Recent Developments in Electronics and Communication Systems
KVS Ramachandra Murthy et al. (Eds.)

doi:10.3233/ATDE221269

Prevention of SQL Injection Using a Comprehensive Input Sanitization Methodology

Parveen SULTANA H[1] and Nishant SHARMA and Nalini N and Gaurav PATHAK and Ayush PANDEY
School of Computer Science and Engineering, Vellore Institute of Technology, Vellore, Tamil Nadu, 632014, India

Abstract. Databases are essential to modern-day web applications. Structured Query Language (SQL) used to retrieve information from relational database is prone to injection attack by a malicious user. Losing access to application database destroys all credibility for the organization. Mitigating against SQL injection attacks is a critical concern. This paper proposes a methodology for the safe flow of SQL query from the user layer to the database layer. The proposed methodology uses the process of client-side input sanitization to validate user input before a dynamic SQL query is constructed. On the server-side, another process provides server-side query sanitization to ensure a full-proof validation of the SQL query before it is run on the database. The client-side input sanitization process is presented in the paper. On comparison with related methodologies, the proposed methodology is robust, lightweight, and fast.

Keywords. SQL, Vulnerability, Injection Attack, Authentication, Elevation of Privilege, Security

1. Introduction

Modern-day web applications use database as an essential component to store data. A popular choice is a relational database. Structured Query Language (SQL) is a language used to retrieve information from relational databases. Figure 1 shows a three-tier architecture for information retrieval in web applications. A business logic layer serves as an interface between the user layer and the database layer. SQL injection is a technique of malicious intent, where an SQL query is injected with the intention of violating authentication, and elevating privileges. Injection ranks third in OWASP top ten web application security risks for the year 2021 [1]. A username and password field form the login page of a web service. A user enters the login information. Ideally, a login is allowed only if the entered information for both the fields is correct. Otherwise, the login is denied. Table 1 presents a reference for two propositions, 'prop 1' and 'prop 2' evaluated for Boolean values, 'true' and 'false' under common logical operators, and ('&&'), or ('||'), and not ('!').

[1] Corresponding Author, Parveen Sultana H, School of Computer Science and Engineering, Vellore Institute of Technology, Vellore, Tamil Nadu, 632014, India; E-mail: hparveensultana@vit.ac.in.

SQL injection attack makes use of the vulnerabilities in the backend code that dynamically create an SQL query without properly sanitizing the entered username and password in the login form. A flowchart depicting the use case when the username field of a login form is injectable is given in figure 2. When the username field is injectable, the malicious user can execute a series of commands that leak sensitive information. In figure 2, the malicious user finds all the tables in the database using the generic information_schema. The malicious user then finds the tables whose schema is public, all columns from the desired table, and then the desired sensitive information. Even with attacks that cause denial of service at the server, the end goal is to gain access to the database. Once the access to the database is obtained, the entire system collapses.

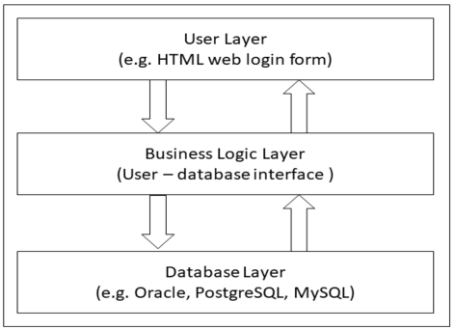

Figure 1. Three tier architecture for information retrieval in web applications.

To perform an SQL injection attack, the attacker will be perseverant and have strong inference. While studying SQL injection, one has the benefit to peek at the server-side code. However, realistically that may not be the case. A lot of trial-and-error goes into bypassing the system. Types of response may vary depending on the implementation of response at server. Some servers may respond with an error message, while others may just emulate a refresh on the page. Some web services may show a blank page. It is then up to the attacker to patiently find an injectable parameter to perform SQL injection. It is evident but worth noting, that the attacker may only win once. Backend codes that construct dynamic queries are written by humans and a mistake is possible. The defense against SQL injection attacks should be robust. An SQL injection attack detection and mitigation system must be created assuming that the attacker is well versed in the technical know-how and has good infrastructure. In this paper, a methodology is proposed for the safe flow of SQL query from the user layer to the database layer.

Table 1. Propositions, 'prop 1' and 'prop 2' evaluated for Boolean values, 'true' and 'false' under common logical operators *and* ('&&'), *or* ('||'), and *not* ('!').

Prop 1	Prop 2	Prop 1 && Prop 2	Prop 1 || Prop 2	! (Prop 1)	! (Prop 2)
False	False	False	False	True	True
False	True	False	True	True	False
True	False	False	True	False	True
True	True	True	True	False	False

2. Literature Survey

SQL injection attack can be of many types. This umbrella term includes attacks that make use of tautologies, response errors, union queries, piggybacked queries, response time, malicious code injection to stored procedures, inference, and alternate encodings. The prevention of SQL injection can be done using input sanitization and validation, stored procedures, restricting user access and input length, encoding user information, and validating results [2,3]. Sanitization of input, prepared statements and tokens can be used in a secure coding approach to prevent SQL injection attack [4,5,6,7]. SQL injection attack is relevant to the modern day. IoT and Cloud are vulnerable to SQL injection attacks, and mitigation against these attacks is a major concern [8,9,10,11]. Machine learning methods can bring more insights to understanding SQL injection attacks. Reinforcement learning approach is seen as an important frontier [12,13].

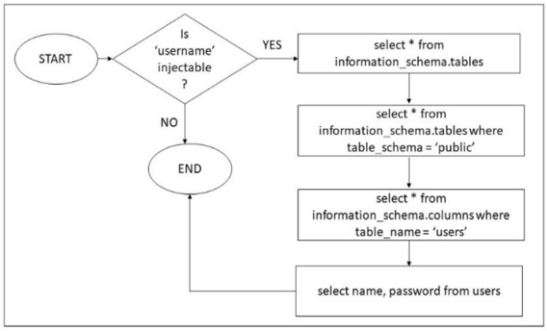

Figure 2. Flowchart depicting the steps that lead to leakage of sensitive information when a user input field is left vulnerable to injection.

3. System Setup

A user enters login information at the user layer as shown in figure 1. The process employed by the business logic layer assists in retrieving the appropriate information from the database layer. In this paper, a custom setup is created to model information retrieval from a database. There are three aspects to the system. The user, the connector, and the database. A user is prompted to enter their username and password. This is like a login form present in modern websites. A table named *users* stores the username and password for various users in the system. The database used is PostgreSQL. The python - psycopg2 library is used to interface the user with the database. The database postgres stores *users*. The database connector has a query of the form *"select * from users where name='"+username+"' and password= '"+password+"';"* which uses the information provided by the user to retrieve information from *users*. The information should be successfully retrieved only if the username and password field are correct. Table 2 presents what happens when a normal user uses the system, and when a malicious attacker attempts an SQL injection attack on *users* and injects the username or password field with a payload. Note that for any username, if a payload such as *abc' or '1' = '1* is injected into the password field, the considered query returns all tuples in the database. This is evident if we substitute the password field in *"select * from users where name='"+username+"' and password= '"+password+"';"* with *abc' or '1' = '1* and

note that the expression becomes *name = username and password = abc or '1' = '1'* which evaluates to true (because *and* has higher precedence than *or*.) Such attacks are referred to as *tautology* attacks in literature.

Table 2. Normal and attack vector for the execution of a dynamic SQL query on a remote database. 'X' represents normal input or payload. '!' represents non-existence of the given normal input in the column.

Username	Input Type	Password	Input Type	Result
Incorrect	Normal	Incorrect	Normal	No result
Correct	Normal	Incorrect	Normal	No result
Incorrect	Normal	Correct	Normal	No result
Correct	Normal	Correct	Normal	Correct tuple retrieved
X	X	abc' or '1' = '1	Payload	All tuples retrieved
abc' or '1' = '1	Payload	!	Normal	No result
abc' or '1' = '1	Payload	Correct	Normal	Tuple with respective password retrieved
abc' or '1' = '1	Payload	abc' or '1' = '1	Payload	All tuples retrieved
abc' or '1' = '1' - -	Payload	X	X	All tuples retrieved

Some systems like MySQL allow *alternate encodings* like hexadecimal to be executed on the system. The hexadecimal input starts with 0x. When an input is hexadecimal, the rest of the string following 0x is decoded and runs on the database. It is possible to inject multiple SQL queries separated by a semicolon. These are referred to as *piggybacked queries* in the literature. A payload such as *abc'); select current_database(), version;* -- or its hexadecimal counterpart 0x61626327293b73656c65637420637572726e745f6461746261736528292c7665 7273696f6e28293b202d2d may be inserted into the username. The correct details of the database and the version are displayed as a tuple ('postgres', 'PostgreSQL 12.10 (Ubuntu 12.10-0ubuntu0.20.04.1) on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0, 64-bit').

The transaction between the business logic layer and the database layer is not immune to SQL injection. An SQL injection attack may occur even though proper input sanitization has been done between the user layer and the business logic layer. The process flow is as follows. The login information is obtained at the user layer and is sanitized. The user layer invokes the service provided by the business logic layer. A dynamic SQL query is constructed. The SQL query must now be sent over a communication channel for its execution on a remote database. However, an interceptor intercepts the SQL query on the communication channel. After intercepting the SQL query, the malicious agent performs changes to the SQL query and further sends it on the communication channel for its execution on the remote database.

4. Proposed Methodology

Figure 3 presents the proposed methodology and the safe flow of SQL query from the user layer to the database layer. The user provides login information. The obtained information is sanitized for malicious code. If the user information is validated to be clean, it is input to the dynamic query creator. The SQL query is transferred via the

communication channel to the remote site that manages the database. A server-side query sanitization process ensures that the query received on the remote site is fit to run on the database. If so, the query is executed on the database. If the query is found to be malicious during the server-side query sanitization process, the database layer applies a mechanism suitable to deal with the compromised clients. A compromised client may be block-listed, and further communication may be stopped indefinitely. During the dynamic query construction, the query *"select * from users where name='"+username+"' and password= '"+password+";"* used previously is modified to "select * from users where (name='"+username+"') and (password='"+password+ "');". This modification provides protection against certain SQL injection attacks. The exploit by the malicious agent works only if the payload is inserted into both username and password fields. The SQL injection attack using comment characters is also prevented. Thereafter, algorithm 1 is used to sanitize the user information at the client side. Although essential to the proposed methodology, the server-side query sanitization process is not presented in this paper.

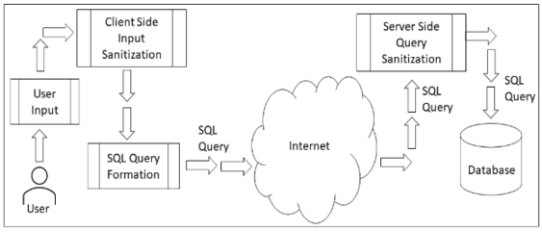

Figure 3. The safe flow of SQL query from the user layer to the database layer.

5. Results and Discussion

The proposed methodology uses the process of client-side input sanitization to mitigate against SQL injection attack. Several other techniques can also be used to mitigate against SQL injection attacks. However, such techniques have their drawbacks. Restricting user access requires extensive rule setting on part of the database administrator and falsely assumes that attackers cannot belong to the same organization. Restricting input length restricts flexibility. The proposed methodology in contrast to mitigation approaches presented in [2, 4, 5, 6, 7] is lightweight and fast. It provides a cumber less validation of SQL query. There is no overhead of token generation and processing. In dynamic query construction, the overhead created by construction, compilation and optimization of prepared statements is avoided. Table 3 presents the type of attacks successfully mitigated against by using algorithm 1. The running time of algorithm 1 is of the order of the length of the input fields.

Table 3. Types of attacks successfully mitigated against by using the proposed input sanitization approach compared with the recommendations given in [2]

Recommendation	Proposed client-side input sanitization
Tautologies	✓
Error attacks	×
Union queries	×
Piggybacked Queries	✓
Inference	×
Alternate encodings	✓

6. Conclusion

The contributions of the paper are as follows:

- The paper presents the severity of an SQL injection attack on modern-day web applications. It presents the flow of the leakage of sensitive information from an application database.
- The paper presents the background and simulation for three popular types of SQL injection attacks namely tautologies, piggybacked queries, and alternate encodings.
- The paper proposes a methodology for the safe flow of SQL query from the user layer to the database layer using a client-side input sanitization and a server-side query sanitization process.
- An algorithm to perform client-side input sanitization that is lightweight and fast is presented in the paper.

The proposed methodology mitigates against SQL injection attacks namely tautologies, piggybacked queries, and alternate encodings. The proposed methodology can be used as a lightweight alternative in mitigating against SQL injection attacks. A lightweight server-side query sanitization process that complements the client-side input sanitization process is left as future work.

References

[1] OWASP. “OWASP Top Ten” [Online]. Accessed from: https://owasp.org/www-project-top-ten/, last accessed 21-05-2022.

[2] Kausar MA, Nasar M, Moyaid A. SQL Injection Detection and Prevention Techniques in ASP .NET Web Application. International Journal of Recent Technology and Engineering (IJRTE). 2019 Sep;8(3):7759-66.

[3] Zaid Sabih. “Website Hacking / Penetration Testing & Bug Bounty Hunting”. Udemy [Online]. Accessed from: https://www.udemy.com/course/learn-website-hacking-penetration-testing-from-scratch/, last accessed: 21-05-2022.

[4] Gautam B, Tripathi J, Singh S. A secure coding approach for prevention of SQL injection attacks. International Journal of Applied Engineering Research. 2018;13(11):9874-80.

[5] Dhivya K, Kumar PP, Saravanan D, Pajany M. Evaluation of web security mechanisms using vulnerability & Sql attack injection. International Journal of Pure and Applied Mathematics. 2018;119(14):989-96.

[6] Patel D, Dhamdhere N, Choudhary P, Pawar M. A system for prevention of SQLi attacks. In2020 International Conference on Smart Electronics and Communication (ICOSEC) 2020 Sep 10 (pp. 750-753). IEEE.

[7] Ahmad K, Karim M. A Method to Prevent SQL Injection Attack using an Improved Parameterized Stored Procedure. International Journal of Advanced Computer Science and Applications. 2021;12(6).

[8] Xiao F, Zhijian W, Meiling W, Ning C, Yue Z, Lei Z, Pei W, Xiaoning C. An old risk in the new era: SQL injection in cloud environment. International Journal of Grid and Utility Computing. 2021;12(1):43-54.

[9] Ferrara P, Mandal AK, Cortesi A, Spoto F. Static analysis for discovering IoT vulnerabilities. International Journal on Software Tools for Technology Transfer. 2021 Feb;23(1):71-88.

[10] Saxena R, Gayathri E. A study on vulnerable risks in security of cloud computing and proposal of its remedies. InJournal of Physics: Conference Series 2021 Oct 1 (Vol. 2040, No. 1, p. 012008). IOP Publishing.

[11] Durai KN, Subha R, Haldorai A. A Novel Method to Detect and Prevent SQLIA Using Ontology to Cloud Web Security. Wireless Personal Communications. 2021 Apr;117(4):2995-3014.

[12] Ben Dickson. "Machine learning offers fresh approach to tackling SQL injection vulnerabilities". The Daily Swig [Online]. Accessed from: https://portswigger.net/daily-swig/machine-learning-offers-fresh-approach-to-tackling-sql-injection-vulnerabilities, last accessed: 21-05-2022.

[13] Erdődi L, Sommervoll ÅÅ, Zennaro FM. Simulating SQL injection vulnerability exploitation using q-learning reinforcement learning agents. Journal of Information Security and Applications. 2021 Sep 1;61:102903.