Notatki - JDBC - cz.3

Spis treści

PreparedStatement

Trochę teraz o security. Zwróć uwagę, że jak na razie cały czas wpisywaliśmy `query`, które nie miały przekazywanych żadnych parametrów. W życiu raczej się takie rzeczy nie zdarzają, raczej będziemy tworzyć zapytania, które będą przyjmowały parametry z kodu Javowego.

Naturalnym wydaje się zatem napisanie czegoś takiego (potrzebujemy skasować użytkownika z naszego sklepu):

```
private static Optional<Integer> deleteUser(final Statement statement, String userName) {
    try {
        String query = "DELETE FROM CUSTOMER WHERE USER_NAME = '" + userName + "';";
        return Optional.of(statement.executeUpdate(query));
    } catch (SQLException e) {
        System.err.printf("Failed to executeUpdate: %s%n", e.getMessage());
    }
    return Optional.empty();
}
```

Wyobraźmy teraz sobie, że mamy napisaną aplikację w taki sposób, że nazwę użytkownika do usunięcia podaje sam użytkownik (w formie potwierdzenia, że na pewno to ten `user_name` ma być usunięty).

No i wszystko w porządku ☺, użytkownik na formularzu strony internetowej wpisuje swój `user_name` do usunięcia, query wygląda wtedy w ten sposób:

```
DELETE FROM CUSTOMER WHERE USER_NAME = 'asterix';
```

W tym momencie bardzo możliwe, że dostaniemy komunikat mówiący o tym, że naruszamy klucz obcy, czyli, że ten użytkownik jest wciąż używany w inych miejscach w bazie danych. Zatem dopiszmy zapytania, które usuwają również tego użytkownika z innych tabel:

```
DELETE FROM OPINION
    WHERE CUSTOMER_ID IN (SELECT ID FROM CUSTOMER WHERE USER_NAME = 'asterix');
DELETE FROM PURCHASE
    WHERE CUSTOMER_ID IN (SELECT ID FROM CUSTOMER WHERE USER_NAME = 'asterix');
DELETE FROM CUSTOMER
    WHERE USER_NAME = 'asterix';
```

I usunięcie przebiegło pomyślnie, usunęliśmy tylko użytkownika o `user_name = asterix`.

Potem przychodzi kolejny użytkownik i wpisuje w parametrze w formularzu coś takiego:

```
whatever' or 1=1 or USER_NAME = 'whateverAgain
```

Wtedy po uzupełnieniu naszego SQL parametrem user_name dostajemy takie SQL:

```
DELETE FROM OPINION
    WHERE CUSTOMER_ID
        IN (SELECT ID FROM CUSTOMER WHERE USER_NAME = 'whatever'
            or 1=1 or USER_NAME = 'whateverAgain');
DELETE FROM PURCHASE
    WHERE CUSTOMER_ID
        IN (SELECT ID FROM CUSTOMER WHERE USER_NAME = 'whatever'
            or 1=1 or USER_NAME = 'whateverAgain');
DELETE FROM CUSTOMER
    WHERE USER_NAME = 'whatever'
        or 1=1 or USER_NAME = 'whateverAgain';
```

Jeżeli spróbujemy wykonać takie zapytanie na bazie danych, to dostaniemy błąd, że nie możemy usunąć rekordów z tabeli CUSTOMER, gdyż istnieją klucze obce w innych tabelach wskazujące na rekordy z tej tabeli, ale skonstruowane zapytanie stara się usunąć nam wszystkich użytkowników z bazy danych.

Czyli gdybyśmy wykonali tego rodzaju akcję na tabeli, która nie ma wskazujących na nią kluczy obcych, np PURCHASE albo OPINION, to takie zapytanie skasowałoby nam wszystkie dane z tabeli. To, co teraz pokazałem nazywa się **SQL Injection** i jest szeroko znanym rodzajem ataku na aplikacje webowe.

Wiesz już, czemu nie należy za pomocą konkatenacji konstruować SQLi, które potem mają zostać wykonane na bazie danych?

Dlatego teraz na białym koniu wjeżdża PreparedStatement i rozwiązuje ten problem i kilka innych:

- **bezpieczeństwo** - jest bezpieczniejsze, ze względu na inną konstrukcję podawania parametrów do zapytania i niestraszne mu **SQL Injection**
- **czytelność** - w przypadku wielu parametrów, PreparedStatement jest czytelniejsze dla osoby czytającej kod
- **wydajność** - PreparedStatement jest bardziej wydajne niż Statement

Wykorzystanie w praktyce

Klasa JdbcPreparedStatementExample

```
public class JdbcPreparedStatementExample {

    public static void main(String[] args) {
        String userName = "asterix";
        String query1 = "DELETE FROM OPINION " +
            "WHERE CUSTOMER_ID IN (SELECT ID FROM CUSTOMER WHERE USER_NAME = ?);";
        String query2 = "DELETE FROM PURCHASE " +
            "WHERE CUSTOMER_ID IN (SELECT ID FROM CUSTOMER WHERE USER_NAME = ?);";
        String query3 = "DELETE FROM CUSTOMER WHERE USER_NAME = ?;";
```

```
        String databaseURL = "jdbc:postgresql://localhost:5432/zajavka";
        String user = "postgres";
        String password = "password";
        try (
            Connection connection = DriverManager.getConnection(databaseURL, user, password);
            PreparedStatement statement1 = connection.prepareStatement(query1);
            PreparedStatement statement2 = connection.prepareStatement(query2);
            PreparedStatement statement3 = connection.prepareStatement(query3)
        ) {
            statement1.setString(1, userName);
            statement2.setString(1, userName);
            statement3.setString(1, userName);

            System.out.println("Changed: " + statement1.executeUpdate());
            System.out.println("Changed: " + statement2.executeUpdate());
            System.out.println("Changed: " + statement3.executeUpdate());

        } catch (Exception e) {
            System.err.printf("Error while working on database: %s%n", e.getMessage());
        }
    }
}
```

Czyli korzystamy z gotowego mechanizmu podstawiania zmiennych `statement.setString(1, userName);` Nie musimy się wtedy martwić o cudzysłowia, apostrofy itp. Jednocześnie zabezpiecza nas to przed atakiem pokazanym poprzednio.