Analysis of SQL Injection Attack

Jasvinder Singh*

Abstract

These days almost all information is online. To store such information one needs huge databases. Over a past few years a new type of cyber security threat has come into existence namely Structured Query Language (SQL) injection attack. The same can be launched by means of web browsers. Typical uses of SQL injection leak confidential information from a database, by-pass authentication logic, or add unauthorized accounts to a database. The severity of this attack can be judged from the fact that in seconds of its inception all critical information is routed to malicious user, thus causing a tremendous amount of loss in terms of authentication, data integration and privacy. As a result, the system could bear heavy loss in giving proper services to its users or it may face complete destruction. Sometimes such type of collapse of a system can threaten the existence of a company or a bank or an industry.

Keywords: Structured Query Language Injection Attack (SQLIA), Java Database Connectivity (JDBC), Preprocessor Hypertext (PHP), Intrusion Detection System (IDS), Security Policy Descriptor Language (SPDL)

Introduction

SQL injection vulnerabilities are the most serious threats for Web applications [4, 15]. Such applications that are vulnerable to SQL injection may allow an attacker to gain complete access to their underlying databases. Attackers can even use an SQL injection vulnerability to take control of and corrupt the system that hosts the Web application. Web applications that are vulnerable to SQL Injection Attacks (SQLIAs) are widespread—a study by Gartner Group on over 300 Internet Web sites has shown that most of them could be vulnerable to SQLIAs. In fact, SQLIAs have successfully targeted high-profile victims such as Travelocity, FTD.com, and Guess Inc. SQL injection refers to a class of code-injection attacks in which data provided by the user is included in an SQL query in such a way that part of the user's input is treated as SQL code. By leveraging these vulnerabilities, an attacker can submit SQL commands directly to the database. These attacks are a serious threat to any Web application that receives input from users and incorporates it into SQL queries to an underlying database. Most Web applications used on the Internet or within enterprise systems work this way and could therefore be vulnerable to SQL injection. The cause of SQL injection vulnerabilities is relatively simple and well understood: insufficient validation of user input. To address this problem, developers have proposed a range of coding guidelines [9] that promote defensive coding practices, such as encoding user input and validation. A rigorous and systematic application of these techniques is an effective solution for preventing SQL injection vulnerabilities. However, in practice, the application of such techniques is human-based and, thus, prone to errors. Furthermore, fixing legacy code-bases that might contain SQL injection vulnerabilities can be an extremely labor-intensive task.

Jasvinder Singh*
IITM, GGSIPU

SQL Injection Vulnerability Problems

SQL injection vulnerability results from the fact that most web application developers do not apply user input validation and they are not aware about the consequences of such practices [11]. These inappropriate programming practices enable the attackers to trick the system by executing malicious SQL commands to manipulate the backend

database [11, 12]. One of the most important properties of SQL injection attack is that it is easy to be launched and difficult to be avoided. These factors make this kind of attack preferred by most cyber criminals, and it is getting more attention in the recent years [12]. Furthermore, the available scanning tools have limited features in shaping efficient attacking patterns which are required to detect hidden SQL injection vulnerability [11, 12]. Moreover, the available scanning tools use brute force techniques to extract data from the targeted websites. These tools do not show meaningful and detailed information about the detected vulnerability. Obtaining this critical detailed information would be very useful for web developers who are not aware about hacking techniques in helping them to fix the bugs, thus to eliminating these vulnerabilities.

SQL Injection Detection Technique

Detection approach to identify the SQL injections is as follows:

1. **Initial Attack-Validation**: It is the fastest step which is based on the historical attack analysis. Algorithm for the attack detection is as follows:
   a) Get input string
   b) Match Input string in attack table
   c) if result=true
   d) Declare "Attack" else Exit ();

The above algorithm validates the input SQL string using the Initial Knowledge Base which stores all the frequent SQL attacks of each category and is managed by the probabilistic approach. If the new input string pattern matches with the any of the patterns already stored in initial knowledge base, then it is declared as an SQL attack and a warning message will be generated automatically.

2. **AND-OR Validation:** This step determines the AND-OR word in the input String of web form(s). This Step uses some string comparing operations and parsing techniques to find the AND-OR locations and also counts AND & OR keywords separately before performing the SQL execution (mysql_query) operation. The algorithm for this step is as follows:
   a) Get input string
   b) Explore each word
   c) Repeat step d) and e) until string ends;
   d) Match each word with AND & OR keyword;
   e) If result=true /*Comment condition is true for any word*/
   f) Declare "Attack" else Exit ();

This step only contains parsing operations to validate the AND & OR keywords in the final SQL string of the user parameters.

3. **Equal Sign ('=') Validation:** This step analyzes the '=' syntax in the input String of web form(s) using Knuth–Morris–Prattalgorithm [17]. This Step compares the final SQL string of the user parameters with the standard format of SQL query. Algorithm for this step is as follows:
   a) Get input string
   b) Repeat step c) and d) until string ends.
   c) Match each letter with '=' keyword;
   d) If result=true /*Comment condition is true for any word*/ Declare "Attack" else Exit ();

This operation is also executed before execution of the commit statement of the SQL string (mysql_query).

4. **Attack Keywords Analysis:** In this algorithm, each letter of input word is explored and matched with a specific set of attack letters (:"', #, —, ~). Algorithm for this step is as follows:
   a) Get input string
   b) Explore Each letter
   c) Repeat step d) and e) until string ends.
   d) Match each letter with specific set of letters (:,",' ,#,— ,~);

e) If result=true /*Comment condition is true for any word*/ Declare "Attack" else Exit ();

Static Analysis

To perform static analysis of the stored procedure, a stored procedure parser which extracts the control flow graph from the stored procedure. All the EXEC (@SQL) statements in the control flow graph are labeled and then backtrack to identify all the statements involved in the construction of the @SQL statement in the control flow graph. In this process, an SQL-graph as explained below is generated. From the SQL-graph, SQL statements which depend on user inputs are selected and flagged to monitor their structure at runtime. At runtime, we compare the structure of the original intended SQL statement with the dynamically generated SQL statement having user inputs by using a Finite State Automaton. An SQLIA which alters the original structure will be flagged as unsafe and related information would be logged.

Prevention of Sqlias

Researchers have proposed a wide range of techniques to address the problem of SQL injection. These techniques range from development best practices to fully automated frameworks for detecting and preventing SQLIAs.

Defensive Coding Practices

The root cause of SQL injection vulnerabilities is insufficient input validation. Therefore, the straightforward solution for eliminating these vulnerabilities is to apply suitable defensive coding practices. Some of the best practices proposed in the literature for preventing SQL injection vulnerabilities are as follows:

1. **Input type checking:** SQLIAs can be performed by injecting commands into either a string or numeric parameter. Even a simple check of such inputs can prevent many attacks. For example, in the case of numeric inputs, the developer can simply reject any input that contains characters other than digits. Many developers omit this kind of check by accident because user input is almost always represented in the form of a string, regardless of its content or intended use.
2. **Encoding of inputs:** Injection into a string parameter is often accomplished through the use of meta-characters that trick the SQL parser into interpreting user input as SQL tokens. While it is possible to prohibit any usage of these meta-characters, doing so would restrict a non-malicious user's ability to specify legal inputs that contain such characters. A better solution is to use functions that encode a string in such a way that all meta-characters are specially encoded and interpreted by the database as normal characters.
3. **Positive pattern matching:** Developers should establish input validation routines that identify *good* input as opposed to *bad* input. This approach is generally called *positive validation*, as opposed to negative validation, which searches input for forbidden patterns or SQL tokens. Because developers might not be able to envision every type of attack that could be launched against their application, but should be able to specify all the forms of legal input, positive validation is a safer way to check inputs.
4. **Identification of all input sources:** Developers must check all input to their application. There are many possible sources of input to an application. If used to construct a query, these input sources can be a way for an attacker to introduce an SQLIA. Simply put, all input sources must be checked. Although defensive coding practices remain the best way to prevent SQL injection vulnerabilities, their application is problematic in practice. Defensive coding is prone to human error and is not as rigorously and completely applied as automated techniques. While most developers do make an effort to code safely, it is extremely difficult to

apply defensive coding practices rigorously and correctly to all sources of input. In fact, many of the SQL injection vulnerabilities discovered in real applications are due to human errors , developers forgot to add checks or did not perform adequate input validation [23, 18, 5]. In other words, in these applications, developers were making an effort to detect and prevent SQLIAs, but failed to do so adequately and in every needed location. These examples provide further evidence of the problems associated with depending on developer's use of defensive coding. Moreover, approaches based on defensive coding are weakened by the widespread promotion and acceptance of so-called "pseudo remedies" [9]. We discuss two of the most commonly-proposed pseudo-remedies. The first of such remedies consists of checking user input for SQL keywords, such as "FROM," "WHERE," and "SELECT," and SQL operators, such as the single quote or comment operator. The rationale behind this suggestion is that the presence of such keywords and operators may indicate an attempted SQLIA. This approach clearly results in a high rate of false positives because, in many applications, SQL keywords can be part of a normal text entry, and SQL operators can be used to express formulas or even names (e.g., O'Brian). The second commonly suggested pseudo-remedy is to use stored procedures or prepared statements to prevent SQLIAs. Unfortunately, stored procedures and prepared statements can also be vulnerable to SQLIAs.

Detection and Prevention Techniques

Researchers have proposed a range of techniques to assist developers and compensate for the shortcomings in the application of defensive coding.

1. **Black Box Testing:** Huang and colleagues [22] propose WAVES, a black-box technique for testing Web applications for SQL injection vulnerabilities. The technique uses a Web crawler to identify all points in a Web application that can be used to inject SQLIAs.

   It then builds attacks that target such points based on a specified list of patterns and attack techniques. WAVES then monitors the application's response to the attacks and uses machine learning techniques to improve its attack methodology. This technique improves over most penetration-testing techniques by using machine learning approaches to guide its testing. However, like all black-box and penetration testing techniques, it cannot provide guarantees of completeness.

2. **Static Code Checkers**: JDBC-Checker is a technique for statically checking the type correctness of dynamically-generated SQL queries [2, 3]. This technique was not developed with the intent of detecting and preventing general SQLIAs, but can nevertheless be used to prevent attacks that take advantage of type mismatches in a dynamically-generated query string. JDBC-Checker is able to detect one of the root causes of SQLIA vulnerabilities in code—improper type checking of input. However, this technique would not catch more general forms of SQLIAs because most of these attacks consist of syntactically and type correct queries. Wassermann and Su propose an approach that uses static analysis combined with automated reasoning to verify that the SQL queries generated in the application layer cannot contain a tautology [7].

   The primary drawback of this technique is that its scope is limited to detecting and preventing tautologies and cannot detect other types of attacks.

3. **Combined Static and Dynamic Analysis:** AMNESIA is a model-based technique that combines static analysis and runtime monitoring [20]. In its static phase, AMNESIA uses static analysis to build models of the different types of queries an application can legally generate at each point of access to the

database. In its dynamic phase, AMNESIA intercepts all queries before they are sent to the database and checks each query against the statically built models. Queries that violate the model are identified as SQLIAs and prevented from executing on the database. In their evaluation, the authors have shown that this technique performs well against SQLIAs. The primary limitation of this technique is that its success is dependent on the accuracy of its static analysis for building query models. Certain types of code obfuscation or query development techniques could make this step less precise and result in both false positives and false negatives.

Similarly, two recent related approaches, SQLGuard [6] and SQLCheck [24] also check queries at runtime to see if they conform to a model of expected queries. In these approaches, the model is expressed as a grammar that only accepts legal queries. In SQLGuard, the model is deduced at runtime by examining the structure of the query before and after the addition of user-input. In SQLCheck, the model is specified independently by the developer. Both approaches use a secret key to delimit user input during parsing by the runtime checker, so security of the approach is dependent on attackers not being able to discover the key. Additionally, the use of these two approaches requires the developer to either rewrite code to use a special intermediate library or manually insert special markers into the code where user input is added to a dynamically generated query.

4. **Taint Based Approaches:** WebSSARI detects input-validation related errors using information flow analysis [23]. In this approach, static analysis is used to check taint flows against preconditions for sensitive functions. The analysis detects the points in which preconditions have not been met and can suggest filters and sanitization functions that can be automatically added to the application to satisfy these preconditions. The WebSSARI system works by considering as sanitized input that has passed through a predefined set of filters. In their evaluation, the authors were able to detect security vulnerabilities in a range of existing applications. The primary drawbacks of this technique are that it assumes that adequate preconditions for sensitive functions can be accurately expressed using their typing system and that having input passing through certain types of filters is sufficient to consider it not tainted. For many types of functions and applications, this assumption is too strong.

Livshits and Lam [18] use static analysis techniques to detect vulnerabilities in software. The basic approach is to use information flow techniques to detect when tainted input has been used to construct an SQL query. These queries are then flagged as SQLIA vulnerabilities. The authors demonstrate the viability of their technique by using this approach to find security vulnerabilities in a benchmark suite. The primary limitation of this approach is that it can detect only known patterns of SQLIAs and, because it uses a conservative analysis and has limited support for untainting operations, can generate a relatively high amount of false positives. Several dynamic taint analysis approaches have been proposed. Two similar approaches by Nguyen-Tuong and colleagues [1] and Pietraszek and Berghe [16] modify a PHP interpreter to track precise per-character taint information. The techniques use a context sensitive analysis to detect and reject queries if untrusted input has been used to create certain types of SQL tokens. A common drawback of these two approaches is that they require modifications to the runtime environment, which affects portability.

A technique by Haldar and colleagues [19] and SecuriFly [10] implement a similar approach for Java. However, these techniques do not use the context sensitive analysis employed by the other two approaches and track taint information on a per-string basis (as opposed to per character). SecuriFly also attempts to sanitize query strings

that have been generated using tainted input. However, this sanitization approach does not help if injection is performed into numeric fields. In general, dynamic taint-based techniques have shown a lot of promise in their ability to detect and prevent SQLIAs. The primary drawback of these approaches is that identifying all sources of tainted user input in highly-modular Web applications and accurately propagating taint information is often a difficult task.

5. **New Query Development Paradigms:** Two recent approaches, SQL DOM [13] and Safe Query Objects [21], use encapsulation of database queries to provide a safe and reliable way to access databases. These techniques offer an effective way to avoid the SQLIA problem by changing the query-building process from an unregulated one that uses string concatenation to a systematic one that uses a type-checked API. Within their API, they are able to systematically apply coding best practices such as input filtering and rigorous type checking of user input. By changing the development paradigm in which SQL queries are created, these techniques eliminate the coding practices that make most SQLIAs possible. Although effective, these techniques have the drawback that they require developers to learn and use a new programming paradigm or query-development process. Furthermore, because they focus on using a new development process, they do not provide any type of protection or improved security for existing legacy systems.

6. **Intrusion Detection Systems:** F. Valeur [8] proposed the use of an Intrusion Detection System (IDS) to detect SQLIAs. Their IDS system is based on a machine learning technique that is trained using a set of typical application queries. The technique builds models of the typical queries and then monitors the application at runtime to identify queries that do not match the model. In their evaluation, Valeur and colleagues have shown that their system is able to detect attacks with a high rate of success. However, the fundamental limitation of learning based techniques is that they can provide no guarantees about their detection abilities because their success is dependent on the quality of the training set used. A poor training set would cause the learning technique to generate a large number of false positives and negatives.

7. **Proxy Filters:** Security Gateway [5] is a proxy filtering system that enforces input validation rules on the data flowing to a Web application. Using their Security Policy Descriptor Language (SPDL), developers provide constraints and specify transformations to be applied to application parameters as they flow from the Web page to the application server. Because SPDL is highly expressive, it allows developers considerable freedom in expressing their policies. However, this approach is human-based and, like defensive programming, requires developers to know not only which data needs to be filtered, but also what patterns and filters to apply to the data.

8. **Instruction Set Randomization:** SQLrand [14] is an approach based on instruction-set randomization. SQLrand provides a framework that allows developers to create queries using randomized instructions instead of normal SQL keywords. A proxy filter intercepts queries to the database and de-randomizes the keywords. SQL code injected by an attacker would not have been constructed using the randomized instruction set. Therefore, injected commands would result in a syntactically incorrect query. While this technique can be very effective, it has several practical drawbacks. First, since it uses a secret key to modify instructions, security of the approach is dependent on attackers not being able to discover the key. Second, the approach imposes a significant infrastructure overhead because it requires the integration of a proxy for the database in the system.

Conclusion

In this paper a rigorous discussion has been done regarding SQL injection attack. The vulnerabilities and threats caused by the same have been highlighted. Besides, the SQL injection and prevention techniques proposed by various researchers have been explained. Thus one can conclude that SQL injection is a grave threat to cyber security. From time to time new algorithms have to be proposed and implemented to protect the most valuable asset to human kind in today's world i.e. their databases.

References

1. A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. "Automatically Hardening Web Applications Using Precise Tainting Information." In Twentieth IFIP International Information Security Conference (SEC 2005), May 2005.
2. C. Gould, Z. Su, and P. Devanbu. JDBC Checker: "A Static Analysis Tool for SQL/JDBC Applications." In Proceedings of the 26th International Conference on Software Engineering (ICSE 04) –Formal Demos, pages 697–698, 2004.
3. C. Gould, Z. Su, and P. Devanbu. "Static Checking of Dynamically Generated Queries in Database Applications." In Proceedings of the 26th International Conference on Software Engineering (ICSE 04), pages 645–654, 2004.
4. D. Aucsmith. "Creating and Maintaining Software that Resists Malicious Attack." http://www.gtisc.gatech.edu/bioaucsmith.html,September 2004. Distinguished Lecture Series.
5. D. Scott and R. Sharp. "Abstracting Application-level Web Security." In Proceedings of the 11th International Conference on the World Wide Web (WWW 2002), pages 396–407, 2002.
6. G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti. "Using Parse Tree Validation to Prevent SQL Injection Attacks." In International Workshop on Software Engineering and Middleware (SEM), 2005.
7. G. Wassermann and 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), pages 70–78, 2004.
8. F. Valeur, D. Mutz, and G. Vigna. "A Learning-Based Approach to the Detection of SQL Attacks." In Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), Vienna, Austria, July 2005.
9. M. Howard and D. LeBlanc. "Writing Secure Code." Microsoft Press, Redmond, Washington, second edition, 2003.
10. M. Martin, B. Livshits, and M. S. Lam. "Finding Application Errors and Security Flaws Using PQL: A Program Query Language." In Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA 2005), pages 365–383, 2005.
11. Kemalis, K., & Tzouramanis, T. (2008). "SQL-IDS: A Specification-based Approach for SQL-Injection Detection." SAC '08. 2153-2158. Fertaleza, Ceara, Brazil.
12. Kiezun, A., Guo, P. J., Jayaraman, K., & Ernst, M. D. (2009). "Automatic Creation of SQL Injection and Cross-Site Scripting Attacks." ICSE '09. 199-209. Vancouver, Canada.

13. R. McClure and I. Kr¨uger. “SQL DOM: Compile Time Checking of Dynamic SQL Statements.” In Proceedings of the 27th International Conference on Software Engineering (ICSE 05), pages 88–96, 2005.

14. S. W. Boyd and A. D. Keromytis. “SQLrand: Preventing SQL Injection Attacks.” In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference, pages 292–302, June 2004.

15. T. O. Foundation. “Top Ten Most Critical Web Application Vulnerabilities, 2005”. http://www.owasp.org/documentation/topten.html.

16. T. Pietraszek and C. V. Berghe. “Defending Against Injection Attacks through Context-Sensitive String Evaluation.” In Proceedings of Recent Advances in Intrusion Detection (RAID2005), 2005.

17. Thomas H. Cormen, Charles E. Leiserson, Ronald L.Rivest and Clifford Stein, “Introduction to Algorithms” MIT Press/McGraw-Hill, 2001.

18. V. B. Livshits and M. S. Lam. “Finding Security Errors in Java Programs with Static Analysis.” In Proceedings of the 14th Usenix Security Symposium, pages 271–286, Aug. 2005.

19. V. Haldar, D. Chandra, and M. Franz. “Dynamic Taint Propagation for Java.” In Proceedings 21st Annual Computer Security Applications Conference, Dec. 2005.

20. W. G. Halfond and A. Orso. “AMNESIA: Analysis and Monitoring for Neutralizing SQL-Injection Attacks.” In Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005), Long Beach, CA, USA, Nov 2005.

21. W. R. Cook and S. Rai. “Safe Query Objects: Statically Typed Objects as Remotely Executable Queries.” In Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), 2005.

22. Y. Huang, S. Huang, T. Lin, and C. Tsai. “Web Application Security Assessment by Fault Injection and Behavior Monitoring.” In Proceedings of the 11th International World Wide Web Conference (WWW 03), May 2003.

23. Y. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo. “Securing Web Application Code by Static Analysis and Runtime Protection.” In Proceedings of the 12th International World Wide Web Conference (WWW 04), May 2004.

24. Z. Su and G. Wassermann. “The Essence of Command Injection Attacks in Web Applications.” In The 33rd Annual Symposium on Principles of Programming Languages (POPL 2006), Jan. 2006.