Detecting SQL Injection Attacks based on Text Analysis

Lu Yu *, Senlin Luo, Limin Pan

School of Information and Electronics, Beijing Institute of Technology, Beijing 100081, China.

* ylmmyy@163.com

Abstract. SQL injection attacks have been a major security threat to web applications for many years. Detection of SQL injection attacks has been a great challenge to researchers due to its diversity and complexity. In this paper, we present a novel approach to detect SQL injection attacks based on text analysis. We utilize query tokenization to express information of each SQL query, then we use Skip-gram model in Word2vec to generate word embedding expressing eigenvectors for each query, and finally we train an SVM classifier with eigenvectors to identify malicious queries. Experimental results confirm the effectiveness of our approach to all types of SQL injection attacks, especially tautological attacks, with good accuracy and negligible performance overhead. The approach does not require access to the source code, moreover, it can be easily implemented on other platforms with minimal changes.

Keywords: SQL; Skip-gram model; SQL injection; Word2vec.

1. Introduction

SQL injection attacks (SQLIA) have been the greatest security risk to web applications from 2013 to 2017 in WOASP TOP 10[1], due to the simple attack methods and serious consequences. SQLIA is one of the unverified input vulnerabilities, attackers attempt to insert other keywords or operators into the parameters in URLs, web forms to change the syntax, semantics, logic and behaviors of the original SQL query, to arrive a malicious intent such as get data or drop the database without permission. There are several types of SQLIA depending on different attack intentions and techniques used. The classic example of "OR 1=1", known as a tautological attack which means the value keeps true all the time, is detected more difficult.

To protect against SQLIA, Web Application Firewalls (WAF) and Intrusion Detection Systems (IDS) are used, but they are still penetrable. Most of them rely on regular expression-based filters created from known attack signatures that are easy to bypass. For example, the following injection statements are inserted into the parameters by an attacker:

OR 'AB' = 'AB'

OR 'ABCD' = ConCat('A','B','C','D')

OR 'ABCD' = SubsTRing (conCAT('ABC','DE','FG'),1,4)

All of these are tautological attacks due to the value of each is true always. It's extremely difficult to construct regular expression or rule-based filters for so many malicious expressions, which are well-constructed by attackers to bypass the filters.

In this paper, we have proposed a lightweight approach to prevent SQLIA by employing word embedding and SVM. We transform a SQL query into a sequence of tokens, then use Word2vec to generate word embedding of these tokens and train an SVM classifier, finally use the classifier to identify the malicious code. This method was implemented in a prototype named SQLIWE (SQL injection Detection using word embedding). Our experimental results show that SQLIWE is effective to prevent a variety of SQLIA compared to other state-of-the-art techniques.

The rest of the paper is organized as follows: Section 2 reviews the related works about research in this area. We detail our approach in Section 3 and present the experimental results, discussion in Section 4. Finally, we conclude the paper with a note on future directions of research in Section 5.

2. Related Works

In order to prevent SQLIA many techniques have been proposed by a number of researchers. Based on the approaches used in the literature, we emphasize 4 categories as follows.

Defensive coding is a first-hand protection against SQLIA. It requires programmers to write the application code in a specific manner. McClure et al. [2] designed SQL-DOM, which can generate safe SQL statements using a set of classes strongly typed to the database schema. The approach must be reconstructed when the database schema changes and the time complexity is very high. Gil et al. [3] used C++ templates to produce safe SQL queries in applications. The process of these approaches is always complicated.

Code analysis compares the application's behavior during normal-use with real behavior executed at runtime, an alarm will occur if they don't match. Halfond et al. [4] designed AMNESIA and Bisht et al. [[5]] proposed CANDID, both of the two approaches are combining static analysis of source code and runtime monitoring. Code analysis generally requires access to the source code, once changes to the application the model need to be rebuilt. Further, this method is suitable for a specific database platform and language.

Taint-based approaches have been used to track data flow during execution of a program. Halfond et al. [[6]] proposed positive tainting with syntax aware evaluation to identify SQLIA. Papagiannis et al. [7] used taint-based approach to prevent applications in PHP from injection attacks.

Fingerprinting and learning based methods generally exploit some machine learning algorithms to detect SQLIA. These methods train a model using collected SQL queries and then use the trained model to identify the abnormal query. Choi et al. [8] used N-Gram analysis for feature selection and SVM to further classify.

3. Proposed Approach

The overall process of SQLIWE is depicted in Fig. 1. The core of our approach consists of 4 phases. (1) Transform a SQL query into a sequence of tokens presenting its semantics, syntax and structural composition. (2) Use Word2vec to generate word embedding for each token and connect them to an eigenvector of the corresponding SQL query. (3) Train an SVM classifier using eigenvectors in phases 2. (4) Use the classifier to identify malicious queries at runtime. The rest of this section describes each phase in detail.

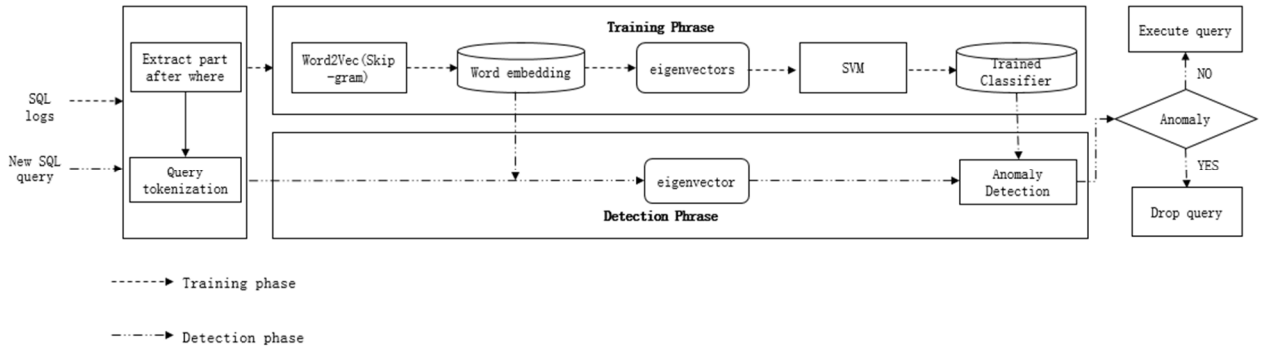

Figure 1. Principle diagram of the SQLIWE

3.1 Query Tokenization

Kar et al. [9] proposed that the malicious code of SQLIA only exists in the statement after keyword "where", so the statement after keyword "where" is extracted for detection. Each SQL query can be regarded as a sentence consisting of keywords, operators, and other symbols, so we can transform a SQL query into a sentence which can express semantics, syntax and structural composition of the corresponding SQL query. They also proposed a transformation scheme splitting the query with spaces and using the uppercase letter A-Z to identify the keywords, operators, values, etc. The detail scheme is in Tab. 1 and Tab. 2.

Table 1. Transformation scheme

Component	Token	Component	Token
Space characters(\t, \r, \n)	space	System objects	
Anything in double/single quotes		a) System database	SYSDB
a) Hexadecimal	HEX	b) System table	SYSTBL
b) Decimal	DEC	c) System column	SYSCOL
c) Integer	INT	d) System variable	SYSVAR
d) IP address	IPADDR	e) System view	SYSVW
e) Single character	CHR	f) System stored procedure	SYSPROC
f) String	STR	User objects	
Anything outside double/single quotes		a) User database	USRDB
a) Hexadecimal	HEX	b) User table	USRTBL
b) Decimal	DEC	c) User column	USRCOL
c) Integer	INT	d) User view	USRVW
d) IP address	IPADDR	e) User stored procedure	USRPROC
Other symbols and special characters	In Table 2	f) User functions	USRFUNC
SQL keywords, reserved words, functions	To uppercase	Multiple spaces	Single space
The entire query	To uppercase		

Table 2. Transformation for special symbols

Symbol	Token	Symbol	Token	Symbol	Token	Symbol	Token
`	Remove	:	CLN	( )	REMOVE	$	DLLR
! = or <>	NEQ	;	SMCLN	(	LPRN	%	PRCNT
/**/	Remove	"	DOUT	)	RPRN	^	XOR
&&	AND	‘	SOUT	{	LCBR	+	PLUS
||	OR	<	LT	}	RCBR	=	EQ
/*	CMTST	--+ or --	NOTE	[	LSQBR	>	GT
*/	CMTEND	&	BITAND	,	CMMA	]	RSQBR
!	EXCLM	|	BITOR	.	DOT	\	BSLSH
@	ATR	*	STAR	?	QSTN	/	SLSH
#	HASH	-	MINUS				

There are 567 keywords in MySQL 5.5, including 277 system functions and 226 reserved words, all of them are converted to uppercase. There are 40 operators divided into symbols and keywords. The symbols are converted to tokens according to the Tab. 2. The keywords are as the same as the Tab. 1. There are 3 system databases, 81 system tables, 328 system columns and 480 variables in MySQL 5.5, we convert all of them to SYSDB, SYSTBL, SYSCOL, SYSVAR, respectively. Any component not substituted so far needs to be converted to the corresponding token based on its position in the SQL statement and the role it plays in the statement. We should deal with 3 special cases in particular ways:

Attackers generally use /* */ to bypass detection. /* */ is the comment of the SQL statement, in query tokenization process, if /* */ is empty, the purpose is to bypass the filter other than to play the role of annotation, we can remove it to simplify the statement, if /**/ contains something else, we can convert it to CMTST or CMTEND.

plays the role of affiliation in the SQL statement, for example, market. tables represents the tables in market database. In query tokenization process, we can identify the role through the component after. plays, and convert it to corresponding token to reduce the complexity.

We generally name an alias for a table or column to enhance the readability of the statement. In query tokenization process, we can identify the role the alias plays and convert it to corresponding token. For example, SQL statement "select * from (select name, pswd from t1) as t2" can be converted to "SELECT STAR FROM SELECT USRCOL CMMA USRCOL FROM USRTBL AS USRTBL", t2 in the SQL statement is an alias of a table, so we can transform it into USRTBL.

There is a query tokenization process of a typical SQL statement. We transform the SQL statement "select money from account where money = 100 union select count (*) from menu" into the sequence of tokens "USRCOL EQ INT UNION SELECT COUNT STAR FROM USRTBL".

Any SQL query can be transformed into a sequence of tokens which can present its semantics, syntax and structural composition regardless of the length and complexity. A major benefit of query tokenization is reducing the number and complexity of samples, cutting down overhead.

3.2 Word2vec and Word Embedding

Word2vec [10] is a distributed representation method in Natural Language Processing. We can use it to generate word embedding for each word. Word embedding is a term for characterization learning techniques, it maps a word or phrase to a real vector in a low-dimensional space. Word2vec is a three-layer neural network model with input layer, projection layer and output layer, and contains two models, namely CBOW model and Skip-gram model. In our method we use Skip-gram model to generate word embedding. Fig. 2 is the structure of Skip-gram model. The window size is 2, w(t) represents the position of the current word in sentence is t, we can predict w(t-2), w(t-1), w(t+1), w(t+2) according to w(t). The objective function formula of the Skip-gram model is expression 1:

$$L = \sum_{W \in C} logp(Context(w)|w) \quad (1)$$

Context(w) indicates the context of the word, C is the corpus including all sentences which are converted from SQL queries through query tokenization phrase. L is the objective function. We use the gradient descent method to find the maximum value of the objective function and the optimal model parameters.

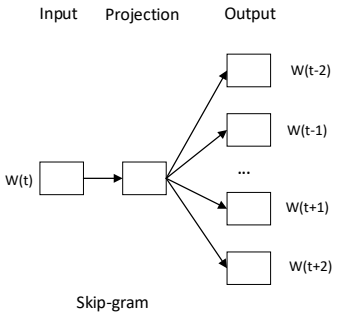

Figure 2. Skip-gram model

Word2vec can represent words as real-value vectors through training a given corpus with the use of the neural networks, and simplify the processing of text content into vector operations in vector space. In SQLIWE, first of all, we use query tokenization method to process normal SQL queries and we get all sequences of normal queries. And then, we use Skip-gram model to train normal sequences and we obtain the word embedding for each word, the length of each word embedding is 4. Next, we concatenate the word embedding for each word of the corresponding sequence to an eigenvector, in this step, we use the longest vector as a standard, and use -1 to pad the vector if the length is smaller than the standard. Finally, we use all of the eigenvectors to train an SVM classifier.

3.3 Support Vector Machine (SVM)

Support vector Machine (SVM) is a supervised machine learning algorithm providing high accuracy while dealing with high-dimensional data. In the most basic sense, SVM is a binary classifier, it can find the optimal hyperplane to divide samples into 2 classes. Given l linearly separable training samples $\{x, y\}$, $x \in R^N$ is the eigenvector, $y \in \{-1, +1\}$ is the label, we can express the optimal hyperplane as follows:

$$f(x) = \sum_{i=1}^{l} y_i a_i k(x, x_i) + b \tag{2}$$

$k(x, x_i)$ is a kernel function. We can use kernel functions to transform a nonlinear problem into a linear separable problem in the feature space. There are 9 kernel functions in SVM, we use Gaussian kernel in the task which is given by:

$$K(x_1, x_2) = exp(-\frac{||x_1 - x_2||^2}{2\sigma^2}) \tag{3}$$

In this phase, we treat injected SQL queries and normal SQL queries as positive samples and negative samples, respectively. We train the SVM classifier with eigenvectors for each sample in phase 2.

3.4 Anomaly Detection

We use the SVM classifier to identify a new coming SQL query whether it is a normal or abnormal. If it is a normal query, we will execute it in database, we will drop it or alarm the administrator otherwise.

4. Experiments and Discussion

The experimental setup consisted of a standard desktop computer with Intel Core-i7 6700 CPU and 8GB RAM, running with windows7 OS and VMware loaded with Kali Linux and Ubuntu14.04. We built a public application system and started logging to get SQL queries. Over 5000 lines were written into the log files which 3042 SQL normal queries were unique with each other. 157493 abnormal SQL queries were collected using sqlmap in Kali Linux to attack the WAVS system.

The performance of SQLIWE was evaluated in terms of FPR (False Positive Rate), FNR (False Negative Rate) and ADT (Average Detection Time). FPR is the ratio of normal queries incorrectly identified by the method (FP) to total negative samples (FP + TN). FNR is the ratio of injected queries falsely identified by the method (FN) to total positive samples (TP + FN). ADT is the average time to detect a new query. The value of FRP and FNR are lower, the accuracy of the model is higher. The ADT is smaller, the efficiency of the model is higher.

Figure 3 shows the result of FPR and FNR when the length of word embedding changed from 1 to 8. The values of FPR and FNR fall with increase of the length, while the values are the smallest with the length is 6. When the length is smaller than 6, the word embedding cannot express enough information of the SQL query, while the length is larger than 6, the model is too complex to improve the accuracy of the method.

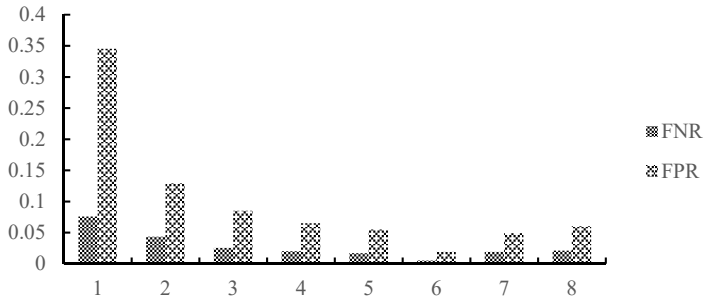

Figure 3. FPR and FNR with different word embedding dimension

Figure 4 show the result of FPR and FNR in different methods. We compare SQLIWE with SQLIGOT [11], SQLIDDS [12] and HMM [13], respectively. Note that our method has the best effect no matter in FPR or FNR. FPR is 0.35% and FNR is 1.9%. We check the final result, there are 12 tautological attacks checked by our method than SQLIGOT. We make sure that SQLIWE has a better performance in detecting tautological attacks than other methods. We use word embedding to express total information of the corresponding SQL query, so we can detect the injected queries effectively.

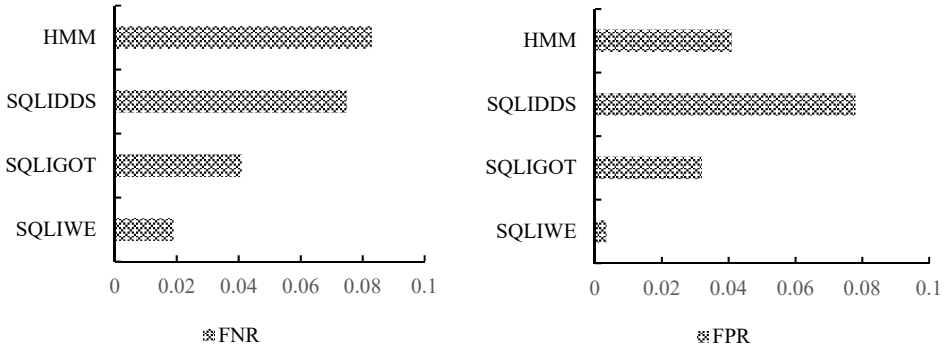

Figure 4. Comparison of FNR and FPR

Table 3 shows the average time of different methods. Obviously, the ADT of SQLIWE is the least due to the use of word embedding. Word embedding was built in training phrase, we can use it directly in detecting phrase, so the time it takes is small. It is clear from the result that our method outperforms other methods in terms of FNR, FPR and ADT.

Table 3. Comparison of ADT

	SQLIWE	SQLIGOT	SQLIDDS	HMM
ADT(ms)	0.89	1.67	9.3	15.8

5. Conclusions and Future Work

This paper presented a novel, efficient approach to detect SQL injection attacks based on text analysis. Firstly, we transformed a SQL query into a sequence of tokens presenting its semantics, syntax and structural composition to reduce the number and complexity of the SQL queries. And then, we used Skip-gram model in Word2vec to generate word embedding expressing eigenvectors for each query. Next, we utilized the eigenvectors to train an SVM classifier. Finally, we used it to identify the malicious code. Experiments confirmed the effectiveness of our approach, especially in detecting the tautological attacks. This approach neither requires access to the source code, nor builds a normal-use model of queries, it is regardless of database platforms and programming languages besides. Future works will be focused on merging the information of the malicious code into query transformation scheme to improve the accuracy of the approach.

References

[1]. Information on https://www.owasp.org/index.php/Top_10_2017-Top_10.

[2]. Mcclure R A. SQL DOM: compile time checking of dynamic SQL statements[C]// International Conference on Software Engineering. IEEE, 2005.

[3]. Gil J., Lenz K. [ACM Press the 6th international conference - Salzburg, Austria (2007.10.01-2007.10.03)] Proceedings of the 6th international conference on Generative programming and component engineering, - GPCE \"07 - Simple and safe SQL queries with c++ templates[J]. 2007:13.

[4]. Halfond W G J, Orso A. AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks[C]// IEEE/ACM International Conference on Automated Software Engineering. DBLP, 2005:174-183.

[5]. Bisht P, Madhusudan P, Venkatakrishnan V. CANDID: dynamic candidate evaluations for automatic prevention of SQL injection attacks. [C]//ACM Trans Inf Syst Secur (TISSEC) .2010b. 13(2):14.

[6]. Halfond W G J, Orso A, Manolios P. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation[J]. IEEE Transactions on Software Engineering, 2008, 34(1)(1):65-81.

[7]. Papagiannis I, Migliavacca M, Pietzuch P. PHP Aspis: Using Partial Taint Tracking to Protect Against Injection Attacks[C]// Proceedings of the 2nd USENIX conference on Web application development. USENIX Association, 2011.

[8]. Choi J, Kim H, Choi C, et al. Efficient Malicious Code Detection Using N-Gram Analysis and SVM[C]// International Conference on Network-based Information Systems. IEEE, 2011.

[9]. Kar D, Panigrahi S. Prevention of SQL Injection attack using query transformation and hashing[C]// 2013 IEEE International Advance Computing Conference. IEEE, 2013:1317-1323.

[10]. Mikolov T, Chen K, Corrado G, et al. Efficient Estimation of Word Representations in Vector Space[J]. Computer Science, 2013.

[11]. Kar D, Panigrahi S, Sundararajan S. SQLiGoT: Detecting SQL injection attacks using graph of tokens and SVM[J]. Computers & Security, 2016, 60:206-225.

[12]. Kar D, Panigrahi S, Sundararajan S. SQLiDDS: SQL Injection Detection Using Query Transformation and Document Similarity[J]. Distributed computing and internet technology. Springer,2015:377-390.

[13]. YANG Lianqun, MENG Kui, WANG Bin, et al. A New Detection Technique of SQL Injection Based on Hidden Markov Mode[J]. Netinfo Security, 2017(9):115-118.