Detection and Prevention of SQL Injection Attacks

[1]Sailor Pratik, [2]Prof. Jaydeep Gheewala
[1]Computer Department
[1]Sarvajanik College of Engineering and Technology, Surat, Gujarat, India
[1]pratik_sailor@ymail.com,[2]jaydeep.gheewala@scet.ac.in

***Abstract*—The Internet and web applications are playing very important role in our today's modern day life. Several activities of our daily life like browsing, online shopping and booking of travel tickets are becoming easier by the use of web applications. Most of the web applications use the database as a back-end to store critical information such as user credentials, financial and payment information, company statistics etc. An SQL injection attack targets web applications that are database-driven. This is done by including portions of SQL statements in a web form entry field in an attempt to get the website to pass a newly formed rogue SQL command to the database. Multiple client side and server side vulnerabilities like SQL injection and cross site scripting are discovered and exploited by malicious users. The principle of basic SQL injection is to take advantage of insecure code on a system connected to the internet in order to pass commands directly to a database and to then take advantage of a poorly secured system to leverage an attacker's access. Even if the some security mechanisms can protect database successfully, we must need to know what kinds of attacks are happening. However, there are many SQL injection attacks which can bypass data filters, which makes it difficult for the application to effectively defend the database from attacks.**

***Index Terms*—SQL Injection, Vulnerabilities, Web Security, Threat, Risks, Cross Site Scripting attack**

I. INTRODUCTION

Although a web application is simply recognized as a program running on a web browser, a web application generally has a three-tier construction as shown in Fig. 1 [1, 2]. In Fig. 1, a presentation tier is sent to a web browser by request of the browser.

1. Presentation Tier: This tier receives the user input and shows the result of the processing to the user. It can be thought of as the Graphical User Interface (GUI). Flash, HTML, Java script, etc. are all part of the presentation tier, which directly interacts with the user. This tier is analyzed by a web browser.
2. CGI Tier: Also known as the Server Script Process, this is located in between the presentation and database tiers. This tier will process the input by user and result calculated will be sent to the Database tier. The database tier sends the stored data back to the CGI tier, and it is finally sent to the presentation tier to be viewed by the user. Therefore, data processing within the web application is performed at the CGI Tier and can be programmed in various server script languages such as JSP, PHP, ASP, etc.
3. Database Tier: This tier only stores and retrieves all of the data. All sensitive data are Protected and managed here. Since this tier is directly connected to the CGI tier without any security check, data in the database can be revealed and modified if an attack on the CGI tier succeeds.

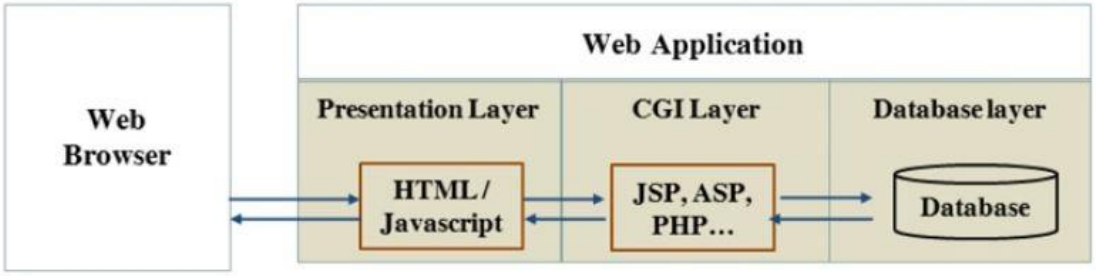

Figure 1: Web Application Architecture [1, 2]

II. EXAMPLE OF SQL INJECTION ATTACK

The SQL injection vulnerabilities are in between the presentation and CGI tiers, thus attacks occur between these tiers. Most of the vulnerabilities accidentally emerge in the development stage of the application program. The data flows among the three tiers using both normal and malicious input data are shown in Fig. 2 [3] as an example. This kind of attack is called a tautology and occurs at the user authentication step. When a genuine user enters their genuine ID and password, the presentation tier uses the GET and POST method to send the correct data to the CGI tier. The SQL query within the CGI tier connects to the database and processes the authentication procedure. The following is based on Fig. 2 [3].

1. Tautologies

If a malicious user enters an ID such as 1‘ or ‗1 = 1‘—, the query within the CGI tier becomes SELECT ∗ FROM user WHERE id = ‗1‘ or ‗1 = 1‘—‘ AND password = ‗1111‘. Because the rest of the string following—becomes a comment and ‗1 = 1‘ is always true, the authentication step is bypassed [3].

2. **Illegal/Logically Incorrect Queries**
   This attack derives the CGI tier replies error message by inserting a malicious SQL query such as query 1.
   Query 1:
   SELECT * FROM user WHERE id=‘1111‘ AND password=‘1234‘ AND CONVERT(char, no) --‘;
   The purpose of this attack is to collect the structure and information of CGI [3].
3. **Union Queries**
   This attack uses the ‗‗Union‘‘ operator which performs unions between two or more SQL queries. This attack performs unions of malicious queries and a normal SQL query with the ‗‗union‘‘ operator. Query 2 shows an example.
   Query 2:
   SELECT * FROM user WHERE id=‘1111‘ UNION
   SELECT * FROM member WHERE id=‘admin‘ --‘ AND password=‘1234‘;
   All subsequent strings after — are recognized as comments, and two SQL queries are processed in this example. The result of the query process shows administrator‘s information of the DBMS [3].
4. **Piggy-Backed Queries**
   This attack inserts malicious SQL queries into a normal SQL query. It is possible because many SQL queries can be processed if the operator ‗‗;‘‘ is added after each query. Query 3 is an example. Note that the operator ‗‗;‘‘ is inserted at the end of query.
   Query 3:
   SELECT * FROM user WHERE id=‘admin‘ AND password=‘1234‘; DROP TABLE user; --‘;
   The result of query 3 is to delete the user table [3].
5. **Stored Procedures**
   Recently, DBMS has provided a stored procedures method with which a user can store his own function that can be used as needed. To use the function, a collection of SQL queries is included. An example is shown in query 4.
   Query 4:
   CREATE PROCEDURE DBO @userName varchar2, @pass varchar2,
   AS
   EXEC("SELECT * FROM user WHERE id=‘" + @userName + "‘ and password=‘" + @password + "‘);
   GO [3]

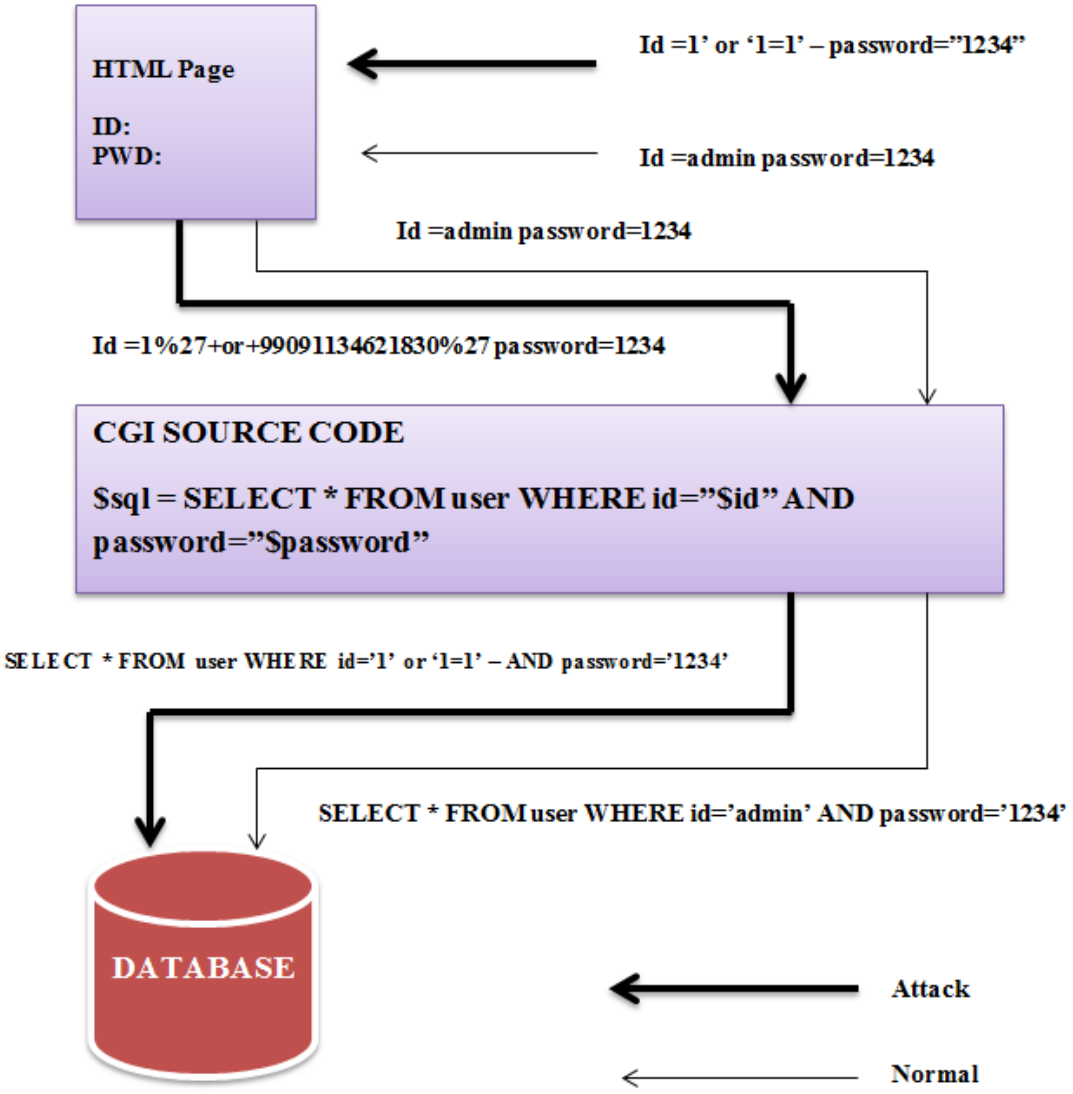

Figure 2: SQL normal and SQL injection attack data flow. [3]

III. Related Work

JDBC-Checker [4] prevents from SQL injection attacks by validating user inputs using Java String Analysis (JSA). However, if input data are in correct format or syntax, it fails to prevent from SQL injection attack. Other thing is, the JSA library can only support the Java programming language. Wassermann [5] used a static analysis method which was combined with automated reasoning. This method cannot detect attacks other than Tautologies. Stephen [6] created a fix generation SQL query by collecting plain text SQL statements, SQL queries, and execution calls to validate user input types. Instead of actually preventing and detecting SQL injection attacks, it is deleting vulnerabilities in advance. But the problem is that it is only available for web applications written with Java, and requires the AST and ZQL libraries [7].

Paros [8] is an open source, detects not only SQL injection attacks, but also other vulnerabilities too. Paros is not effective because it uses predetermined attack codes.Sania [9] protects against SQL injection attacks by using the following procedures. (a) It collects normal SQL queries between client and web applications and between the web application and database, and analyzes the vulnerabilities. (b) It generates SQL injection attack codes which can reveal vulnerabilities. (c) After attacking with the generated code, it collects the SQL queries generated from the attack. (d) The normal SQL queries are compared and analyzed with those collected from the attack, using a parse tree. (e) Finally, it determines whether the attack succeeded or not.

AMNESIA [10] consists of four main steps. (a) Identify hotspots: Study whole application to find the hotspots. Hotspots are points in the application code that issue SQLqueries to the underlying database. (b) Build SQL-query models: For each and every hotspot in the application, Form a model that represents all of the possible SQL queries that may be generated at the particular hotspot. An SQL-query model is a non-deterministic finite state automaton in which the transition labels consist of SQL tokens, de-limiters, and placeholders for string values [10]. (c) Instrument Application: At each hotspot in the application, add calls to the runtime monitor [10]. (d) Runtime monitoring: At runtime, check the dynamically generated queries against the SQL-query model and reject and report queries that violate the model [10].

IV. Cross Site Scripting Attack (XSS Attack)

XSS attack the hacker infects a legitimate web page with his malicious client-side script. When a user visits this web page the script is downloaded to his browser and executed. A malicious website might employ JavaScript to make changes to the local

system, such as copying or deleting files. A malicious website might employ JavaScript to monitor activity on the local system, such as with keystroke logging. [28]A malicious website might employ JavaScript to interact with other Websites the user has open in other browser windows or tabs. Exploited XSS is commonly used to achieve the following malicious results:

- Identity theft
- Accessing sensitive or restricted information
- Gaining free access to otherwise paid for content
- Spying on user's web browsing habits
- Altering browser functionality
- Public defamation of an individual or corporation
- Web application defacement
- Denial of Service attacks

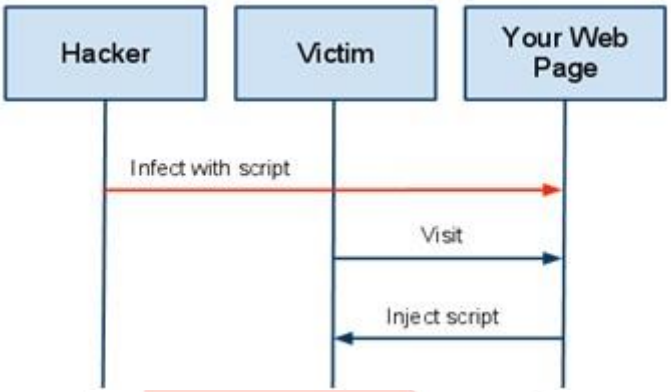

Figure 3: Cross Site Scripting Attack [18]

V. PROPOSED METHODOLOGY

If "Union" keyword is found in input field, Chances are there that attack may be Union Query Attack. If "And", "convert", "order", "char", "varchar", "nchar" keywords are found in input field , Chances are there that attack may be Illegal or Illogical Queries Attack. If "Drop", "Delete", "Alter" keywords are found in input field, Chances are there that attack may be Piggy Backed Query Attack. If "Script" keyword is found in input field, Chances are there that attack may be Cross Site Scripting Attack. If "=" keyword is found in input field, Chances are there that attack may be Tautologies Attack.

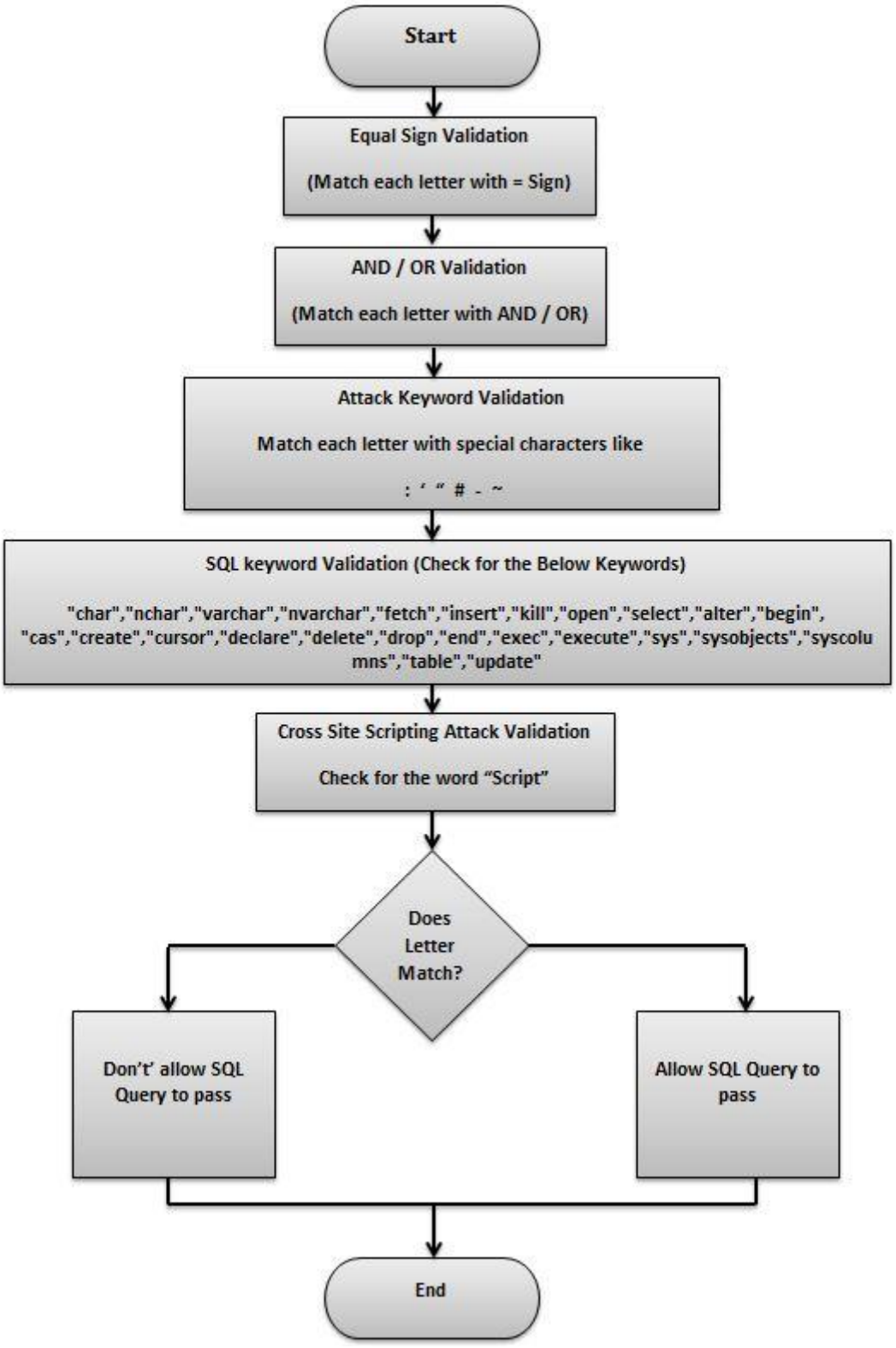

Figure 4: Proposed Methodology

Proposed Methodology is Validation Based Approach which prevents attack like Tautology, Illegal/Incorrect Query Attack, Union Query Attack, Stored Procedure Attack, Piggy Back Query Attack and Cross Site Scripting Attack and Detection Ration is almost 100% but we are getting some cases of False Alarm. We have compared our approach with some existing methods.

VI. COMPARISON OF PROPOSED METHODOLOGY WITH SOME EXISTING METHODS

Table 1: Comparison of methods for SQL injection attacks [3]

Detection/ Prevention methods	Tautologies Attack	Illegal/Incorrect Queries Attack	Union queries	Piggy-Backed queries	Stored procedures	Cross Site Scripting Attack

AMNESIA [10]	✓	✓	✓	✓	✖	✖
CSSE [11]	✓	✓	✓	✓	✖	✖
Web App. Hardening [12]	✓	✓	✓	✓	✖	✖
Safe Query Objects [13]	✓	✓	✓	✓	✖	✖
SQLCheck [14]	✓	✓	✓	✓	✖	✖
SQLrand [15]	✓	✖	✓	✓	✖	✖
SQL – DOM [16]	✓	✓	✓	✓	✖	✖
Tautology-checker [17]	✓	✖	✖	✖	✖	✖
Proposed Methodology	✓	✓	✓	✓	✓	✓

✓ Possible ✖ Not Possible

VII. Classification Of Attacks

With this we can classify the attacks to which category they belong to.

1. If "Union" keyword is found in input field , Chances are there that attack may be Union Query Attack
2. If "And", "convert", "order", "char", "varchar", "nchar" keywords are found in input field , Chances are there that attack may be Illegal or Illogical Queries Attack
3. If "Drop", "Delete", "Alter" keywords are found in input field , Chances are there that attack may be Piggy Backed Query Attack
4. If "Script" keyword is found in input field , Chances are there that attack may be Cross Site Scripting Attack
5. If "=" keyword is found in input field , Chances are there that attack may be Tautologies Attack

Below the comparison is shown in which method Classification of Attack is possible

Table 2: Comparison of methods in terms of Classification of Attacks

Detection/ Prevention methods	**Tautologies Attack**
AMNESIA [10]	✖
CSSE [11]	✖
Web App. Hardening [12]	✖
Safe Query Objects [13]	✖
SQLCheck [14]	✖
SQLrand [15]	✖
SQL – DOM [16]	✖
Tautology-checker [17]	✖
Proposed Methodology	✓

✓ Possible ✖ Not Possible

VIII. Chances of "False Alarm"

Problem: Our proposed methodology will detect all types of attack and will detect almost every attack but it has a limitation that it also gives "False Alarm". False Alarm is the case when entered string is genuine user but it is treated as Attack. For examples "Havmor" is genuine user but it contains the word "or", so here it is treated as Attack, "Island" has keyword "and", so it will be treated as Attack here.

Solution: How to reduce False Alarm? We should look for the Individual Malicious word in the Input Field than finding malicious word as a substring in the input field. That means instead of finding "or" in "Havmor", look for the "or" as individual word and instead of finding "and" in "Island", look for the "and" as individual word.

IX. Conclusion and Future Work

Our work is inspired by a situation of large number of SQL injections attacks and Cross Site Scripting attacks. We have recorded all the input strings which are responsible for the query execution and analyzed them thoroughly using the described steps. We have executed the methodology on PHP website and analyzed all the SQL strings. The experiment results provide complete scenario of the problem and accuracy of proposed steps. Our system indicated all types of SQL Injection attacks and all SQL Injection attacks. Advantage of the methodology is that it also prevents against stored procedure attack and Cross site scripting attack. Other Advantage is that this methods can classify the attacks to which category they belongs to but Major Drawback of this method is False Alarm, If user inputs SQL keywords as a User Name and however they are neither attack nor a Genuine User Id, They will treat as an attack. if user inputs User name as "Convert", it's neither an attack exactly nor a genuine user name, but it will treat as an attack here. So we cannot reduce False Alarm Completely and Future work is to reduce False Alarm as much as possible and to develop various solutions to reduce False Alarm.

REFERENCES

[1] G. Buehrer, B.W. Weide, P.A. Sivilotti, Using parse tree validation to prevent SQL injection attacks, in: Proceedings of the 5th International Workshop on Software Engineering and Middleware, 2005, pp. 105–113.

[2] Z. Su, G. Wassermann, The essence of command injection attacks in web applications, in: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2006, pp. 372–382.

[3] Lee, Inyong, Soonki Jeong, Sangsoo Yeo, and Jongsub Moon. "A novel method for SQL injection attack detection based on removing SQL query attributes values." Mathematical and Computer Modelling 55, no. 1 (2012): 58-68.

[4] C. Gould, Z. Su, P. Devanbu, JDBC checker: a static analysis tool for SQL/JDBC applications, in: Proceedings of the 26th International Conference on Software Engineering, ICSE, 2004, pp. 697–698.

[5] G. Wassermann, Z. Su, An analysis framework for security in web applications, in: Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems, SAVCBS, 2004, pp. 70–78.

[6] S. Thomas, L. Williams, Using automated fix generation to secure SQL statements, in: Proceeding of the 29th International Conference on Software Engineering Workshops, ICSEW, IEEE Computer Society, 2007, p. 54.

[7] P.-Y. Gibello, Zql: A java SQL parser. http://www.gibello.com/code/zql.

[8] Paros. Parosproxy.org. http://www.parosproxy.org/.

[9] Y. Kosuga, K. Kernel, M. Hanaoka, M. Hishiyama, Y. Takahama, Sania: yntactic and semantic analysis for automated testing against SQL injection, in: Proceedings of the Computer Security Applications Conference 2007, 2007, pp. 107–117.

[10] Halfond, William GJ, and Alessandro Orso. "Preventing SQL injection attacks using AMNESIA." In Proceedings of the 28th international conference on Software engineering, pp. 795-798. ACM, 2006.

[11] T.C. Pietraszek, V. Berghe, Defending against injection attacks through context–sensitive string evaluation, in: Proceeding of Recent Advances in Intrusion Detection, in: LNCS, vol. 3858, 2006, pp. 124–145.

[12] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, D. Evans, Automatically hardening web application using precise tainting information, in: Twentieth IFIP Iternational Information Security Conference, in: LNCS, vol. 181, 2005, pp. 295–307.

[13] W.R. Cook, S. Rai, Safe query objects: statically typed objects as remotely executable queries, in: Proceedings of the 27th International Conference on Software Engineering, 2005, pp. 97–106.

[14] Z. Su, G. Wassermann, The essence of command injection attacks in web applications, in: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2006, pp. 372–382.

[15] J. Park, B. Noh, SQL injection attack detection: profiling of web application parameter using the sequence pairwise alignment, in: Information Security Applications, in: LNCS, vol. 4298, 2007, pp. 74–82.

[16] R. McClure, I. Krüger, SQL DOM: compile time checking of dynamic SQL statements, in: Proceedings of the 27th International Conference on Software Engineering, 2005, pp. 88–96.

[17] G. Wassermann, Z. Su, An analysis framework for security in web applications, in: Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems, SAVCBS, 2004, pp. 70–78.

[18] Cross Site Scripting Attack, http://www.acunetix.com/websitesecurity/cross-site-scripting