**INTRODUCTION**

The aim of this study is improve knowledge about SQL Injection Vulnerabilities.

**DETAILS**

**What is SQLi?**

SQL injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It generally allows an attacker to view data that they are not normally able to retrieve. This might include data belonging to other users, or any other data that the application itself is able to access. This means that with SQLi attacker can compremise sensitive information about users of an application such as credit cards, passwords etc. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior.

In some situations, an attacker can escalate an SQL injection attack to compromise the underlying server or other back-end infrastructure, or perform a denial-of-service attack. There is an article in the reference part about how we can use SQLi for DoS purposes. After we find the vulnerable column we can inject this paylaod to that column = select tab1 from (select decode(encode(convert(compress(post) using latin1),des\_encrypt(concat(post,post,post,post),8)),des\_encrypt(sha1(concat(post,post,post,post)),9)) as tab1 from table\_1)a;

Also for DoS purposes we can also use Sleep to make connections live for long that will also accomplish the task. Using Sleep we can also Pool out the connection in ASP.net where by default maximum 100 connections are allowed at a time in 30 seconds. so if we can make our connection live using Sleep command it wont allow the server to reply other users.

**SQL injection examples**

here are a wide variety of SQL injection vulnerabilities, attacks, and techniques, which arise in different situations. Some common SQL injection examples include:

* Retrieving hidden data, where you can modify an SQL query to return additional results.
* Subverting application logic, where you can change a query to interfere with the application's logic.
* Union Attacks, where you can retrieve data from different database tables.
* Examining the database, where you can extract information about the version and structure of the database.
* Blind SQL injection, where the results of a query you control are not returned in the application's responses.

**1-) Retrieving Hidden data**

Consider a shopping application that displays products in different categories. When the user clicks on the Gifts category, their browser requests the URL:

https://insecure-website.com/products?category=Gifts

This causes the application to make an SQL query to retrieve details of the relevant products from the database:

SELECT \* FROM products WHERE category = 'Gifts' AND released = 1

This SQL query asks the database to return:

* all details (\*)
* from the products table
* where the category is Gifts
* and released is 1.

The restriction released = 1 is being used to hide products that are not released. For unreleased products, presumably released = 0.

The application doesn't implement any defenses against SQL injection attacks, so an attacker can construct an attack like:

https://insecure-website.com/products?category=Gifts'--

This results in the SQL query:

SELECT \* FROM products WHERE category = 'Gifts'--' AND released = 1

The key thing here is that the double-dash sequence -- is a comment indicator in SQL, and means that the rest of the query is interpreted as a comment. This effectively removes the remainder of the query, so it no longer includes AND released = 1. This means that all products are displayed, including unreleased products.

Going further, an attacker can cause the application to display all the products in any category, including categories that they don't know about:

https://insecure-website.com/products?category=Gifts'+OR+1=1--

This results in the SQL query:

SELECT \* FROM products WHERE category = 'Gifts' OR 1=1--' AND released = 1

The modified query will return all items where either the category is Gifts, or 1 is equal to 1. Since 1=1 is always true, the query will return all items.

**LAB 1-)** This lab contains an SQL injection vulnerability in the product category filter. When the user selects a category, the application carries out an SQL query like the following:

SELECT \* FROM products WHERE category = 'Gifts' AND released = 1

To solve the lab, perform an SQL injection attack that causes the application to display details of all products in any category, both released and unreleased.

1-) I tried to filter the products so maybe I can see any sql query for filtering process.



2-) There were no SQL query, but in the URL we can see that category=Lifestyle parameter. So maybe this parameter is used for input as in the SQL query. Thus, I will try to manipulate it.

Graphical user interface, text, email

Description automatically generated

3-) And The lab is completed

A picture containing text

Description automatically generated

**2-) Subverting application logic**

Consider an application that lets users log in with a username and password. If a user submits the username wiener and the password bluecheese, the application checks the credentials by performing the following SQL query:

SELECT \* FROM users WHERE username = 'wiener' AND password = 'bluecheese'

If the query returns the details of a user, then the login is successful. Otherwise, it is rejected.

Here, an attacker can log in as any user without a password simply by using the SQL comment sequence -- to remove the password check from the WHERE clause of the query. For example, submitting the username administrator'-- and a blank password results in the following query:

SELECT \* FROM users WHERE username = 'administrator'--' AND password = ''

This query returns the user whose username is administrator and successfully logs the attacker in as that user.

**LAB 2-)** This lab contains an SQL injection vulnerability in the login function. To solve the lab, perform an SQL injection attack that logs in to the application as the administrator user.

1-) There is a my account page in the application. we want to bypass the authorization in the application. So I capture and will modify the request according to that.

Graphical user interface, text, application, email

Description automatically generated

2-) Lab is done.

A picture containing text

Description automatically generated

**3-) Retrieving data from other database tables**

In cases where the results of an SQL query are returned within the application's responses, an attacker can leverage an SQL injection vulnerability to retrieve data from other tables within the database. This is done using the UNION keyword, which lets you execute an additional SELECT query and append the results to the original query.

For example, if an application executes the following query containing the user input "Gifts":

SELECT name, description FROM products WHERE category = 'Gifts'

then an attacker can submit the input:

' UNION SELECT username, password FROM users--

This will cause the application to return all usernames and passwords along with the names and descriptions of products.

**\*\*\*SQL Injection UNION Attacks**

**STEPS WHEN PROCESSING SQLI UNION ATTACK**

**1-) Find how many columns are returned from the original query with NULL queries**

**2-) Find how many of them suitable with string with replacing null values with ‘a’**

**3-) Use malicious payload on the one that can contain string values.**

When an application is vulnerable to SQL injection and the results of the query are returned within the application's responses, the UNION keyword can be used to retrieve data from other tables within the database. This results in an SQL injection UNION attack.

The UNION keyword lets you execute one or more additional SELECT queries and append the results to the original query. For example:

SELECT a, b FROM table1 UNION SELECT c, d FROM table2

This SQL query will return a single result set with two columns, containing values from columns a and b in table1 and columns c and d in table2.

For a UNION query to work, two key requirements must be met:

* The individual queries must return the same number of columns.
* The data types in each column must be compatible between the individual queries.

To carry out an SQL injection UNION attack, you need to ensure that your attack meets these two requirements. This generally involves figuring out:

* How many columns are being returned from the original query? (This can be found with ‘Union Select Null,Null,Null--%20 (count of the null words is changes according to the columns that table has. If we find the true column count, then the app will return true. After that we will try to find a column which has a type that is compatible with string (varchar, long etc.) by changing null value to ‘a’ one column at a time to see which one is compatible with string. After we find the column that we are searching for, then we can inject a malicious sql payload in to that column like: ‘Union select Null,username || ‘~’ || password from users--)
* Which columns returned from the original query are of a suitable data type to hold the results from the injected query?

**Determining The Number Of Columns Required In An SQL Injection UNION Attack**

Submitting a series of UNION SELECT payloads specifying a different number of null values is a perfect method for determining how many columns are being returned from the original query.

' UNION SELECT NULL--

' UNION SELECT NULL,NULL--

' UNION SELECT NULL,NULL,NULL--

etc.

If the number of nulls does not match the number of columns, the database returns an error, such as:

All queries combined using a UNION, INTERSECT or EXCEPT operator must have an equal number of expressions in their target lists.

Again, the application might actually return this error message, or might just return a generic error or no results. When the number of nulls matches the number of columns, the database returns an additional row in the result set, containing null values in each column. The effect on the resulting HTTP response depends on the application's code. Sometimes the null values might trigger a different error, such as a NullPointerException. Worst case, the response might be indistinguishable from that which is caused by an incorrect number of nulls, making this method of determining the column count ineffective.

**LAB 3-) SQL injection UNION attack, determining the number of columns returned by the query**

This lab contains an SQL injection vulnerability in the product category filter. The results from the query are returned in the application's response, so you can use a UNION attack to retrieve data from other tables. The first step of such an attack is to determine the number of columns that are being returned by the query. You will then use this technique in subsequent labs to construct the full attack.

To solve the lab, determine the number of columns returned by the query by performing an SQL injection UNION attack that returns an additional row containing null values.

1-) First I tried to capture the filtering request and try to inject Union query. At first 2 Null was not enough. So I increase it accordingly and saw that original query returns 3 row. And the lab is completed.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, Word

Description automatically generated with medium confidence

* **Important Note:** The reason for using NULL as the values returned from the injected SELECT query is that the data types in each column must be compatible between the original and the injected queries. Since NULL is convertible to every commonly used data type, using NULL maximizes the chance that the payload will succeed when the column count is correct.
* On Oracle, every SELECT query must use the FROM keyword and specify a valid table. There is a built-in table on Oracle called dual which can be used for this purpose. So the injected queries on Oracle would need to look like:
* ' UNION SELECT NULL FROM DUAL—%
* To comment out the remainder of the query use **“--%20 or --+”** This comment indicator is suitable for all database types. Use these as a comment indicatior.

**Finding Columns With A Useful Data Type In An SQL Injection UNION Attack**

The reason for performing an SQL injection UNION attack is to be able to retrieve the results from an injected query. Generally, the interesting data that you want to retrieve will be in string form, so you need to find one or more columns in the original query results whose data type is, or is compatible with, string data.

Having already determined the number of required columns, you can probe each column to test whether it can hold string data by submitting a series of UNION SELECT payloads that place a string value into each column in turn. For example, if the query returns four columns, you would submit:

' UNION SELECT 'a',NULL,NULL,NULL--

' UNION SELECT NULL,'a',NULL,NULL--

' UNION SELECT NULL,NULL,'a',NULL--

' UNION SELECT NULL,NULL,NULL,'a'--

If the data type of a column is not compatible with string data, the injected query will cause a database error, such as:

Conversion failed when converting the varchar value 'a' to data type int.

If an error does not occur, and the application's response contains some additional content including the injected string value, then the relevant column is suitable for retrieving string data.

**\*\*\*Important Note:** Data types that are compatible with strings are **CHAR, VARCHAR, BINARY, VARBINARY, BLOB, TEXT, ENUM, and SET**.

**LAB 4-) SQL injection UNION attack, finding a column containing text**

This lab contains an SQL injection vulnerability in the product category filter. The results from the query are returned in the application's response, so you can use a UNION attack to retrieve data from other tables. To construct such an attack, you first need to determine the number of columns returned by the query. You can do this using a technique you learned in a previous lab. The next step is to identify a column that is compatible with string data. The lab will provide a random value that you need to make appear within the query results. To solve the lab, perform an SQL injection UNION attack that returns an additional row containing the value provided. This technique helps you determine which columns are compatible with string data.

1-) First I started to find how many columns are returning from the original query. After some tries, I saw that 3 columns are returned.

Graphical user interface, text, application, email

Description automatically generated

2-) Then I tried to find which column that returned is-are compatible with string.

Graphical user interface, text, application

Description automatically generated

Not the third one, so maybe the second one?

Graphical user interface, text, application

Description automatically generated

Second one is the one that we are looking for.

3-) Labs ask us to return the string from the query: '3jqSyK'. And When we did that lab is completed.

Graphical user interface, text, email

Description automatically generated

Text

Description automatically generated

**\*\*\*Examining The Database In SQL Injection Attacks**

When exploiting SQL injection vulnerabilities, it is often necessary to gather some information about the database itself. This includes the type and version of the database software, and the contents of the database in terms of which tables and columns it contains.

**Querying the database type and version**

Different databases provide different ways of querying their version. You often need to try out different queries to find one that works, allowing you to determine both the type and version of the database software.

The queries to determine the database version for some popular database types are as follows:

|  |  |
| --- | --- |
| **Database type** | **Query** |
| **Microsoft, MySQL** | **SELECT @@version** |
| **Oracle** | **SELECT \* FROM v$version** |
| **PostgreSQL** | **SELECT version()** |
|  |  |

For example, you could use a UNION attack with the following input:

' UNION SELECT @@version--

This might return output like the following, confirming that the database is Microsoft SQL Server, and the version that is being used:

Microsoft SQL Server 2016 (SP2) (KB4052908) - 13.0.5026.0 (X64)

Mar 18 2018 09:11:49

Copyright (c) Microsoft Corporation

Standard Edition (64-bit) on Windows Server 2016 Standard 10.0 <X64> (Build 14393: )

**LAB 5-) SQL injection attack, querying the database type and version on Oracle**

This lab contains an SQL injection vulnerability in the product category filter. You can use a UNION attack to retrieve the results from an injected query. To solve the lab, display the database version string.

1-) First, I need to find how many columns are returning from original query(2).

2-) Then, I need to find which columns are compatible with strings(both)

3-) Then I need to send this request to the server

Graphical user interface, text

Description automatically generated

A picture containing text

Description automatically generated

Text

Description automatically generated with low confidence

**LAB 6-) SQL injection attack, querying the database type and version on MySQL and Microsoft**

This lab contains an SQL injection vulnerability in the product category filter. You can use a UNION attack to retrieve the results from an injected query. To solve the lab, display the database version string.

1-) Same steps with above lab. Until step 3.

3-) Then I need to send this request to the server

Graphical user interface, text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

**Listing The Contents Of The Database**

Most database types (with the notable exception of Oracle) have a set of views called the information schema which provide information about the database.

We can query information\_schema.tables to list the tables in the database:

SELECT \* FROM information\_schema.tables

This returns output like the following:

TABLE\_CATALOG TABLE\_SCHEMA TABLE\_NAME TABLE\_TYPE

=====================================================

MyDatabase dbo Products BASE TABLE

MyDatabase dbo Users BASE TABLE

MyDatabase dbo Feedback BASE TABLE

This output indicates that there are three tables, called Products, Users, and Feedback.

You can then query information\_schema.columns to list the columns in individual tables:

SELECT \* FROM information\_schema.columns WHERE table\_name = 'Users'

This returns output like the following:

TABLE\_CATALOG TABLE\_SCHEMA TABLE\_NAME COLUMN\_NAME DATA\_TYPE

=================================================================

MyDatabase dbo Users UserId int

MyDatabase dbo Users Username varchar

MyDatabase dbo Users Password varchar

This output shows the columns in the specified table and the data type of each column.

**LAB 7-) SQL injection attack, listing the database contents on non-Oracle databases**

This lab contains an SQL injection vulnerability in the product category filter. The results from the query are returned in the application's response so you can use a UNION attack to retrieve data from other tables.

The application has a login function, and the database contains a table that holds usernames and passwords. You need to determine the name of this table and the columns it contains, then retrieve the contents of the table to obtain the username and password of all users.

To solve the lab, log in as the administrator user.

1-) When I captured The request, as always, I tried to find how many columns are returned from the original query. There were 2 and both of them can contain strings.

2-) After that I try to inject a database reconnaissance query. In this query we need to use 2 column like as original query. One of our columns need to hold tablename datas. So inject a term that can contain tablename which is “table\_name”. From **information.schema.tables**. Most database types (with the notable exception of Oracle) have a set of views called the information schema which provide information about the database.

We can query information\_schema.tables to list the tables in the database:

Graphical user interface, text, email

Description automatically generated

3-) Then show response in browser to see the tablenames (so much tablenames. Lets use the ones with usernames)

Graphical user interface, application

Description automatically generated with medium confidence

4-) There is a table which named as users\_zvqgla. This can contain user credentials. Lets query that database.

Graphical user interface, text, application

Description automatically generated

5-) Responses are contains password\_tgoyhr and username\_isckwy. Lets query those columns.

Graphical user interface, text, chat or text message

Description automatically generated

6-) And we found our answer

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

**Equivalent to Information Schema On Oracle**

On Oracle, we can obtain the same information with slightly different queries.

we can list tables by querying all\_tables:

SELECT \* FROM all\_tables

And we can list columns by querying all\_tab\_columns:

SELECT \* FROM all\_tab\_columns WHERE table\_name = 'USERS'

**LAB 7-) SQL injection attack, listing the database contents on Oracle**

This lab contains an Sql injection vulnerability in the product category filter. The results from the query are returned in the application's response so you can use a UNION attack to retrieve data from other tables.

The application has a login function, and the database contains a table that holds usernames and passwords. You need to determine the name of this table and the columns it contains, then retrieve the contents of the table to obtain the username and password of all users.

To solve the lab, log in as the administrator user.

1-) Watchin same steps with above lab but syntax differs.

Graphical user interface, text, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, application

Description automatically generated

**Using An SQL Injection UNION Attack To Retrieve Interesting Data**

When you have determined the number of columns returned by the original query and found which columns can hold string data, you are in a position to retrieve interesting data.

Suppose that:

* The original query returns two columns, both of which can hold string data.
* The injection point is a quoted string within the WHERE clause.
* The database contains a table called users with the columns username and password.

In this situation, you can retrieve the contents of the users table by submitting the input:

' UNION SELECT username, password FROM users--

Of course, the crucial information needed to perform this attack is that there is a table called users with two columns called username and password. Without this information, you would be left trying to guess the names of tables and columns. In fact, all modern databases provide ways of examining the database structure, to determine what tables and columns it contains.

**LAB 9-) SQL injection UNION attack, retrieving data from other tables**

This lab contains an SQL injection vulnerability in the product category filter. The results from the query are returned in the application's response, so you can use a UNION attack to retrieve data from other tables. To construct such an attack, you need to combine some of the techniques you learned in previous labs.

The database contains a different table called users, with columns called username and password.

To solve the lab, perform an SQL injection UNION attack that retrieves all usernames and passwords, and use the information to log in as the administrator user.

1-) Default steps are made (null injection, original column count found, string suitable columns found). Then made the request to gather information about tablenames

Graphical user interface, text, email

Description automatically generated

2-) There is a table named users. Lets query it.

Graphical user interface, text, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, email

Description automatically generated

Graphical user interface, text

Description automatically generated with medium confidence

Graphical user interface

Description automatically generated with medium confidence

**Retrieving Multiple Values Within A Single Column**

In the preceding example, suppose instead that the query only returns a single column.

You can easily retrieve multiple values together within this single column by concatenating the values together, ideally including a suitable separator to let you distinguish the combined values. For example, on Oracle you could submit the input:

' UNION SELECT username || '~' || password FROM users--

This uses the double-pipe sequence || which is a string concatenation operator on Oracle. The injected query concatenates together the values of the username and password fields, separated by the ~ character.

The results from the query will let you read all of the usernames and passwords, for example:

...

administrator~s3cure

wiener~peter

carlos~montoya

...

Note that different databases use different syntax to perform string concatenation. We can find some useful payloads at [SQL injection cheat sheet](https://portswigger.net/web-security/sql-injection/cheat-sheet).

**LAB 9-) SQL injection UNION attack, retrieving multiple values in a single column**

This lab contains an SQL injection vulnerability in the product category filter. The results from the query are returned in the application's response so you can use a UNION attack to retrieve data from other tables.

The database contains a different table called users, with columns called username and password.

To solve the lab, perform an SQLi Unionattack th at retrieves all usernames and passwords, and use the information to log in as the administrator user.

1-) Same Default Steps. Only 2nd column can take string data

2-) Lets make our concetenation queryGraphical user interface, text, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, text, application, website

Description automatically generated

**BLIND SQL INJECTION VULNERABILITIES**

Many instances of SQL injection are blind vulnerabilities. This means that the application does not return the results of the SQL query or the details of any database errors within its responses. Blind vulnerabilities can still be exploited to access unauthorized data, but the techniques involved are generally more complicated and difficult to perform.

Depending on the nature of the vulnerability and the database involved, the following techniques can be used to exploit blind SQL injection vulnerabilities:

* We can change the logic of the query to trigger a detectable difference in the application's response depending on the truth of a single condition. This might involve injecting a new condition into some Boolean logic, or conditionally triggering an error such as a divide-by-zero.
* We can conditionally trigger a time delay in the processing of the query, allowing you to infer the truth of the condition based on the time that the application takes to respond.
* We can trigger an out-of-band network interaction, using [OAST](https://portswigger.net/burp/application-security-testing/oast) techniques. This technique is extremely powerful and works in situations where the other techniques do not. Often, you can directly exfiltrate data via the out-of-band channel, for example by placing the data into a DNS lookup for a domain that you control.

**How to detect SQL injection vulnerabilities**

* + Submitting the single quote character **‘** and looking for errors or other anomalies
  + Submitting some SQL-specific syntax that evaluates to the base (original) value of the entry point, and to a different value, and looking for systematic differences in the resulting application responses. For example add ‘AND 1=1 and look for differences on the response
  + Submitting Boolean conditions such as OR 1=1 and OR 1=2, and looking for differences in the application's responses.
  + Submitting payloads designed to trigger time delays when executed within an SQL query, and looking for differences in the time taken to respond.
  + Submitting OAST payloads designed to trigger an out-of-band network interaction when executed within an SQL query, and monitoring for any resulting interactions(Generally for blind SQL injections)

**\*\*\*How To Detect SQL Injection Vulnerabilities**

Most SQL injection vulnerabilities arise within the WHERE clause of a SELECT query. This type of SQL injection is generally well-understood by experienced testers.

But SQL injection vulnerabilities can in principle occur at any location within the query, and within different query types. The most common other locations where SQL injection arises are:

* **In UPDATE statements, within the updated values or the WHERE clause.**
* **In INSERT statements, within the inserted values.**
* **In SELECT statements, within the table or column name.**
* **In SELECT statements, within the ORDER BY clause.**

**SQL Injection In Different Contexts**

In all of the labs so far, we have used the query string to inject our malicious SQL payload. However, it's important to note that we can perform SQL injection attacks using any controllable input that is processed as a SQL query by the application. For example, some websites take input in JSON or XML format and use this to query the database.

These different formats may even provide alternative ways for you to [obfuscate attacks](https://portswigger.net/web-security/reference/obfuscating-attacks-using-encodings" \l "obfuscation-via-xml-encoding) that are otherwise blocked due to WAFs and other defense mechanisms. Weak implementations often just look for common SQL injection keywords within the request, so we may be able to bypass these filters by simply encoding or escaping characters in the prohibited keywords. For example, the following XML-based SQL injection uses an XML escape sequence to encode the S character in SELECT:

<stockCheck>

<productId>

123

</productId>

<storeId>

999 &#x53;ELECT \* FROM information\_schema.tables

</storeId>

</stockCheck>

This will be decoded server-side before being passed to the SQL interpreter.

**Second-Order SQL Injection (Stored SQL Injection)**

First-order SQL injection arises where the application takes user input from an HTTP request and, in the course of processing that request, incorporates the input into an SQL query in an unsafe way.

In second-order SQL injection (also known as stored SQL injection), the application takes user input from an HTTP request and stores it for future use. This is usually done by placing the input into a database, but no vulnerability arises at the point where the data is stored. Later, when handling a different HTTP request, the application retrieves the stored data and incorporates it into an SQL query in an unsafe way.

A picture containing diagram

Description automatically generated

Second-order SQL injection often arises in situations where developers are aware of SQL injection vulnerabilities, and so safely handle the initial placement of the input into the database. When the data is later processed, it is deemed to be safe, since it was previously placed into the database safely. At this point, the data is handled in an unsafe way, because the developer wrongly deems it to be trusted.

**Database-Specific Factors**

Some core features of the SQL language are implemented in the same way across popular database platforms, and so many ways of detecting and exploiting SQL injection vulnerabilities work identically on different types of databases.

However, there are also many differences between common databases. These mean that some techniques for detecting and exploiting SQL injection work differently on different platforms. For example:

* Syntax for string concatenation.
* Comments.
* Batched (or stacked) queries.
* Platform-specific APIs.
* Error messages.

To see some examples of SQL injection payloads on different databases we can visit [SQL injection Cheat sheet on portswigger](https://portswigger.net/web-security/sql-injection/cheat-sheet).

**\*\*\*How To Prevent SQL Injection**

Most instances of SQL injection can be prevented by using parameterized queries (also known as prepared statements) instead of string concatenation within the query.

The following code is vulnerable to SQL injection because the user input is concatenated directly into the query:

String query = "SELECT \* FROM products WHERE category = '"+ input + "'";

Statement statement = connection.createStatement();

ResultSet resultSet = statement.executeQuery(query);

This code can be easily rewritten in a way that prevents the user input from interfering with the query structure:

PreparedStatement statement = connection.prepareStatement("SELECT \* FROM products WHERE category = ?");

statement.setString(1, input);

ResultSet resultSet = statement.executeQuery();

**Parameterized queries can be used for any situation where untrusted input appears as data within the query, including the WHERE clause and values in an INSERT or UPDATE statement. They can't be used to handle untrusted input in other parts of the query, such as table or column names, or the ORDER BY clause. Application functionality that places untrusted data into those parts of the query will need to take a different approach, such as white-listing permitted input values, or using different logic to deliver the required behavior.**

For a parameterized query to be effective in preventing SQL injection, the string that is used in the query must always be a hard-coded constant and must never contain any variable data from any origin. Do not be tempted to decide case-by-case whether an item of data is trusted and continue using string concatenation within the query for cases that are considered safe. It is all too easy to make mistakes about the possible origin of data, or for changes in other code to violate assumptions about what data is tainted.

**REFERENCES**

<http://www.securityidiots.com/Web-Pentest/SQL-Injection/ddos-website-with-sqli-siddos.html>

<https://portswigger.net/web-security/sql-injection/cheat-sheet>

<https://portswigger.net/web-security/sql-injection/examining-the-database>

<https://portswigger.net/web-security/sql-injection/union-attacks>