A Survey of SQL Injection Defense Mechanisms

Kasra Amirtahmasebi, Seyed Reza Jalalinia and Saghar Khadem
Chalmers University of Technology, Sweden
akasra, seyedj, saghar{@student.chalmers.se}

Abstract

SQL Injection Attack (SQLIA) is a prevalent method which makes it possible for the attackers to gain direct access to the database and culminates in extracting sensitive information from the firm's database. In this survey, we have presented and analyzed six different SQL Injection prevention techniques which can be used for securing the data storage over the Internet. The survey starts by presenting Variable Normalization and will continue with AMNESIA, Prepared statements, SQL DOM, SQLrand and SQLIA prevention in stored procedures respectively.

1. Introduction

Since the dawn of web programming, companies started putting their databases on the Internet for public access. These databases sometimes contained confidential and valuable information which were good targets of attack. SQL injection attacks (SQLIA) are among the most common database attacks which try to access the sensitive data directly. They work by injecting malicious SQL codes through the web application and cause unexpected behavior from the database. The 2002 Computer Security Institute and FBI revealed that on a yearly basis, over half of all database experience at least one security breach and an average episode results in close to $4 million in losses [4]. We have presented six SQL injection prevention techniques in this paper which will cover a wide range of SQL injection attacks. A combination of these prevention techniques may lead to a more secure and reliable database system.

2. Variable Normalization

Here we introduce a method that uses a virtual database connectivity drive along with a special method named "variable normalization" using these methods we can determine the basic structure of a SQL statement therefore we will be able to decide if a SQL statement is legal or not. This method does not require changing the source code of database applications and can also be used for auto-learning the allowable list of SQL statements. There is also a very minimal overhead in the system due to the fact that determining whether a SQL statement is allowable or not is done by checking the existence of normalized statement in the ready-sorted allowable list.

2.1. Background

Many web pages ask users to input some data and make a SQL queries to the database based on the information received from the user i.e. username and passwords. By sending crafted input a malicious user can change the SQL statement structure and execute arbitrary SQL commands on the vulnerable system. Consider the following username and password example, in order to login to the web site, the user inputs his username and password, by clicking on the submit button the following SQL query is generated:

SELECT * FROM user_table WHERE user_id = 'john' and password = '1234'

Now consider what will happen if the user input the following password:

' or 1=1 –

The SQL query will become:

SELECT * FROM user_table WHERE user_id = 'john' and password = '' or 1=1 --'

The "or 1=1" will result in returning all the records in the "user_table" and the "--" comments out the last ' appended by the system. Therefore, the query will return a non-empty result set without any error. SQL injection problem can be solved by checking all SQL statements before sending them to the database; however, with respect to dynamic generation of SQL queries by web applications, each statement might be different, so we are not able to predefine the allowable SQL statements [2].

2.2. Goal

The goal of variable normalization is try to take away the variables and get the basic structure of the SQL query, so that although the supplied variables differ every time, the basic structure remains the same. In the case of a SQL injection, the injected

code will change the structure of the SQL statement, and hence we will be able to detect it.
In order to normalize variables in a SQL statement, this method converts all single quoted strings to a single character such as "a" for instance, and all numbers to single digit "0". Then the normalized SQL statement is stored in a data structure, called a "rule", along with its corresponding normalized variable information, including their types, positions and the original values. The only parameters which will be modified by the normalization process are variables, and everything else including SQL comments, carriage returns, white spaces, or character cases will remain unmodified [2].

2.3. Defining the Allowable List

We will need to define an allowable list in order to verify if a normalized SQL statement is allowed. This list is a set of rules which were discussed before, but somehow different, because here the variable requirements are stored instead of variable values.
In this mode the variable value is stored as the expected value. This assumption is true until another SQL statement with the same normalized form but with different variable value is found. Therefore, theoretically we need to "learn" each SQL statement twice; otherwise, all variables will be assumed to be static.
For example, when the system first learns the following SQL statement

SELECT * FROM jobs WHERE completed = 0 and start_day > '1/1/2009'

The result would be like table 1.

SELECT * FROM jobs WHERE completed = 0 and start_day > 'a'			
Variable 1	Position 39	Integer type	Expected value: 0
Variable 2	Position 58	String type	Expected value: "1/1/2009"

Table 1. Normalized SQL statements

And after that the system comes to this SQL statement:

SELECT * FROM jobs WHERE completed = 0 and start_day > '1/1/2008'

The result will be stored as in table 2.

SELECT * FROM jobs WHERE completed = 0 and start_day > 'a'			
Variable 1	Position 39	Integer type	Expected value: 0
Variable 2	Position 58	String type	Expected value: "1/1/2008"

Table 2. Normalized SQL statements

Since both normalized SQL statement expect value 0 for variable 1, this expectation is kept. But for variable 2 we have two different expected values ("1/1/2009" and "1/1/2008"), now we can perform more analysis on these two expected values in order to form a new requirement with say, character set constrain, or simply do not have any requirement.

2.4. SQLBlock Implementation

In order to check the authorization of the executing SQL statement, we need a way to get it. This can be done by implementing a proxy driver which is simply another database connectivity driver. This proxy driver calls another database driver as a client instead of passing the database query to the database server directly. This proxy driver also hides the error messages so the attacker will not be able to benefit from the information about the database schema contained in these error messages. After normalizing
the SQL statement, the proxy driver will search the ready list to see if that particular normalized SQL statement exists or not. In case of existence, if the variables are within expected bounds this SQL execution is allowed and otherwise not [2].

3. AMNESIA

In this part we will introduce a tool, AMNESIA, which implements a new technique for detecting and preventing SQL injection attacks. This technique detects illegal queries before being executed on the database using a model-based approach. This technique consists of a static part in which it automatically builds a model of legal queries using program analysis, and also a dynamic part in which it inspect and checks the dynamically-generated queries against the statically built model using runtime monitoring. Queries violating the model are potential SQLIAs and are prevented from execution. The technique is consisted of four steps which we will go through respectively.

3.1. Step 1: Identifying the hotspots

Hotspots are some particular places in the application program which generate SQL queries to the database. In this step a simple scanning is performed to identify these points in the application code [3].

3.2. Step 2: Build SQL Query Model

In this step a SQL-query model is built for each identified hotspot in order to compute the possible values of the query string passed to the database. Using the Java String Analysis (JSA) [1] library a

flow graph is constructed which abstracts away the control floe of the program represents the operations performed on string variables. This technique analyzes the flow graph for each desired string and simulates the string manipulation operations that are performed on the string resulting in a Non-Deterministic Finite Automata (NDFA) which represents all the possible values the considered string can take. A depth first traversal on the NDFA for the hotspot is performed in order to build the SQL-query model for that particular hotspot. Variable strings are presented using symbol β e.g. the value of password [3].

3.3. Step 3: Instrument Application

In this step, we instrument the application in such a way that, there will be a call to the monitor before the call to the database, so the queries will be checked at runtime. By the time of calling the monitor, a string that contains the query and also a unique identifier for the hotspot are passed to it as its parameters. This unique identifier is used by the runtime monitor in order to relate the hotspot with the specific SQL-query model generated for that point.

3.4. Step 4: Runtime Monitoring

While executing normally when an application reaches a hotspot, the query will be sent to the monitor where it is parsed to a sequence of tokens according to SQL syntax. Empty string constants are identified by their syntactic position and are presented using ε. The important issue about this technique is that, it interprets the query the same way as the database does according to the SQL grammar, therefore it doesn't result in false positives and problems with user input caused by a simple keyword matching. After parsing the query, the runtime monitor checks it against the SQL query model associated with the hotspot from which the monitor has been called. To check whether a query is compliant with the model, the runtime monitor can check whether the NDFA accepts the query or not. In case of acceptance the monitor lets the query to be executed, otherwise considers it as an SQLIA.

4. Prepared Statements

SQL (Structured Query Language) is a standard and common language which is designed for modifying and querying data and managing database.SQL statements are queries which are created with a standardized language for defining and manipulating data in a database that returns a single result set from a relational database. If the attacker can change the logic of the SQL statement by inserting SQL characters and keywords into SQL statement's input it is shown that SQL injection vulnerability exists. This vulnerability permits the attacker to modify SQL statement's structure and performs the SQL injection attack. A brief explanation about SQL structure will be presented which will be followed by its related vulnerabilities:

4.1. Structure

An SQL statement's structure is the SQL code including the logical meaning of the statements. For instance the SQL statement [5]:
SELECT password FROM user WHERE userName = 'user1'
Has the following structure:
SELECT password FROM user WHERE userName =
This structure consists of the SELECT, FROM and WHERE clause that they specify the purpose of the SQL statement. The SQL statement's structure can include these elements:

- Clause as SELECT, FROM and WHERE
- Identifiers as table, attribute names
- Comparators as equal(=), AND, OR and LIKE

An SQL statement's input is the part of the statements which is combined with the identifier to change the result of the conditional clause and it is changed by each user. For instance 'user1' is the SQL input in the previous example.

4.2. SQL Injection Attack

In order to explain the SQL injection attack we start with an example [5]:
Take the following SQL statement into account:

SELECT password FROM users WHERE userName = ''+inputUserName +''

If a user enters 111' OR true# as an input, the SQL statement executed is:

SELECT password FROM users WHERE userName = '111' OR true#'

When the database performs this SQL statement, the WHERE clause will always be true and the database will interpret the structure of this SQL statement as follows:

SELECT password FROM users

Because the single quote character ‘ comes out of the variable inputUserName_, the keyword OR makes the WHERE_clause conditional and the word true_which is a keyword, makes the conditional true and the character # is used for commenting the SQL statements, In this example the SQL injection attack is happened and the attacker can remove the WHERE clause from the SQL statement and change the SQL statement. Therefore the result of this statement is all password columns from the user table regardless of inputUserName [5].
As already mentioned, the SQL injection attack happens where an input isn’t apart from the SQL statement structure and the attacker can change the logic of the statement by inserting an insufficiently-confirmed input at runtime. By doing so, the application merges the input and structure of the statement together as one request and sends it to the database. Therefore in the previous example the vulnerable statements merge with the input inputUserName as a statement structure before sending to the database which permits that inputUserName_to alter the_WHERE_clause. To overcome this vulnerability, developers should change the logic of the SQL statement and they should restrict the range of the acceptable SQL structure.

4.3 SQL Injection Prevention

There are several solutions to solve this problem. One common solution is to discrete the SQL structure from the SQL input by applying prepared statements. Prepared statements specify the SQL structure explicitly. Since replacing vulnerable SQL statements with prepared statements manually is time consuming, boring and error-prone, by using a prepared statement replacement algorithm (PSR-Algorithm) the task will be performed automatically. This algorithm collects information from source code which is containing SQL injection vulnerability and creates secure prepared statement code with respect to functional integrity. It means that it analyzes source code including SQL Injection Vulnerabilities and creates a specific recommended code structure which includes prepared statements and makes apart the SQL structure from SQL statement’s input. For each string object applied to generate the SQL statement the PSR Algorithm generates an additional string object and an assistant vector which are used for finding any SQL input in the original string objects. Therefore the SQL injection vulnerability can be eliminated from vulnerable SQL statements by converting them with secure prepared statements. The logic of this algorithm is not restricted to any specific language so it can be extended to fit the syntax of any language [5].
The SQL statements that segregate statement structure from statement input are called prepared statements. By specifying the structure of the statement and defining the limitation for inputs a prepared statement is prepared. Then the SQL statement structure with the limited or bind variable is sent to the database and the database compiles and saves the structure of the statement for implementing with input variables in the future. Therefore the statement structure cannot be altered by input parameters when prepared statements are generated because the statement structure set clearly before the runtime. Consequently the risk of the SQL injection vulnerability will be decreased. Consider the following prepared statement:

SELECT password FROM users WHERE userName = ?

In this prepared statement the question mark (?) is the limited variable. So through this method which is a setter method a bind variable is set and also strong type checking is done and the result of the invalid characters such as single quotes in the middle of the string becomes ineffective. For example the bind variable can be set through the setter method setString (index, input) in the SQL structure which is specified by the index to input. Another setter method is setObject (index, input) that calls the suitable setter method depending on the object type of the input. After a preparation of the SQL statement one setter method is used for each bind variable to fill the bind variable with input.
In Java prepared statements are applied not only for securing against injection attacks, but also for efficiency. In java implementation, prepared statements are compiled one time but executed multiple times.

5. SQL DOM

One of the most common interaction mechanisms between applications and databases is a call level interface (CLI) such as ODBC and JDBC. CLIs provide a low level interface to the database engine. By using a CLI for interfacing with a database engine, SQL statements will be constructed by concatenating and substituting strings that let the developer to create very powerful and flexible queries. The result of the queries won’t be checked for correctness until the runtime when they are sent to the database engine [6].
Therefore by constructing SQL statements in this way many errors and problems such as bad syntax, misspelled column and table names and data mismatches can arise. Also by changing the database schema, SQL strings that contain SQL statements will be problematic. The number of SQL strings can be large in medium or large sized projects so by

changing an application and the database schema, maintaining the SQL strings becomes difficult therefore SQL strings make an application vulnerable to SQL injection attacks by inserting a malicious code into a web form. This problem can be solved by applying a set of classes that are strongly-typed to a database which is called SQL DOM [6]. These classes are used to generate SQL statements instead of manipulating strings. This solution involves an executable sqldomgen which is performed against a database. Sqldomgen generates a Dynamic Link Library (DLL) that includes SQL DOM. By applying this method, the compiler is secured to remove the possibility of SQL syntax, data type mismatch and misspelling problems. By using class names or enumeration members, names of tables and column are associated into the SQLDOM and also types of constructor and method parameters are extracted from data type of columns.
For enhancing the maintainability of the application, the compiler is needed to be support because during the life of an application the database schema alters. Sqldomgen is re executed if the database schema alters and then a modified SQL DOM is generated. When the application is rebuilt with the modified SQL DOM, some compiler errors will be arisen such as [6]:

Data type conversion error: If the data type of a column was changed this error would occurred.

No such class exists: If a table or column was renamed or eliminated this error would occurred.

Missing constructor parameter: If a new column was added to a table this error would occurred.

By applying SQL DOM, the compiler would be able to help in the sustainment process. Hence the reliability of the application will be increased and also developers can change the database schema easily according to the requirements of their customers. The application will be secured because SQL DOM constructs all the SQL statements and remove all syntax and data type mismatch bugs, and as a consequence, all known SQL injection attacks will be omitted. While in an application that uses SQL strings, it is difficult to defend against SQL injection attacks.
Sqldomgen perform three main steps. The first step is to get the schema of the database. This step is fulfilled by applying some methods provided by an OLEDB provider. The second step is to repeat through the tables and columns included in the schema and obtain a number of files encompassing a strongly-typed instance of the abstract object model. The third step is to compile these files into a dynamic link library (DLL). To guarantee that changes to the database schema happen in compile time errors, we can imagine that sqldomgen being implemented daily as a part of the build of an application.

6. SQLrand

Many applications take the user input and put this input into a pre-defined SQL query and send the resultant query to the database. As another prevention technique, SQLrand can be discussed. SQLrand assumes that SQL commands are provided by the web application and users should not enter SQL commands as their input data. The basic idea behind SQLrand is randomizing SQL commands in which the template query inside the application will be randomized. In this way SQL commands which are injected by a malicious user will not be encoded, therefore the proxy will not recognize the injected commands and the attack will not be successful.

6.1. SQLrand Architecture

SQLrand uses the logic of Instruction-Set Randomization [7] [9]; the SQL keywords are manipulated by appending a random integer to them which the attacker cannot easily guess [9]. By doing so, the SQL keywords which are pre-defined in the web application are randomized. Before entering the database, the manipulated SQL keywords are decoded into standard SQL commands by a proxy. The random integers will not be appended to any other SQL commands (which are injected by malicious users); therefore the proxy will not recognize these injected commands and will lead to invalid expressions.
The proxy in SQLrand is an independent module which can be placed anywhere. Having a separate proxy which is not an internal part of the database and is in charge of decoding the randomized SQL commands will culminate in flexibility, simplicity and security [9]. A few benefits of having a separate proxy can be mentioned as follows:
There can be multiple proxies to decode the SQL commands with different random keys and they can all send their decoded keywords to the same database.
Since the randomized integers are removed in the proxy, the database will never be aware of the randomized integers and won't generate any error messages to reveal the information about the random integers.
The proxy can be used in order to filter the error messages that are generated by the database. Error messages can reveal invaluable information about the database and its tables and column names. By having a proxy, database generated error messages will never return to the web application.

The proxy has a simple structure which makes it possible and easy to protect it against the attacks.
The following simplified example shows how the SQLrand works [9]. If the random integer would be 123 then the randomized query which will be sent to the proxy will be

SELECT123 gender, avg123 (age) FROM123 cs101.students WHERE123 dept = %d GROUP123 BY123 gender

Since the proxy is in charge of receiving the randomized keywords from the application, the application must connect to the proxy instead of the database. Upon receiving a connection, the proxy will connect to the database and will transmit the commands sent by the client after having them decoded. If the proxy could not decode the received query, it will send an error message back to the application and will disconnect from the database. If the query was legitimate, the proxy will decode the randomized keywords and will transmit the standard query to the database.
SQLrand makes it possible for the developers to generate randomized SQL commands instead of normal keywords [8].

7. Stored Procedures

Stored procedures are subroutines in the database which the applications can make call to. They are becoming more widespread due to the fact that they add an additional abstraction level to the database which means that the underlying database structure can easily change if the interface on the stored procedure stays the same. SQL queries can be built at runtime based on user inputs which leads to flexibility. However, this feature will make the SQL injection attacks possible in stored procedures.

7.1. SQLIA Prevention

The SQL injection attack prevention in stored procedure works by combining static analysis with runtime validation [10]. The static analysis uses a stored procedure parser which identifies the commands that build the SQL statement. In the runtime analysis, a new function (SQLIAChecker) will identify the user input. If the input does not conform to the standard SQL commands, then the query will be marked as SQLIA.

7.1.1. Static Analysis

In this technique, there exists a stored procedure parser which extracts the control flow graph from the stored procedure [10]. Static analysis is an offline procedure and is done by analyzing the stored procedure's source code. In this phase, an SQL graph will be generated and all the user input will be marked. These user inputs will be checked further at run time to compare the final generated statement with the original SQL statement. It should also be mentioned that SQL statements which do not require user input, are not included in the SQL graph because they are not vulnerable to SQL injection attacks. The static analysis phase helps reducing the overhead in the runtime phase.

7.1.2. Runtime Analysis

In this phase, a function named SQLIACHECKER() will identify the user input and a finite state automaton will be built. After that, the user input will be checked with the finite state automaton. If the user input does not match with the original SQL statements which were identified by the finite automata, they will be flagged as a potential SQL injection attack; otherwise, the query will be valid and will be passed through.
An approach to reduce the computational overhead would be to track user inputs. If a user input passes through the SQLIACHECKER() function, it can be concluded that the input is not malicious and can be used over and over again on the condition of not being exploited. By doing so, the overhead will be reduced dramatically while SQL injection attacks are still detected and managed.

8. Discussion

So far, six different SQL Injection defense mechanisms has been presented and discussed somehow in detail. We believe the mentioned mechanisms can provide security against SQL Injection Attacks. To sum up, a summary of all the discussed SQL Injection defense mechanisms will be presented with a comparison of the techniques as well as their strengths and weaknesses.
The paper starts with discussing Variable Normalization. It works by extracting the basic structure of the SQL statement, so an SQL injection attack which alters the original structure of the SQL statement, can be detected. The introduced SQLBlock has a minimal overhead and works with all database servers without requiring any change in the client source code. The auto-learning option for the allowable list makes the system very convenient to setup for clients that issue many different SQL commands. There exist some cases that cannot be handled by Variable Normalization

Technique	Tautology	Illegal	Piggy-Back	Union	Stored Procedure	Inference	Alternate Encoding
AMNESIA	✓	✓	✓	✓	X	✓	✓
SQL DOM	✓	✓	✓	✓	X	✓	✓
SQL Rand	✓	X	✓	✓	X	✓	X

Table 3. Comparison of techniques with respect to attack types

effectively. Imagine a website which allows the user to select different types of a particular product form a multi-line selection box. The normalized SQL command may be something like this:

SELECT * FROM Product_Tbl WHERE Product_name LIKE 'a' AND Product_type in ('a' , 'a' , 'a')

The SQLBlock will be able to handle these kinds of queries only by learning all different variants of the SQL statement. This is possible only in case that there are limited and also expected numbers of items in the selection box. [2]

AMNESIA is a fully automated technique for detecting and preventing SQL injection attacks. A web application code contains a policy which helps to differentiate between legitimate and illegal queries. This technique uses static and dynamic analysis to obtain the mentioned policy and use it to distinguish illegal SQL queries. It is important to mention that this technique targets SQLIAs, which happen when an attacker attempts to inject SQL statements into a query which is sent to the database. SQLIAs do not include other types of web-application related attacks. [3]

Most of the existing techniques to deal with SQLIAs are focusing on queries generated within the application. Enabling the techniques to embody the queries that are executed on the database is a complex task. For this reason attacks based on stored procedures and also alternate encoding are problematic and difficult to handle. Only three techniques, AMNESIA, SQLCheck, and SQLGuard explicitly address these types of attacks and that is because they use the database parser to interpret a query in the same way that the database would [8].

The degree of success in finding the real SQLIAs by this technique is dependent on the accuracy of the generated query models which takes place in the static analysis phase. Depending on the precision of this step, the technique might result in both false positives and false negatives. [3]

SQL DOM is an approach in which changes the process of query-building to a systematic process of API type-checking. SQL DOM creates new SQL commands instead of generating a string of SQL keywords that will later be sent to the database. By doing so, there are no more SQL string generations in the applications which are the main target of SQL injection attacks.

SQL DOM offers protection against tautologies, illegal/logically incorrect statements, piggyback and union queries as well as protection against inference attacks and alternate encodings [8]. On the other hand, it does not have any protection against attacks in stored procedures. While SQL DOM seems like an effective prevention technique against SQL injection attacks, it has the drawback which requires new programming paradigms that may not be desirable for developers.

Prepare statements allow the clients to pre-issue a template SQL query at the beginning of a session. It has the advantage of performance while, at the same time, it has two disadvantages. First, it is vulnerable to SQL injection attacks if the same query is used many times and secondly, this technique does not work for dynamically constructed queries [9].

SQLrand is a useful prevention method which is based on instruction-set randomization. It makes it possible for the developers to generate randomized SQL commands instead of normal keywords [8]. A proxy is in charge of decoding the randomized keywords to standard keywords which is then transferred to the database for execution. On the condition of an attack, the proxy will not recognize the injected SQL commands which will result in rejecting the query. The proxy will disconnect from the database and an error message will be sent back to the application. Although SQLrand needs key management, developer training and proxy creation, it will have little overhead on the applications and systems. Apart from its advantages, SQLrand requires a complex setup which will make it harder for developers to use this prevention technique. From security point of view, SQLrand uses a secret key in order to change the instructions. On the condition that the secret key is revealed, the attacker will have access to randomized queries. SQLrand provides protection against tautology and piggyback queries as well as union query and inference attacks. However, this technique does not have any protection against illegal or logically incorrect queries, stored procedure attacks and alternate encodings [8].

Although some may think that stored procedures are secure and reliable, they are open to SQL injection attacks. A malicious user can perform denial of

service attacks, execute remote commands and perform privilege escalation attacks. A prevention technique has been presented here which builds an SQL graph. The graph will be validated later at the runtime analysis against different user inputs in order to detect malicious activities [10]. In conclusion, this technique has proved to be useful with little computational overhead and as the name implies, it provides protection in stored procedures.

9. Conclusion

This paper is a survey of different SQL Injection prevention techniques. Generally, the presented techniques focus on standardization of SQL statements in a way that prevent attackers from gaining unauthorized access to an organization's database. First we started by studying different attacks which were more common and advanced. The research continued with organization and classification of different prevention techniques which we found and finally a discussion with a summary of the best SQL injection prevention techniques were presented. The paper discusses the appropriate SQL prevention techniques for a given attack. We believe that each prevention technique cannot provide complete protection against SQL Injection Attacks but a combination of the presented mechanisms will cover a wide range of SQL injection attacks which will culminate in a more secure and reliable database which is protected against SQL Injection Attacks.

10. References

[1] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In *Proceedings of the 10th International Static Analysis Symposium, SAS 03*, volume 2694 of*LNCS*, pp 1–18. Springer-Verlag, June 2003.

[2] Sam M.S. N.G, "SQL Injection Protection by Variable Normalization of SQL Statement", www.securitydocs.com/library/3388, 06/17/2005.

[3] William G.J. Halfond, Allesandro Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL Injection Attacks", ACM, USA, 2005, pp 174-183.

[4] C.S. Institute. Computer crime and security survey. http://www.gocsi.com/press/20020407.jhtml, 2002

[5] Stephen Thomas, Laurie Williams, Tao Xie, "An Automated Prepared Statement Generation To Remove SQL Injection Vulnerabilities", Elsevier, USA, 2008.

[6] Russell A. McClure, Ingolf H. Krüger, "SQL DOM: Compile Time Checking of Dynamic SQL Statements", IEEE Explore, USA, 2005.

[7] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks with Instruction-Set Randomization", In *Proceedings of the ACM Computer and Communications Security (CSS) Conference,* October 2003, pp. 272-280

[8] WG Halfond, J Viegas, A Orso , "A Classification of SQL Injection Attacks and Countermeasures" *In Proceedings of the International Symposium on Secure Software Engineering*, 2006

[9] S. W. Boyd, and A. D. Keromytis, "SQLrand: Preventing SQL Injection Attacks", Springer Berlin / Heidelberg, Department of Computer Science, Columbia University, 2004, pp. 292-302

[10] Ke Wei, M. Muthuprasanna, Suraj Kothari, "Preventing SQL Injection Attacks in Stored Procedures", Proceedings of the 2006 Australian Software Engineering Conference (ASWEC'06), IEEE, Australia, 2006, pp 1-7.