# SQL Generation to Prevent SQL Injection via Obfuscation

Nick Vadlamudi
nrv434@utexas.edu
UT Austin, USA

## Abstract

Describe the overall area of contribution, the crux of the problem, and end with highlights of results. For the initial report, end with the proposed experiments and what you aim to find out.

**ACM Reference Format:**
Nick Vadlamudi. 2023. SQL Generation to Prevent SQL Injection via Obfuscation. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnn.nnnnnnn

## 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. 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 vs non-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.


*Conference'17, July 2017, Washington, DC, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 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.

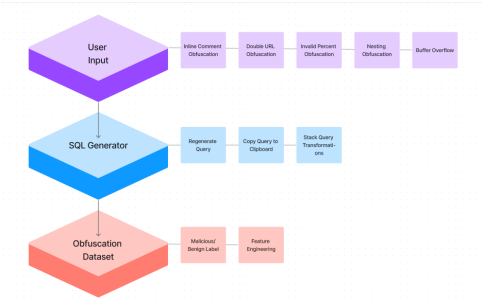


**Figure 1.** SQL Generator Architecture

### 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**

```
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.** Pseudocode of Inline Obfuscation => Replace with actual pseudocode

In this transformation, the statement has url encodings substituting characters of the string with their corresponding encoding in hexadecimal format. This format is possible because a web server processing requests accepts requests in this encoded format. 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.

```
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.** Pseudocode of URL Encoding => Replace with actual pseudocode

#### 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.

#### 3.1.4 Nesting Stripped Expressions

**. SELSELECTECT * FROFROMM table**

In this transformation, the statement nests expressions within themselves. Some sanitizing filters that prevent SQL

```
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.** Pseudocode of the Percent Encoding => Replace with actual pseudocode

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.

```
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.** Pseudocode of the Nesting Expressions => Replace with actual pseudocode

#### 3.1.5 Buffer Overflow

**. SELECT * FROM table UNION #AAAA....**

Firewalls written in C/C++ can crash if too many user-specified parameters are added to the query. This is a buffer overflow that causes a vulnerability to the database. If there is no sanitizing to the size of the input, this attack is possible. Randomization can be applied to where to specify parameters and the filler characters that are used for the overflow attack.

### 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.

```
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 6.** Pseudocode of the Nesting Expressions => Replace with actual pseudocode

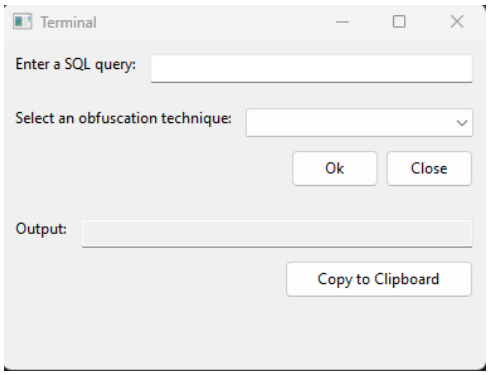


**Figure 7.** Generator GUI

### 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 unobfuscated 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

preprocessing and feature engineering techniques. As well, the type of classification model used will be tested, including Support Vector Machines (SVM), Boosting Models (XGBoost and Catboost), and Random Forests.

#### 3.4.1 Preprocessing

. If we expect that obfuscation is a transformation of the text, we can plan how to reverse the transformation. Knowing the types of attacks that can work on the attack surface allow us to experiment with how to apply transformations to the input so that we can effectively reveal the unobfuscated statement. For example, creating a method to process all strings so that all hex encodings are converted back to ASCII characters will reveal the real string. This will also counter the random aspect of the attack.

#### 3.4.2 Feature Engineering

. Capturing important characteristics of malicious and benign obfuscations help significantly in classification. Tokenization of the data allows us to look for specific keywords or obfuscations associated with an attack. Query length can also reveal the maliciousness of an input.

#### 3.4.3 Model Selection

. SVMs, boosting models, and random forests are common classification models for other text-related classification, like sentiment analysis. By returning the Softmax of these models, we can view what the model predicts the input to be and how confident it is in its selection. Having multiple models allows for ensembling and hyperparameter tuning for better refinement of the final classification.

## 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

#### 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.

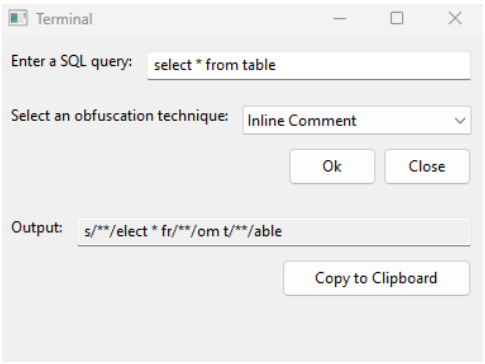


**Figure 8.** Generated Inline Obfuscation Example

### 4.2 URL Encoding Obfuscation

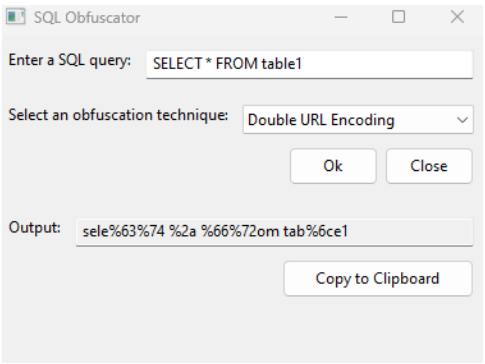


**Figure 9.** Generated URL Encoding Obfuscation

#### 4.2.1 GUI Output

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

#### 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'**

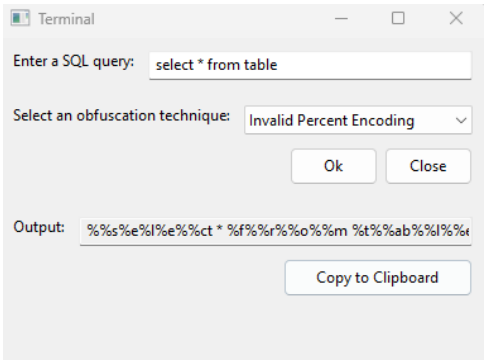


**Figure 10.** Generated Percent Encoding Obfuscation

#### 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

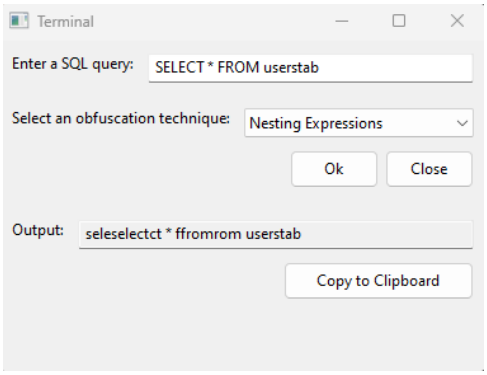


**Figure 11.** Generated Nesting Obfuscation

#### 4.4.1 GUI Output

. **Benign: selecselectt * frfromom Person.Person wwhere-here 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

#### 4.5.1 GUI Output

. **Benign: %73e%6ce%63%73%65%6ce%63%2f%2a%2a%2ft%74**

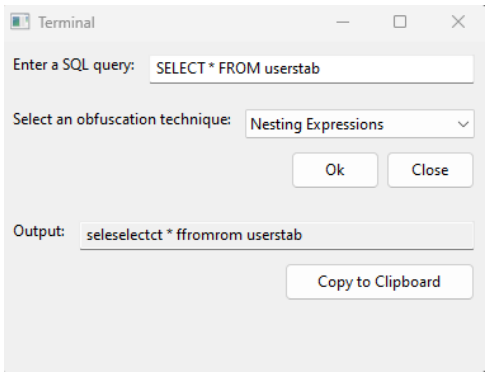


**Figure 12.** Generated Nesting Obfuscation

**%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 futher 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 a combination of preprocessing, feature engineering, and model ensembling.

TODO: FINISH DATASET CREATION TO START TRAINING

## 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 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.

## 6 Conclusions

In conclusion, ...

## References