

SQLBlock: SQL Injection Protection by Variable Normalization of SQL Statement

Sam M.S. NG samng@computer.org

"Make everything as simple as possible, but not simpler."
-- Albert Einstein

Abstract. We present here a method to protect from SQL injection attack. The method involve using a virtual database connectivity drive as well as a special method named "variable normalization" to extract the basic structure of a SQL statement so that we could use that information to determine if a SQL statement is allowed to be executed. The method can be used in most scenarios and does not require changing the source code of database applications (i.e. the CGI web application). The presented method can also be used for auto-learning the allowable list of SQL statements, which makes the system very easy to setup. And since the decision of whether a SQL statement is allowed is to check if the normalized statement exists in our ready-sorted allowable list, the overhead of the system is very minimal.

1. Background

SQL injection [1, 2] is now one of the most common attacks in the Internet. It is an application layer attack to inject SQL commands along with other valid inputs possibly via web pages. Many web pages take parameters from web user, and make SQL queries to the database. Take for instance when a user login, web page ask for user name and password and make SQL query to the database to check if a user has valid name and password. With SQL injection, it is possible for malicious users to send crafted user name and/or password field that will change the SQL statement structure and thus allowing the attacker to alert the intended SQL statement and execute arbitrary SQL commands on the vulnerable system.

For example, consider an e-banking web site login page, the program logic may be expecting a user name and password. For instance, when the user hit the submit button on the page with user name 'john' and password 'mysecret' the program may issue the following SQL command to the database:

```
SELECT * FROM user_table  
WHERE user_id = 'john' and password = 'mysecret'
```

The program then authorized the login if the result returns a non-empty result set.

However, instead of a valid user name and password, a malicious user may input the following password to by pass the authentication process:

```
' or 1=1 --
```

The SQL command would then become

```
SELECT * FROM user_table  
WHERE user_id = 'john' and password = '' or 1=1 --'
```

The “or 1=1” basically makes the query to return all the records in the “user_table” and the “--“ comments out the last ‘ appended by the system. Therefore, the query will return a non-empty result set without any error.

Verifying all SQL statements before sending it to the database server can solve SQL injection problem. However, since the SQL statements are dynamically created by the web application, every SQL statements may be different. Therefore, we cannot pre-define the allowable SQL statements in a straightforward way. For instance, the previous login page will issue

SELECT * FROM user_table WHERE user_id = 'john' and password = 'mysecret'
SELECT * FROM user_table WHERE user_id = 'mary' and password = 'love777'
SELECT * FROM user_table WHERE user_id = 'peter' and password = '123jump'
...
...

The situation becomes worse if the system may add new users to the system.

It is possible to use regular expression to check if the input is of our expected pattern. However, regular expression is slow, it is not feasible if there are more than 100 SQL statements.

Instead of verifying the SQL statements directly, our invention uses a special method to “normalize” the variables in a SQL statement. Since client application should only issue SQL statements in a predictable way (unless the system allow users to issue arbitrary SQL commands), there should be only a limited set of normalized SQL statements. Therefore, we will be able to verify the normalized SQL statement against a pre-definable allowable list.

We will explain the detail of variable normalization in Section 2, how we can define the allowable list of SQL statements in Section 3. In Section 4, we will explain the details of SQLBlock that implements variable normalization. We will discuss the performance of SQLBlock in Section 5 and will have an overview of other related technologies in Section 6.

2. Variable Normalization

The propose of variable normalization is try to strip away the variables and get the basic structure of the SQL statement, so that although the supplied variables differ every time, the basic structure remains the same. If SQL injection happens, the injection code will change the structure of the SQL statement, and hence we should be able to detect it.

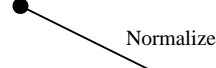
There are only two types of variables in a SQL statement, strings and numbers. Numbers can

further be sub-categorized as integers and floating point numbers, and they can be positive or negative numbers. As specified by SQL standards, strings variables are single quoted, numbers are not quoted. In our design, we will consider quoted numbers to be strings as well. (Ironically, these “variables” are in fact, “constants” in SQL language terminology.)

To normalize variables in a SQL statement, we will convert all single quoted strings to say, the single character “a”, and we will convert all positive or negative integer, or floating point numbers to single digit “0”. We will store the normalized SQL statement in a data structure, called a “rule”, along with its corresponding pre-normalized variable information, including their types, positions and the original values. The normalization process will only modify variables, and it will keep everything else un-modified, including SQL comments, carriage returns, white spaces, or character cases.

Example 1:

`SELECT * FROM employee WHERE name = 'john'`



Normalized SQL	<code>SELECT * FROM employee WHERE name = 'a'</code>			} Rule
Variable 1	Position <u>38</u>	String type	Original value: "john"	

Example 2:

`SELECT * FROM assets
WHERE price > 5000 and category = 'computer'`



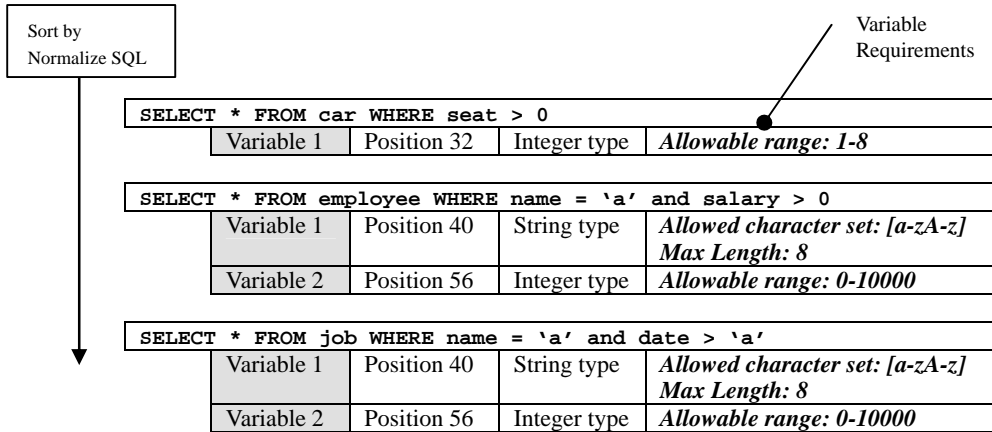
Normalized SQL	<code>SELECT * FROM assets WHERE price > 0 and category = 'a'</code>			} Rule
Variable 1	Position <u>37</u>	Integer type	Original value: 5000	
Variable 2	Position <u>65</u>	String type	Original value: "computer"	

3. Defining the allowable list

In order to verify if a normalized SQL statement is allowed, we will need to pre-define an allowable list. The list is a set of rules that is described in the previous section, but different in a sense that we do not store the variable values, but the variable requirements instead.

We can define the allowable list manually by defining the normalized SQL statement along with the requirements of its variables.

For example, we can define three rules in the list



Instead of defining the allowable list manually, which is tedious and error probing. We can also have the system to “learn” the allowable list by itself. By running the system in “learning” mode, in which the system will not verify if a SQL statement is allowed, but recording all the SQL statements that go through it.

In learning mode, we will store the variable value as the expected value, i.e. we first assume the variables are static. This assumption is true until we found another SQL statement with the same normalized form but with different variable value. Therefore, theoretically we need to “learn” each SQL statement twice; otherwise, all variables will be assumed to be static.

For example, when we first learn this SQL statement

```
SELECT * FROM jobs WHERE completed = 0 and start_day > '3/14/2004'
```

<code>SELECT * FROM jobs WHERE completed = 0 and start_day > 'a'</code>			
Variable 1	Position 39	Integer type	<i>Expected value: 0</i>
Variable 2	Position 58	String type	<i>Expected value: "3/14/2004"</i>

And then we learn this SQL statement

```
SELECT * FROM jobs WHERE completed = 0 and start_day > '7/20/2004'
```

<code>SELECT * FROM jobs WHERE completed = 0 and start_day > 'a'</code>			
Variable 1	Position 39	Integer type	<i>Expected value: 0</i>
Variable 2	Position 58	String type	<i>Expected value: "7/20/2004"</i>

Since both normalized SQL statement have variable 1 expecting value 0, this expectation is retained. However, for variable 2, the two expected values are different (“3/14/2004” and “7/20/2004”), we can perform further analysis to group the two expected values to form a new requirement with say, character set constrain, or simply do not have any requirement.

If we do not perform the extra analysis after merging with the existing rule, the rule would become

<code>SELECT * FROM jobs WHERE completed = 0 and start_day > 'a'</code>			
Variable 1	Position 39	Integer type	<i>Expected value: 0</i>
Variable 2	Position 58	String type	<i>No requirement</i>

4. SQLBlock Implementation

Since we need to check if the SQL statement executing is authorized or not, we need a way to get the SQL statement that is executing. This can best be done by implementing as a proxy driver.

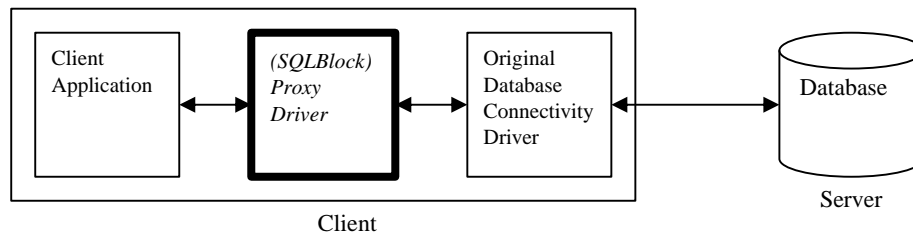


Fig 1. SQLBlock architecture

The proxy driver is basically another database connectivity driver (e.g. ODBC driver or JDBC driver), but instead of passing the database query to the database server directly, the proxy driver calls another database driver as a client to complete its task. Therefore, the client application should be compatible with our proxy since the proxy and the original driver conform to the same connectivity driver standard. Hence, we only need to change the “connection string” which is usually just a configuration parameter in order to instruct the client application to use the proxy driver instead of original database connectivity driver.

Implementing proxy driver provides another feature. If there is any error executing the SQL commands, the database server may send error message to the client. However, these messages may be good for debug during development life cycle, but obviously contain too much information about the database schema, and are often used by attackers to recon the system [6]. The proxy driver can help to solve this by hiding these error messages sending back to the client.

To check if a SQL statement is allowed, the proxy driver will normalize the SQL statement, and search if this statement already exists in our ready-sorted list. If the normalized SQL statement does exist, we will allow this SQL execution if the variables are within our expectation and block the execution if otherwise.

If the normalized SQL statement is not in the allowable list, the system check against another user supplied list of regular expressions. If the normalized SQL statement does not match to any regular expression in this list, we will then block this SQL execution. This design allows the system to handle exceptional case that might not be compatible with current algorithm of variable normalization. Since system checks against the regular expression list after variable

normalization, therefore, attackers should not be able to by pass the authorization process. And since most SQL statements do not need to be match against the regular expression, performance impact should be minimal.

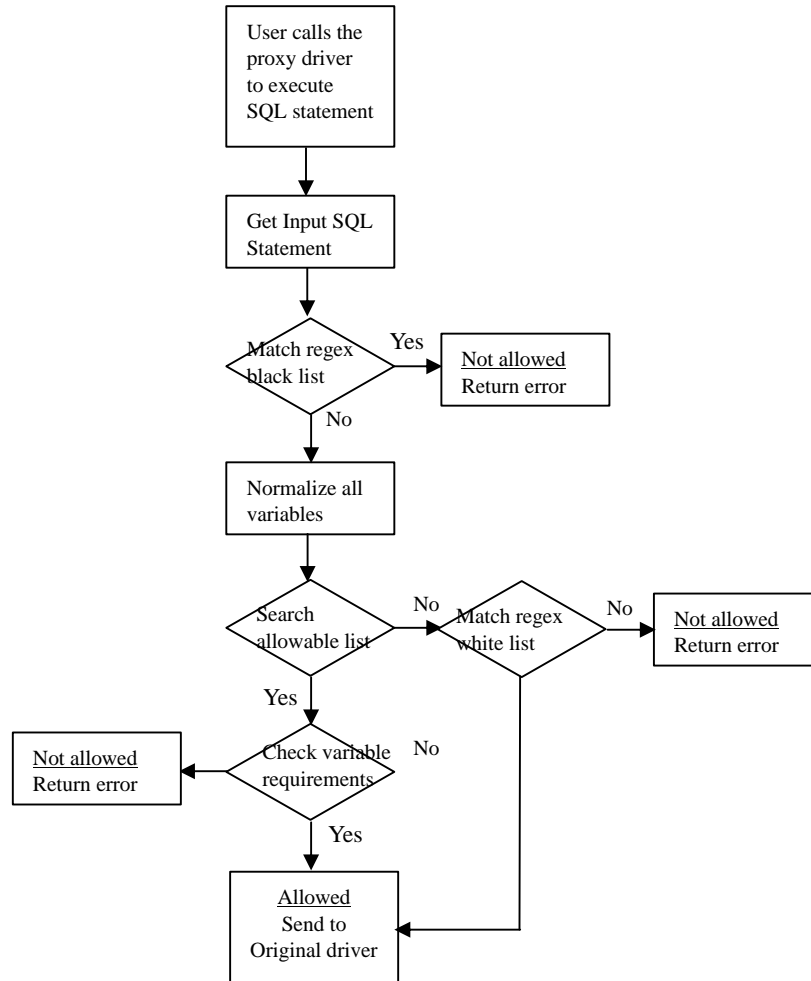
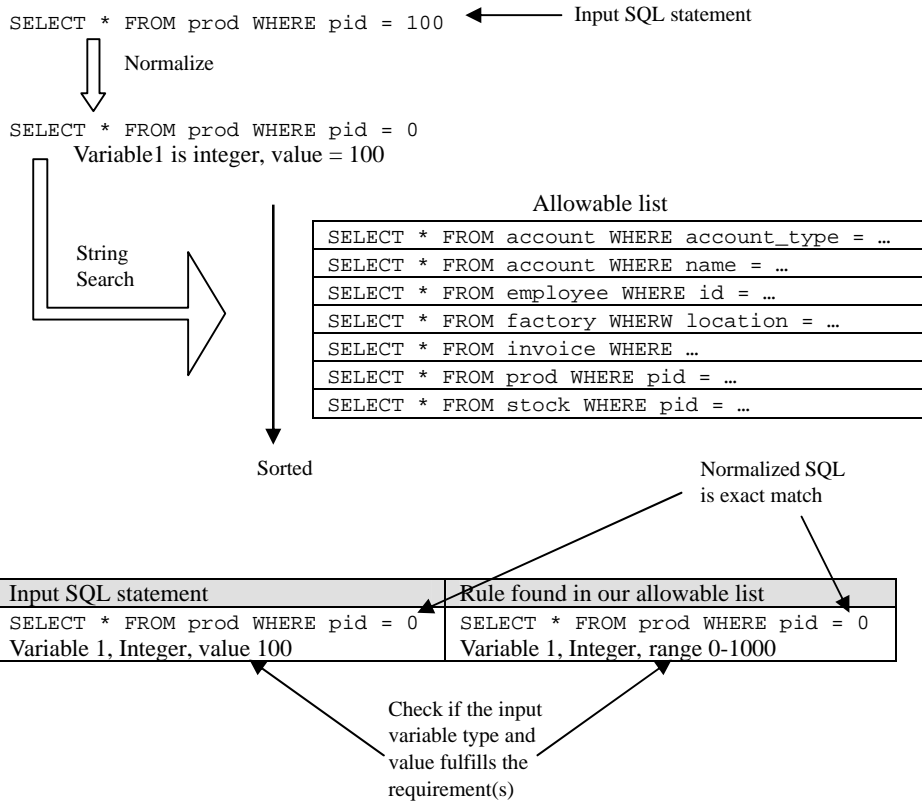


Fig 2. The flow of executing a SQL statement

Example



5. System Limitation

There are some types of SQL statements that cannot be handled by variable normalization effectively. For example, consider a web page allowing user to select the product type by using a multi-line selection box as shown in Fig 3.

The client application “may” generate a SQL statement as follows:

```
SELECT * FROM tbl_prod WHERE prod_name LIKE '%hello%' AND prod_type in ('CD', 'DVD', 'GAMES')
```

And the normalized SQL statement will be

```
SELECT * FROM tbl_prod WHERE prod_name LIKE 'a' AND prod_type in ('a', 'a', 'a')
```

Product Name Search:	<input type="text" value="hello"/>
Product Type:	<div style="border: 1px solid black; padding: 2px;"> <div style="background-color: #e0e0e0; padding: 2px;">Book</div> <div style="background-color: #000080; color: white; padding: 2px;">CD</div> <div style="background-color: #000080; color: white; padding: 2px;">DVD</div> <div style="background-color: #000080; color: white; padding: 2px;">Games</div> <div style="background-color: #e0e0e0; padding: 2px;">Software</div> </div>
<input type="button" value="Submit"/> <input type="button" value="Reset"/>	

Fig 3. A web page input with a multi-line selection box

However, if the user selects 2 product types only, then the normalized SQL statement will be different (only two 'a' terms at the end)

```
SELECT * FROM tbl_prod WHERE prod_name LIKE 'a' AND prod_type in ('a', 'a')
```

In order for SQLBlock to be able to handle these SQL statements, SQLBlock has to learn all the variants of the SQL statement. That is, SQLBlock has to learn all of the 5 SQL statements in this example. This is practical only if there are limited (and expectable) number of items in the selection box.

Another solution to handle this special type of SQL statement is by regular expression white list. However, since regular expression operation is much more CPU intensive, and more important, may lead to security vulnerability if not securely designed, we should only use regular expression if strictly necessary.

6. Performance evaluation

The major tasks for SQLBlock are as follows:

1. Normalize the SQL statement
2. Search if the normalized SQL statement is in our allowable list
3. Match the normalized SQL statement against the regular expression list if it is not in our allowable list

The normalization process is in fact, very simple, we just need to replace all quoted strings with 'a' and all numbers with '0'. The program code should be very simple. And since most SQL statements should be only around 100-1000 bytes long, the normalization process should be negligible.

As for searching the normalized SQL statement, because we will sort the allowable list before use, the searching is $O(\log n)$ operation. For typical database application with less than 1000 SQL actions, the search time is also negligible.

However, the performance would be greatly affected if there were many SQL statements need to be authorized by regular expressions. However, since most SQL statements can be handled

by variable normalization, we expect there will only be 5% of SQL statements need to be authorized by regular expression.

7. Related Technology

SQL injection likes buffer overflow attack, can of cause be solved by better programming practices [3] like code review, input validation and SQL parameter binding. There are also automatic source code scanning tools [4] to check for specific class of programming error.

Web base SQL injection attack can also be solved by web application gateway, which implements user input validation.

Network base intrusion detection tools (e.g. SNORT), can detection certain types of SQL injection attacks (both at HTTP protocol layer or database connection).

SQLrand [6] protects from SQL injection by randomizing the SQL statement, creating instances of the language that are unpredictable to the attacker. It is implemented as database server proxy but requires source code modification.

Parasoft Automated Error Prevention [7] technology protects from SQL injection by ensuring all SQL statements are executing as parameter binding (JDBC preparedStatement) and are not dynamically created.

8. Conclusion

We presented “variable normalization” for SQL statements, which can extract the basic structure of a SQL statement. If SQL injection happens, the structure of the SQL statement will be altered and hence normalized SQL statement will also be altered and we will be able to detect it. We use this method to implement SQLBlock, a database connectivity layer proxy driver that can block SQL injection attacks.

SQLBlock has very minimal overall performance impact. Theoretically, it works will all database servers without the need to change the client source code. Auto-learning the allowable list makes the system easy to deploy even for complex clients that will issue many different SQL commands. And since SQLBlock is a connectivity layer proxy, it works even for SSL web applications. We believe SQLBlock is an effective and practical solution to solve this class of attacks.

Reference:

1. Chris Anley. Advanced SQL Injection In SQL Server Applications.
http://www.nextgenss.com/papers/advanced_sql_injection.pdf
2. Kevin Spett. SQL Injection: Are you web application vulnerable?
<http://www.spidynamics.com/support/whitepapers/WhitepaperSQLInjection.pdf>
3. Microsoft. Writing More Secure IIS Applications
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/iissdk/iis/securing_iis_content.asp
4. Sanctum. Ensuring Security in the Web Application Development Lifecycle
<http://www.watchfire.com/resources/ensuring-security-wdlc.pdf>
5. StephenW. Boyd and Angelos D. Keromytis. SQLrand: Preventing SQL Injection

- Attacks. <http://www1.cs.columbia.edu/~angelos/Papers/sqlrand.pdf>
6. D. Litchfield. Web Application Disassembly with ODBC Error Messages.
<http://www.nextgenss.com/papers/webappdis.doc>
 7. Parasoft. Automated Error Prevention
http://www.parasoft.com/jsp/printables/WhitePaper_AEP_Security_255.255.pdf