International Journal of Scientific & Engineering Research, Volume 4, Issue 8, August 2013
ISSN 2229-5518

SQL Injection Attack Solutions: A Review

Sruthy Manmadhan, Manesh T, Varghese Paul

Abstract— Web applications are becoming an important part of our daily life. So attacks against them also increases rapidly. Of these attacks, a major role is held by SQL Injection Attacks (SQLIA). This attack is launched through specially crafted user inputs and target web applications that used backend databases. Characteristics feature of this attack is that, it will change the intended query structure. To avoid this type of attack, the best solution is to do not allow user to enter any part of the SQL query directly. In this paper, we describe SQL Injection attack, various types and a detailed review of its solution techniques.

Index Terms— Attack , Injection, SQL, Vulnerability, Web.

1 Introduction

Nowadays, for most of the activities in our life, we depend on internet or web applications. There exists a natural trend that as the usage of a particular service increases; the attacker's interest on it also increases. The same thing happened in case of web applications. Of many kinds of attacks against web applications, SQL Injection Attack (SQLIA) is one of the top most threats against them[1]. So it is highly requires in the current scenario to have a good solution to prevent such attack to secure the information. This is the motivation behind this work.

SQL Injection targets the web applications that use a back end database. Working of a typical web application is as follows: User is giving request through web browsers, which may be some parameters like username, password, account number etc. These are then passed to the web application program where some dynamic SQL queries are generated to retrieve required data from the back end database.

SQL Injection attack is launched through specially crafted user inputs. That is attackers are allowed to give requests as normal users. Then they intentionally create some bad input patterns which are passed to the web application code. If the application is vulnerable to SQLIA, then this specially created input will change the intended structure of the SQL query that is being executed on the back end database and will affect the security of information stored in the database. The tendency to change the query structure is the most characteristics feature of SQLIA which is being used for its prevention also.

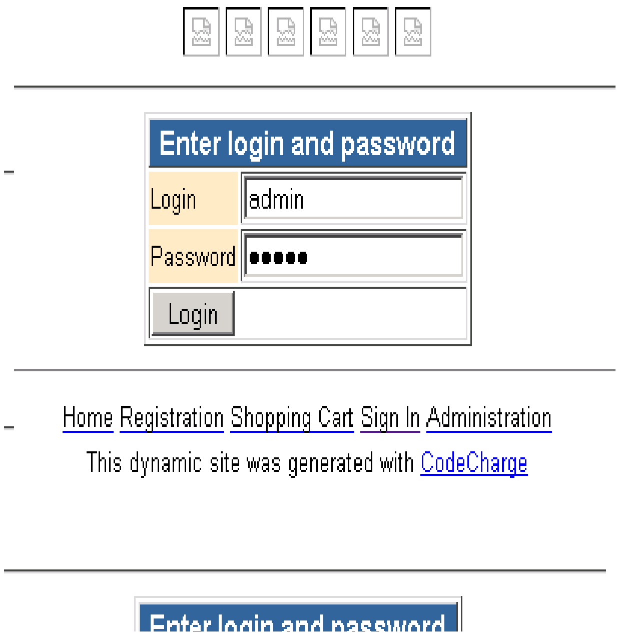
For better understanding let us have look at the following example. We all know that most of the applications that we are accessing through internet will have a login page to authenticate the user who is using the application. Figure 1 show such a login page. Here when a user is submitting his username and password, an SQL query is generated in the back end to check whether the given credentials are valid or not. Suppose the given username is 1 and password is 111, the query will be:

*Select * from login where user='admin' and pass='admin'*

This is the normal case and if any rows are selected by the query, the user is allowed to log in. Now, figure 2 shows an attack scenario. That is an attacker wants to log in without correct username and password. Instead of entering valid username if he uses injection string like "hacker' OR '1'='1' –" as username and "something" as password, the query formed will be like this:

*Select * from login where user='hacker' or '1'='1' –' and pass=''*

When this query is executed in the database, it will always return a true and the authentication will succeed.


Enter login and password
Login: admin
Password: ●●●●●
Login
Home Registration Shopping Cart Sign In Administration
This dynamic site was generated with CodeCharge

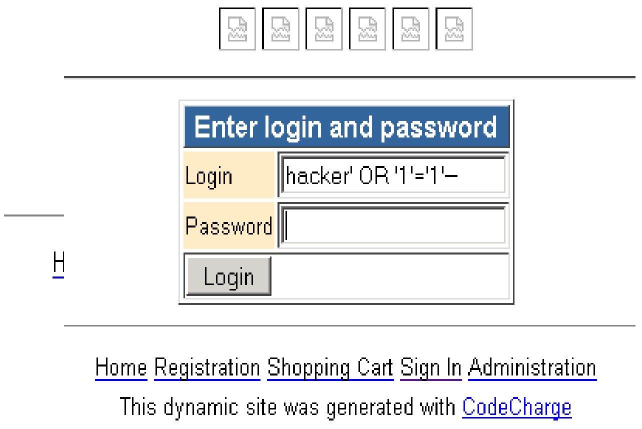

Enter login and password
Login: hacker' OR '1'='1'–
Password:
Login
Home Registration Shopping Cart Sign In Administration
This dynamic site was generated with CodeCharge

Fig 2: Example Login- Attack

Here the pattern “1=1” will always be true and is called *tautology*. Since, “OR” operator is inserted by the attacker, the query will return true even though the username and password are wrong. Also “ —“ will have special purpose. It will comment the remaining part of the query so that password will not be checked.

The rest of this paper is organized as follows: section 2 describes different types of SQL Injection attacks. Section 3 describes different solution to this attack which is categorized into three, defensive coding, static analysis, and defense mechanisms. Section 4 concludes the review.

2 Types Of SQLIA

The SQLIA can be broadly classified into two: *first order* and *second order* attacks. First of these will have direct effect on the system whereas other doesn’t have any direct harm. Different types of first order attacks are listed below[2]:

Tautologies: The main intention of this attack is to bypass authentication. For this they attack the field that is used in a query’s WHERE conditional. Transforming the conditional into a tautology causes all of the rows in the database table to be returned so that he can login successfully without having a valid username and password. The attack shown in figure 2 is an example of tautology attack.

Illegal/Incorrect Queries: This is the first step of SQL injection attack. Here the intention of the attacker is to gather information about the type and structure of the back end database that is being used in the web application. This attack exploits very descriptive default error pages returned by the application servers.

Union Queries: This type of attack is mainly used to bypass authentication and to extract data by changing the data set returned for a given query. Format is ‘UNION SELECT <part of injected query>’, where the query after the UNION keyword is fully under control of the attacker so that he/she can retrieve data from any table which is not intended by the actual query.

Piggybacked Queries: This attack mainly aims at extracting data. Like the concept of piggybacked acknowledgement in computer networks where, acknowledgement of a packet is sent along with the next packet, here, the attacker tries to inject additional queries with original one.

Stored procedure Attack: This type of attack tries to execute stored procedures present in the database with malicious inputs.

Inference: Main aim of this kind of attack is to identify injectable parameters. The information can 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.

There are lots of prevention methods proposed against these types of attacks. Table 1 includes an overview of these techniques.

TABLE 1
Overview Of Solution Methods

Solution	Overview
SQL DOM	A set of classes that are strongly-typed to a database schema are used to generate SQL statements instead of string manipulation.
MUSIC	A system based on mutation based testing.
SQLrand	A strong random integer is inserted in the SQL keywords.
AMNESIA	This scheme identifies illegal queries before their execution. Dynamically-generated queries are compared with the statically- built model using a runtime monitoring.
CANDID	Programmer-intended query structures are guessed based upon evaluation runs over non-attacking candidate inputs.
Tainting	This method will check if any keywords in a query are tainted before executing the query.
Parse Tree Validation	Comparing, at run time, the parse tree of the SQL statement with and without user inputs.
IDPS	Combines signature based and anomaly based detection.
Obfuscation	A method which includes obfuscation and reconstruction of queries
AIIDA	An agent based system which integrated the use of CBR, ANN and SVM.
Using Biological algorithms	Uses an algorithm for pair wise sequence alignment of amino acid code from web applications.

3 Literature Review

3.1 Defensive Coding Practices

It includes input validation and use of prepared statements. Input validation is a burden for programmers because, they have to manually decide valid inputs for each point of input and do an extensive search for special characters, alternate encodings and presence of back end commands.

PREPARED statements semantically separate the role of keywords and data literals. Its use is very effective for new web applications to be developed. Retrofitting already launched applications with PREPARED statements is a huge task and is not practical.

In general defensive coding practices can be applied only at the time of programming. It doesn’t consider millions of applications that are already in use.

• *Sruthy Manmadhan is currentlyworking in Computer Science Department of Adi Shankara Institute of Engineering and technology ,Kalady,India , E-mail: sruthym.88@gmail.com*

3.1.1 SQL DOM
SQL DOM[3] is proposed for Object Oriented Programming (OOP) environment. It doesn't consider stored procedure SQL Injection attack. But, it is a slight improvement of defensive coding practices because instead of relying completely on programmers to do all input validation, use a safe API which will take care of security. For the generation of API, they proposes an API generation tool, sqldomgen, which analyses database schema at compile time and writes code for a custom set of SQL query construction classes, which are directly called by developers to build queries.

3.2 Static Analysis Techniques
Limited to identifying points of inputs and query issuing locations, and checking whether every data flow from point of input to query location is subject to proper input validation. In this method also, programmer must manually evaluate and declare the sanitizing blocks of code for each web application and so this approach is not fully automatable.

3.2.1. MUSIC-Mutation Based Testing
Mutation is the act of deliberately altering a program's code, then re-running a suite of valid unit tests against the mutated program[4]. Mutation testing is a method of software testing, which involves modifying programs' source code or byte code in small ways. Mutation testing is done by selecting a set of mutation operators and then applying them to the source program one at a time for each applicable piece of the source code. The result of applying one mutation operator to the program is called a *mutant*. Mutants for SQL injection are Remove WHERE keywords and conditions , Negate each of the unit expression inside where conditions , Add parentheses in where conditions and prepend "*FALSE AND*" after the WHERE keyword ,Unbalance parentheses of where condition expressions , Set multiple query execution flags to true , Override commit and rollback options , Set the maximum number of record returned by a result set to infinite , Set query execution delay to infinite and Override the escape character processing flags. In mutation base testing author has suggested to do the checking for SQL injection before uploading the web service on the server. Advantage of this technique is it identifies the vulnerabilities in advance that is it's like a precautionary measure for SQL injection.

3.3 Defense Mechanisms

3.3.1. Randomization Based Method
One can randomize SQL keywords in parts of the query generated by an application and look for correctly randomized keywords in SQL statements issued to the database to detect attacks. This is the approach taken by *SQLrand* [5]. *SQLRand* applies the concept of instruction-set randomization to SQL, creating instances of the language that are unpredictable to the attacker i.e create randomized instances of the SQL query language, by randomizing the template query inside the CGI script and the database parser. To allow for easy retrofitting of our solution to existing systems, we introduce a de-randomizing proxy, which converts randomized queries to proper SQL queries for the database. Code injected by the rogue client evaluates to undefined keywords and expressions. The SQL standard keywords are manipulated by appending a random integer to them, one that an attacker cannot easily guess. Therefore, any malicious user attempting an SQL injection attack would be thwarted, for the user input inserted into the "randomized" query would always be classified as a set of non-keywords, resulting in an invalid expression. Our design consists of a proxy that sits between the client and database server . Note that the proxy may be on a separate machine. By moving the de-randomization process outside the DataBase Management System (DBMS) to the proxy, we gain in flexibility, simplicity, and security. Fig 3 shows the proposed architecture.

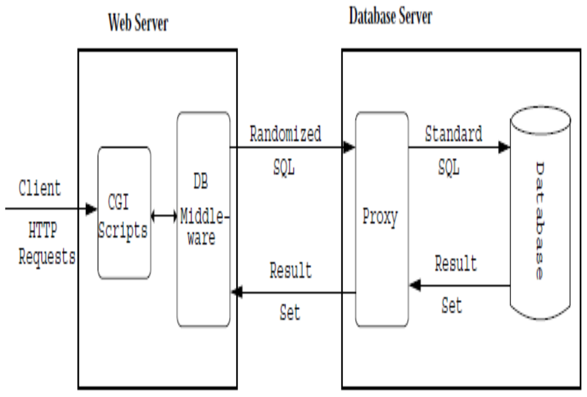


Fig 3: SQLrand System Architecture[5]

For example, in the C language, an SQL query, which takes user input, may look like the following:
select gender, avg(age)
from cs101.students
where dept = %d
group by gender
The utility will identify the six keywords in the example query and append the key to each one (e.g., when the key is "123"):
select123 gender, avg123 (age)
from123 cs101.students
where123 dept = %d
group123 by123 gender
This SQL template query can be inserted into the developer's web application. The proxy, upon receiving the randomized SQL, translates and validates it before forwarding it to the database. Note that the proxy performs simple syntactic validation — it is otherwise unaware of the semantics of the query itself. Problems of this method are:

- Limits scalability – due to manual retrofitting.
- It could result in a change of semantics even on benign inputs – due to randomization.

3.3.2. Learning Intentions Statically
One approach in the literature has been to learn the set of all intended query structures a program can generate and check at runtime whether the queries belong to this set. This is used in *AMNeSIA* [6], which is a tool which uses a model based

approach to detect and prevent SQL injection attacks in java based web applications. In its static part, the technique uses program analysis to automatically build a model of the legitimate queries that could be generated by the application. In its dynamic part, the technique uses runtime monitoring to inspect the dynamically-generated queries and check them against the statically-built model. The technique consists of four main steps.

- **Identify hotspots:** Scan the application code to identify *hotspots*— points in the application code that issue SQL queries to the underlying database.
- **Build SQL-query models:** For each hotspot, build a model that represents all the possible SQL queries that may be generated at that hotspot. A *SQL-query model* is a non-deterministic finite-state automaton in which the transition labels consist of SQL tokens (SQL keywords and operators), delimiters, and place holders for string values.
- **Instrument Application:** At each hotspot in the application, add calls to the runtime monitor.
- **Runtime monitoring:** At runtime, check the dynamically-generated queries against the SQL-query model and reject and report queries that violate the model.

Drawback of this method is that, the core component of AMNeSIA is the JSA (Java String Analyzer) tool which extracts the SQL query model from java source code. An issue with the implementation of AMNeSIA method with ASP.NET with C# is that there is no JSA kind of tool available for this technology. Hence the AMNeSIA method cannot be directly used to prevent SQL injection in ASP.NET based applications.

3.3.3. Learning Intentions Dynamically

To deduce (at runtime) the query structure intended by a programmer, the high-level idea is to dynamically construct the structure of the programmer intended query whenever the execution reaches a program location that issues a SQL-query. Here the approach is to compute the intended query by running the application on *candidate* inputs that are self-evidently non-attacking. An approach in literature using this idea is *CANDID* [7]. To deduce (at runtime) the query structure intended by a programmer, their high-level idea is to dynamically construct the structure of the programmer intended query whenever the execution reaches a program location that issues a SQL-query. This approach is to compute the intended query by running the application on candidate inputs that are self-evidently non attacking. The crux of this approach is to avoid the problem of finding candidate inputs that exercise a control path, and instead derive the intended query structure directly from the same control path. It suggest that we can simply execute the statements along the control path on any benign candidate input, ignoring the conditionals that lie on the path. The idea of executing the statements on a control path, but not the conditionals along it, is a new idea.

Drawback is that, this technique using dynamic candidate evaluation is inefficient in dealing with external functions and when applied at wrong level.

3.3.4. Dynamic Tainting

Dynamic approaches based on tainting input strings, tracking the taints along a run of the program, and checking if any keywords in a query are tainted before executing the query are a powerful formalism for defending against SQL injection attacks. This is used in the work named *Automatically hardening web applications using precise tainting*[8] and in many others. Preventing SQL injections requires taking advantage of precise taint information. Before sending commands to the database, e.g. mysql_query, we run the following algorithm to check for injections:

1. Tokenize the query string; preserve taint markings with tokens.
2. Scan each token for identifiers and operator symbols (ignore literals, i.e., strings, numbers, boolean values).
3. Detect an injection if an operator symbol is marked as tainted. Operator symbols are ,()[].;:+-*/\%^<>=~!?@#&|`
4. Detect an injection if an identifier is tainted and a keyword. Example keywords include UNION, DROP, WHERE, OR, AND.

Example Query : $cmd="SELECT user FROM users WHERE user = ' " . $user . "' AND password = ' " . $password . " ' ";

The resulting query string (with $user set to ' OR 1 = 1 ; -- ') is tainted as follows: SELECT user FROM users WHERE user = ' ' OR 1 = 1 ; -- ' AND password = 'x'.

They detect an injection since OR is both tainted and a keyword.

Problem is that, even though this approach sounds good in many cases, there are some difficulties in its implementation; especially the propagation of taints across function calls is very difficult.

3.3.5. Dynamic Bracketing

This is an approach where the application program is manually transformed at program points where input is read, and the programmer explicitly brackets these user inputs (using random strings) and checks right before issuing a query whether any SQL keyword is spanned by a bracketed input. An example approach using this method is *Parse Tree validation to prevent SQL Injection Attacks* [9].

The technique is based on comparing, at run time, the parse tree of the SQL statement before inclusion of user input with that resulting after inclusion of input. A parse tree is a data structure for the parsed representation of a statement. Parsing a statement requires the grammar of the statement's language. By parsing two statements and comparing their parse trees, we can determine if the two queries are equal. When a malicious user successfully injects SQL into a database query, the parse tree of the intended SQL query and the resulting SQL query do not match.

Problem is that, it relies on the programmer to correctly handle the strings at various stages; for example, if the input is checked by a conditional, the brackets must be stripped away before evaluating the conditional.

3.3.6.Signature Based System

It is a combination of pattern-based detection and anomaly-based detection for a robust intrusion detection system[10]. There are two models of detection used by this system.

- Signature-based Detection Model
- Anomaly-based Detection Model

The signature-based detection fails to detect unknown attacks, while anomaly-based detection will detect unusual activity and behavior. Here, signature means known SQL Injection patterns. In the signature-based detection model, the input obtained from an HTML form is compared to signatures. If the input is found to match a signature, access is denied and the user is given a generic invalid username/password screen. This is to reduce the information returned to attacker through error messages. The drawback of the signature-based detection model is it cannot detect attacks that are unknown. For this they use anomaly-based detection model. In the anomaly-based detection model, the number of times a user attempts to log into the system, successful or not, is considered. If the attempts from a user exceed a predetermined number, the system will lock out this user's IP for a period of time. Further attacks are not possible because the attacker's IP address is subsequently blocked.

3.3.7. Obfuscation Based Method

This technique combines static and dynamic analysis[11]. In the static phase, the queries in the application are replaced by queries in obfuscated form. The main idea behind obfuscation is to isolate all the atomic formulas from other control elements of the query. During the dynamic phase, the user inputs are merged into the obfuscated atomic formulas, and the dynamic verifier analysis the presence of possible SQLIA at atomic formula level. Finally, a de-obfuscation step is performed to recover the original query before submitting it to the DBMS.

The proposed scheme has three phases, the first one is performed statically, while the latter two are performed dynamically.

- Obfuscating the legitimate query *Q* into *Q*0 at each hotspot of the application.
- After merging the user inputs into the obfuscated query at run-time, the dynamic verifier checks the obfuscated query at atomic formula level in order to detect the presence of possible SQLIA.
- Reconstruction of the original query *Q* from the obfuscated query *Q*0 before submitting it to the database, if no possible SQLIA was detected.

3.3.8. Agent Based Systems

AIIDA SQL agent[12] is a hybrid intelligent agent which integrates the use of Case Based Reasoning (CBR) engine for adaptation and learning capability and a mixture of Artificial Neural Networks (ANN) and Support Vector Machine (SVM) for classification. The lifetime of this agent includes four stages - retrieval, reuse, revise and retain. The retrieval stage includes selection of queries based on their type and memory classification models. Reuse phase includes the prediction of new query using a Multilayer Perceptron (MLP) and a SVM simultaneously. Once the output values for the ANN and the SVM are obtained, compute the weighted average of the error rate of each one of the techniques. If different classifications are obtained from each technique, the query would then be classified as suspicious, and subsequent revision would be launched. The revise phase can be manual or automatic depending on the output values. The automatic review is given for non-suspicious cases. Retain phase includes reconstruction of classifiers offline to made it available for new classifications.

3.3.9. Using Biological Algorithms

In [13] they used a biological algorithm- Hirschberg algorithm, which is a pair wise sequence alignment of amino acid code formulated from Web application form parameter sent via Web server. Then it analyzes the transaction to find out the malicious access. The Hirschberg algorithm is a divide and conquers approach to reduce the time and space complexity. This system was able to stop all of the successful attacks and did not generate any false positives. This algorithm finds least cost sequence alignment between two strings and this capability is utilized in finding SQL injection attacks in an optimal way. Other alternative algorithms are BLAST and FASTA which are suboptimal heuristics.

Table 2 includes overview of the prevention techniques discussed above. The comparison is based on the different types of SQL Injection Attacks.

TABLE 1
COMPARISON OF SOLUTION METHODS

Technique	Tautology	Illegal	Piggy Back	Union	Stored Pro-ced-ure	Inference	Alternate en-coding
SQL DOM	*	*	*	*	X	*	*
SQLrand	*	X	*	*	X	*	X
AMNESIA	*	*	*	*	X	*	*
Tainting	*	*	*	*	*	*	*
SQL Check	*	*	*	*	X	*	*
SQL Guard	*	*	*	*	X	*	*
CANDID	*	*	*	*	*	*	*

4 CONCLUSION

Nowadays, many organizations use web applications to provide services to users. Web applications depend on the backend database to supply with correct data. However, data stored in databases are often targets of attackers. SQL injection is a prominent technique that attackers use to compromise databases. Even though many solutions are proposed against this attack by researchers, database vendors and developers, still, SQL injection vulnerability is one of the top vulnerabilities present in the web applications. In this paper we describe this attack in detail with its different types. Also we classified different proposed solutions into main three categories, defensive coding, static analysis and defense mechanisms and explained specific properties of each type. This is an exclusive review on methods proposed in the literature.

REFERENCES

[1] OWASP, O.W.(2010). OWASP Top 10 for 2010‖. *Category: OWASP Top Ten Project* http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project (Apr. 14, 2011).

[2] Halfond, W.G, Viegas, J, and Orso, A. (2006) A classification of SQL Injection Attacks and countermeasures.

[3] Russel A. McClure and Ingolf H. Kruger, 2005 SQL DOM: Compile Time Checking of Dynamic SQL Statements.

[4] Hossain Shahriar Mohammad Zulkernine, MUSIC: Mutationbased SQL Injection Vulnerability Checking , The Eighth International Conference on Quality Software

[5] Boyd, S. W., and Keromytis, A. D. Sqlrand: Preventing sql injection attacks. In ACNS (2004), pp. 292–302.

[6] Halfond, W., and Orso, A. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In ASE (2005), pp. 174–183.

[7] Prithvi Bisht, P. Madhusudan, V. N. VENKATAKRISHNAN. CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks. ACMTransactions on Information and System Security,Vol. 13, No. 2, Article 14, Publication date: February 2010.

[8] Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D. Automatically hardening web applications using precise tainting. In SEC (2005), pp. 295–308.

[9] Buehrer, G., Weide, B. W., and Sivilotti, P. A. G. Using parse tree validation to prevent sql injection attacks. In SEM (2005).

[10] Varian Luong, 2010. Intrusion Detection and Prevention Systems: SQL Injection Attacks.

[11] Raju Halder and Agostino Cortesi, 2010 IEEE, Obfuscation-based Analysis of SQL Injection Attacks.

[12] Cristian PinZon, Alvaro Herreno, Juan F. De Paz, Javier Bajo and Emilio Corchado, AIIDA-SQL: An Adaptive Intelligent Intrusion Detector Agent for Detecting SQL Injection Attacks.

[13] Ezhumalai R and Aghila G, 2009 IEEE. Combinatorial Approach for Preventing SQL Injection Attacks.