ISSN: 0974-5823 Vol. 7 No. 1 January, 2022
International Journal of Mechanical Engineering

SQL Injection Vulnerabilities: Understanding Eliminating Approaches in Web Applications

[1]Najla'a Ateeq Mohammed Draib, [2]Abu Bakar Md Sultan, [3]Abdul Azim Abdul Ghani, [4]Hazura Zulzalil
Dept. of Software Engineering and Information System, Faculty of Computer Science and Information Technology
Universiti Putra Malaysia

Abstract - Structured Query Language Injection Vulnerabilities (SQLIVs) have consistently been top-ranked for the past few years, as eventually specified by the Open Web Applications Security Project (OWAPS). SQL Injection Attack (SQLIA) is a technique that exploits SQLIVs that occur in the database layer of a web application. The consequence of SQL injection attack would be devastating. A successful attack can threaten data confidentiality, data integrity and application availability. Finding the proper solution to stop or mitigate SQL injection is necessary. Researchers introduce different techniques to develop secure codes, eliminate SQL injection vulnerabilities, and prevent SQL injection attacks for addressing this problem. This paper concentrates on various security approaches for eliminating SQL injection vulnerabilities in the early stages of the software development life cycle. It also describes some exist gaps in the current state of the art of eliminating SQL injection vulnerabilities.

Index Terms - Web application; SQL injection; Vulnerabilities; Removal; Automated vulnerabilities elimination

INTRODUCTION

In today's era, Web applications form the backbone of the modern Internet. Their popularity and acceptance are rapidly growing due to the utmost convenience, accessibility, and omnipresence they provide. These online applications are reliable and efficient solutions to business challenges, delivering information and services, and are a great communication medium. At the time of writing, the coronavirus disease (COVID-19) pandemic is still ongoing. Governments imposed restrictions by putting citizens under several forms of lockdown to avoid the warned Covid-19 new cases' numbers. Therefore, people's dependence on using the Internet increased significantly. In particular, Web applications are widely used to fulfil almost all their daily needs and activities such as social activities, work, education, and entertainment. Thus, Web applications have become a truly ubiquitous and transformative force in people daily life.

Unfortunately, web applications are designed with hard time restrictions, and therefore, are often deployed with varying degrees of unexpected security vulnerabilities that are exploitable by hackers through different types of attacks [1]–[3].

The high ubiquity of web applications, their high global exposure, the growing reliance on them, and the importance and sensitivity of the information stored in their database have made them an attractive target for cyber-attacks who always try to uncover and maliciously exploit such security vulnerabilities. Among web application vulnerabilities, Structured Query Language Injection Vulnerabilities (SQLIVs) have consistently been top-ranked for the past few years, as eventually specified by [4]–[7]. SQL Injection Attack (SQLIA) is a technique that exploits a vulnerability that occurs in the database layer of a web application. It takes advantage of SQLIVs in the input validation and improper handling of submitted requests in server-side programs which interact with the database server.

The consequence of the SQL injection attack would be devastating. A successful attack hinders integrity, privacy, and information availability in the database. The attacker uses this attack intending to bypass the authentication procedure (authentication lost), extracting data from the backend database (confidentiality is lost), and/or altering existing data (integrity is lost)[8]–[12].

In order to protect the application from a malicious user, test procedures for identifying and removing SQLIVs must be implemented earlier in the software development life cycle (SDLC) of Web applications before being deployed into production and open to a potentially malicious attacker[13].

Over the past years, various mitigation approaches have been identified for securing Web applications by both the academic field and industries. Several papers have concentrated on SQL injection vulnerabilities [8][11] [14]–[17]. However, most existing review studies focus on SQLIVs detection, SQLIA detection, or/and SQLIA prevention. Very few studies discuss SQLIVs eliminating technique in the testing phase. This paper emphasised the techniques used to eliminate SQL injection vulnerability from the source code in the testing phase of web applications development.

The rest of the paper is organised as follows. Section II gives an overview of related work. Section III and Section IV explain SQL injection vulnerabilities and SQL injection attacks. Section V introduces several SQL vulnerabilities elimination approaches in the early stages of the software development life cycle. Section VI makes a brief description. Finally, Section VII is the conclusion.

RELATED WORK

References [2][12][15]–[19] discuss the various aspects of SQLI, such as SQLIA types, mechanisms, and prevention approaches, but they do not shed alight on the vulnerabilities removal approaches.

Sadeghian, Zamani, and Manaf [20] presented a review of different types of SQL injection detection and prevention techniques. The paper focuses on the classification of available manual solutions and proposed solutions to prevent vulnerabilities after the

software deployment but does not discuss the available solutions to remove vulnerabilities automatically before the software deployment.
Reference [19] presented a review on SQLI types, evasion techniques, and countermeasures. This paper classifies countermeasure for removing or blocking SQLIVs into eight classes; parameterised query, least privilege, customised error message, system stored procedure reduction, SQL keyword escaping, and input variable length checking. However, it does not highlight the solutions that have been proposed to get rid of the vulnerabilities.
Kindy and Pathan [21] presented a detailed survey on types of SQL injection vulnerabilities, attacks, and their prevention techniques. However, this study focused on approaches that automate SQLIVs prevention at runtime.
We can conclude from previous research that there is a shortage of papers dealing with deleting loopholes after their discovery. This research presents a classification of the existing techniques to eliminate such vulnerabilities and discusses their problems and strengths in solving the gaps. It also highlights the areas of research available for development.

SQL INJECTION VULNERABILITIES

SQL injection vulnerability (SQLIV) refers to possible software security flaws associated with database-driven web applications, which could be exploited through SQLIAs. Typically, SQLIV takes place in the code when user-supplied data (i.e., URL parameters or HTML form inputs) is allowed to propagate from input source to security-critical operation (e.g., database queries) without proper sanitisation. The vulnerability is caused by code fragments where unsanitised input is interpreted as SQL code instead of being treated as data. Depending on the environment, these security flaws might enable an attacker to compromise underlying databases of web applications resulting in unwanted extraction or insertion of data from or into a database.
In order to prevent SQLIVs, prepared statements can be used, or vulnerabilities can be patched by sanitising or validating the user input before it is embedded into the query/beforehand. For this purpose, a security mechanism is applied between the user input (input source) and the sensitive operation (sensitive sink) so that malicious data cannot reach the sensitive operation[22].

SQL INJECTION ATTACKS

SQL injection attack (SQLIA) is a notorious hacking technique in which the attacker exploits SQLIVs of web applications connected to a database to inject SQL code fragments into vulnerable input parameters (e.g. HTTP requests). The malicious code masquerades as user input and is embedded in the SQL query. Executing such SQL statements by the database might enable the attacker to view, modify the database structure, or manipulate critical data[17]. SQLIAs are possible due to insufficient input validation or improper construction of SQL statements in web applications[22].

The main consequences of successful SQLIA include loss of confidentiality, authentication, and integrity of information in the database. The attacker can obtain control over and access the backend database without providing an authentic user name and password, which is a major problem with SQLIA and may lead to damaging ramifications since the backend database usually stores sensitive assets [11][23].

A. Basic SQL injection attack types

There are many types of SQL injection attacks that hackers have developed for exploiting SQL injection vulnerabilities. Related studies such as [24][25] have classified SQL injection attacks as follows:
Tautologies
Illegal/Logically incorrect queries
Piggy-backed
Stored Procedure
Alternate encoding
Inference Based Attacks
Blind Injection (True/False)
The interested reader can refer to [17] and [19] for more details about SQLIA basic types.

B. SQL injection types based on input mechanisms

Practically, SQL injection can be introduced into vulnerable web applications using two main mechanisms based on the injection order: first-order SQL injection and second-order SQL injection [17] [24][11].
First-order SQL injection
First-order attack is the basic type of SQL injection attack. In this attack, the attacker inserts SQL commands into a vulnerable input field that flows directly from an entry point (e.g., $_GET) to a sensitive sink (e.g., mysqli_query); the successful injection results are delivered immediately upon user-input submission. First-order SQL injection attacks can be launched using any aforementioned attack types by injecting malicious input throw user input, cookies, or server vulnerabilities[2][11].

C. Second order-SQL injection

Second-order SQLIA, also called stored or persistent SQLIA, is a particular type of SQLIA that is more severe and more difficult to be detected and has strong concealment [17] [25]–[27] Second-order attacks belong to multi-step attacks and are accomplished by combining two-step inputs[28]. A Second-order SQL injection attack is developed on a first-order SQL injection attack. In

such attacks, the attacker first seeds SQL commands into the database and uses that input later in a sensitive sink to launch the attack. Web applications must have a vulnerability that allows malicious code injection into the database before the second attack input is entered. To illustrate, Figure 1. presents the Web application state transition diagram of such an attack.

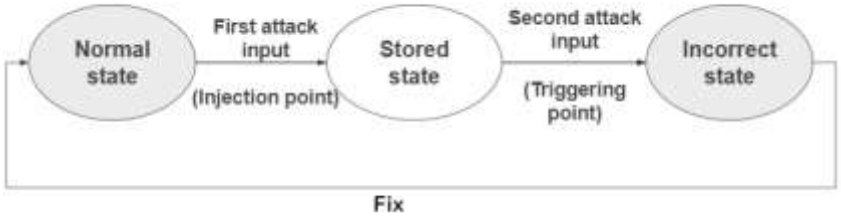

Fig. 1 Web application state transition diagram Adapted from [28].

D. SQLIVs elimination approaches

This section elaborates on the security techniques/mechanisms proposed by researchers to eliminate and prevent the introduced vulnerabilities. Various mitigation techniques have been identified for securing Web applications from SQLIVs. Some of the most relevant literature for defending SQLIVs are going to be discussed.

Existing approaches for eliminating SQLIVs can be categorised into two main categories; manual defensive coding practices and automated vulnerability removal approaches[18].

Manual defensive coding practices

Secure coding practice is a common and very effective way for defeating SQLIVs and minimising their exploitation probability. Secure coding enables the developers to follow secure code practices during the application development to avoid such vulnerabilities[29][30].

These code practices involve escaping, properly sanitising user-supplied input, data type checking, parameterised queries, using stored procedures, whitelist filtering, and using the principle of least privilege. OWASP's SQL injection prevention cheat sheet provides useful manual defensive coding guidelines [39]. For example, parameterised queries, where the developer must define all the SQL code first, use a placeholder (e.g. "?") as a reference to user inputs, and then pass each input to the query later. This coding style allows the database to distinguish between code and data, regardless of what user input is supplied, thereby avoiding attacks. Another example is proposed in [31], where a high-level input validation mechanism is introduced in order to block malicious input to Web applications.

Reference [32] proposed a secured coding approach for SQL injection prevention. Developers can use this approach at the time of development to secure their applications against SQL injection attacks. In this approach, all user input must be checked thoroughly before interacting with the database. The main focus was on; input and URL validation, data sanitisation, prepared statement, and query and session tokenisation.

E. Automated vulnerability removal approaches

Usually, after having detected vulnerabilities by using detection methods, other methods could be applied to cure the web application, such as applying the prepared statements technique or inserting sanitisation methods to the vulnerable code. Existing approaches for automating vulnerability prevention can be divided into two main approaches: fixing the vulnerability in the early stages of the web application development and curing the application once it is under attack. In this section, we are going to discuss the former one. Approaches for eliminating SQL injection vulnerabilities at the testing phase, before the application deployment, can be classified as follows:

F. Parameterised query insertion

Prithvi et al. [33] applied a method to eliminate SQL injection attacks in legacy web applications by automatically retrofitting prepared statements through automated code transformation. This method uses symbolic execution to identify the query's arguments and requires the original source code to be maintained so that the method can regenerate it upon the modifications.

Thomas and Williams [34] proposed an automated method for retrofitting SQL statements with prepared statements. This method automatically transforms the code to secure SQL statements by injecting prepared statements to the abstract syntax tree of the code so that it does not allow altering the query structure during the runtime.

Dysart and Sherriff [35] introduced a solution that identified potentially vulnerable queries and then generated a new solution using the prepared statement technique. The framework established vulnerability detection by parsing the SQL statement to identify if any variable is used to build the statement without sanitisation. The aim was to provide the developer with quick feedback that could help them maintain their software.

Reference [36] proposed a learning-based approach for mitigating SQLIVs. The authors use vulnerable source code collected from GitHub [37], and handcrafted SQLIV fixes based on OWASP prepared statement guidelines[38] to generate training data and then feed training data to a hierarchical clustering-based model. The abstract syntax tree (AST) of the vulnerable code is

compared with AST of a similar cluster to find a match and suggest fixes for the vulnerable code. This approach relies on a static analysis technique to spot the vulnerable SQL statements.

Abadi, Feldman, and Shomrat [39] presented Code-Motion for automatic eradication of SQLIVs, an algorithm that automatically replaces the code that makes use of Statement API to use PreparedStatement API instead. However, this algorithm is restricted to particular scenarios; it cannot perform code refactoring if the number of variables and the variables' type is on different sides of a conditional query.

Reference [40] presented an approach to automatically detect and fix three Web security risks: cross-site scripting, server misconfiguration, and SQL injection. This approach used a static analysis tool, ESLint [41], to detect and fix vulnerabilities automatically. The authors implemented four ESLint rules for eliminating vulnerabilities. According to the authors, these rules enhanced the tool to fix XSS and server misconfiguration automatically. However, it just suggested cod fixes using prepared statements to guide the developer in eliminating SQLIVs.

G. Data Sanitization

Medeiros, Neves, and Correia [1] introduced an approach for detecting and removing SQLIVs. This approach used a combination of taint analysis and data mining to detect the web application source code vulnerabilities. After performing the vulnerability detection, the approach applies the code correction automatically to the source code using the information returned by taint analysis about the slice of the vulnerable code. The vulnerability removal is done by inserting fixes to the SQL statement before reaching a sensitive sink (e.g., mysql_query). These fixes apply PHP sanitisation functions (e.g., mysql_real_escape_string) to the statements' arguments to make them sanitised before executing the statement by a sensitive sink. However, this approach is limited to removing first-order SQLIVs. It applies to escape methods to sanitise the data before inserting it into the database; however, this data can be retrieved later to build a new SQL query, leading to second-order SQLIV.

Mui and Frankl [42] presented an automatic technique for identifying and sanitising vulnerable SQL statements to SQLI. This technique uses a combination of static analysis and program transformation techniques. The static analysis is devoted to identifying and locating the vulnerable SQL statement, and then instrumentation is used for inserting sanitisation function calls. The location where the calls are inserted is determined based on the location of where the tainted variable is concatenated into the SQL query. The sanitisation function type is determined based on the type of the tainted variable corresponding attribute in the database schema; an SQL parser and the database schema are used to identify the attribute type.

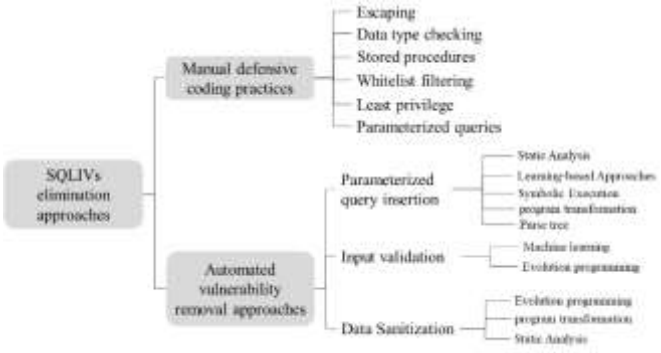

Fig. 2 General Mapping of SQLIV eliminating approaches

H. Input validation

Scholte et al. [43] presented a novel technique for eliminating SQLIVs and XSS vulnerabilities in Web applications by transparently learning data types of Web application parameters during testing and then enforcing robust validation for these parameters at runtime. That technique was implemented into a tool called IPAAS (Input Parameters analysis System). However, the proposed technique I incapable of preventing SQLIAs through parameters that accept free text.

Reference [44] presented a platform for predicting and correcting SQL injection and XSS vulnerabilities using machine learning. The proposed system first scans the Web application using different payloads for different vulnerabilities and then use the scanner's output as an input to a fortifier that suggests a secure code to the user using built-in sanitise functions.

Reference [45], propose a framework for detecting and removing SQL injection vulnerabilities in Islamic websites. The framework employs evolution programming to model web application SQLIVs fixing as a search problem by establishing co-evolution of web applications and test sets. This framework establishes a competitive co-evaluation of vulnerable web application sets and test sets; only those web applications that are able to defend test attacks and pass legitimate input tests are involved to the next generation. Consequently, a more secure source code that can produce a secure version of the original code is generated.

Discussion

In practice, secure coding practices do not guarantee a secure Web application. Most developers are usually under time-to-market pressure and often misuse/unknowingly neglect these defeating methods when using database technologies, resulting in SQLIVs.

That might result from the lack of training, lack of security skills, lack of effective preventing techniques, or/and lack of experience [46] [47].

Additionally, some programming languages, such as PHP and Java, provide various defense mechanisms for developing secure web applications. Unfortunately, that does not guarantee secure web applications. The developer may not enforce the security mechanisms during coding and may sometimes not use the proper validation, leading to advanced attacks, partly due to inexperience, lack of security concern, or time [48].

Fixes suggestion tools provide quick feedback to the developers that could aid them in maintaining their software. However, these techniques only aid the developer by suggesting a safer code to eliminate the vulnerabilities and leave their removal as a burden on the programmer.

The prepared statement is one of the most used approaches for fixing SQL injection vulnerabilities. It is known that separating the code from the data sent by the user is one of the best ways to implement sentences safely. However, practically not all vulnerable columns/parameters are applicable to this separation technique. Therefore, there must be another technique to deal with such cases.

Moreover, most of the discussed approaches only addressed fixing the first-order SQLIVs and did not consider the second-order SQLIVs fixing. However, a mechanism to fix SQLI vulnerabilities cannot afford to defend against the second-order SQLIA attack because the malicious inputs supplied by the attacker are concatenated with the SQL statement at the database level, not at the application level. To the best of our knowledge, none of the current automated methods, but [36], is able to do this. Hence, the actual fixing of the vulnerabilities is left for the human developer to handle. Manual removal of such SQL injection vulnerabilities is tedious, error-prone, and costly. Second-order injections are difficult to prevent as the point of injection is different from the point of attack. Thus, more care should be taken in order to detect and prevent them. Both attack points should be validated carefully (Point of injection as well as point of attack).

CONCLUSION

SQL injection is one of the severe threats to web applications security. Research is ongoing for an effective way to test the source code early before the software deployment and eliminate this threat. SQL injection attack is a notorious hacking technique in which the attacker exploits SQLIVs of web applications connected to a database to inject SQL code fragments into vulnerable input parameters (HTTP requests). The malicious code masquerades as user input and is embedded in the SQL query. The main consequences of successful SQLIA include loss of confidentiality, authentication, and integrity of information in the database. Although many reliable solutions have been proposed for SQLIVs detection and prevention in web applications, it still exists and has become a giant threat to many organisations. This paper focused on the approaches introduced to eliminate SQLIVs in web applications source code during the testing phase. There is a lack of techniques to eliminate or remove these vulnerabilities from the source code of the web applications prior to deployment. Therefore, more research is needed in the field of SQLIVs detection and removal from the source code of web applications in the testing phase before deployment. The study states current approaches to eliminating SQLIVs on which can be further extended and analysed.

Acknowledgements

This research was supported by the Malaysian Ministry of Higher Education through grants FRGS 08-01-19-2159FR to Universiti Putra Malaysia.

REFERENCES

[1] I. Medeiros, N. Neves, and M. Correia, "Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining," *IEEE Trans. Reliab.*, vol. 65, no. 1, pp. 54–69, 2016.

[2] P. Kaur and K. P. Kour, "SQL injection: Study and augmentation," *Proc. 2015 Int. Conf. Signal Process. Comput. Control. ISPCC 2015*, pp. 102–107, 2016.

[3] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL Injection and cross-site scripting attacks," *2009 IEEE 31st Int. Conf. Softw. Eng.*, pp. 199–209, 2009.

[4] Owasp, "OWASP Top 10:2021," 2021. [Online]. Available: https://www.owasptopten.org/the-release-of-the-owasp-top-10-2021. [Accessed: 25-Sep-2021].

[5] SANS/CWE, "CWE - 2019 CWE Top 25 Most Dangerous Software Errors," 2019.

[6] Acunetix, "Web Application Vulnerability Report 2020," 2020.

[7] TRUSTWAVE, "2020 SECURITY GLOBAL TRUSTWAVE REPORT," 2020.

[8] R. Johari and P. Sharma, "A survey on web application vulnerabilities (SQLIA, XSS) exploitation and security engine for SQL injection," *Proc. - Int. Conf. Commun. Syst. Netw. Technol. CSNT 2012*, no. May 2012, pp. 453–458, 2012.

[9] N. Mishra, S. Chaturvedi, A. K. Sharma, and S. Choudhary, "XML-based authentication to handle SQL injection," in *Advances in Intelligent Systems and Computing*, 2014, vol. 236, pp. 739–749.

[10] Z. S. Alwan and M. F. Younis, "Detection and Prevention of SQL Injection Attack: A Survey," *Int. J. Comput. Sci. Mob. Comput.*, vol. 6, no. 8, pp. 5–17, 2017.

[11] S. A. Faker, M. A. Muslim, and H. S. Dachlan, "A Systematic Literature Review on SQL Injection Attacks Techniques and Common Exploited Vulnerabilities," *Int. J. Comput. Eng. Inf. Technol.*, vol. 9, no. 12, pp. 284–291, 2017.

[12] J. Hu, W. Zhao, and Y. Cui, "A Survey on SQL Injection Attacks, Detection and Prevention," in *ACM International Conference Proceeding Series*, 2020, pp. 483–488.

[13] A. J. Nathan and A. Scobell, “How China sees America,” *Foreign Affairs*, vol. 91, no. 5. pp. 1365–1367, 2012.
[14] M. . Lawal, M. A. B. Sultan, and A. O. Shakiru, “438. Systemic Literature Review on SQL Injection Attacks,” *International Journal of Soft Computing*, vol. 11, no. 1. pp. 26–35, 2016.
[15] S. M. H. Chaki and M. Mat Din, “A Survey on SQL Injection Prevention Methods,” *Int. J. Innov. Comput.*, vol. 9, no. 1, pp. 47–54, 2019.
[16] V. Prokhorenko, K. K. R. Choo, and H. Ashman, “Web application protection techniques: A taxonomy,” *Journal of Network and Computer Applications*, vol. 60. pp. 95–112, 2016.
[17] W. G. J. Halfond, J. Viegas, and A. Orso, “A Classification of SQL Injection Attacks and Countermeasures,” 2008.
[18] L. K. Shar and H. B. K. Tan, “Defeating SQL injection,” *Computer (Long. Beach. Calif).*, vol. 46, no. 3, pp. 69–77, 2013.
[19] A. Kumar, S. Choudhary, and A. K. Jain, “A Detail Survey on Various Aspects of SQLIA Posture recognition for safe driving View project A Detail Survey on Various Aspects of SQLIA,” *Artic. Int. J. Comput. Appl.*, vol. 161, no. 12, pp. 975–8887, 2017.
[20] A. Sadeghian, M. Zamani, and A. A. Manaf, “A taxonomy of SQL injection detection and prevention techniques,” *Proc. - 2013 Int. Conf. Informatics Creat. Multimedia, ICICM 2013*, pp. 53–56, 2013.
[21] D. A. Kindy and A. S. K. Pathan, “A detailed survey on various aspects of sql injection in web applications: Vulnerabilities, innovative attacks and remedies,” *Int. J. Commun. Networks Inf. Secur.*, vol. 5, no. 2, pp. 80–92, 2013.
[22] W. Du, K. Jayaraman, X. Tan, T. Luo, and S. Chapin, “Position paper: Why are there so many vulnerabilities in web applications?,” in *Proceedings New Security Paradigms Workshop*, 2011, pp. 83–93.
[23] N. Singh, M. Dayal, R. S. Raw, and S. Kumar, “SQL Injection: Types, Methodology, Attack Queries and Prevention,” *2016 Int. Conf. Comput. Sustain. Glob. Dev.*, pp. 2872–2876, 2016.
[24] C. Sharma and S. C. Jain, “Analysis and classification of SQL injection vulnerabilities and attacks on web applications,” in *2014 International Conference on Advances in Engineering and Technology Research, ICAETR 2014*, 2014.
[25] A. Muraleedharan, “A Robust Method for Prevention of Second Order and Stored Procedure based SQL Injections,” pp. 20–23, 2015.
[26] H. Choudhury, B. Roychoudhury, and D. K. Saikia, “Efficient Detection of Multi-step Cross-Site Scripting Vulnerabilities,” *Int. J. Netw. Secur.*, vol. 18, no. 6, pp. 1041–1053, 2016.
[27] C. Ping, “A second-order SQL injection detection method,” *Proc. 2017 IEEE 2nd Inf. Technol. Networking, Electron. Autom. Control Conf. ITNEC 2017*, vol. 2018-Janua, pp. 1792–1796, 2018.
[28] M. Liu and B. Wang, “A Web Second-Order Vulnerabilities Detection Method,” *IEEE Access*, vol. 6, pp. 70983–70988, 2018.
[29] R. A. McClure and I. H. Krüger, “SQL DOM: Compile time checking of dynamic SQL statements,” in *Proceedings - 27th International Conference on Software Engineering, ICSE05*, 2005, pp. 88–96.
[30] C. Musciano and B. Kennedy, *HTML & XHTML : The Definitive Guide 4th edition*. 2000.
[31] D. Scott and R. Sharp, “Abstracting application-level web security,” in *Proceedings of the 11th International Conference on World Wide Web, WWW ’02*, 2002, pp. 396–407.
[32] B. Gautam, J. Tripathi, S. Singh, and M. T. Student, “A Secure Coding Approach For Prevention of SQL Injection Attacks,” *Int. J. Appl. Eng. Res.*, vol. 13, no. 11, pp. 9874–9880, 2018.
[33] P. Bisht, A. P. Sistla, and V. N. Venkatakrishnan, “Automatically preparing safe SQL queries,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6052 LNCS, pp. 272–288, 2010.
[34] S. Thomas and L. Williams, “Using automated fix generation to secure SQL statements,” in *Proceedings - ICSE 2007 Workshops: Third International Workshop on Software Engineering for Secure Systems, SESS’07*, 2007, p. 9.
[35] F. Dysart and M. Sherriff, “Automated fix generator for SQL injection attacks,” *Proc. - Int. Symp. Softw. Reliab. Eng. ISSRE*, no. May, pp. 311–312, 2008.
[36] M. L. Siddiq, M. R. R. Jahin, M. R. Ul Islam, R. Shahriyar, and A. Iqbal, “SQLIFIX: Learning Based Approach to Fix SQL Injection Vulnerabilities in Source Code,” *Proc. - 2021 IEEE Int. Conf. Softw. Anal. Evol. Reengineering, SANER 2021*, pp. 354–364, 2021.
[37] “GitHub: Where the world builds software · GitHub.” [Online]. Available: https://github.com/. [Accessed: 23-Jul-2021].
[38] “SQL Injection Prevention - OWASP Cheat Sheet Series.” [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html. [Accessed: 29-Sep-2021].
[39] A. Abadi, Y. a Feldman, and M. Shomrat, “Code-motion for API migration: Fixing SQL injection vulnerabilities in Java,” in *WRT 2011 - Proceedings of the 4th Workshop on Refactoring Tools, co-located with ICSE 2011*, 2011, pp. 1–7.
[40] W. Rafnsson, R. Giustolisi, M. Kragerup, and M. Høyrup, “Fixing Vulnerabilities Automatically with Linters.”
[41] K. F. Tomasdottir, M. Aniche, and A. Van Deursen, “The Adoption of JavaScript Linters in Practice: A Case Study on ESLint,” *IEEE Trans. Softw. Eng.*, vol. 46, no. 8, pp. 863–891, 2020.
[42] R. Mui and P. Frankl, “Preventing SQL Injection through Automatic Query Sanitisation with ASSIST,” *Electron. Proc. Theor. Comput. Sci.*, vol. 35, pp. 27–38, 2010.
[43] T. Scholte, W. Robertson, D. Balzarotti, and E. Kirda, “Preventing input validation vulnerabilities inweb applications through automated type analysis,” *Proc. - Int. Comput. Softw. Appl. Conf.*, pp. 233–243, 2012.
[44] R. Tommy, G. Sundeep, and H. Jose, “Automatic Detection and Correction of Vulnerabilities using Machine Learning,” in *2017 International Conference on Current Trends in Computer, Electrical, Electronics and Communication (CTCEEC)*, 2017, pp. 1062–1065.
[45] K. Umar, A. B. Sultan, H. Zulzalil, N. Admodisastro, and M. T. Abdullah, “Prevention of attack on Islamic websites by fixing SQL injection vulnerabilities using co-evolutionary search approach,” in *The 5th International Conference on*

Information and Communication Technology for The Muslim World (ICT4M), 2014, pp. 1–6.
[46] S. Steiner, D. C. de Leon, and J. Alves-Foss, "A structured analysis of SQL injection runtime mitigation techniques," in *Proceedings of the Annual Hawaii International Conference on System Sciences*, 2017, vol. 2017-Janua, pp. 2887–2895.
[47] M. Saidu Aliero, I. Ghani, S. Zainudden, M. Murad Khan, and M. Bello, "Review on sql injection protection methods and tools," *Jurnal Teknologi*, vol. 77, no. 13. pp. 49–66, 2015.
[48] T. Scholte, D. Balzarotti, and E. Kirda, "Have things changed now? An empirical study on input validation vulnerabilities in web applications," *Comput. Secur.*, vol. 31, no. 3, pp. 344–356, 2012.