SQL Generation to Prevent SQL Injection via Obfuscation

Nick Vadlamudi
nrv434@utexas.edu
UT Austin, USA

Abstract

SQL injections are attacks done on web applications that modify queries to inject harmful code into a database. A user can potentially read, create, update, or delete code within a database that they do not have permission to do. As such, modern web applications use Web Application Firewalls (WAFs) to filter user inputs to ensure that code is safe before executing SQL commands. However, newer attacks use obfuscation techniques to bypass WAFs by modifying the input to appear safe to the input filter.

Obfuscation is not accounted for in a lot of input filtering since WAFs use static, grammar and syntax based rules that target certain keywords that need to be filtered out. New static rules implemented can potentially stop certain obfuscation, but the attack methodology can quickly evolve to bypass new rules. As such, machine-learning-based detection can create better detection for obfuscated code since it can adapt to the numerous types and adjustments of obfuscation.

SOQL is a SQL obfuscation generator that provides an algorithmic transformation of a SQL query to produce an obfuscated version of the query. The purpose of this generator is to create training data to create a robust and informative dataset to use for classification models. In this paper, we will show the importance of having plenty of obfuscated entries to ensure that detection mechanisms can properly identify obfuscation attacks. Using detection models trained only on regular injection data will result in a significantly higher amount of missed detections.

The SOQL generator allows for several obfuscation types to be made. The resulting obfuscated data will be tested on some basic, popular text classification models to show how well obfuscation can be detected. Using Logistic Regression and Multinomial Naive Bayes Classification showed to be quite accurate in detecting the obfuscation techniques provided by the generator given enough data.

ACM Reference Format:
Nick Vadlamudi. 2023. SQL Generation to Prevent SQL Injection via Obfuscation.
, 9 pages.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

1 Introduction

Databases are prevalent in virtually every industry as they are an easy way to store structured and unstructured information in large quantities. Data is leveraged in numerous aspects of business and research in order to carry out operations and provide insight into problems. As such, data storage needs to be secure so that we can maintain the integrity of the data while ensuring that sensitive information cannot be seen without proper authorization. Modern database servers like MySQL, SQLite, PostgreSQL, etc. allow for easy data storage using the Structured Query Language (SQL) that executes commands by a user to select, update, insert, and delete data in the database. Malicious users with the ability to query a database can format these queries in a way that abuses the privileges of the database to access or tamper with data that the user does not have permission to. One of the methods that attackers commonly use is obfuscated SQL commands which are seemingly benign parameters provided to SQL commands that execute malicious commands when run on the database. The issue with these obfuscated commands is that they can be hard to detect since there are numerous obfuscation methods where it can be disguised as a hex string, or wrapped in SQL CHAR() commands, etc. In modern web applications, Web Application Firewalls (WAF) are used to check user inputs before they are executed on the database. When a query is passed through the firewall, the query is directly executed as a command on the database. Hence, there are a static set of rules that are implemented on the firewall to prevent malicious queries from being executed on the database. Commonly, input filters consists of grammar checking and parsing the input to check for specific keywords that are blocked by the developer. Alternatively, there are machine-learning-based filters that are trained on plain injection queries that label if a query is safe or not. Thus, having an obfuscated SQL injection can potentially bypass both of these filtering types as it passes the static format check while also looking too different from what the classification model is trained to detect. This paper proposes

a SQL Generation application that generates SQL injection calls based on provided user input. The input could be benign or malicious, but either way, an obfuscated command string will be generated that can be run on the database to inject the data. Given the same inputs, a random factor is implemented into the generation such that the generated strings are unique. Additionally, the generator provides multiple types of obfuscation transformations. From the list of techniques, the user can select the one transformation they want to apply to the obfuscated string. Considering that there are many platforms with unique SQL command filtering, it is left to the user to validate whether the obfuscation technique is usable on the attack surface they want to test. The SQL generator will provide insight into SQL obfuscation detection, particularly on how we can detect malicious obfuscated SQL commands. This allows database security developers to design better iterations of SQL injection detectors that can adapt to the wide range of obfuscation methods that can be done by attackers.

2 Motivation

One of the primary motivations of the SQL Generator is the Bashfuscator framework that generates random convoluted Bash code that evaluates the desired input upon execution. Similar to bash commands, SQL commands have a lot of vulnerabilities that can be abused to attain privileged abilities. By having the application generate numerous unique obfuscated commands, it is easier to create datasets used to train models to detect malicious obfuscated SQL. In terms of how to approach the obfuscation generation, the main motivation is to encompass all of the different ways that an obfuscated string can be made. There a prevalent research papers (Queries, Obfuscation Research) that detail the type of obfuscations that are bypassing most Web Application Firewalls (WAF) and Intrusion Detection/Prevention Systems (IDS/IPS). Our generation library can use these methods as the main way to create obfuscation strings since we know that they are used by attackers. Finally, we can approach detection mechanisms by looking at existing solutions that try to detect SQL obfuscation attacks. Some current implementations of obfuscation detection involve parsing trees that de-obfuscate commands and then validate the syntax. A paper by Halfond, et. al. also identifies some other classification models that have worked at identifying "alternate encoding" attacks which fit the function of what obfuscated SQL aims to do. Some of the proposed methods, like AMNESIA, are dynamic models that could better adapt to the uniqueness of SQL obfuscation attacks provided by the set of commands made by the SQL generator.

3 Our Architecture

Our generation architecture consists of the generation application and a data set that we use to train a classification model. The flow diagram below describes the attack process. Firstly, the user will input a regular SQL command. The application will provide options for the formatting of the obfuscation transformation. Once processed through the generator, the user will be provided the obfuscated command with the ability to copy the text. The text can then used on a test bed chosen by the user to see if the obfuscation is suitable for the tested firewall. Should the query run and produce the desired output, we can then add the command as an entry to the data set that we can use for detection.

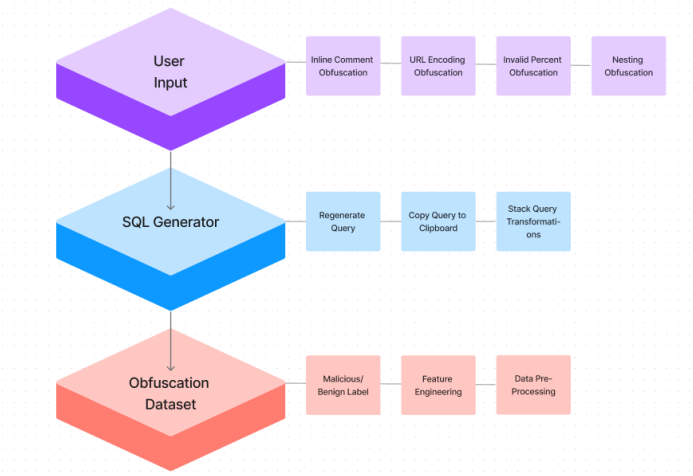

Figure 1. SQL Generator Architecture

3.1 Obfuscation Techniques

Obfuscated SQL code has numerous implementations. As such, we will provide a user interface that allows for customization for the type of attack to be run. The following subsection shows the types of transformations that can be implemented to a sample command. For the transformations below, consider the command **SELECT * FROM table**

3.1.1 Inline Comment Transformation

. SEL/*_*/EC/*_*/T * F/*_*/RO/*_*/M table

In this transformation, the statement has inline comments added to the original query. The obfuscation creates spacing to avoid blacklisting. In many SQL Injection checking libraries, commands like "SELECT" and "FROM" are filtered out on a blacklist to prevent injection. With comments, the filter reads the command differently, so the command bypasses. Randomization can be applied to the placement and quantity of comments in the obfuscated string, allowing for multiple obfuscation results for this one example command.

3.1.2 URL Encoding

. S%45LECT%20%2A%20FRO%4D%20tabl%45

In this transformation, the statement has url encodings substituting characters of the string with their corresponding

```python
def ManipulateInput_Inline(string):
    # Obfuscate user input
    for word in commands:
        if word in string.lower():
            #  find all that match the sql command
            wordmatched = re.findall(word, string, flags=re.IGNORECASE)
            for i in range(len(wordmatched)):
                wordmatched = re.search(word, string, flags=re.IGNORECASE)
                RandomPlace = random.randint(1, len(word) - 1)
                listofchr = [char for char in wordmatched[0]]

                newword = ""
                for i, c in enumerate(listofchr):
                    if i == RandomPlace:
                        newword += "/**/"
                    newword += c

                resault = re.compile(re.escape(word.lower()), re.IGNORECASE)
                string = resault.sub(newword, string, 1)
    return string
```

Figure 2. Inline Comment Obfuscation Code

encoding in hexadecimal format. This format is possible because a web server processing requests accepts requests in this encoded format and the database decodes the hex during execution. In the example, %45 corresponds to "e", %20 represents a *space*, %2A represents a "*", and %4D represents an "m". Randomization can be applied to the amount of characters that are encoded, allowing for multiple obfuscation results for this one example command.

```python
def ManipulateInput_URL(user_input):
    # for each word in the user input, split each word into characters
    # generate a random number between 1 and len(listofchr)
    # replace random character with % + hex value of character for random number of characters
    user_input = user_input.lower()
    words = user_input.split()
    for word in words:
        # split word into list of characters
        listofchr = [char for char in word]
        random_subs = random.randint(1, len(listofchr))
        index_list = []
        for i in range(random_subs):
            random_index = random.randint(0, len(listofchr) - 1)
            while random_index in index_list:
                random_index = random.randint(0, len(listofchr) - 1)
            index_list.append(random_index)

            listofchr[random_index] = "%" + to_hex(listofchr[random_index])
        # join the list back into a string
        newword = "".join(listofchr)
        # replace the word with the new word
        result = re.compile(re.escape(word.lower()), re.IGNORECASE)
        user_input = result.sub(newword, user_input)

    return user_input
```

Figure 3. URL Encoding Obfuscation Code

3.1.3 Invalid Percent Encode . %SE%LE%CT * F%RO%M ta%b%le

In this transformation, the statement has percent symbols added throughout the string. However, they are not in a valid position to be properly encoded and interpreted. In .NET and IIS applications, the invalid "%" symbols are stripped from the command when executed, making the command valid. Therefore, the obfuscated string could bypass filters checking form malicious commands due to masking the commands with invalid symbols while still being executed. Randomization can be applied to the amount of "%" symbols and where they are placed in the string.

```python
def ManipulateInput_Percent(user_input):
    # obfuscate user input with invalid percent encoding      You, 1 second ago
    # for every word in the user input, add a random number of %'s to the word
    for word in commands:
        # split word into list of characters
        listofchr = [char for char in word]
        # for every character in the list
        for i, c in enumerate(listofchr):
            # add a random number of %'s to the character
            randomnumber = random.randint(0, 2)
            listofchr[i] = (randomnumber * "%") + c
        # join the list back into a string
        newword = "".join(listofchr)
        # replace the word with the new word
        resault = re.compile(re.escape(word.lower()), re.IGNORECASE)
        user_input = resault.sub(newword, user_input)

    return user_input
```

Figure 4. Ivalid Percent Encoding Obfuscation Code

3.1.4 Nesting Stripped Expressions . SELSELECTECT * FROFROMM table

In this transformation, the statement nests expressions within themselves. Some sanitizing filters that prevent SQL injection simply strip expressions in one iteration. So, when the filter detects "SELECT" or "FROM" nested in the larger string and removes it. However, when the expression is removed, the resulting expression is "SELECT" and "FROM". If the filter is not applied recursively, the obfuscated command bypassed the filter. Randomization can be applied to where the nested expression is placed within the expression.

```python
def ManipulateInput_Nesting(user_input):
    # obfuscate user input with nesting expressions
    # for every word in the user input that is a sql command
    # split the word into a list of characters
    # choose a random position in the list
    # add the duplicate of the word starting at the random position
    # join the list back into a string
    # replace the word with the new word
    for word in commands:
        if word in user_input.lower():
            # split word into list of characters
            listofchr = [char for char in word]
            # choose a random position in the list
            RandomPlace = random.randint(1, len(listofchr) - 1)
            # add the duplicate of the word starting at the random position
            listofchr.insert(RandomPlace, word)
            # join the list back into a string
            newword = "".join(listofchr)
            # replace the word with the new word
            resault = re.compile(re.escape(word.lower()), re.IGNORECASE)
            user_input = resault.sub(newword, user_input)
    return user_input
```

Figure 5. Nesting Expressions Obfuscation Code

3.2 Generation Application

The SQL Generator will be provided as a Graphical User Interface using wxPython. Within the interface, the user will be prompted to enter a SQL query. They will also be provided a drop down menu where they can select the obfuscation technique they want to implement. They will be able to see the output and be able to copy it to their clipboard. The initial design of the GUI is shown in the figure below.

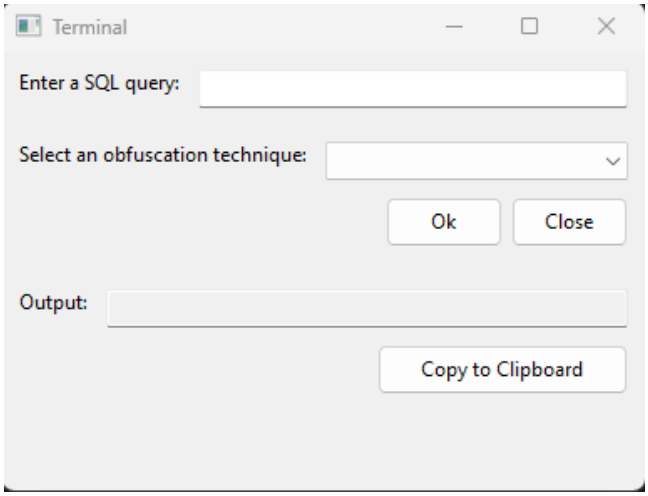

Figure 6. Generator GUI

3.3 Data Set Creation

The data set is designed to be an accumulation of obfuscated commands that have a label of whether they are malicious or benign to the database. This classification model can be treated as binary classification in this case, where 0 indicates *benign* and 1 is *malicious*.

3.3.1 Adding all Transformations

. Since the generator allows for multiple types of transformations, it is important to include all viable transformations of the attack surface. This would mean that one normal query could have multiple entries where the obfuscation differs by transformation type. Also, the combination of obfuscations allows for more training data for just one command.

3.3.2 Utilizing Randomization

. The generator uses randomization to handle the placement or encoding of characters such that one transformation of a command has multiple unique obfuscations. This allows for a more robust dataset that aids in model accuracy.

3.4 Classification Model

Classification model accuracy will vary to each firewall that is being tested. As such, we will experiment with multiple classification models on each type of obfuscation. The two models that are tested in this paper is Logistic Regression and Multinomial Naive Bayes Classification since they are popular in the sphere of Natural Language Processing (NLP) and text classification. These models also return probabilities of each text input being malicious which allows for thresholding so that developers can create rules set on percent confidence rather than a binary label.

3.4.1 Logistic Regression

. Logistic Regression is a well-suited model for binary classification. These models are also very simple to implement, interpret, and fit. So, logistic regression can provide sufficient classification for firewall developers that may not be too familiar with machine learning.

3.4.2 Multinomial Naive Bayes Classification (MultiNB)

. This classification model uses a bag-of-words approach to learn word and term frequencies and the interaction of the words together in a string to create a classification on the text. This approach helps reduce missed detections by analyzing the words using condition independence.

4 Experimental Results

The goal of the generator is to produce an obfuscated version of a SQL query, whether it is malicious or benign. As such, we should expect to see a similar output for both types of queries. In this section, we observe how the output looks after generation to ensure that it is the correct format. We will show an example of a benign query, **SELECT * FROM Person.Person WHERE FirstName = 'Nick'** and a malicious query, **SELECT * FROM Person.Password WHERE 'Id' = 1 OR 'a' = 'a'**

4.1 Inline Obfuscation

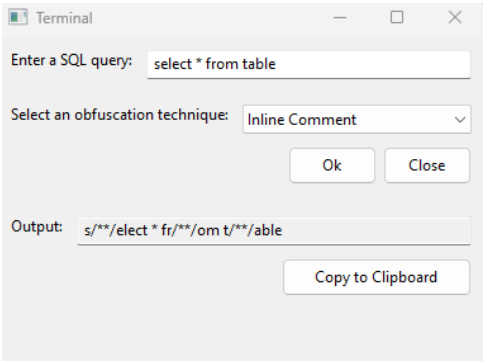

Figure 7. Generated Inline Obfuscation Example

4.1.1 GUI Output

. Benign: SELEC//T * F/**/ROM Person.Person WHER/**/E FirstName = 'Nick'**
Malicious: S//ELECT * FR/**/OM Person.Passwo/**/rd WHE/**/RE 'Id' = 1 O/**/R 'a' = 'a'**

4.1.2 Query Results

. The obfuscation here achieves the results we expected in theory where there is a random amount of comments separating the commands. This hinders readability reasonably.

4.2 URL Encoding Obfuscation

4.2.1 GUI Output

. Benign: sele%63t %2a %66%72om %70e%72%73%6fn%2epers%6fn

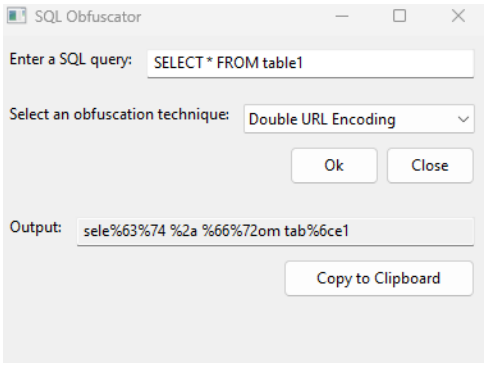

Figure 8. Generated URL Encoding Obfuscation

%77%68%65%72%65 %66i%72%73tn%61%6d%65 %3d '%6eic%6b%27
Malicious: %73ele%63%74 %2a %66r%6f%6d person%2epa%73swo%72d
%77%68%65%72%65 %27%69%64%27 %3d %31 o%72 %27%61%27
%3d %27%61%27

4.2.2 Query Results

. The transformation matches what was expected. The obfuscation performs well in reducing readability. Just visually, it is hard to observe any malicious intent from this type of query.

4.3 Invalid Percent Obfuscation

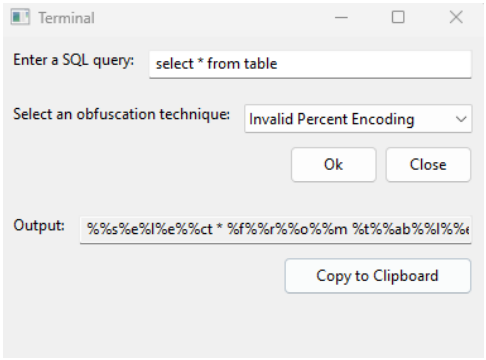

Figure 9. Generated Percent Encoding Obfuscation

4.3.1 GUI Output

. Benign: se%le%c%%t * f%r%o%m Person.Person %%wh%%e%%re FirstName = 'Nick'
Malicious: %se%%l%e%c%t * fro%m Person.Passw%o%rd w%%h%e%r%e 'Id' = 1 %o%r 'a' = 'a'

4.3.2 Query Results

. The generated command achieved what it was designed to do. Readability is hindered by spacing out the commands with the invalid percent symbols. It may be worth looking into modifying the code itself to adjust which words get encoded since the "or" of *Password* was also transformed, as seen in the malicious command.

4.4 Nesting Expressions Obfuscation

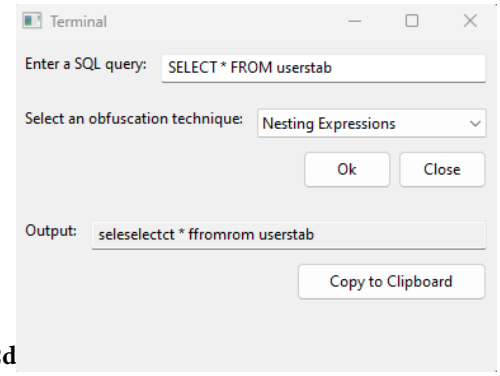

Figure 10. Generated Nesting Obfuscation

4.4.1 GUI Output

. Benign: selecselectt * frfromom Person.Person wwherehere FirstName = 'Nick'
Malicious: sselectelect * ffromrom Person.Passwoorrd wherwheree 'Id' = 1 oorr 'a' = 'a'

4.4.2 Query Results

. The transformation achieves the expected result. The readability is still very much intact since the elements are just nested, but achieves the bypass result from the filtering type it targets.

4.5 Combining Obfuscation Techniques

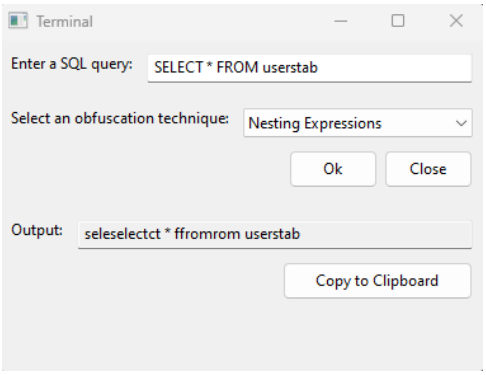

Figure 11. Generated Nesting Obfuscation

4.5.1 GUI Output

. Benign: %73e%6ce%63%73%65%6ce%63%2f%2a%2a%2ft%74
%2a frfr/%2a%2a/omom %70%65%72%73%6f%6e%2e%70%65%72%73%6f
wherw/%2a%2a/heree firs%74%6eame %3d %27%6e%69%63%6b'
Malicious: %73%73%65%6c%2f%2a%2a%2f%65%63%74%65%6c%65%63%

%2a ffr/%2a%2a/omrom person.passwoo/%2a%2a/rrd wh-wher/%2a%2a/eere '%69d%27 %3d %31 oo/%2a%2a/rr %27a' %3d %27a'

4.5.2 Query Results

. It is also worth looking into how the combination of these transformations can be combined to further reduce the readability of a query. In the example above, the queries were nested, inline commented, and then url encoded. The readability here is the most hindered and it is really hard to visually tell whether the commands are malicious visually.

4.6 Building a Classification Model

When creating a data set for training a classification model, we end up with one string feature of the command and the label of whether it is malicious or benign. As described in Our Architecture, we implemented two classification models, Logistic Regression and MultiNB. There are five datasets derived from two Kaggle datasets (SQLi and ModSQL) that have a combination of benign SQL queries and SQL injections with their corresponding labels (0 benign, 1 malicious). Combined, the Kaggle datasets amount to 35,120 entries and the train test split used 20 percent of the combined datasets for a total of 7,024 test entries. Each dataset is a union of the two Kaggle datasets with unique modifications. One was left completely unaltered and was used to train a baseline classifier using MultiNB. For the four remaining datasets, the malicious queries were obfuscated using one of the transformations available from the generator, and then they were used to train a classification model for Logistic Regression and MultiNB. This results in nine models total. All models were evaluated with accuracy, precision, and recall. Since this is a detection model, we would like to minimize the amount of false negatives (missed detections) for the model. As such, when evaluating the differences between the two model types, there will be an analysis on which model performs best at reasonably minimizing the recall while maintaining competitive accuracy. The results are as shown below.

4.6.1 Inline Detection. When trained on obfuscated data, the MultiNB model had an accuracy of 0.97 with a precision of 0.97 and recall of 0.97. The logisctic regression model also had an accuracy of 0.97 with precision 0.97 and recall 0.97. The models performed very similarly and have high rates of detection. MultiNB performed best in missed detections with 128 false negatives to 61 false positives, however, the logistic regression model had a significantly less amount of false positives at 4 compared to a higher 193 missed detections. Their precision-recall curves are shown below.

4.6.2 Invalid Percent Detection. The MultiNB model had an overall accuracy of 0.97 with a precision of 0.97 and recall of 0.97. The logistic regression was slightly under those metrics with an accuracy of 0.96, 0.96 precision, and 0.96 recall. Although the logistic regression model had a small

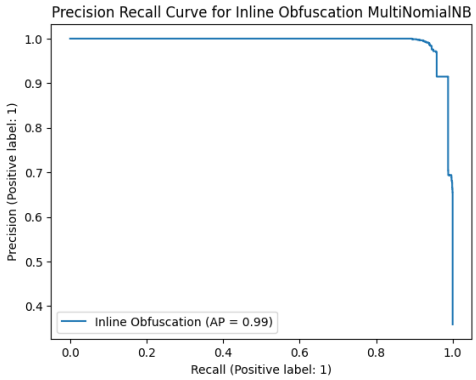

Figure 12. PR Curve for MultiNB

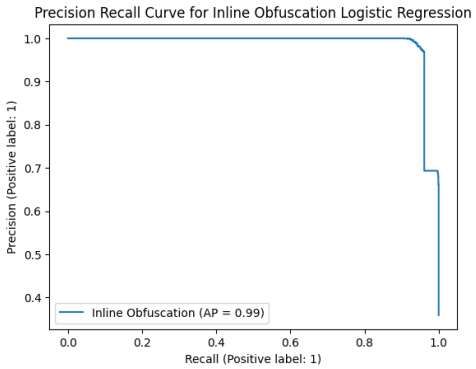

Figure 13. PR Curve for Logistic Regression

amount of false positives at 9, there were 285 missed detections compared to 159 missed detections for the MultiNB model. Although there are more false positives at 73, the tradeoff of over 100 more true detections outweighs this case in our evaluation.

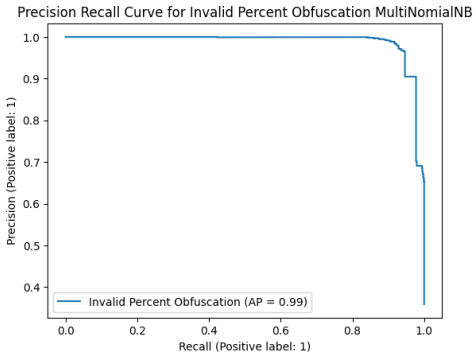

Figure 14. PR Curve for MultiNB

4.6.3 URL Encoding Detection. Overall, URL encoding was easiest to detect out of all the obfuscation techniques tested. The MultiNB had a rounded accuracy of 1.00 with 1.00

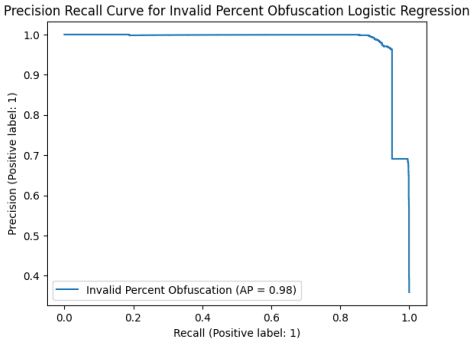

Figure 15. PR Curve for Logistic Regression

precision and recall. The logistic model also had a rounded accuracy of 1.00 and 1.00 precision and recall. The MultiNB model had 8 missed detections and 19 false positives while the logistic model 28 missed detections to 2 false positives. Although readability is significantly hindered using URL encoding, the text classification models are easily able to detect the use of hex encodings as malicious while not having too many false detections.

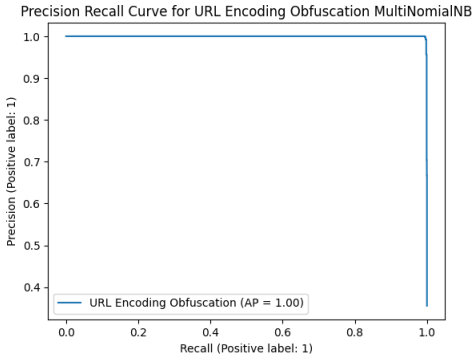

Figure 16. PR Curve for MultiNB

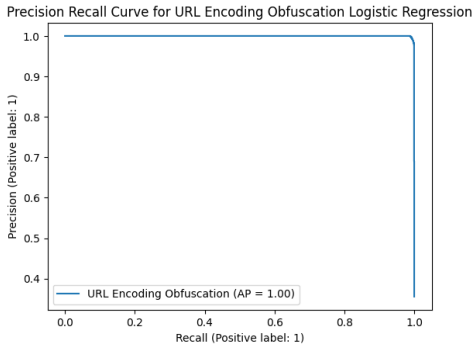

Figure 17. PR Curve for Logistic Regression

4.6.4 Nested Expression Detection.

The MultiNB model and logistic regression model both had an accuracy of about 0.99 with a 0.99 precision and recall. The MultiNB had 48 missed detections to 56 false positives while the logistic regression model notably had no false positives to 67 missed detections.

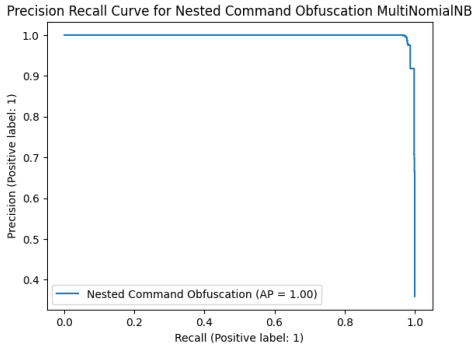

Figure 18. PR Curve for MultiNB

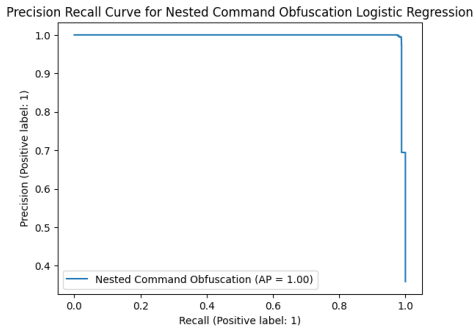

Figure 19. PR Curve for Logistic Regression

4.6.5 Baseline Model.

This model was trained on unobfuscated data. So, while it is designed to do well at detecting plain SQL injections at 0.96 accuracy and 0.96 precision and 0.95 recall with a general MultiNB model, it should not perform well when it encounters an obfuscated entry. From the confusion matrix below, we can see that there is a significant jump in missed detection, mostly in the 1000's of false negatives. This shows the importance of having obfuscation training data. Most models trained to detect SQL injection are likely only detecting plain injections, so obfuscated injections have a much higher likelihood of bypassing the filter.

4.6.6 Overall Evaluation.

The two figures below are the confusion matrices of the MultiNB and Logistic Regression classifiers, respectively. Overall, we can see that the MultiNB model did better at minimizing missed detections while Logistic Regression minimized false positives. In terms of security, missed detections are more important to reduce

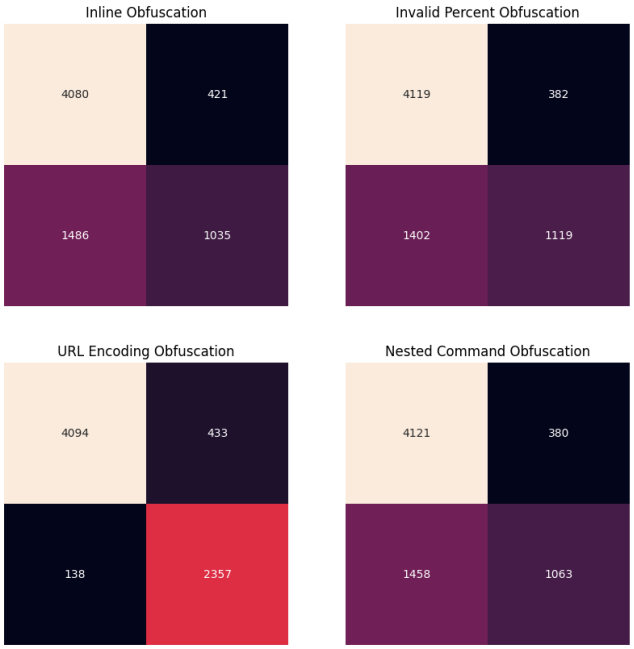

Figure 20. Confusion Matrix of Base Model Detection of Obfuscated Data

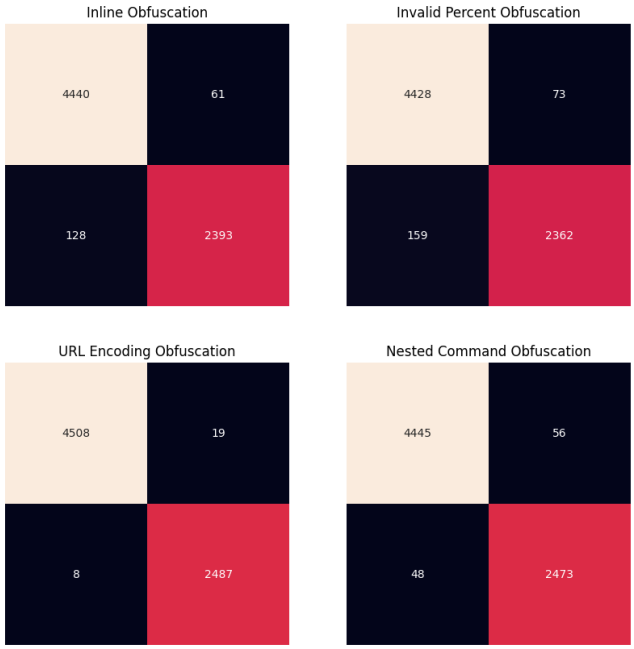

Figure 21. Confusion Matrices for MultiNB

since we don't want more injections bypassing the filter, but an overwhelming amount of false positives can create too much overhead for analysts and applications to handle. A good compromise could be to analyze an ensemble of the two models together. We can expect that the ensemble may not have the same minimal false positives and negatives, but can be more reasonable for a security system.

5 Related Work

SQL Injections and Obfuscation attacks have been prevalent for some time and there are numerous research topics in attack prevention. As a result, there is a lot of analysis on the type of attacks that occur on databases, especially injections and obfuscation.

In terms of injections, OWASP has aggregated a cheat sheet of defense mechanisms that can be applied in database security. The primary defense implementations involve using prepared statements, similar to generating SQL statements. As well, it recommends the use of properly constructed stored procedures, allow-list input validation, and escaping all user-supplied input. OWASP Cheat Sheet

A paper by Balasundaram, et al. also covered detection techniques using ASCII-based string matching, ASCII based detection

Research on evasion techniques revealed how attackers have tried to bypass these existing prevention mechanisms by using obfuscation. An article by Imperva details an analysis of the type of obfuscation methods attackers used in a

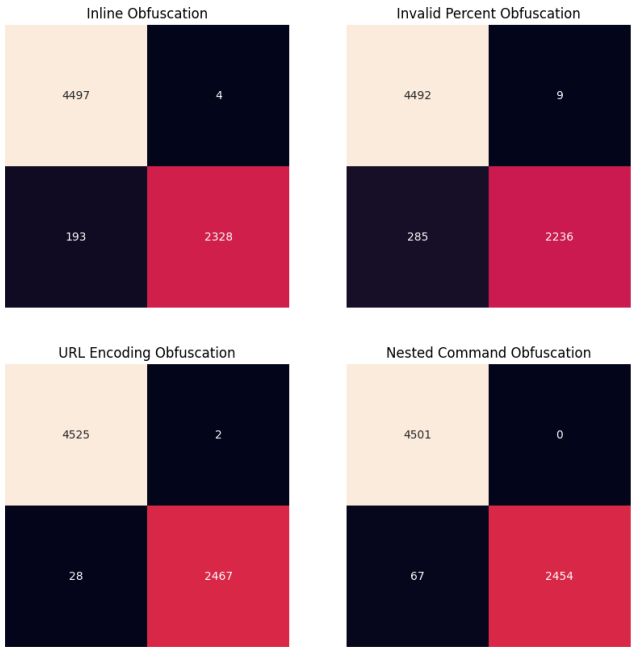

Figure 22. Confusion Matrices for Logistic Regression

honeypot that was set up to examine attack methods. The article detailed HEX Encoding attacks that disguise malicious queries as hex strings. This paper provides some patterns that can be observed when an attacker attempts hex encoding injections for multiple database platforms which could assist in obfuscation detection for SQL generators.

A paper by Robert Salgado covers SQL injection using optimization and obfuscation. It has detailed explanations of how obfuscation attacks work and how to obfuscate queries for different database environments.

6 Conclusions

In conclusion, having a robust dataset full of obfuscated data to train text classification models is a crucial part of developing new firewalls to better protect database information. As such, the SOQL generator provides an algorithmic transformation of multiple obfuscation techniques used by attackers today to provide insight on how to combat the evolving techniques used for SQL injections.

Further work can be done with SOQL to cover more attacks and create better detection models. In our results, our detection models were trained on only one attack. Depending on the attack environment, a training dataset can consist of a sample of different obfuscation attacks so that the model can recognize several types of obfuscations. As recommended in Experimental Results, experimenting with more text classification models and the ensembling of these models could prove to have better detection metrics for security systems.

The SOQL generator only covers four syntax-based obfuscation attacks. However, there are numerous other types of attacks that involve encoding, buffer overflow, and fragmentation. Adding more transformations to the generator is a recommended next step since it would cover a larger attack surface. Papers covering other attack methods are in related work.

All code and results can be found in the Github Repository for SOQL. The code is written in Python and all of the CSVs are included. The Jupyter notebooks contain the data obfuscation and classification model results. obfuscator.py contains all of the transformation algorithms. terminal.py is the generator application.

References

[1] Sql injection dataset.
[2] Sql injection dectection using neural network.
[3] A deep dive into database attacks [part i]: Sql obfuscation, 2018.
[4] Netspi sql injection wiki, 2023.
[5] owasp.org. Double encoding | owasp.
[6] R.Salgado. Sql injection optimization and obfuscation techniques, 2013.
[6] [5] [1] [2] [3] [4]