Exploring SQL Injection Classification Using Fuzzing Payloads

Project GitHub Repository: https://github.com/ChrisAHolland/ML-SQL-Injection-Detector

Chris Holland (V00876844)

Abstract

SQL fuzzing payloads are lists of malformed SQL statements that are used to discover vulnerabilities in web applications that use a SQL database. While machine learning and artificial intelligence approaches to defending against SQL injections are often overlooked compared to more power and memory efficient solutions, performing data mining tasks on fuzzing payloads may provide some insight on how to detect whether a SQL query is an attempted injection or not.

For those interested in both cybersecurity and machine learning, trying to find connections between the two is an interesting and rewarding task. The goal of this study was to perform data mining techniques on popular SQL injection fuzzing payload datasets to find what characteristics make them useful for being used in a classification system to detect SQL injections. Fuzzing payloads were chosen to train the classifier because they provide a very readily available dataset of malicious SQL queries that may cause harm to a web application's database. For testing purposes in this study, a customized Naïve Bayes Multinomial text classifier was written using Python that can be trained with any given dataset, test any given dataset, report on the accuracy of the classifications, and provide a report of how each document was classified. The code for this classifier, along with the datasets and results can be found in the GitHub repository linked here, or above. A Jupyter Notebook was also used to create graphs for this report, and can be found in the repository.

The tests in this report revealed that fuzzing payload datasets can definitely be used to create accurate and reliable classifiers for detecting SQL injections. The tests discovered that as the training datasets grow in size, their accuracy in correctly classifying one another converges to nearly 100%. Ultimately a classification system was implemented with test documents being classified correctly with 75% accuracy. After an introductory analysis of the datasets it is apparent that in order to use this approach for creating a SQL injection detector, high amounts of features engineering and preprocessing of very specified datasets would be needed in order to create a system that is 100% accurate and reliable.

Table of Contents

1. Introduction

For multiple years in a row, OWASP has classified injection attacks as the most common and dangerous type of attack that web applications face; with SQL injections being the most common form [1]. Analyzing web applications for SQL injection vulnerabilities usually involves a process known as fuzzing; the process of sending a web applications input points random and malformed data, called fuzz, in hopes of exposing some sort of weakness. The payloads used in fuzz testing are created in such a way to simulate small SQL injections, and tries to uncover how the web application reacts to potentially malicious input.

It has long been established that the best way to secure a web application from SQL injections are the following four options:

1. Prepared Statements
2. Stored Procedures
3. Whitelisted Input
4. Escaping User Input

These techniques are largely favored over machine learning or artificial intelligence solutions because they require far less computing power and memory. However, data mining and machine learning approaches can still be very useful for detecting and preventing SQL injections, and may even be useful for attackers who wish to attack a system. Fuzzing is a technique that can be used both by security analysts to find flaws in a system and fix them, or by an attacker for the same reason, except for exploitation. In this study, fuzzing payloads from popular security testing tools, used by both security analysts and hackers, will be analyzed to determine how data mining techniques can prove useful while trying to find, prevent, and exploit SQL injection vulnerabilities.

2. Dataset Collection

In this study, SQL injection fuzzing payloads from popular fuzzing tools and resources are analyzed, including:

- Kali Linux Burp Suite
- OWASP JBroFuzz
- FuzzDB

Each of these fuzzing payloads was taken from their respective repository on GitHub, however, needed large amounts of preparation in order to be used in the data mining techniques. See Table 1 for the head (first five entries) from each dataset to understand the type of data dealt with in this study. Note that overall the datasets will be much more similar than Table 1 suggests, however very different in size.

Table 1: Head of Fuzzing Datasets

Kali Linux Burp Suite [2] Total Entries: 650	OWASP JBroFuzz [3] Total Entries: 167	FuzzDB [4] Total Entries: 331

`OR 1=1`
`OR 1=0`
`OR x=x`
`OR x=y`
`OR 1=1#`
	`a`
	`a'`
	`a'--`
	`a'or 1=1; --`
	`@`
		`Sleep(__TIME__)#`
		`1 or sleep(__TIME__)#`
		`" or sleep(__TIME__)#`
		`' or sleep(__TIME__)#`
		`" or sleep(__TIME__)=#`

As seen in Table 1, the fuzzing data is comprised of syntactically incorrect SQL statements that may cause errors or exploits in vulnerable databases. Initially, the datasets did not contain any class attributes and were simply collections of documents (each document being a statement of fuzz input). However, the data mining experiments performed on these datasets required special preparation, pre-processing, and feature engineering that will be explained later in this report.

3. Initial Comparisons of the Datasets

The first experiment was a test to find the level of similarity between all three of the datasets. For this method of analysis, the label for each document was either 1 for dangerous, or 0 for safe; representing whether the documents had the potential to harm a web application/database. Each document in the datasets were initially labelled as dangerous since all of the fuzzing information is designed in such a way as to try and cause abnormalities in the system. Using a customized implementation of Naïve Bayes Multinomial Model for text classification, all the datasets were used to train a model and then all of the datasets were used as test sets to find the accuracy using each dataset as the training model. The results are represented in Table 2.

Table 2: Initial Naive Bayes Accuracy Between Datasets

		Training Set		
		Kali Linux Burp Suite Total Entries: 650	OWASP JBroFuzz Total Entries: 167	FuzzDB Total Entries: 331
Test Set	Kali Linux Burp Suite	99.84%	99.84%	99.84%
	OWASP JBroFuzz	99.40%	99.40%	99.40%
	FuzzDB	99.69%	99.69%	99.69%

As seen in Table 2, all of the datasets have a very high accuracy when trained and tested with one another, implying their similarity and therefore effectiveness as fuzzing payloads. However, an interesting pattern is also present. No matter which dataset was used for training the model, the testing datasets were all classified with the same accuracy. Interestingly, the accuracy rates were higher for datasets that contained more documents, however only within fractions of a percentage. This may imply two conclusions; (a) the datasets are equal in their abilities to generate Naïve Bayes Multinomial training models, and (b) the larger that dataset, the more likely it is to be classified correctly.

3.1 Classifying Resized Datasets

In order to test hypothesis (b), the smaller datasets were duplicated to be closer in size to the largest dataset (the Kali Linux Burp Suite payload). The Owasp JBroFuzz and FuzzDB payloads were multiplied three and two times respectively to create datasets that were more similar in size to see if size has any effect on the Naïve Bayes Multinomial text classification accuracy for our datasets. The contents of the datasets remained in equal proportions to ensure that individual documents were not causing a sway in the classifications. The Naïve Bayes Multinomial text classification accuracy with the resized datasets can be seen in Table 3.

Table 3: Naive Bayes Accuracies After Dataset Resizing

		Training Set		
		Kali Linux Burp Suite Total Entries: 650	OWASP JBroFuzz Total Entries: 668	FuzzDB Total Entries: 662
Test Set	Kali Linux Burp Suite	99.84%	99.84%	99.84%
	OWASP JBroFuzz	99.85%	99.85%	99.85%
	FuzzDB	99.84%	99.84%	99.84%

As seen above in Table 3, the resizing of the datasets to make them more similar in size had an impact on the accuracy in which the Naïve Bayes Multinomial text classifier classified the documents. The OWASP JBroFuzz dataset saw an accuracy increase of 0.45% while the FuzzDB dataset saw an accuracy increase of 0.15%. The increase in the size of the datasets had no effect on the pattern witnessed in the initial classification of the datasets; the fact of datasets being classified to the same accuracy no matter which dataset is used to train the model. The increase in accuracy may not seem large or significant, however the growth is interesting and seems much more significant when represented as a graph as seen in Figure 1.

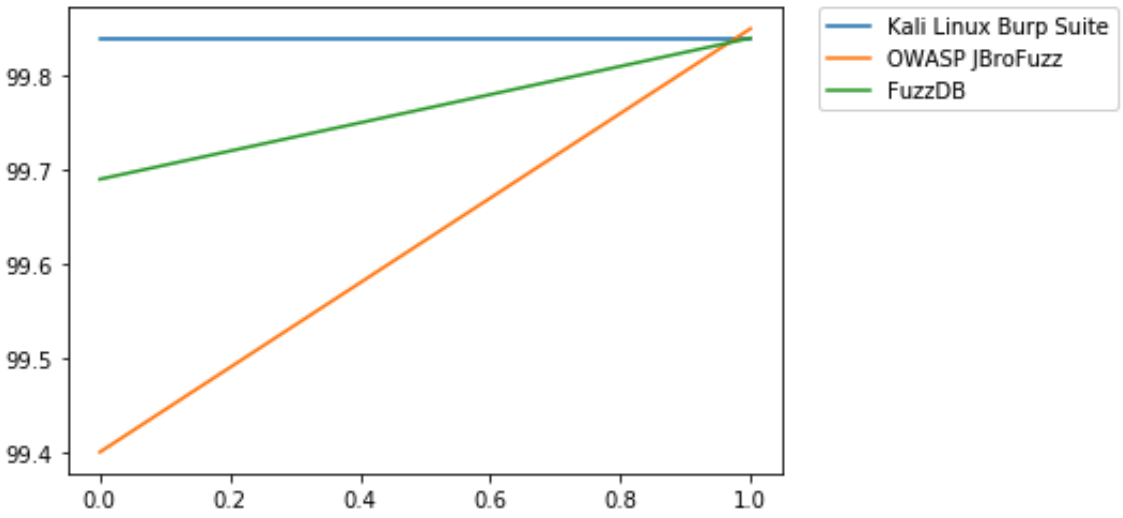

Figure 1: Naive Bayes Accuracies Convergence After First Resizing

3.2 Second Classification of Resized Datasets

After seeing an increase in classification accuracy after resizing the OWASP JBroFuzz and FuzzDB datasets in order to be more similar in size with respect to the Kali Linux Burp Suite dataset, it seemed necessary to once again resize the data sets, this time including the Kali Linux Burp Suite dataset. For this experiment, all of the datasets were doubled in size. The Naïve Bayes Multinomial text classification accuracy with the resized datasets can be seen in Table 4; please note that an additional significant digit was added to represent the miniscule differences between the accuracy.

Table 4: Naive Bayes Accuracies After Doubling Datasets

		Training Set		
		Kali Linux Burp Suite Total Entries: 1300	OWASP JBroFuzz Total Entries: 1336	FuzzDB Total Entries: 1324
Test Set	Kali Linux Burp Suite	99.923%	99.923%	99.923%
	OWASP JBroFuzz	99.925%	99.925%	99.925%
	FuzzDB	99.924%	99.924%	99.924%

With the datasets reengineered to a larger size, the Naïve Bayes Multinomial text classification accuracy seems to be optimal, and for remaining experimentation the reengineered datasets will be used. Figure 2 provides a representation of how the accuracies converge as the amount of documents in the datasets grow in size. Figure 2 contains data that is an extension of Figure 1, and provides some unique insights regarding the increase in classifier accuracy for the datasets. Take special note of how Figure 2 compares to Figure 1; in Figure 2 after point x = 1.00, the accuracies for all three datasets increase in an almost perfectly uniform linear way. This convergence highly implies that the reengineered size of the datasets is optimal for Naïve Bayes Multinomial text classification.

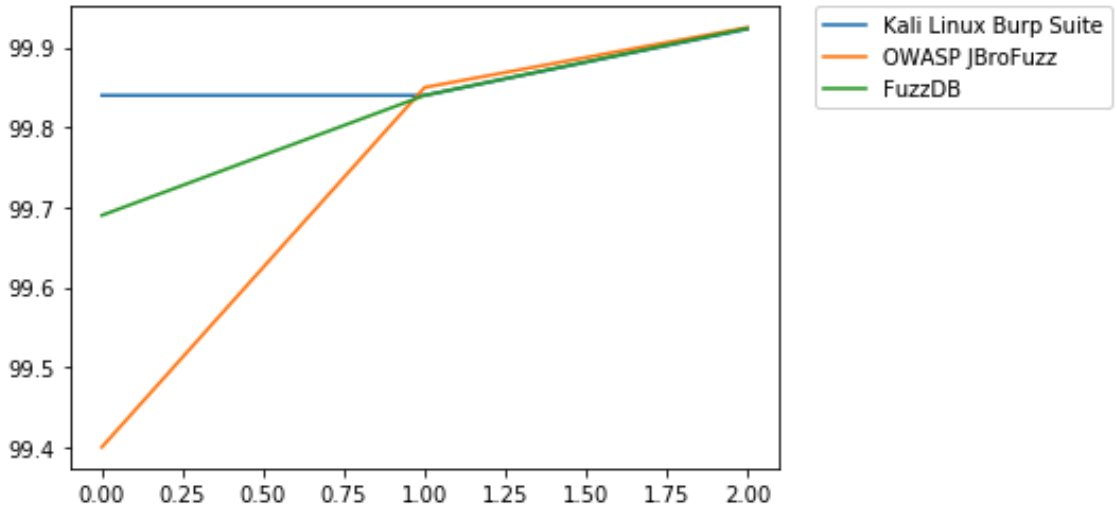

Figure 2: Naive Bayes Accuracies Convergence After Second Resizing

4. Classifying New Documents

Now that the datasets have been preprocessed and reengineered to be optimal in training the custom implementation of the Naïve Bayes Multinomial text classifier, tests will be ran to classify new documents, deciding whether they are malicious or safe database query payloads.

4.1 Classifying New Documents

For the purpose of this test, the OWASP JBroFuzz dataset was used to train the model as it had the highest classification accuracy. Given the results in Section 3.2, there would be no noticeable effect in the classification of new documents if the other datasets were used to train the model. The Naïve Bayes Multinomial text classification model was successful in classifying new documents expected to be malicious, see Table 5, however, a problem arose when attempting to classify documents expected to be safe, see Table 6.

Table 5: Initial Classification of New Malicious Documents

Document	**Expected Classification**	**Given Classification**
TRUE OR 1=1; --	1	1
1' AND (SELECT * FROM Users) = 1--+	1	1
DROP TABLE members --	1	1
SELECT * FROM members; DROP members--	1	1
WAITFOR DELAY ‘0:0:10’--	1	1

Table 6: Initial Classification of New Safe Documents

Document	**Expected Classification**	**Given Classification**
SELECT COUNT(*) FROM games;	0	1
SELECT * FROM Users;	0	1
SELECT * FROM Customers ORDER BY Country;	0	1

Immediately it is apparent that the Naïve Bayes Multinomial text classifier is not correctly classifying documents that are expected to be safe. This is because the original datasets only contain documents that are classified as 1. In order to work towards a classification model that can accurately classify for documents that are both malicious and safe (0 or 1), a process of features engineering must take place.

4.2 Feature Engineering the Datasets

The process of features engineering for the datasets involved adding documents that would be classified as safe and retraining the Naïve Bayes Multinomial text classifier. The process of adding safe documents to the dataset was a long and tedious one, since it involved creating hundreds of entries that represent very general purpose and syntactically correct SQL statements. For the purposes of this test, 300 documents with class safe (or 0) were added alongside each of the datasets to train the Naïve Bayes Multinomial text classifier, and the inputs from Section 4.1 were retested with each of the three datasets training the model. See below for the results of the

Naïve Bayes Multinomial text classifier after the features engineering was performed on the training datasets.

Table 7: Classification of Documents After Features Engineering (OWASP JBroFuzz)

Document	**Expected Classification**	**Given Classification**
TRUE OR 1=1; --	1	1
1' AND (SELECT * FROM Users) = 1--+	1	1
DROP TABLE members --	1	0
SELECT * FROM members; DROP members--	1	0
WAITFOR DELAY '0:0:10'--	1	0
SELECT COUNT(*) FROM games;	0	0
SELECT * FROM Users;	0	0
SELECT * FROM Customers ORDER BY Country;	0	0

Table 8: Classification of Documents After Features Engineering (Kali Linux Burp Suite)

Document	**Expected Classification**	**Given Classification**
TRUE OR 1=1; --	1	1
1' AND (SELECT * FROM Users) = 1--+	1	0
DROP TABLE members --	1	0
SELECT * FROM members; DROP members--	1	0
WAITFOR DELAY '0:0:10'--	1	0
SELECT COUNT(*) FROM games;	0	0
SELECT * FROM Users;	0	0
SELECT * FROM Customers ORDER BY Country;	0	0

Table 9: Classification of Documents After Features Engineering (FuzzDB)

Document	**Expected Classification**	**Given Classification**
TRUE OR 1=1; --	1	1
1' AND (SELECT * FROM Users) = 1--+	1	0
DROP TABLE members --	1	1
SELECT * FROM members; DROP members--	1	0
WAITFOR DELAY '0:0:10'--	1	1
SELECT COUNT(*) FROM games;	0	0
SELECT * FROM Users;	0	0
SELECT * FROM Customers ORDER BY Country;	0	0

As seen in Tables 7-9, none of the training datasets resulted in a model that was 100% accurate, with the FuzzDb having the highest accuracy of 75% with the given test documents. See Figure 3 to analyze how the test results compare between the datasets.

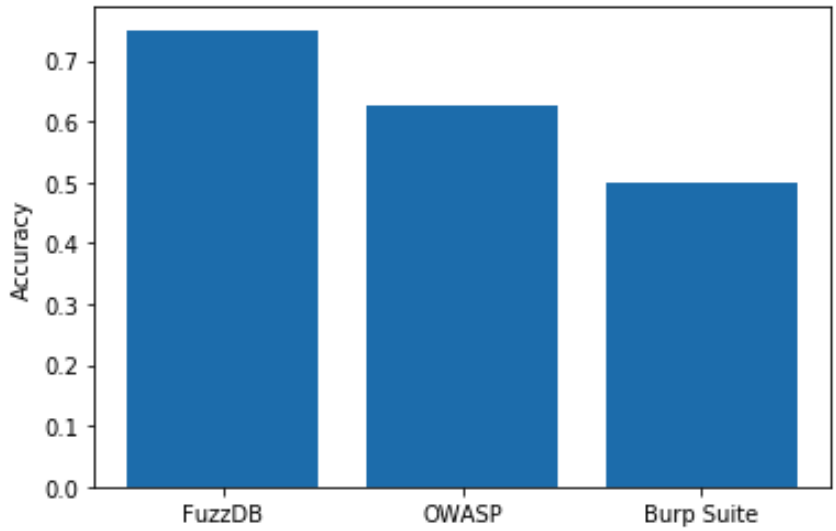

Figure 3: Accuracies of Training Models with Feature Engineered Datasets

Interestingly, despite that high levels of accuracy (>99.9%) while training and testing between datasets, the accuracy is much lower when new documents are classified. This may be the result of having fuzzing payload datasets that are very similar to one another, since they were created for the same purpose, and in the case of classifying new documents, that variable parts of the SQL query (such as table or column names) differ too much between the dataset used to train the model and the test documents.

Conclusion

After running a multitude of tests on the datasets, and performing large amounts of preprocessing and features engineering, it is safe to conclude that there is potential in creating a SQL injection detection system using fuzzing payload datasets. It appears that the size of the dataset has a significant impact on the accuracy of the classifier; a fact that seems obvious since having a large vocabulary in the classification model would cover more test instances when new documents arrive. It is also important to use a payload dataset that includes safe documents as well, since most SQL injection fuzzing payloads will only include malformed and malicious documents.

While this study has proven that detecting SQL injections using fuzzing payloads is possible and relatively feasible, in order to create a system that is robust, efficient, and highly accurate would take a bit of intelligent planning. Since most production databases are going to have different table names, column names, value types, etc., there is no “on size fits all” dataset that can be used for building a classifier. Therefore, the implementers of such a system would need to analyze the potential SQL queries to the database of interest, and think like an attacker in order to create their own payload dataset to train a classification model with.

Future Ambitions

Being a student who is highly interested in both data mining and cybersecurity, and having Co-op experience working in both fields, I found the results and methods of this study to be very interesting and useful for developing a strong skillset within my interests. I plan to further the

work I have performed as part of this report, and develop a more robust and accurate SQL injection classification system as a side project.

Related Readings

Out of my interest for the topics of this report, I have come across the following reports that are related to this project.

- https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1727&context=etd_projects
- https://hal.archives-ouvertes.fr/hal-01138604/document
- https://pdfs.semanticscholar.org/6607/c88a1db2858de0b26f506cccaa51471a56c3.pdf

Bibliography

[1] OWASP, "OWASP Top 10 - 2017," 10 October 2017. [Online]. Available: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf. [Accessed 19 November 2019].

[2] xer0dayz@xerosecurity.com, "Intruder Payloads," 20 February 2017. [Online]. Available: https://github.com/1N3/IntruderPayloads. [Accessed 19 November 2019].

[3] OWASP, "Payloads," 9 March 2016. [Online]. Available: https://github.com/foospidy/payloads/tree/master/owasp/jbrofuzz. [Accessed 19 November 2019].

[4] fuzzdb-project, "sql-injection," 25 May 2016. [Online]. Available: https://github.com/fuzzdb-project/fuzzdb/tree/master/attack/sql-injection. [Accessed 19 November 2019].