**Jaharsh Samayan - 1210406220**

**Vahini Mohan - 1210414124**

**PROJECT REPORT**

**Problem Description:**

The goal of the project was to build a security library that helps in preventing SQL injection vulnerabilities which causes unauthorized personnel to gain access to sensitive information stored on a database server. Malicious input for SQL query could be provided through user input in forms, manipulating URLS, cookies, referrer header etc;

**Implementation:**

We wrote down a small crawler (attached in folder) in python using scrapy framework seeding from websites like w3schools, kali linux , acm.org to return top results with the keywords sql ,injection and attacks. We crawled through hundreds of web sites to gain information on understanding the different methods through which sql injections can occur and made a list of all possible injection mechanism. We then wrote down a sanitazation function that sanitizes the user input covering each of the different possibility that has been discovered. Our sanitization function covered a wide range of attacks .On research, we found the following class of SQL injection:

* Tautologies – ex: user input = d’ or 1=1 # - This type of sql injection always executes due to the OR condition. It is mostly used in bypassing authentication, identifying injectable parameters, extracting data
* Illegal/logically incorrect queries – The error caused by the query helps in revealing the metadata about the database used. It is used a precursor to perform future injection.
* Multiple queries/Piggy backed queries – ex: userinput = value’; drop table tblname; # - When the mysqli\_mutli\_query is used to execute the query, malicious query could be included as part of user input.
* Union query – ex: a' union select content from files #"; - This type of injection is useful in returning data from other tables in addition to the one intended by the developer
* URL encoding – The sql is injection into URL. Ex: %27%20UNION%20SELECT%20password%20FROM%20Users%20WHERE%20name%3D%27admin%27--
* Character encoding – the user input is encoding using the char() method. Ex: ' UNION SELECT password FROM Users WHERE name=char(114,111,111,116)--
* Hex encoding – Hexa decimal encoding is used to replace the input. Ex: Select user from users where name = 726F6F74
* Injection through cookies – This type of injection is difficult to detect if it is encoded in a particular base format.
* Injection through HTTP headers like X-Forwarded-For, User-Agent, Referer etc;
* Column truncation – ex : if username in database is 10 chars in length and the application does not have limit, and user tries to register with a username like “admin               x” with 11 characters in length, the last ‘x’ would be truncated in database, thereby storing admin and acquiring administrative privileges.
* Second order SQL injection – the malicious input(code) is stored in the database and later used to perform a database attack

We have built a library in PHP for MYSQL database which provides two functions that could be used by the developer in his code thereby helping in preventing most of the SQL injection vulnerabilities intended through the user input.

Two functions implemented are:

* Sanitize\_input($con,$input,$type) : This method takes in database connection details, the user input, data type of the input. It check for all the possible vulnerabilities that could be injected depending on the data type and sanitizes the input for variations of all the above discussed cases using mysql\_real\_escape\_string() so that it would be treated as exact input and not as an SQL query. It also takes care of URL encoded input.
* pwi($con ,$query,$input,) : We came across and studied more about PHP Data Objects (PDO) extension which defines a lightweight, consistent interface for accessing databases in PHP. This helps us define a clear boundary between the query and user input . Thus we found that no matter what method of injection we try since we have clearly set the boundaries for the query to be executed the sql injection did not work including second order sql injection. We felt this approach to be more reliable and secure .The challenges we faced in working towards this approach was to construct dynamic PDO equivalent for a given query during run time and at the same time keeping the effort from developer as minimal as possible.

Keeping these two aspects in mind, we started to work and wrote our own user defined function pwi() .The pwi() function accepts the connection details , user query and its input parameters as arguments. Our function parses the given input query (it can also be any complex nested query ) and dynamically binds the input parameters passed to construct an equivalent PDO object . We safely execute this and return the results of the executed query back to the function callee similar to what would have happened in a conventional way . A simple example

Eg: if you were to execute

$res = mysqli\_query($con,"select content from files where name = '${\_POST['name']}' and password = '${\_POST['password']}'") or die(mysqli\_error());

You would simply be writing it as

prevention($con,” select content from files where name = %s and password = %s'" , ,  ${\_POST['name']}  ,${\_POST['password']});

as simple as writing a printf statement ; it is even simpler than writing it the conventional way .Our function handles different types of exceptions and returns its appropriate error message in case of any. This method would prevent both the first order and second order SQL injection.

A simple **diff** command between actual code and the code after implementing our function gave us encouraging results of less than 6% change compared to the actual code.

A wide range of test cases over which we have tested varying including nested SQL injection have been attached as a separate file .

**Developer Usage:**

The developer needs to use the library by calling any of the function according to his convenience.

* For using the sanitize\_input($con,$input,$type) function, the developer needs to pass the connection details, data type and the input he fetches using $\_POST or $\_GET. The sanitized input returned by the method could be used in the developer’s query.
* For using the pwi($con ,$query,$input) function, the developer needs to pass the connection object, query and the user input as simple as that.It works exactly as if the developer wrote his conventional execute query statement.

**Advantages of the library:**

1. Using the sanitize\_input() –

* Reduces the overhead on passing the inputs to the method.
* Less changes required in the developer’s code.
* Mysql\_query() is slightly faster compared to executing PDO’s

1. Using the Make\_PDO()-

* Guarantees complete security by preventing all the possible first order and second order sql injections.
* Provides great portability as it provides single interface across multiple databases. Useful in case of projects using multiple databases.

**Limitation of the library:**

1. Using the sanitize\_input() –

* This method has covered all the possible kinds of scenarios we have come across but this might imply a guarantee to cover all the possible scenarios of injecting the vulnerabilities with complete security.
* If the developer had already sanitized the input before calling the method, then it would a double sanitization on the input which would break the protection.
* Type of base encoding of the cookie value needs to be provided
* The limitation of passing the connection details could be overcome as most of the real time projects will have multiple files and a common database connection file would be used by all of them.

1. Using the Make\_PDO() –

* The limitation in using the method is passing the query as well. But the developer needs to make this tradeoff to make his application secure which is an integral part of any application looked up by the users in today’s world.
* He might be passing the connection object again , but this could be avoided in
* Prepared statement is a bit slower than the mysql\_\*
* The limitation of passing the connection details could be overcome as most of the real time projects will have multiple files and a common database connection file would be used by all of them.

**Future Work:**

This library could be extended such that it takes in the developer’s code as input, reads every statement delimited by a semicolon, converts every sql query into a prepared statement on the fly and replaces the mysql execute statement with the prepared statement. This approach guarantees complete security to the developer’s application without any overhead of modifying the code by the developer.

**References**

[1] [*http://www.cc.gatech.edu/~orso/papers/halfond.viegas.orso.ISSSE06.pdf*](http://www.cc.gatech.edu/~orso/papers/halfond.viegas.orso.ISSSE06.pdf)

[2] [*https://www.owasp.org/index.php/Testing\_for\_SQL\_Injection\_(OTG-INPVAL-005)#Example\_1\_.28classical\_SQL\_Injection.29:*](https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-005)%23Example_1_.28classical_SQL_Injection.29:)

[3] <https://haiderm.com/column-truncation-sql-injection-vulnerability/>

[4] <http://resources.infosecinstitute.com/sql-injection-http-headers/>

[5] <https://haiderm.com/second-order-sql-injection-explained-example/>