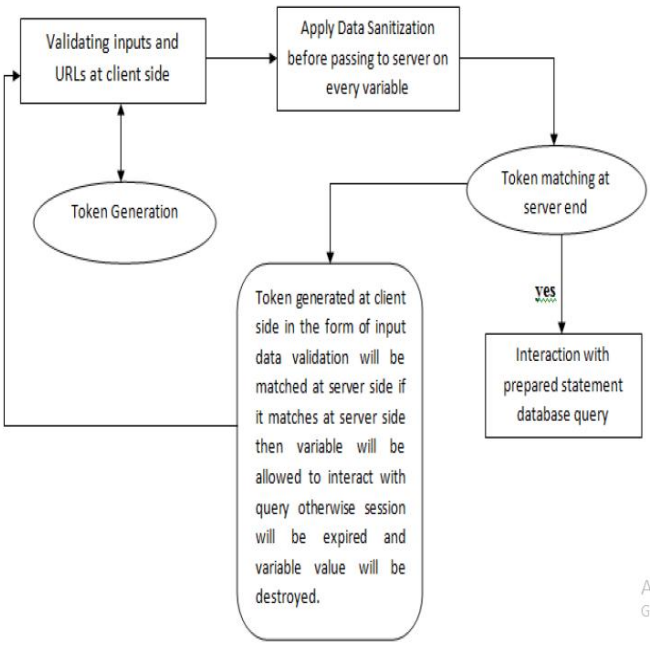
4. Query and session tokenization:- Query tokenization is a technique of converting the input query into various tokens. All string before a single quote, before double dashes and before a space constitutes a token. In this method, the original query and query with Injections are considered differently.

Working Scenario of proposed approach:

In this approach, following steps will be performed at the time of coding to secure web application from such type of attacks at some extent:

- Input and URL validation with the help of regular expression at client side.
- Token will be generated, that will be used at server side for database interaction of query.
- Data Sanitization will be applied on every variable before passing to server.
- Token matching at server side takes place.
- If token matches, database interaction will be allowed.
- Otherwise variable cannot interact with database query for further operation at server side and page will be redirected to default page.

Form input validations work as a token for interaction with database query. For example if any variable say '$a' which interacts with the database with the help of some query and values are passing in this variable by some user form then there will be some input validation for this particular variable at the client end. Now in this approach the client side validation for this particular variable '$a' will work as a token to interact with database query at server end and without having this token variable '$a' will not be able to interact with database. This approach reduces the chances of SQL injection attacks.



RESULTS AND DISCUSSION

In this section, experiments are performed on selected real time projects and after applying proposed technique on them, again same tests are performed on project in same environment to check the efficiency of proposed approach in terms of security.

To achieve the objectives, a detailed study of SQL injection attacks has been carried out. On the basis of study and analysis of different prevention techniques and proposed approach, a comparative study has been carried out between different type of prevention techniques and proposed prevention technique, the comparison is shown below:

Technique	Tautology	Illegal/Inc.	Piggy-back	Union	Stored Procedure	Inference	Alt. Encoding
JDBC-Checker	○	○	○	○	X	X	X
JAVA-Static Tainting	•	•	•	•	•	•	•
Safe-Query Ob	•	•	•	•	X	○	•
Security Gateway	○	○	○	○	○	○	○
SQL DOM	•	•	•	•	X	•	○
WAVES	○	○	○	○	○	○	○
WebSSARI	•	•	•	•	○	X	○
Proposed Approach	•	•	•	•	•	○	○

Comparison between prevention-focused techniques and proposed approach with respect to attack types

- • Denotes that technique can prevent attack successfully
- ○ Denotes that technique can address attack type but cannot guarantee of completeness
- X Denotes that technique is not able to address attack type

To achieve the other objectives, first of all a regressive security testing is performed on selected real time projects regarding SQLI attacks by using VEGA (a web application vulnerability detection tool) and SQLMAP (An exploitation tool).

Attacker Ip Address	172.158.110.79
Attacker's OS	KALI
Target Ip Address	172.158.30.156
Target OS	Windows 8.1
Target URL	172.158.30.156/convocation2017

Results after Scanning by VEGA to Target machine:

SQL injection Flaws

Classification	Input Validation Error
Resource	http://172.158.30.156/convocation2017/convo_submit.php
Parameter	Reg
Method	GET
Detection type	Blind Arithmetic Evaluation Differential
Risk	High

Request

GET/convocation2017/convo_submit.php?reg=1'"&dob=1

Discussion

Vega has detected a possible SQL injection vulnerability. These vulnerabilities are present when externally-supplied input is used to construct a SQL query. If precautions are not taken, the externally-supplied input (usually a GET or POST parameter) can modify the query string such that it performs unintended actions. These actions include gaining unauthorized read or write access to the data stored in the database, as well as modifying the logic of the application.

Impact

- Vega has detected a possible SQL injection vulnerability.
- These vulnerabilities can be exploited by remote attackers to gain unauthorized read or write access to the underlying database.
- Exploitation of SQL injection vulnerabilities can also allow for attacks against the logic of the application.
- Attackers may be able to obtain unauthorized access to the server hosting the database.

Exploitation of Found Vulnerability through SQLMAP:

Vulnerable Parameter:

'reg' and 'dob'

Vulnerable URL:

http://172.158.30.156/convocation2017/convo_submit.php?reg=1&dob=1

SQLMAP Command :

$sqlmap –url "http://172.158.30.156/convocation2017/convo_submit.php?reg=1&dob=1" –schema

Backend Database and server information disclosure:

Hence from above experiments and results, it can be concluded that the real time projects are vulnerable regarding sqli attacks and remote os-shell privilege escalation is also possible, which is totally undesirable.

Projects	SQLI Attack	Backend Info	Data Dumping	OS Shell Escalation
Convocation2017	Yes	Yes	Yes	Yes
Student info	Yes	Yes	Yes	Yes
Dvwa	Yes	Yes	Yes	Yes

After applying the proposed approach:

Following steps are performed to apply this approach in any source code:

- Validate each and every variable by applying javascript validation at client side with the help of regular expression. For example, there is a form in which one text field is password type; this variable is validated with the help of regular expression at client side. Also validate the url.

```
re = /[0-9]/;
 if(!re.test(form.pwd1.value)) {
 alert("Error: password must contain at least one number (0-9)!");
form.pwd1.focus();
 return false;
}
// regular expression to match required at least one lowercase letter in passord field
re = /[a-z]/;
   if(!re.test(form.pwd1.value)) {
   alert("Error: password must contain at least one lowercase letter (a-z)!");
   form.pwd1.focus();
   return false;    }
 // regular expression to match required at least one uppercase letter in passord field
   re = /[A-Z]/;
   if(!re.test(form.pwd1.value)) {
   alert("Error: password must contain at least one uppercase letter (A-Z)!");
   form.pwd1.focus();
   return false;}}
```

- After validating each variable at client side, the pattern validation for each variable works as a token at server side at the time of database query interaction. In this step, each variable will be sanitized before passing these to server side.

  filter_var(variable,FILTER_SANITIZE_STRING);

- Now when variables are reached at server side, these variables are matched at server side with same regular expression pattern in php script that are used at client side in java script validation.
- If pattern is matched, means token matches, then only variables are allowed to interact with database query otherwise variables will not be allowed foe query interaction.
- This approach reduces the chances of malicious code insertion in form or in url. So that chances of SQLI attack also reduces.

After applying the proposed approach in same project, source code is again tested in same environment and the results are satisfactory in terms of SQL injection security.

Results after Scanning by VEGA to Target machine after applying proposed approach:

SQL injection Flaws

Classification	No
Resource	Not found
Parameter	Not found
Method	POST
Detection type	Not found
Risk	No risk against this type attack

When url is scanned by SQLMAP to exploit the target. It shows that:

" 'reg' & 'dob' parameters might not be injectable…… "

Now the results of same real time projects are quite satisfactory in term of security regarding SQL injection attack, when we use same tools and same environment after applying proposed approach.

Projects	Backend Info	Data Dumping	OS Shell Escalation
Convocation2017	No	No	No
Student info	No	No	No
DVWA	No	No	No

In this approach, form input validations work as a token for interaction with database query. For example if any variable say '$a' which interacts with the database with the help of some query and values are passing in this variable by some user form then there will be some input validation for this particular variable at the client end. Now in this approach the client side validation for this particular variable '$a' will work as a token to interact with database query at server end and without having this token variable '$a' will not be able to interact with database. This approach reduces the chances of SQL injection attacks.

CONCLUSION AND FUTURE WORK

In this work, different prevention techniques of SQL injection attack in any web application have been studied and analyzed. And now it can be stated that these prevention techniques are efficient in few cases and at some extent. For example, JAVA-Static Tainting prevention technique can identify and prevent almost all type of attacks but it is only useful for JAVA based web applications and not for PHP or ASP based projects. The main goal of this research is to propose a secure coding approach for the prevention of SQLI attacks so that it will be very easy for developers and security professional to secure their applications at the time of development because at the time coding application's complexity level is not high if we compare it after the development of application. The result of proposed approach is quite satisfactory because it is very effective in preventing plenty of SQL injection attacks. This research has proposed a secure coding approach for the prevention of SQL injection attacks on PHP based web

applications at the time of coding and on the basis of analysis of results it can be stated that this approach reduces the efforts for security professionals to secure the web applications in the whole life of particular application at some extent. It is possible just because this approach is applied at the time of developing phase and to make any changes in project module at the time of development phase is much easier than to make changes in project after development.

In future, the proposed approach can be automated by developing some algorithm and real time application to facilitate web developers in more efficient manner. The whole research is done on PHP based projects, in future it can be extended to some other programming languages like Python, ASP etc.

From this paper, it can be concluded that SQL injection is one of most dangerous threat for web application security. Because it is the attack which is responsible for disclosure of sensitive data to unauthorized users that can cause severe harm to legitimate users depending upon the importance of information such as disclosure of debit/credit card details, bank related information etc. The proposed approach for prevention of SQL injection attack can be very effective and efficient to secure web applications against such type of attacks because user input validation acts as a token to interact with database query at server side.

REFERENCES

[1] M. Khosrow-Pour (ed.): *Encyclopedia of E-Commerce, EGovernment, and Mobile Commerce, Idea Group Reference,*2006.

[2] Preventing SQL Injections in Online Applications: Study, Recommendations and Java Solution Prototype Based on the SQL DOM Etienne Janot, Pavol Zavarsky.

[3] Janot, E.: SQLDOM4J: Preventing SQL Injections in Object-Oriented Applications. Master thesis, Concordia University College of Alberta (2008), http://waziboo.com/thesis

[4] OWASP Foundation: SQL Injection http://www.owasp.org/index.php/SQL_injection

[5] Power, R.: 2002 CSI/FBI Computer Crime and Security Survey. Computer Security Issues & Trends, 8, 1, 1--22 (2002).

[6] Y.W. Huang, F. Yu, C. Hang, C.H. Tsai, D. T. Lee, S.Y.Kuo: "Securing Web Application Code by Static Analysis and Runtime Protection", In *Proceedings of the 13th international conference on World Wide Web, New York,*USA, May 2004.

[7] OWASP Top Ten 2007, http://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf

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

[9] C. Andrews, D. Litchfield, B. Grindlay and NGS Software: *SQL Server Security*, McGraw-Hill/Osborne, 2003.

[10] P. Bisht, P. Madhusudan, and V.N. Venkatakrishnan, "CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks," *ACM Trans. Information and System Security*, www.cs.illinois.edu/~madhu/tissec09.pdf.

[11] W. Halfond, A. Orso, and P. Manolios, "WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation," *IEEE Trans. Software Eng.,* Jan. 2008,pp. 65-81.

[12] Web Application Disassembly with ODBC Error Messages, David Litchfield http://www.nextgenss.com/papers/webappdis.doc.

[13] Mittal Piyush, "A fast and secure way to prevent SQL injection attacks.[C]", *2013 IEEE Conference on Information and Communication Technologies*, pp. 730-734, 2013.