SQL Injection Attack Detection Using Fingerprints and Pattern Matching Technique

Benjamin Appiah *, Eugene Opoku-Mensah * and Zhiguang Qin *†
*School of Information and Software Engineering, UESTC, Chengdu, China.
† Director of UESTC–IMB Technology Center, Chengdu, China.
Emails: 1746627510@qq.com, ugino@ymail.com, qinzg@uestc.edu.cn

***Abstract*—Web-Based applications are becoming more increasingly technically complex and sophisticated. The very nature of their feature-rich design and their capability to collate, process, and disseminate information over the Internet or from within an intranet makes them a popular target for attack. According to Open Web Application Security Project (OWASP) Top Ten Cheat sheet-2017, SQL Injection Attack is at peak among online attacks. This can be attributed primarily to lack of awareness on software security. Developing effective SQL injection detection approaches has been a challenge in spite of extensive research in this area. In this paper, we propose a signature based SQL injection attack detection framework by integrating fingerprinting method and Pattern Matching to distinguish genuine SQL queries from malicious queries. Our framework monitors SQL queries to the database and compares them against a dataset of signatures from known SQL injection attacks. If the fingerprint method cannot determine the legitimacy of query alone, then the Aho Corasick algorithm is invoked to ascertain whether attack signatures appear in the queries. The initial experimental results of our framework indicate the approach can identify wide variety of SQL injection attacks with negligible impact on performance.**

***Index Terms*—SQL Injection Attack Detection, Pattern Matching, String Search, SQL Injection**

I. Introduction

SQL injection attacks can be performed in many ways like modifying the data, query manipulation, data extraction etc. Through the execution of altered SQL query, the attackers can get unauthorized access to the application, steal sensitive identity information. Web Application Firewalls (WAF), such as Apache ModSecurity, offer some level of protection. However, they require lots of complex configuration and high processing time in order to detect malicious traffic. As SQL injection vectors can be formed in numerous ways, attacks can be specifically devised from cleverly crafted injection vectors to bypass detection techniques [5–7].

To address the above issues, we propose a signature based SQL Injection attack detection framework, integrating fingerprinting method and Pattern Matching (PM) to distinguish genuine SQL queries from malicious queries. Pattern Matching (PM) has been employed in Snort to detecting SQLIA as well as other intrusion detection/prevention IDs/IPs systems [18, 19, 21], which seeks the occurrence of pattern P in text T. PM based (IDs/IPs) treat intrusion identification as pattern search process by matching malicious traffics against all telltales in the attack signature database. In PM operations the PM-engines determines the performance of the system, thus PM-engines operation demands high system processing time [12]. PM algorithms routinely used by PM engines such as Boyer Moore [13] and Aho Corasick [10] are computationally intensive and have a shortfall of generating false positives. In our framework we use Aho-Corasick algorithm. The complexity of implementing the Boyer-Moore algorithm has been cited as one of the reasons why it has not been used more and also for high input Aho-Corasick algorithm out performs Boyer Moore PM algorithm when input data is much high [18]. Our fundamental design goals aim at reducing high processing time of the PM engines, and reducing the false positive rate achieve an insignificant (zero) false negative rate. We subjects SQL queries to a fingerpringting process before lunching PM engine. SQL queries and attack signature patterns are represented with fingerprints. Initially incoming SQL queries fingerprints are matched against digest of signatures instead of the actual signatures, queries receives a negative verdict and is declared as legitimate if it queries fails to match any fingerprints from the signature database, hence no Pattern Matching processing is needed. Our framework quickly identify attack queries using the Rabin fingerprint method, the pattern matching algorithm Aho-Corasick is invoked to perform an exact pattern match. The Rabin fingerprint techniques has been successfully used to detect document similarity and established plagiarism [17,21] and they have also been adopted in Pattern Matching process in IDs/IPs [21]. Our proposed framework is designed to work between the application and database data server, therefore multiple web applications hosted on a shared web server can be protect.

The paper is organized as follows. Section II provides the background information of SQLIA. Details of the framework is given in Section III. The experimental results and performance evaluation is given in Section IV. Section V outlines related research on SQLIA. Finally in Section VI, we provide concluding remarks and present areas of future directions of research.

II. Background of SQL Injection Attack

To be able to counter the problem of SQLIA, we need to explore several ways these attacks are launched. SQL injection is a very simple attack technique but can result in devastating damage to the database of the web application. In this section, we will explain SQL injection mechanism and typical attacks.

978-1-5386-0497-7/17/$31.00 ©2017 IEEE

A. First-order Injection

The attackers inject SQL statements by providing crafted user input via HTTP GET or POST, cookies, or a collection of server variables that contain HTTP, network headers, and other environmental parameters. For example, the UNIONS command is added to an existing statement to execute a second statement, a subquery is added to an existing statement, or an query condition such as "OR 1=1" is added to bring back all the data from a table.

B. Second-order injection

The attackers inject SQL statements into persistent storage (such as a table record) which is considered as a trusted source but would indirectly trigger an attack when that input is used at a later time. For example, an attacker registers on a website by using a seeded user name, such as "'admin' – ". Assume that the Web application failed to validate the input before storing it in the database. The attacker later modifies his or her password by using the following SQL statement: *UPDATE tblname SET password = "' + newPassword +"' WHERE username = "" + userName +"' AND password = "' + oldPassword + "'"*. In this case the name of the attacker currently logged-in is "admin'–'", and the above SQL statement will then be read as: UPDATE users SET password='newpassword' WHERE userName= 'admin'–' AND password='oldpassword'. Since the "–" is the SQL comment operator, everything after it will be ignored by the SQL engine. Consequently, the attacker's SQL statement will change the administrator's username ("admin") to an attacker-specified value.

C. Illegal/Logically Incorrect Queries

An attacker gathers important information about the type and structure of the back-end database of a Web application by injecting illegal or logically incorrect SQL syntax which will make the application return default error pages that often reveal vulnerable/injectable parameters to the attacker. This attack is considered as a preliminary, information-gathering step for other SQL injection attacks. For example: Probing column name, Input (username): 'ddd". Sql: *SELECT * FROM students WHERE username = "ddd"' AND password =*. Result: "Incorrect syntax near 'ddd'. Unclosed quotation mark after the character string " AND Password="aaa".

D. Tautologies

An attacker injects a query that always evaluates to true for entries in the database to bypass authentication, identify injectable parameters, or extract data. For example: Input (username): *jdoe' or '1'='1'–*. Sql: *SELECT * FROM students WHERE username = 'jdoe' or '1'='1' – AND password =*. Result: All students are retrieved.

E. Union Query

An attacker injects an UNION SELECT to trick the application into returning data from a table different from the one that was intended. Here is a common form using a single quote for this attack: normal SQL statement + "semi-colon" + UNION SELECT <rest of injected query>.

F. PiggyBacked Queries

An attacker injects additional queries into the original query to extract data, add or modify data, perform denial of service, or execute remote commands. In this scenario, the attacker does not intend to modify the original intended query but to include new queries that piggy-back on the original query. As a result, the DBMS receives multiple SQL queries. The first is the normal query which is executed normally, while the subsequent ones are executed to satisfy the attack. Here is a common form using a query delimiter (;) for this attack: normal SQL statement + ";" + INSERT (or UPDATE, DELETE, DROP) <rest of injected query>.

G. Stored Procedures

When a normal SQL statement (i.e., SELECT) is created as a stored procedure, an attacker can inject another stored procedure as a replacement for a normal stored procedure to performing privilege escalation, create denial of service, or execute remote commands. Here is a common form using a query delimiter (;) and the "SHUTDOWN" store procedure for this attack: normal SQL statement + "; SHUTDOWN; " <rest of injected query>.

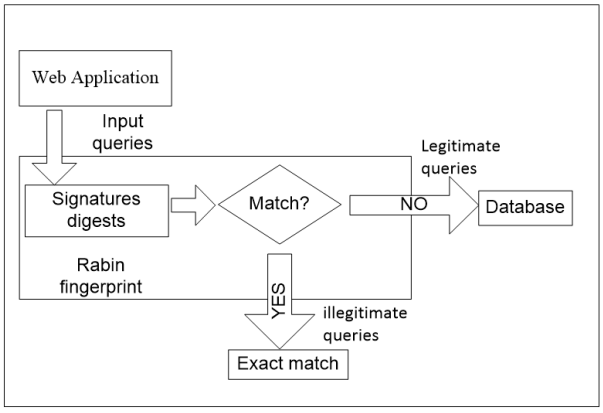

Fig. 1. SQL Injection Attack Detection Framework

III. Outline of SQLIA Detection Framework

SQL query is basically a string consisting of identifiers, keywords, operators, literal values and other symbols; referred to as tokens. Intuitively, the way these tokens are arranged in the query can provide insight to identify malicious ones. Detecting SQL injection attack consist of determining whether the query is injected or genuine at runtime. The basic idea is to quickly examine tokens and probabilistically determine if any of the tokens matches with patterns in the attack signatures before forwarding to the database server as shown in Fig.1. The core ideas behind our approach in detecting and preventing SQL injection attack are; 1) modeling the WHERE clause portion of the query as a token interaction network, 2) fingerprint computation and initial matching. The fingerprint operation is done in two phases - programming phase and query phase this to produce an efficient SQL injection attack

detection framework. During the programing phase repository for attack signatures are created to help make quick decision as far as the existance of exploits in the input query is concerned. While in the query phase our framework deduce the legitimacy of a query if there are no matches with degest found in the attack signature repository. From Fig.1, If matching yields a "YES" our framework proceeds with the exact PM Aho-Corasick algorithm to make the final verdict.

A. Framework design rationale

To avoid high computation of PM engines for reaching negative verdicts, our framework subjects incoming SQL queries to a fingerprinting process before invoking the exact pattern matching operation. Fingerprints are short and compact tags that represent raw data and documents this possible to shift the operations done on the original data to processes taking place on concise signatures [20]. Fingerprints are usually generated by a digesting function defined as character string P is a bit string containing m bits $[b_1, ..., b_m]$. It is then associated to a polynomial of degree $(m-1)$ in indeterminate t as $A(t) = b_1 t^{m-1} + b_2 t^{m-2} + ... + b_{m-1} t + b_m$. Given a polynomial $P(t)$ of degree k, $P(t) = a_1 t^k + a_2 t^{k-1} + ... + a_k + a_k$. The residual $f(t) = A(t) mod P(t)$ will be of degree $(k-1)$. For bit strings all coefficient of $A(t)$ are in Z , thus $P(t)$ is chosen using a_i's in Z, the fingerprint function is defined as $H(t) = A(t) mod P(t)$. Rabin fingerprint represents both query and attack signatures with their fingerprints, so that it can operate on them at a reduced space of fingerprints to detect a match and as long as fingerprints objects collides with low probability the performance of the system enhanced [20,15]. In this regard, our framework transforms its input text, either a query or attack signatures into tokens. Kar et. al. demonstrated that it is sufficient to examine only the WHERE clause portion of a query for detecting SQLIA [8]. Since a query may have multiple WHERE clauses (e.g., UNION queries), the portion from the token following the first WHERE keyword up to the end of the query is consider in order to reduce computational overhead. The incoming SQL queries WHERE clause portion is sanitized and after splitting into tokens. We incorporate the sanitizing enhancements to the transformation scheme due to attackers employing this techniques to bypass detection. The sanitizing operation are performed by:

- removing back-quotes (‘) double-quotes (“ ”) backslash (/) if any because these do not contributes towards the structure from query
- removing newline, carriage return, tabs and other special characters
- replacing characters with their ASCII code in hexadecimal form proceeded by the percentage (%) character.
- converting all characters into lower cases.

To visualize the sanitizing and splitting process with these enhancements, consider the following query with a tautological injection vector crafted for bypassing detection: *SELECT * FROM students WHERE username = ‘jdoe’ or ‘1’=‘1’ AND password =* . Sanitizing and splitting process converts query into the following sequence of x tokens $\{x_0 = jdoe,\ x_1 = or,\ x_2 = 1\}$. The sequence of tokens is used for input to the matching process.

B. Programming and querying phase

In the *programming phase*, attack signatures are fingerprinted and represented with short digests and stored in fingerprint table. Given that $Y = y_0, y_1, y_2, ..., y_m$, s.t $y_j \in [0, 1]$, Y as the signature and m as the signature length. We presents each string as a binary representation of integers. Thus, the binary representation of string Y is expressed as

$$H(Y) = \sum_{j=1}^{m} y_j 2^{m-j} \tag{1}$$

Let M be a positive integer. Defined S ={t | t is prime and t $\leq$ M} and $\emptyset_t(Y) = H_t(Y)$ for all Y. For any integer t, the fingerprint function of Y is represented as

$$H_t(Y) = H(Y) mod(t). \tag{2}$$

The programming phase is performed once and gets updated only when changes in the signature database occur.

In the *query phase*, incoming query fingerprint is computed and its digest are matched against the digests of attack signatures created during the programming phase. Given $X = x_0, x_1, x_2, ..., x_n$, s.t $x_i \in [0, 1]$. X is the token and n is the token length, the binary representation of string X is expressed as

$$H(X) = \sum_{i=1}^{n} x_i 2^{n-i} \tag{3}$$

fingerprint function of X is represented as

$$H_t(X) = H(X) mod(t) \tag{4}$$

For any n objects X, the number of distinct fingerprint values $H_t(x_i), i = 0...n-1$ is equal to n with high probability; at the same time for any two x_i and x_j $(i \neq j)$, it should be highly unlikely that $H_t(x_i) = H_t(x_j)$. For n randomly chosen strings with degree m, the probability for a Rabin fingerprint collision of two distinct strings is less than $(nm^2/2^k)$. Hence, we controlled collision rate by manipulating parameters k, m and n [20]. We reduce computational complexity in our framework by setting the base b to two so that multiplications in Rabin fingerprint calculation can be replaced with left-shift operations. Matching $H_t(X)$ with $H_t(Y)$ receives a negative verdict and is declared as legitimate, if it fails to match any fingerprint from the attack signature database; thus, no further pattern matching processing is required. For a match yielding a positive identification, our model executes the Aho-Corasick algorithm to ascertain that the token indeed matches a patterns in the attack signature whose fingerprint resulted in the preliminary match.

IV. Experimental Evaluation

In this section we will implement an experiment to evaluate our proposal, we performed our experiment on laptop equipped with Intel 2.3G CPU, 4G physical memory, Windows 8. The evaluation of our framework focused on two aspects (i) the accuracy of our framework such as the number of false negative, false positive, i.e. the unnecessary queries forwarded to the database, and the detection rate for each attack. (ii) the performance of our framework at runtime. We evaluate the performance of each component, following the procedure (1) user query is fed into the PM at runtime, (2) the above step is repeated n-times($n = 100$) and the best processing time achieved within the n cycles is obtained and (3) the average processing time is computed as it best processing time. Our framework components consist of: (a) extracting the WHERE clause, (b) sanitizing and splitting WHERE clause portion into tokens, (c) preliminary matching that is fngerprinting with Rabin technique (RF) and (d) integrating fingerprint technique and pattern matching method. The average processing time for (a) is negligible, therefore ignored.

A. Preparation of Dataset

In order to obtain our dataset SQL Injection attacks were lunched on web applications using several attacked tools download from the internet, such as WebInspect, Havij, HP Scrawler, WebCruser Pro, etc. The tools were applied multiple times with all possible settings so as to capture widest varieties of SQL injection attacks. Altogether, 1134 queries collected. The injected and genuine queries were collected and their WHERE clause portion of the SQL statement were taken into consideration. After removing duplicates, a dataset consisting of 527 injected tokens and 343 genuine tokens were prepared. The distribution of the injected tokens is presented in Table III column 3 with the maximum of 86 recorded for First-order Injection and a minimum of 57 recorded for Tautologies.

TABLE I
Confusion Matrix

Test	Genuine	Injected	Total
Genuine	305 TN	6 FP	311 N
Injected	10 FN	425 TP	435 P
Total	315	431	746

TABLE II
Predictive Performance of our Framework

Metric	Formula	Result
Precision	TP/(TP + FP)	98.53%
Recall	TP/(TP + FN)	97.64%
Fallout (FPR)	FP/(FP + TN)	0.18%
Specificity (TNR)	TN/(FP + TN)	99.67%
Accuracy	(TP + TN)/(P + N)	98.52%
F1-Score	2TP/(2TP + FP + FN)	98.15%

B. Experimental Results

The predictive performance of the system are calculated based on the values in the Confusion matrix in Table I.

TABLE III
Detection rate of attack types

SQL Attacks	Detection Rate	Number of injected pattern
Second-order injection	98.32 %	73
Illegal/Logically Incorrect Queries	97.60%	84
Tautologies	98.53%	57
Union Query	100%	63
PiggyBacked Queries	96.41%	89
Stored Procedures	96.73%	75
First-order Injection	97.92%	86

TABLE IV
Component Processing Time per query in microseconds

Components	Injected	Genuine
Sanitizing and Splitting	0.5214 ms	0.1834 ms
Fingerprinting	10.5730 ms	4.7640 ms
Final Detection	39.8543 ms	———
Total	50.9487 ms	4.9474 ms

The results in Table II show that the false matches are reduced at a very considerable level. Also recorded a high detection rate with very good accuracy low false positive and low false negative. From Table III, all Union injection were detected, followed by Tautologies and Second-order injection which achieved 98.53% and 98.32% at 57 and 73 tokens respectively. Detection rate for Stored procedures and PiggyBacked injection were the lowest among the seven attacks. For PM scenario, the combination of short-length patterns and diverse attack types may produce false matches. However, our framework achieved high efficiency as a result of the sanitizing technique and integrating of Rabin fingerprint as a preliminary match at the early stage before pattern matching do the exact match.

C. Impact on Performance

From Table IV column 2, the processing time is high because of the presences of malicious string, which makes this queries longer than the genuine queries, however, for genuine strings the processing time is much lower. The performance of the Rabin fingerprint method is heavily affected by the size of it's fingerprint table (which is derived from the signature table) as the latter determines the fingerprint collisions probability. For user executing multiple queries, each query will be delayed by $\times 51ms$ over each page load. As a page load times on the Internet are in order of multiple seconds, this will not impact on users experience. A trade-off exist between processing rate and memory consumption in our framework. Injected queries detection could be improved with a large signature database that reduces fingerprint collisions at the cost of increasing memory. Logically reducing the size of the signature database

will result in low processing time and reduction in memory consumption, but this will come at cost of efficiency.

V. Related Work

Web applications are not only vulnerable to SQL injection rather any code that accept inputs from untrusted source and then uses that input to form a dynamic SQL statement [16]. For an attacker being able to influence what is passed to the database, the attacker can leverage the syntax and capabilities of SQL itself, as well as the power and flexibility of supporting database functionality and operating system functionality available to the database.

Current research on SQLIA include dynamic taint analysis [4], distinguishing the SQL statements generated by legal inputs from those by attack inputs. Dynamic taint analysis instruments suspicious inputs at run time, when the input is propagated to a SQL relevant statement, an alert is raised. Wang and Li proposed program tracing along with parsetree and string similarity to train a support vector machine (SVM) classifier to detect malicious queries at run-time [10]. Buehrer et al. developed SqlGuard which detects anomalies by comparing the parse-tree of SQL queries before and after inclusion of user inputs [11].

Wasserman [1] presented a static approach using string analysis to approximate the language of each SQL string used by SQL-relevant methods, if the intersection of the approximated string language and the string language used by existing attacks is empty, then the program is safe.Jovanovic [2] and Qiang [3] use taint analysis to track suspicious inputs, if the invalidated input is propagated to a SQL-relevant statement in an executable path, then a SQLIA vulnerability is found.

VI. Conclusion

For an attacker being able to influence what is passed to the database, the attacker can leverage the syntax and capabilities of SQL itself, as well as the power and flexibility of supporting database functionality and operating system functionality available to the database. Pattern Matching techniques such as Boyer Moore and Aho Corasick algorithm has been employed in detection attacks before they are being passed to the database. The latter are more computationally intensive and produces much false positive and false negatives as attackers are easily ably to circumvent. This paper presented a signature based SQLIA detection framework integrating fingerprinting method and Pattern Matching to distinguish genuine SQL queries from malicious queries. The fingerprint method accelerate the attack detection rate. The Rabin fingerprint monitors SQL queries to the SQL database and compare them against a dataset of signatures before the PM method is invoke. The experimental results confirm that the approach is effective in detecting all types of SQLIA with low false positive rate. Because of the difficulty identifying unknown attack using signature-based detection systems our future work will focus on anomaly-based system that learns the profiles of the normal database access performed by web-based applications using a number of different models to detect unknow attacks.

Acknowledgment

The authors would like to thank the Joint Funds of the National Science Foundation of China (Grant No. 61520106007), supporting the Regional and International Research Projects.

References

[1] G. Wasserman and Z. Su, “Sound and Precise Analysis of Web Applications for Injection Vulnerabilities,” In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2007.

[2] N. Jovanovic, C. Kruegel, and E. Kirda, “Static Analysis for Detecting Taint-style Vulnerabilities in Web Applications,” Journal of Computer Security. Vol 18, N5, August 2010.

[3] H. Qiang, Z. Qing-Kai, “Taint Propagation Analysis and Dynamic Verification with Information Flow Policy,” Journal of Software, n22(9):2036-2048, 2011.

[4] W. Yi, L. Zhoujun, G. Tao, “Literal Tainting Method for Preventing Code Injection Attack in Web Application,” Journal of Computer Research and Development, 49(11): 2414–2423, 2012.

[5] O. Maor and A. Shulman, “SQL Injection Signatures Evasion (Whitepaper),” Imperva Inc, 04 2004.

[6] J. Dahse, “Exploiting hard filtered SQL Injections (Whitepaper),” 03, 2010.

[7] P. Luptk, “Bypassing Web Application Firewalls,” in Proceedings of 6th International Scientific Conference on Security and Protection of Information, pp. 79-88, Czech Republic, 2011.

[8] D. Kar, S. Panigrahi, and S. Sundararajan, “SQLiDDS: SQL Injection Detection Using Query Transformation and Document Similarity,” In Distributed Computing and Internet Technology, pp. 377-390, 2015.

[9] A. Aho and M. Corasick, “Efficient string matching: An aid to bibliographic search,” Communications of the ACM, vol. 18, no. 6, pp. 333-343, June 1975.

[10] Y. Wang and Z. Li, “SQL Injection Detection via Program Tracing and Machine Learning,” in Internet and Distributed Computing Systems, pp. 264274, Springer, 2012.

[11] G. Buehrer, B. Weide, and P. Sivilotti, “Using Parse Tree Validation to Prevent SQL Injection Attacks,” in Proc. of the 5th Int’l Workshop on Software Engineering and Middleware, pp. 106-113, ACM, 2005.

[12] M. Fisk, and G. Varghese, “An Analysis of Fast String Matching Applied to Content-Based Forwarding and Intrusion Detection” Technical Report, CS2001-0670. Univ of California, San Diego. 2002.

[13] R. Boyer, and J. Moore, “A fast string searching algorithm,” Communi. ACM, 20, 762-772, 1977.

[14] C.J. Coit, S. Staniford, and J. McAlerney, “Towards Faster Pattern Matching for Intrusion Detection, or Exceeding the Speed of Snort”. Proc. 2nd DARPA Information Survivability Conf. Exposition (DISCEX II), pp. 367-373,Anaheim, CA, USA, June 2001.

[15] S. Pankanti, S. Prabhakar, and A. Jain, “On the Individuality of Fingerprints,” IEEE Trans. Pattern Anal. Intell., 24, 10101025. Mach 2002.

[16] Justin Clarke, “SQL Injection Attack and Defence (Second Edition)” 2012.

[17] D. Fetterly, M. Manasse, and M. Najork, ”On the Evolution of clusters of Near-Duplicate Web Pages,” IEEE Computer Society, Proc. 1st Latin American Web Cong, Santiago, Chile, pp. 37-45. November 2003.

[18] M. Fisk, G. Varghese, “Applying fast string matching to intrusion detection” Los Alamos National Lab Report, 2002.

[19] Martin Roesch, “Snort - lightweight intrusion detection for networks” in Proc. of the 13th Systems Administration Conference, USENIX, 1999.

[20] M. O. Rabin, “Fingerprinting by random polynomials,” Dept. Math., Hebrew Univ. Jerusalem, Jerusalem, Israel, Tech. Rep. TR-15-81, 1981.

[21] R. Ramaswamy, L. Kencl, and G. Iannaccone, “Approximate Fingerprinting to Accelerate Pattern Matching,” Proc. 6th ACM SIGCOMM on Internet Measurement Conf. (IMC06), Rio de Janeiro, Brazil, pp. 301-306. ACM Press. October 2006