SQL Injection

How SQL Injection attacks work

- Typically, queries are created dynamically by source code that combines a predetermined query structure with data input from a user.
- For example, suppose a simple web application asks the user to login using a user name and password.
- The user enters "BobJ" for the user name and "password1234" for the password.
- A simple query that incorporates this information and checks it against the database might look as follows. Note that the password is inappropriately passed in clear text solely for purposes of simplifying the example:

```
SELECT * from users WHERE username = 'BobJ' and password =
 'password1234';
```

- In order to construct this query, the application code would characteristically create a string containing the command structure of the query and concatenate the data that the user input. For example, the code might appear as follows (additional spaces added for clarity):

```
String username= getUsername();
String password=  getUserPassword();
String query= "SELECT * from users WHERE username = ' " +
username + " ' and password = ' " + password + " ' ";
```

- If an attacker wished to bypass the password check, the attacker could enter the following string for the username:

  ```
  attacker' OR 1 = 1 -- gonofurther
  ```

- Unless the programmer properly validates the data entered, the query would be constructed as follows:

  ```
  SELECT * from users WHERE username = 'attacker' OR 1 = 1
  -- gonofurther and password = ' '
  ```

- The query now includes an alternative test, 1 = 1, which always resolves to true, and the double dashes following the additional test comments out the remainder of the query.
- Consequently, the database server would retrieve all of the login records, thereby allowing the attacker to login without need of a password.

Solutions

- All current approaches for solving this problem may be divided generally into three classes:

    1. One class of approaches is aimed at detecting queries that contain potentially malicious input and preventing them from being executed.
    2. A second class of approaches use a feature of the database server to separate the command structure from the data in the query, known as a **Prepared Statement**. A variation on this theme includes research that has focused on creating alternative means for achieving this separation through additional software libraries that perform the work, rather than relying solely on the database server. Notably, the use of the **Prepared Statement** is regarded by the security community as the best solution to this problem because it always prevents SQL Injection attacks. **Prepared Statement**'s, however, can degrade performance significantly, however, in certain cases, such as queries that call database server functions-- for example, obtaining the current time. Moreover, **Prepared Statement**'s cannot be used for batch queries (which is different from the execution of a single query using batch data).
    3. Finally, the traditional approach is to sanitize user input by replacing potentially malicious data with benign representations or removing it altogether.

- In short, the three classes of approaches include preventing potentially malicious queries from executing, better separating the command structure from the data, and altering the input. Additionally, several proposals have combined these approaches into hybrid methods.
- Despite all of the means available for eliminating SQL Injection attacks, systems often remain vulnerable today. The primary reasons for continued vulnerability include ignorance and lack of budget to fix code.

Root cause

- The root cause of the problem is that the data and command structure of the SQL query are intermingled within the query statement.
- Hence, there is no guarantee that the query that the client intended will be interpreted the same way by the database server receiving the query.
- If database queries were structured so that their meaning was well defined and not open to interpretation, then such attacks could not occur. Thus, the true answer would be a communication protocol that guarantees uniformity in interpretation.