

Project : Lab for SQLi Vulnerabilities

Abhinav Shaw

[Introduction](#)

[Background](#)

[Task 1 - Setting up Database](#)

[Task 2 - Setting up Application](#)

[Task 3 - SQLi vulnerability at application layer](#)

[SQL Attack](#)

[Prevention](#)

[Task 4 - SQLi vulnerability dynamic sql](#)

[SQL Attack](#)

[Prevention](#)

[Things we learned](#)

Introduction

The goal of the lab is to study and simulate SQL injection attack on database servers. SQLi attacks can exploit vulnerabilities in the whole application stack, from application api to database. To perform SQLi, extensive knowledge of SQL , database engine and application api is required. This lab explains the different ways through which SQL databases can be exploited and how we can prevent SQL injection at multiple layers of the application making it extremely difficult for an attacker to effectively gain access to the data.

Background

Databases come in different flavours, namely, SQL server (Popular RDBMS by Microsoft), PostgreSQL (Free database), MySQL , Oracle (PL SQL) etc. These database have different engines and the ways in which an attacker might exploit these databases may differ. For the lab SQL Server has been chosen as the database. SQL server comes from the family of Transact-SQL (T-SQL) which is different from other major SQL flavor Procedural-SQL (PL-SQL) used by oracle. These flavors differ in the syntax, language capabilities, database object organization and transaction control. Generally speaking T-SQL has ways of executing dynamic sql statements. Dynamic sql statements are statements that are prepared as a string first and then that string, which contains various commands is executed. This is one of the many points of SQLi vulnerability.

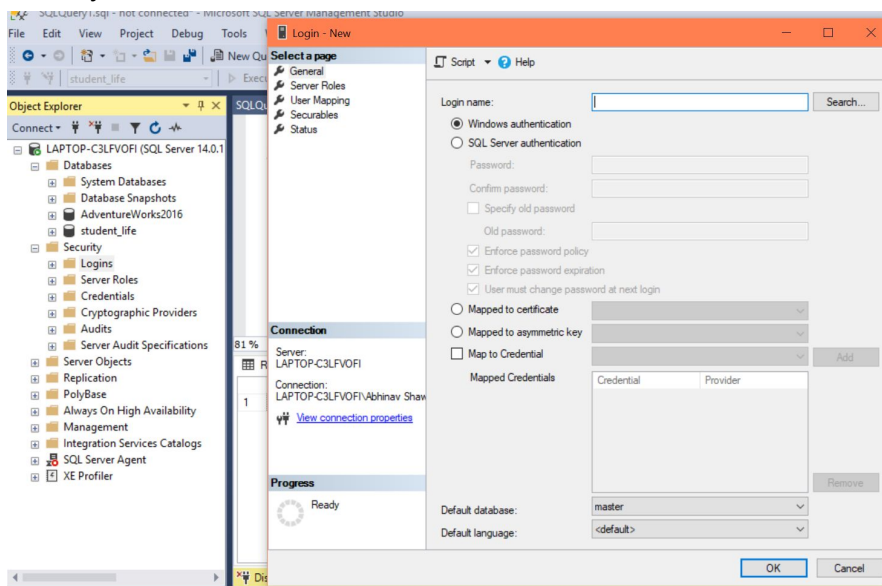
Usually a web application has an application layer that communicates with the database to get data. This data could be obtained either through **Stored Procedure Calls** or via **Query Calls** which are generated in the application itself. Query Calls by the application layer is another point of vulnerability, as these queries are, again, first prepared as strings and then sent to the database system to be executed.

Another important factor to look into is **Access Control**. Access control limits the permissions to the users that have an account in the database. Web application also have user accounts in the database which they use to query the database. This user account should only be given the permissions it requires. Dangerous permissions like Insert/Update/Delete/Alter should never be granted. Granting proper permission can limit the chances of a SQLi attack.

Task 1 - Setting up Database

Environment setup -

- Download and install SQL server from <https://go.microsoft.com/fwlink/?linkid=853016> This comes with an IDE SQL Server Management Studio which is the best IDE for SQL server.
- Once SQL server has been downloaded, you need to create a user account that your application can access. In SQL server management studio sign in using windows authentication (you PC Password). This account has all privileges that are required by you. Locate the object explorer from left side and navigate to "security". Right Click security and create a new login.



- Choose the username password as required. Then goto Databases/user and add your Login to the users of the database.
- To programmatically do all this user instructions in here - <https://searchsqlserver.techtarget.com/tip/Mapping-user-and-login-security>

- Run the following script to create the table that we'll be using -

```
CREATE Database student_life

CREATE TABLE [dbo].[stress_details](
    [student_id] [int] NOT NULL,
    [epoch_time] [bigint] NOT NULL,
    [response_time] [datetime] NULL,
    [latitude] [decimal](9, 6) NULL,
    [longitude] [decimal](9, 6) NULL,
    [stress_level] [tinyint] NULL
) ON [PRIMARY]
GO
```

- Grant permissions to the user that you created, this is done by connecting to the database via windows authentication. Execute the following command -

```
use your_database
Grant select, execute to your_username
```

- We are granting **select** and **execute** permissions to this user. This will be key for avoiding SQLi when we will revoke permission from the user.
- Once the user, database and the table has been created we can move onto writing a DB connector and application layer for the Lab.

Task 2 - Setting up Application

Now that we have created a database which we can query, we need to setup an application layer. The application layer contain APIs which upon being called return the data in some format, usually JSON. internally the API will make a database call by connecting to it. In our lab we are using SQL alchemy to connect to the database.

DB connector / Application Layer

- Create a python script (file ending with .py) named database_connector.py and write the following code -

```
import pandas as pd
# Create a connection with SQL server to get data.
def exec_sql_query(query, param=None):

    from sqlalchemy import create_engine
    import urllib
    params = urllib.parse.quote_plus("DRIVER={SQL Server Native Client
11.0};SERVER=LAPTOP-C3LFVOFI;DATABASE=student_life;UID=web_usr;PWD=26592385
")
```

```

engine = create_engine("mssql+pyodbc:///odbc_connect=%s" % params)
connection = engine.raw_connection()

try:
    cursor = connection.cursor()
    if(param):
        cursor.execute(query, param)
    else :
        cursor.execute(query)

    results = cursor.fetchall()
    columns = [column[0] for column in cursor.description]
    df = pd.DataFrame.from_records(results, columns=columns)
    cursor.close()
    connection.commit()
finally:
    connection.close()

del engine
return df

```

- The above script connects to the Database with an ODBC connection via SQL Alchemy and execute the sql command given to it.
- Make sure you change the connector string to the id pass you assigned in your database
"DRIVER={SQL Server Native Client 11.0};SERVER=LAPTOP-C3LFVOFI;DATABASE=your_database;UID=your_user;PWD=your_pass" , replace the the following appropriately.
 - your_database - with the name of the database that you have created, in my case it was student_life.
 - your_user - the user that was created as login in sql server. You need to make sure that the login is mapped to the database.
 - your_pass - the password of the user that you entered while creating the login.
- In the same folder in another script named **"sqli_script_query_gen_at_app.py"** add -

```

from flask import Flask
from flask import abort
import json as json
from database_connector import exec_sql_query

app = Flask(__name__)

def query_generator(student_id):

```

```

sql = "select * from stress_details where student_id = " + student_id
stress_level = exec_sql_query(sql)

return stress_level.to_json(orient='records')

@app.route('/sqli_query_gen_at_app/<string:student_id>', methods=['GET'])
def get_task(student_id):
    result = query_generator(student_id)
    if not result:
        abort(404)
    return result

if __name__ == '__main__':
    app.run(debug=True)

```

- This is a rest API at route `/sqli_query_gen_at_app/<string:student_id>`
- Run the script by running “python sqli_script_query_gen_at_app.py”
- Once the script is running it can be accessed at “localhost:5000/sqli/sqli_query_gen_at_app/student_id”
- If the student Id exists in the table, you get its data as a json.

Task 3 - SQLi vulnerability at application layer

Now that we have a way to query the database we can try different ways of sql injection. The first point of vulnerability for a SQL attack is at the application layer. This happens when the application generates the query and executes it instead of making a stored procedure call. This is inherently dangerous as the database user that the application uses needs to be granted additional permissions like we did. We granted our user both **SELECT** and **EXECUTE** permissions. In production applications, only EXECUTE permission is preferred.

SQL Attack


- Now, we run the script `sqli_script_query_gen_at_app.py` created in **task 2**.
- Run command in cmd - `python sqli_script_query_gen_at_app.py` this will create a server and expose an api which can be accessed at “/sqli_query_gen_at_app/**student_id**” where `student_id` is the student for which you want the data.

- Enter student id 5 and see what is the response, must get something like this -



```
[
  - {
    student_id: 5,
    epoch_time: 1364122436,
    response_time: 1364122436000,
    latitude: 44,
    longitude: -72,
    stress_level: 4
  },
  - {
    student_id: 5,
    epoch_time: 1364122432,
    response_time: 1364122432000,
    latitude: 44,
    longitude: -72,
    stress_level: 4
  },
]
```

- Now, to attack the server, we need to fool the app. While the python script is running enter in the url “http://127.0.0.1:5000/sqli_query_gen_at_app/5 or 1 = 1” and you get this -



```
[
  - {
    student_id: 4,
    epoch_time: 1367949703,
    response_time: 1367949703000,
    latitude: 44,
    longitude: -72,
    stress_level: 2
  },
  - {
    student_id: 4,
    epoch_time: 1368079040,
    response_time: 1368079040000,
    latitude: 44,
    longitude: -72,
    stress_level: 3
  },
  - {
    student_id: 5,
    epoch_time: 1364122436,
    response_time: 1364122436000,
    latitude: 44,
    longitude: -72,
    stress_level: 4
  },
  - {
    student_id: 5,
    epoch_time: 1364122432,
    response_time: 1364122432000,
    latitude: 44,
    longitude: -72,
    stress_level: 4
  },
]
```

- If you see closely, you got data for student id 4 as well. Infact, the server returned data for all students as if it was as good as running “SELECT * FROM Tabl_name”.

Prevention

This type of attack can be prevented by

- Proper type checking, we were accepting **string** at the application, if we configure it to only accept **int**, then our application will break and it will not make a database call thus keeping the data secure.
- Limited access to user. For this in sql execute `revoke select from your_user` This removes select permission from the user and you won't get any data.

- Using stored procedure calls instead of these naked queries to the database. To do this, run the following script in sql server management studies after selecting your database.

```
CREATE PROCEDURE get_student_stress_level_by_id @student_id INT = NULL
AS
BEGIN
    SELECT student_id
        ,cast(response_time AS DATE) AS [date]
        ,min(adjusted_stress_level) min_stress_level
        ,max(adjusted_stress_level) max_stress_level
        ,round(avg(adjusted_stress_level), 0) avg_stress_level
    FROM stress_details a
    JOIN stress_level_master b ON a.stress_level = b.id
    WHERE response_time <= '2013-05-15'
        AND student_id = @student_id
    GROUP BY student_id
        ,cast(response_time AS DATE)
    ORDER BY DATE
END
```

- This will create a stored procedure which will take **student_id** as a parameter and return the data for only that student.
- We are not done yet, we need to make necessary changes in our app so it calls the stored procedure instead of making a query. In the same folder create a new python script names “sqli_script_proc_call.py” and write the following code in it -

```
import Flask
from flask import abort
import json as json
import database_connector
from database_connector import exec_sql_query

app = Flask(__name__)

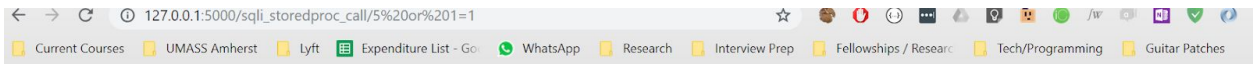
@app.route('/sqli_storedproc_call/<string:student_id>', methods=['GET'])
def get_task(student_id):
    result = exec_sql_query('get_student_stress_level_by_id ?',
[student_id])

    if result.empty:
        abort(404)

    return result.to_json(orient='records')
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

- This script exposes an endpoint at “/sqli_storedproc_call/student_id” where student_id is the student for which we want the data.
- If you access the url at “http://127.0.0.1:5000/sqli_storedproc_call/5” you'll get results as expected, when you try to attack the server by “http://127.0.0.1:5000/sqli_storedproc_call/5 or 1=1” you get an error -



pyodbc.DataError

pyodbc.DataError: ('22018', '[22018] [Microsoft][SQL Server Native Client 11.0]Invalid character value for cast specification (0) (SQLExecDirectW)')

Traceback (most recent call last)

```
File "C:\Users\Abhinav Shaw\Anaconda3\lib\site-packages\flask\app.py", line 1997, in __call__
    return self.wsgi_app(environ, start_response)
File "C:\Users\Abhinav Shaw\Anaconda3\lib\site-packages\flask\app.py", line 1985, in wsgi_app
    response = self.handle_exception(e)
File "C:\Users\Abhinav Shaw\Anaconda3\lib\site-packages\flask\app.py", line 1540, in handle_exception
    reraise(exc_type, exc_value, tb)
```

Task 4 - SQLi Vulnerability Dynamic SQL

Sometimes you need to generate the query dynamically at the database level. This is done by preparing a string of the desired query and then executing it **EXEC (@prepare_string)**. This however leaves the database vulnerable and is prone to SQL injections.

SQL Attack

First we will write a procedure that does this type of dynamic query generation then exploit it through the url.

- Execute the following code in sql, it will create the required procedure for the experiment

```
CREATE PROCEDURE get_student_stress_level_by_id_bad_dynamic_sql @student_id
varchar(20) = NULL
AS
BEGIN

declare @sql varchar(2000)

select @sql = 'SELECT student_id
,cast(response_time AS DATE) AS [date]
,min(adjusted_stress_level) min_stress_level
```



```

        ,max(adjusted_stress_level) max_stress_level
        ,round(avg(adjusted_stress_level), 0) avg_stress_level
    FROM stress_details a
    JOIN stress_level_master b ON a.stress_level = b.id
    WHERE response_time < = '2013-05-15'
        AND student_id = '+' @student_id + '
    GROUP BY student_id
        ,cast(response_time AS DATE)
    ORDER BY DATE'

exec (@sql)

END

```

- Once we have the procedure in place we need to handle the app accordingly. Create a python script named “**sqli_script_bad_dynamic_sql.py**”, why do we name it “**bad**”, because doing dynamic queries in sql like this is a bad practice.

```

from flask import Flask
from flask import abort
import json as json
import database_connector
from database_connector import exec_sql_query

app = Flask(__name__)

@app.route('/sqli_bad_dynamic_sql/<string:student_id>', methods=['GET'])
def get_task(student_id):
    result = exec_sql_query('get_student_stress_level_by_id_bad_dynamic_sql
?', student_id)
    if result.empty:
        abort(404)

    return result.to_json(orient='records')

if __name__ == '__main__':
    app.run(debug=True)

```

- Type in command line / terminal `python sqli_script_bad_dynamic_sql.py` to run the script. Enter “http://127.0.0.1:5000/sqli_bad_dynamic_sql/5” in browser to get the data. You’ll get the data for student 5. Now to attack the server we enter “http://127.0.0.1:5000/sqli_bad_dynamic_sql/5 or `1=1`” and we get all the data.

Prevention

SQL attack at dynamic query generation could be prevented by using **sp_executesql** instead. This is a functionality provided by SQL server and other databases have their own ways of dealing with this. `sp_execute sql` parametrizes the string preparation, thus adding another layer of type check which again is really difficult to get around.

- Execute the following in sql to create the required stored procedure.

```
CREATE PROCEDURE get_student_stress_level_by_id_good_dynamic_sql
@student_id nvarchar(20) = NULL
AS
BEGIN

DECLARE @sql nvarchar(4000)
DECLARE @param_definition nVARCHAR(50) = '@student_id nvarchar(20)'

select @sql = 'SELECT student_id
                ,cast(response_time AS DATE) AS [date]
                ,min(adjusted_stress_level) min_stress_level
                ,max(adjusted_stress_level) max_stress_level
                ,round(avg(adjusted_stress_level), 0) avg_stress_level
FROM stress_details a
JOIN stress_level_master b ON a.stress_level = b.id
WHERE response_time < = ''2013-05-15''
      AND student_id = @student_id
GROUP BY student_id
      ,cast(response_time AS DATE)
ORDER BY DATE'

EXECUTE sp_executesql @sql, @param_definition, @student_id = @student_id

END
```

- You see how you `sp_executesql` accepts a `@param_definition` because it is mapping the parameter and doing an implicit cast to int, which is derived to what field it is equated to.
- Create another python script named “**sqli_script_good_dynamic_sql.py**” the good stands for good practice. Write following code in it -

```
from flask import Flask
from flask import abort
import json as json
import database_connector
from database_connector import exec_sql_query
```

```

app = Flask(__name__)

@app.route('/sqli_good_dynamic_sql/<string:student_id>', methods=['GET'])
def get_task(student_id):
    result =
exec_sql_query('get_student_stress_level_by_id_good_dynamic_sql ?',
student_id)
    if result.empty:
        abort(404)

    return result.to_json(orient='records')

if __name__ == '__main__':
    app.run(debug=True)

```

- The run it by python **sqli_script_good_dynamic_sql.py** and access data at “**http://127.0.0.1:5000/sqli_good_dynamic_sql/5**” you get results as expected, when when you try to attack the server by “**http://127.0.0.1:5000/sqli_good_dynamic_sql/5 or 1=1**” you get an error, this happens because of the binding by sp_executesql.



pyodbc.DataError

pyodbc.DataError: ('22018', "[22018] [Microsoft][SQL Server Native Client 11.0][SQL Server]Conversion failed when converting the nvarchar value '5 or 1=1' to data type int. (245) (SQLExecDirectW)")

Traceback (most recent call last)

File "C:\Users\Abhinav Shaw\Anaconda3\lib\site-packages\flask\app.py", line 1997, in __call__

return self.wsgi_app(environ, start_response)

File "C:\Users\Abhinav Shaw\Anaconda3\lib\site-packages\flask\app.py", line 1985, in wsgi_app

Things we learned

- Setting up a database on sql server.
- Setting up REST apis.
- Exploiting sql vulnerabilities when apps generate their own queries and how to prevent it.
- Writing stored procedures in SQL and using them at the web applications.
- Exploiting sql vulnerabilities in stored procedures that generate sql queries dynamically and are not safe and how to prevent the via sp_executesql which binds parameters adding another layer of implicit type check.

Note: Python scripts, sql scripts for Creating the table and adding data to it has been attached.