Graphical user interface, text, application

Description automatically generated

**Description**

A SQL injection is not a new or overly complicated type of attack, yet it continues to sit atop the OWASP Top Ten Application Security Risks after more than 20 years of it having been publicly utilized. This is primarily due to its inherent, relative ease of use, coupled with its severity of impact when directed toward the staggeringly high number of websites with poorly written, vulnerable code.

SQL is a query language that is designed to access, modify, and delete data stored in relational databases. Numerous web applications and websites use SQL databases as their method of data storage. Applications with a higher prevalence of older functional interfaces such as PHP and ASP are relatively more susceptible to SQL injection flaws than applications based on more recent technologies.

Applications are vulnerable to attacks when user-supplied data is not validated, filtered for escape characters or sanitized by the application.

An attacker can use SQL Injection to manipulate an SQL query via the input data from the client to the application, thus forcing the SQL server to execute an unintended operation constructed using untrusted input.

**Impact**

A successful SQL injection attack can result in a malicious user gaining complete access to all data in a database server with the ability to execute unauthorized SQL queries and compromise the confidentiality, integrity, and availability of the application. Depending on the backend DBMS used, and the permissions granted to the user on the Database, a SQL Injection could result in arbitrary file read/write and even remote code execution.

The severity of attacks that have leveraged SQL Injection should not be understated. Notorious breaches, including the devastating and internationally renowned hacks of Sony Pictures and LinkedIn for example, are reported to have been executed using SQL injection.

**Scenarios**

Subverting application logic through SQL can lead to unpredictable outcomes depending on the context of the SQL statement the strategy of the attacker.

There are a well-known exploitation techniques that attackers leverage depending on the vulnerability within the implementation of the code:

* Manipulating an SQL query logic to bypass access controls.
* Retrieving hidden data to return additional results, including data from other tables within the databases e.g. leveraging the UNION keyword.
* Executing arbitrary SQL code in the context of the database whether stacked queries are allowed.
* Accessing files and executing commands in the operating system, depending of the vulnerable code and the database management system.

It is called *blind* SQL Injection when the injection succeeds, but the code doesn't return the result of the manipulated query to the attacker. Blind injections are still exploitable to retrieve the content using timing analysis, content analysis, or other out-of-bound techniques.

The following is a classic example of subverting application logic to bypass access controls.

Usernames and passwords are ubiquitous as the method for logging into applications. In this benign scenario, a user submits the username user and the password secret. The application then performs a SQL query to verify the credentials:

**SELECT** \* **FROM** **users** **WHERE** username = 'user' **AND** **password** = 'secret'

The login is successful if the query returns the details of the user. If the query doesn't return the user details, it is rejected.

By leveraging single quotes and SQL comments (--), it is possible to log in as any user without a password, as the password check from the WHERE clause is removed from the query.

The following example illustrates this in action. By entering administrator'-- in the username field and leaving the password field blank, the SQL statement would result as the following:

**SELECT** \* **FROM** **users** **WHERE** username = 'administrator'*--' AND password = '*

The database evaluates this statement without the commented out part, executing just the first part:

**SELECT** \* **FROM** **users** **WHERE** username = 'administrator'

Since the manipulated query always returns the details of the administrator user, the attacker can successfully login without knowing the correct password.

**Prevention**

To avoid SQL injection vulnerabilities, developers need to use parameterized queries, specifying placeholders for parameters so that they are not considered as a part of the SQL command; rather, as solely data by the database.

When working with legacy systems, developers need to escape inputs before adding them to the query. Object relational mappers (ORMs) make this easier for the developer; however, they are not a panacea, with the underlying mitigations still entirely relevant: **untrusted data needs to be validated, query concatenation should be avoided unless absolutely necessary, and minimizing unnecessary SQL account privileges is crucial**.

**Testing**

Verify that where parameterized or safer mechanisms are not present, context specific output encoding is used to protect against injection attacks, such as the use of SQL escaping to protect against SQL injection.

* **OWASP ASVS**: [5.3.5](https://github.com/OWASP/ASVS/releases/download/v4.0.2_release/OWASP.Application.Security.Verification.Standard.4.0.2-en.pdf)
* **OWASP Testing Guide**: [Testing for SQL Injection](https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05-Testing_for_SQL_Injection.html)

**.NET**

Entity Framework is the primary .NET object-relational mapper (ORM). Currently, Entity Framework 6 is used in .NET Framework projects, while Entity Framework Core is the recommended version for .NET Core applications.

Generally speaking, Entity Framework uses LINQ-to-Entities parametrized queries, and it is not susceptible to traditional SQL injection attacks.

However, Entity Framework does allow for the use of raw SQL queries when working with a relational database, introducing the risk of writing injectable queries. The dangerous methods are:

EF6:

* DBSet.SqlQuery()
* Database.SqlQuery()
* Database.ExecuteSqlCommand()

EF Core:

* FromSql()

**Vulnerability example**

**EF Core**

The following is a basic raw query designed to find users that match the userEmail parameter.

**var** result = context.Users.FromSql($"SELECT \* from Users WHERE email = '{userEmail}';").ToList();

Since the SQL query is built using interpolation, if the userEmail is provided by the user, an attacker could manipulate the query at her/his discretion.

For example, by injecting ' OR 1 OR ' in the userEmail, the query becomes:

**SELECT** \* **from** **Users** **WHERE** email = '' **OR** 1 **OR** '';

The manipulated query returns all the records in the Users table.

**Prevention**

Do not use raw SQL queries. When possible, use LINQ to include functions in the query.

**EF Core**

The query above can be rewritten securely, as follows.

**var** result = context.Users.Where(u => u.email == userEmail).ToList();

**References**

[CWE - CWE-89: Improper Neutralization of Special Elements used in an SQL Command](https://cwe.mitre.org/data/definitions/89.html)

[OWASP - SQL Injection](https://www.owasp.org/index.php/SQL_Injection)

[OWASP - SQL Injection Prevention Cheat Sheet](https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.md)

[Entity Framework Core - Raw SQL Queries](https://docs.microsoft.com/en-us/ef/core/querying/raw-sql)