SQL injection

In this section, we'll explain what SQL injection (SQLi) is, describe some common examples, explain how to find and exploit various kinds of SQL injection vulnerabilities, and summarize how to prevent SQL injection.

Labs

If you're already familiar with the basic concepts behind SQLi vulnerabilities and just want to practice exploiting them on some realistic, deliberately vulnerable targets, you can access all of the labs in this topic from the link below.

View all SQL injection labs

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 attacker to view data that they are not normally able to retrieve. This might include data belonging to other users, or any other data that the application itself is able to access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior.

In some situations, an attacker can escalate an SQL injection attack to compromise the underlying server or other back-end infrastructure, or perform a denial-of-service attack.

What is the impact of a successful SQL injection attack?

A successful SQL injection attack can result in unauthorized access to sensitive data, such as passwords, credit card details, or personal user information. Many high-profile data breaches in recent years have been the result of SQL injection attacks, leading to reputational damage and regulatory fines. In some cases, an attacker can obtain a persistent backdoor into an organization's systems, leading to a long-term compromise that can go unnoticed for an extended period.

SQL injection examples

There are a wide variety of SQL injection vulnerabilities, attacks, and techniques, which arise in different situations. Some common SQL injection examples include:

- Retrieving hidden data, where you can modify an SQL query to return additional results.
- Subverting application logic, where you can change a query to interfere with the application's logic.
- UNION attacks, where you can retrieve data from different database tables.
- Examining the database, where you can extract information about the version and structure of the database.
- Blind SQL injection, where the results of a query you control are not returned in the application's responses.

Retrieving hidden data

Consider a shopping application that displays products in different categories. When the user clicks on the Gifts category, their browser requests the URL:

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

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

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

This SQL query asks the database to return:

- all details (*)
- from the products table
- where the category is Gifts
- 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=Gifts'--
```

This results in the SQL query:

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

The key thing here is that the double-dash sequence `--` is a comment indicator in SQL, and means that the rest of the query is interpreted as a comment. This effectively removes the remainder of the query, so it no longer includes `AND released = 1`. This means that all products are displayed, including unreleased products.

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.

LAB
APPRENTICE **SQL injection vulnerability in WHERE clause allowing retrieval of hidden data**

Subverting application 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.

LAB
APPRENTICE **SQL injection vulnerability allowing login bypass**

Retrieving data from other database tables

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

Read more

SQL injection UNION attacks

Examining the database

Following initial identification of an 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
```

Read more

Examining the database in SQL injection attacksSQL injection cheat sheet

Blind SQL injection vulnerabilities

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.

Read more

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 an 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 an 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.

LAB
PRACTITIONER **SQL injection with filter bypass via XML encoding**

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 an 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 an SQL query in an unsafe way.

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.

Read more

SQL injection cheat sheet

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.

Which of the following statement is TRUE about SQL Injection?

Answer: C) SQL Injection is a Code Penetration Technique

2. At which of the following stage does SQL Injection occurs?

Answer: D) When the user is asked to input username

3. Which of the following is the type of SQL Injection attack?

Answer: D) All of the above

4. Which of the following is TRUE about the type of SQL Injection attack?

Answer: D) All of the above

5. Select the correct statement which will return all the rows from the Table and then also deletes the Table_Add table?

Answer: A) SELECT * FROM Table; DROP TABLE Table_Add

6. Through which system, we can detect SQL Injection attacks?

Answer: C) Intrusion Detection System

7. Which of the following is TRUE about Intrusion Detection System?

A. Intrusion Detection System is Network-based
B. Intrusion Detection System is Host-based
C. Both A) and B)
D. None of the above

Answer: C) Both A) and B)

8. Network-based IDS can be used to monitor –

Answer: C) All connections to the database server

9. Host-based IDS can be used to monitor –

A. Web server logs
B. When something weird occurs
C. Both A) and B)
D. None of the above

Answer: C) Both A) and B)

10. How can we prevent SQL Injection attack?

A. We should pre-define the input type, input field and length of the user data to validate the input for the user authentication.
B. Access privileges should be restricted for the users
C. Administrator accounts should not be used.
D. All of the above

Answer: D) All of the above

1. What type of attack did the parents in this XKCD comic perform? b. SQL injection

2. Which of the following is not a security exploit? c. Authentication

3. Where does packet sniffing happen? a. Over the network

4. How do you prevent SQL injection? a. Escape queries

5. In cross-site scripting where does the malicious script execute? b. In the user's browser

6. Which of the following is not a CERT security practice? e. None of the above (i.e., all of them are CERT security practices)

7. T or F? Eavesdropping can be countered by using encryption. a. True

8. How do you prevent packet-sniffing exploits? c. Encrypt network communication with SSL

9. Imagine a social networking web app (like Twitter) that allows users to post short blurbs of text. Which type of exploit might be carried out by posting text that contains malicious code?

a. Cross-site scripting b. SQL injection

10. Which of the following are most vulnerable to injection attacks?

a. Session IDs b. Registry keys c. Network communications d. SQL queries based on user input e. None of the above are vulnerable to injection attacks

answer e

11. Which of the following is not a CERT security practice?

a. Use blacklists to restrict access to services and resources

Problem: Consider a web app that displays user posts, similar to Twitter and Facebook. The developers of the web app have accidentally left it vulnerable to cross-site scripting attacks. Explain how you would perform a cross-site scripting attack against the web app. Be thorough in your explanation.

Solution:

I would set up the attack by creating JavaScript that does something harmful. For example, it might redirect the current webpage to one that I made. My web page might try to trick the user into entering his/her username and password, which I would then steal.

To perform the attack, I would make a user post using the web app. My post would contain HTML code that causes my JavaScript to execute when loaded. Thus, any web app user who viewed my post would fall victim to my attack.