Learning Objectives

Learners will be able to…

- List database threats
- Describe best practices of DB security
- Recognize SQL Injection vulnerabilities and implement mitigation techniques for enhanced database security

info

Make Sure You Know

Learners should have a basic grasp of security concepts like authentication, authorization, and encryption.

Database Security Threats

Any database must be protected from the many security threats it faces. The most popular risks are the loss, alteration and theft of important information. Other less critical but equally dangerous threats include performance degradation, privacy violations, and database confidentiality agreements.

Popular database security threats

Below is a list of the most common threats to which databases are vulnerable today.

1. SQL Injection Attacks

This is a type of attack where malicious code is embedded in a front-end (web) application and then transferred to a back-end database. As a result of SQL injections, cybercriminals could gain access to any data stored in the database.

2. Unsecure configuration or wrong permissions

Often, database servers are installed in an organization with their default security settings, which are often never changed. As a result, databases are exposed to attackers who know the default permissions and know-how to exploit them.

Also, assigning users a large number of rights that they do not use can lead to problems and disclosure of confidential information.

The existence of inactive accounts is also a security risk that is often overlooked because attackers can be aware of the existence of these accounts and use them to gain unauthorized access to databases.

3. Database software vulnerabilities

All major commercial database software vendors and open source database management platforms regularly release security patches to address these vulnerabilities, but applying these patches late can increase your vulnerability. But a large amount of time can pass between the release of a patch and the software update on the server, which can be exploited by attackers.

4. Malware

Malicious software is software written specifically to exploit vulnerabilities or otherwise damage a database. The result of a software virus can be the complete destruction of the contents of the database, so it is necessary to have healthy copies of the database.

5. Denial of Service (DoS)

This is a type of attack that affects the availability of the service, it affects the performance of the database server and makes the database service unavailable to users.

For example, if there is a request for very sensitive financial data, and the database is unavailable due to a DoS, then this can lead to a loss of money.

6. Attacks on backups

If the database servers are secured with a high security level, there is a possibility that the backups can be accessed by non-privileged users. In such a situation, there is a risk that unauthorized users can create backup copies and transfer them to their own servers in order to extract sensitive information.

7. Human error

Accidents, weak passwords, password sharing, and other unwise or uninformed user behaviors continue to be the cause a lot of problems.

Security Best Practices

Since databases are almost always available on the network, any security threat to any component within or part of the network infrastructure is also a threat to the database, and any attack that affects a user device or workstation can threaten the database. Thus, database security must go far beyond the DB

When evaluating database security in your environment to decide on your team's top priorities, consider each of the following areas:

- **Physical security**. Whether your database server is on-premise or in a cloud center, it must be located in a secure, controlled area.
- **Network access control**. A practically minimal number of users should have access to the database, and their access should be limited to the minimum levels necessary to do their job. Likewise, network access should be restricted to the minimum required privilege level. If possible, the set of IP addresses that have access to the database should also be limited.
- **End user account/device security**. Always control who and when accesses the database, as well as how and when the data is used. Set up monitoring for unusual activity, such as attempts to connect users who should not have access to the database.
- **Data monitoring solutions** can alert you if data activity is anomalous. For example, the flow of outgoing data from the database has greatly increased, which was not noticed before.
- **Encryption**. All data must be encrypted at rest and in transit. All encryption keys must be handled according to best practices.
- **Database software security**. Always use the latest database management software and apply all patches as soon as they are released.
- **Application/web server security**. Any application or web server that interacts with a database can become an attack vector and should be subjected to ongoing security testing.
- **Auditing**. Performing a database audit is very important and requires regular reading of application and database log files. This log is typically used for auditing purposes such as determining who accessed the database, when it was accessed, and what actions were performed on the database.
- **Backup security**: All backups, copies, or images of the database must be subject to the same (or equally stringent) security controls as the database itself.

What is SQL injection?

An injection is when, instead of the data that is expected of us, we substitute some other data that is formed according to rules different from those that were expected, but still correct within the framework of the task.

This type of vulnerability does not necessarily apply only to SQL, but it is a common type of attack in regards to SQL.

To understand how injections work in general, let's start with a simple example.

Let's imagine that we have a hungry cat that can be fed and that the owner will feed it if a request is sent: **Put X packages of food**, where the **X** parameter can be changed, but the rest of the words cannot be changed.

That is, if you put `X = 5`, then you get the request **Put 5 packages of food** and the owner will put 5 packages of food for the cat. As a result, we will get a happy cat.

However, if we have the same hungry cat and the same type of request **Put X packages of food**, but instead we influence the parameter `X` by specifying **the bulldog and 5**, in the end we get the phrase **Put the bulldog and 5 packages of food**. Of course, nothing good will come of this.

SQL injection 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.

SQL injection usually occurs when you ask a user for input, like their username/userid, and instead of a name/id, the user gives you a SQL statement that you will unknowingly run on your database.
What does SQL Injection look like?
Let's imagine we have the table with stored information about cats and owners.

Name	Breeds	Age	Owner
Whiskers	Sphynx	3	Smith
Felix	Abyssinian	1	Jones
Lucky	Bombay	5	Williams
Oscar	Bengal	4	Taylor

And we have an application, where we can select all cats with a specific age range from user input `ageFromUser`:

```
const query = "SELECT * FROM Cats WHERE Age <= " + ageFromUser
const result = database.executeQuery(query)

// show result to user
```

All works fine while user passes different parameters to `ageFromUser`. For example, passing a `1` will return one row with `Felix`, and for `3`, two rows will be returned - `Felix` and `Whiskers`.

However, a user can pass something like `1 OR 1 = 1`. This will construct the query `SELECT * FROM Cats WHERE Age <= 1 OR 1 = 1`, which returns all records from the database.

A SQL injection attack can occur when:

- Unintended data enters a program from an untrusted source
- A SQL query is used to dynamically modify or return data that is unintended

SQL Injection Examples

SQL Injection Based on 1=1 is Always True

Let’s look at this example from a previous page.

```
"SELECT * FROM Cats WHERE Age <= " + ageFromUser
```

This example can be modified to this:

```
"SELECT * FROM Users WHERE UserId = " + txtUserId;
```

Where `txtUserId` can be something like `105 OR 1=1`.

SQL Injection Based on "“=”" is Always True

```
'SELECT * FROM Users WHERE Name ="' + uName + '" AND Pass ="' +
uPass + '"'
```

In the example above, `uName` and `uPass` are user inputs.

A hacker might get access to user names and passwords in a database by simply inserting `" OR ""="` into the user name or password text box.

The code at the server will create a valid SQL statement like this:

```
SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or
""=""
```

The SQL above is valid and will return all rows from the “Users” table, since `OR ""=""` is always TRUE.

SQL Injection Based on Batched SQL Statements

Most databases support batched SQL statement.

A batch of SQL statements is a group of two or more SQL statements, separated by semicolons.

The SQL statement below will return all rows from the “Users” table, then delete the “Suppliers” table.

```
"SELECT * FROM Users WHERE UserId = " + txtUserId;
```

An attacker can pass something like `105; DROP TABLE Suppliers`.

The valid SQL statement would look like this:

```
SELECT * FROM Users WHERE UserId = 105; DROP TABLE Suppliers;
```

SQL Injection with comments

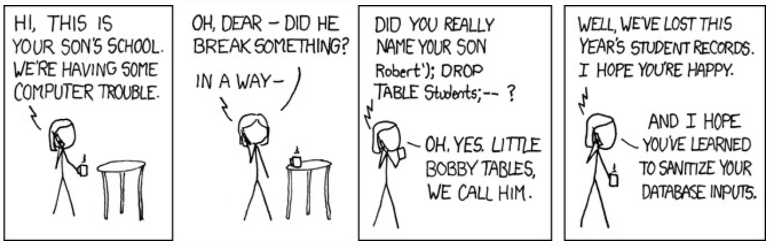
You don't even need to pass something to the `uPass` variable if you provide `uName` with `--` which is a SQL comment. This will cause everything after it to be ignored.

The resulting text will be `" OR ""="--` which produces a valid expression:

```
'SELECT * FROM Users WHERE Name ="" OR ""="--" AND Pass =""'
```

SQL Injection has become a common issue with database-driven web sites. The flaw is easily detected, but easily exploited, and as such, any site or software package with even a minimal user base is likely to be the subject of an attempted attack of this kind.

Essentially, the attack is accomplished by placing a meta character into a data input field which then results in the placement of SQL commands in the control plane which did not exist there before. This flaw depends on the fact that SQL makes no real distinction between the control and data planes.


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.
Comic of mom showing a schools weakness of student records.

SQL Injection Prevention

SQL Injection flaws are introduced when software developers create dynamic database queries constructed with string concatenation which includes user supplied input. Avoiding SQL injection flaws can be simple. For example, developers can either:
1. stop writing dynamic queries with string concatenation; and/or
1. prevent user supplied input which contains malicious SQL from affecting the logic of the executed query.

Unsafe Example:

SQL injection flaws typically look like this (example from a previous page):

```
cursor.execute(
    f"INSERT INTO actor(first_name, last_name) VALUES
('{first_name}', '{last_name}')"
  )
```

The following example is UNSAFE, and would allow an attacker to inject code into the query that would be executed by the database. The unvalidated “first_name” and “last_name” parameter that is simply appended to the query allows an attacker to inject any SQL code they want. Unfortunately, this method for accessing databases is all too common.

Defense Option 1: Prepared Statements (with Parameterized Queries)

The use of prepared statements with variable binding (aka parameterized queries) is how all developers should first be taught how to write database queries. They are simple to write, and easier to understand than dynamic queries. Parameterized queries force the developer to first define all the SQL code, and then pass in each parameter to the query later. This coding style allows the database to distinguish between code and data, regardless of what user input is supplied.

Prepared statements ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker.

Safe example for previous query will be:

```
cursor.execute(
  "INSERT INTO actor(first_name, last_name) VALUES (%s, %s)",
  (first_name, last_name,)
)
```

Defense Option 2: Stored Procedures

Stored procedures are not always safe from SQL injection. However, certain standard stored procedure programming constructs have the same effect as the use of parameterized queries when implemented safely which is the norm for most stored procedure languages.

‘Implemented safely’ means the stored procedure does not include any unsafe dynamic SQL generation. Developers do not usually generate dynamic SQL inside stored procedures. However, it can be done, but should be avoided.

Defense Option 3: Allow-list Input Validation

Various parts of SQL queries aren’t legal locations for the use of bind variables, such as the names of tables or columns, and the sort order indicator (ASC or DESC). In such situations, input validation or query redesign is the most appropriate defense. For the names of tables or columns, ideally those values come from the code, and not from user parameters.

For something simple like a sort order, it would be best if the user supplied input is converted to a boolean, and then that boolean is used to select the safe value to append to the query. This is a very standard need in dynamic query creation.

```
query = "some SQL ... order by Salary " + (sortOrder ? "ASC" :
"DESC");
```

Any time user input can be converted to a non-String, like a date, numeric, boolean, enumerated type, etc. before it is appended to a query, or used to select a value to append to the query, this ensures it is safe to do so.