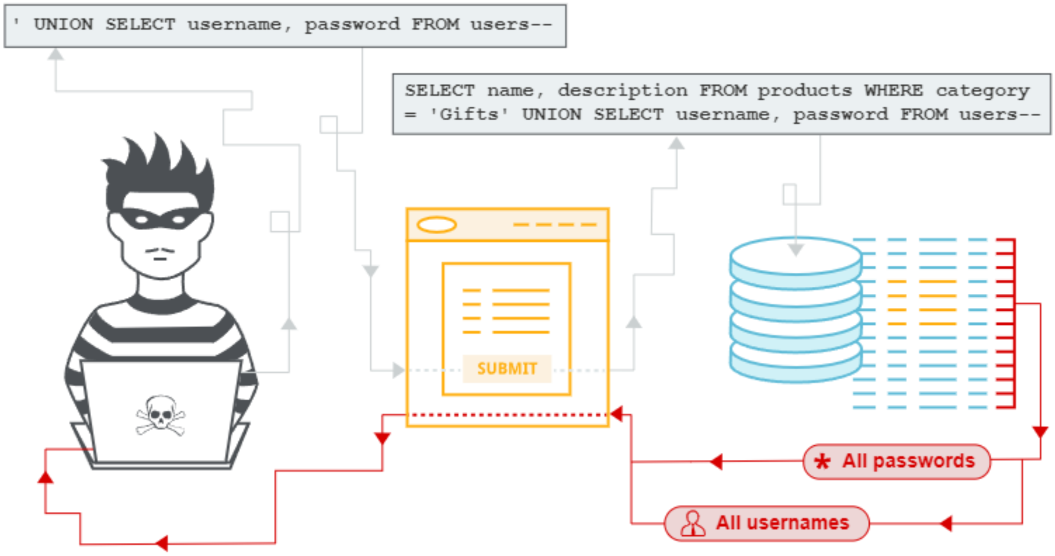
SQL Injection


' UNION SELECT username, password FROM users--
SELECT name, description FROM products WHERE category = 'Gifts' UNION SELECT username, password FROM users--
SUBMIT
All passwords
All usernames

What is 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. It generally allows an attack
er to view data that they are not normally able to retrieve. This might include data belon
ging to other users, or any other data that the application itself is able to access. In m
any cases, an attacker can modify or delete this data, causing persistent changes to the a
pplication's content or behavior.
```

- A web security vulnerability
- Attacker can:
    - interfere with queries to databases through the application
    - View data (which is not normally able to be retrieved)

    - Modify data → Cause persistent changes
    - Escalate → compromise the server/other back-end infrastructure, perform DoS attack

What is the impact?

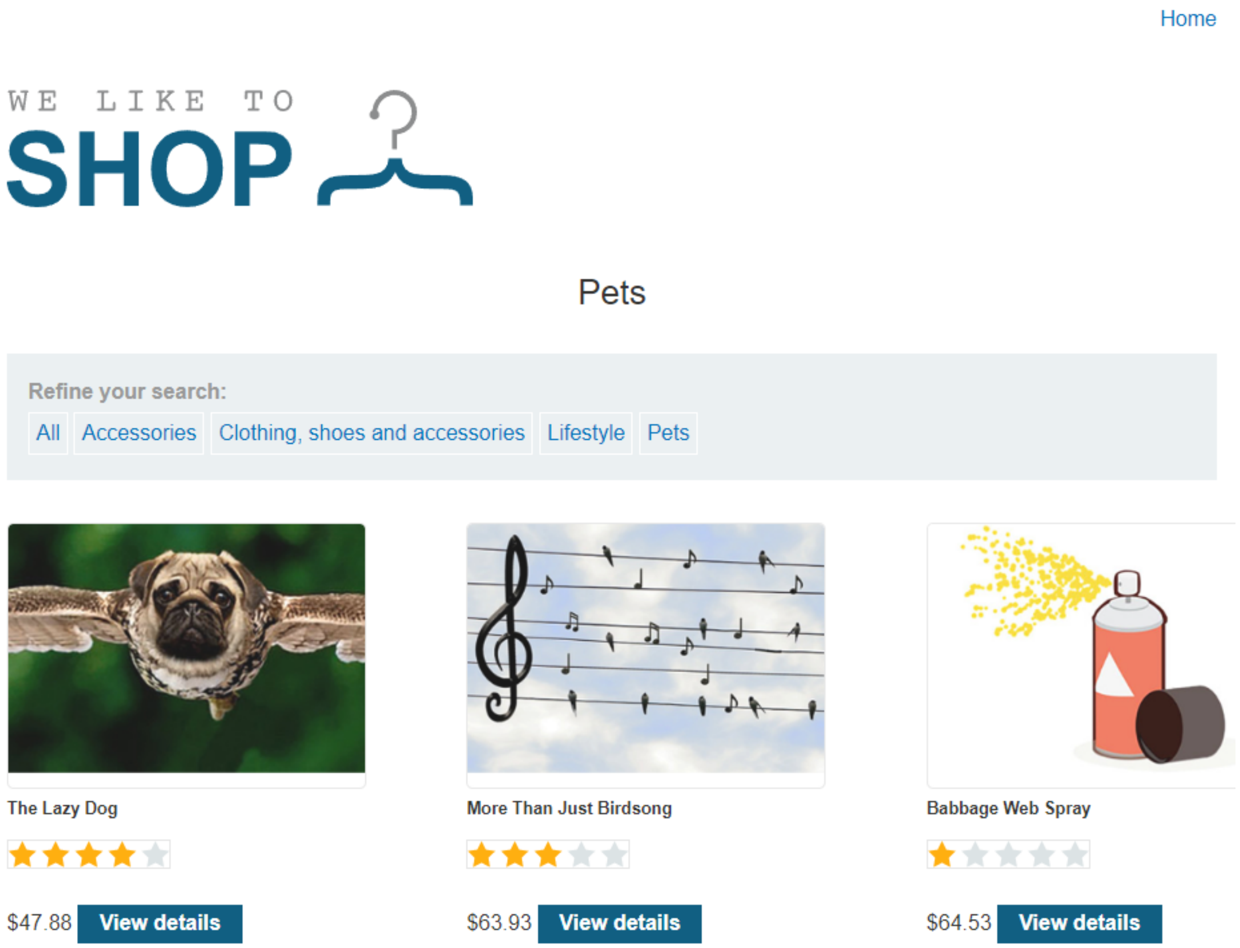
- Unauthorized access to sensitive data (passwords, credits, user information,…)
- Lead to reputational damage (thiệt hại uy tín) and regulatory fines (tiền phạt quy định) caused by data breach
- Lead to long-term compromise (through persistent back-door)

Common SQL Injection

Retrieving hidden data

Modify a SQL query to return additional results

A shopping application displays products in different categories



When user clicks on the Category, their browser requests the URL:

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

And the application makes a SQL query:

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

This SQL query asks the database to return:

- all details (*)
- from the products table

- where the category is Pets
- 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=Pets'--
```

This results in the SQL query:

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

The single quote `'` is used to close the previous quote before `Pets` param → Insert a vulnerable query after that

The double-dash sequence `--` is a comment indicator in SQL → the rest of the query is interpreted as a comment (not query)

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.

Labs: https://portswigger.net/web-security/sql-injection/lab-retrieve-hidden-data

Subverting application logic

Change a query to interfere with the application's 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.

Labs: https://portswigger.net/web-security/sql-injection/lab-login-bypass

Retrieving data from other tables

In cases where the results of a SQL query are returned within the application's responses, an attacker can leverage a 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

Examining the database

Extract information about the version and structure of the database

Following initial identification of a SQL injection vulnerability, it is generally useful to obtain some information about the database itself. This information can often pave the way for further exploitation.

You can query the version details for the database. The way that this is done depends on the database type, so you can infer the database type from whichever technique works. For example, on Oracle you can execute:

```
SELECT * FROM v$version
```

You can also determine what database tables exist, and which columns they contain. For example, on most databases you can execute the following query to list the tables:

```
SELECT * FROM information_schema.tables
```

Examining the database in SQL injection attacks

Blind SQL injection vulnerabilities

The results of a query are not returned in the application’s responses

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:

- You 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.
- You 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.
- You can trigger an out-of-band network interaction, using 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.

Blind SQL Injection

How to detect SQL injection vulnerabilities

The majority of SQL injection vulnerabilities can be found quickly and reliably using Burp Suite's web vulnerability scanner.

SQL injection can be detected manually by using a systematic set of tests against every entry point in the application. This typically involves:

- 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.
- 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 a 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 a SQL query, and monitoring for any resulting interactions.

SQL injection in different parts of the query

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, you've used the query string to inject your malicious SQL payload. However, it's important to note that you 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 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 you 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`:

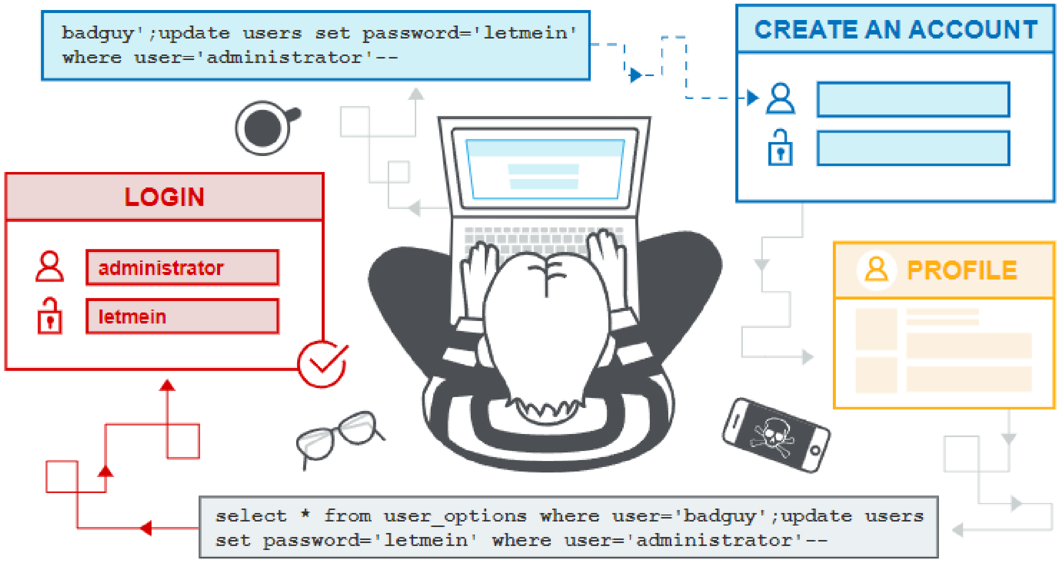
```
<stockCheck>
    <productId>
        123
    </productId>
    <storeId>
        999 &#x53;ELECT * FROM information_schema.tables
    </storeId>
</stockCheck>
```

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

Second-order SQL injection

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 a 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 a SQL query in an unsafe way.



```
badguy';update users set password='letmein'
where user='administrator'--
```

```
select * from user_options where user='badguy';update users
set password='letmein' where user='administrator'--
```

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 database.

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.

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.