JDBC

Java

Lukas Wais

Codersbay

Version: 22. Mai 2023

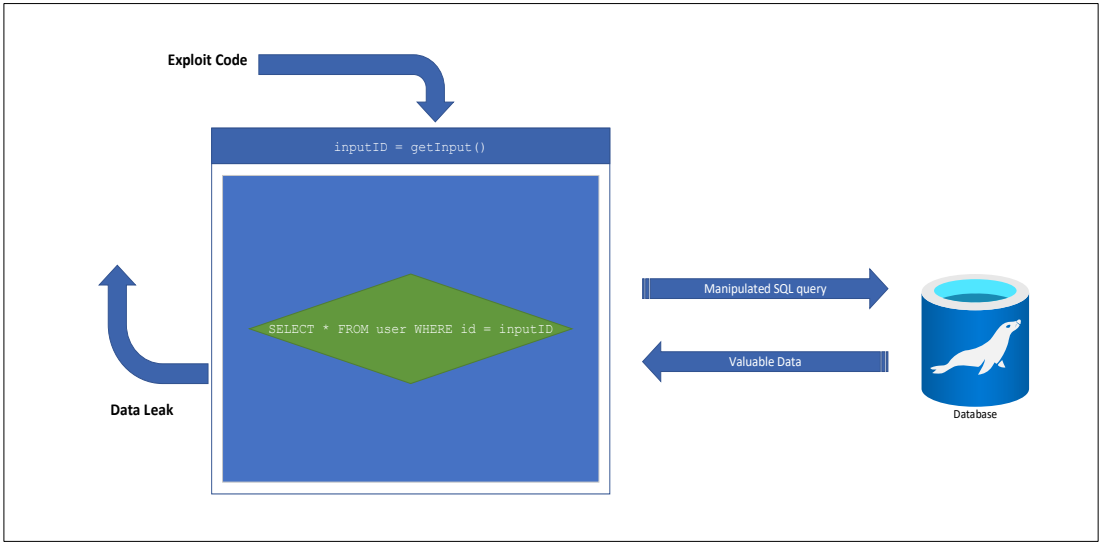
Inhaltsverzeichnis

Datenbankzugriff mit JDBC

Grundlegendes Problem von Code Injektions

- Vermischung von Code und Daten
  - Makros in Office-Dokumenten
  - JavaScript in HTML/PDF
  - Werte in SQL-Abfragen

SQL Injection



Weakness CWE 89

- Code und Daten gemischt
  - Daten unter der Kontrolle des Benutzers (d. h. des Angreifers) werden Teil des SQL-Befehls
- Keine/unzureichende Filterung dieser Daten

CWE 89 ... Common Weakness Enumeration 89.

Anatomie einer SQL Abfrage

Beispiel Blogging Applikation

- Seite zum Anzeigen eines Beitrags
- URL: /blog/article?id=192

```
userID = getParameter("id");
query = "SELECT ..." + id;
```

```
SELECT userID, title, author, text
FROM article WHERE id = 192
```

Einfacher Exploit

```
id = "192 OR 1=1";
```

```
SELECT id, title, author, text
FROM article WHERE id = 192 OR 1=1
```

$1 = 1$ ist immer wahr.

$\rightarrow$ Alle Zeilen werden zurückgegeben.

- (relativ) langweilig, wenn es sich nur um Artikel handelt.
- Problematisch bei Benutzerdaten.
- Ausgabeerzeugung ohne Schleife; einfach LIMIT verwenden.

Verhindern des Einfachen Exploits

Ansatz: Trennung von Daten und Code. Verwendung von Begrenzungszeichen, genannt "Quoting".

```
query = "... id = \'\" + id + \"\'";
```

```
SELECT ... FROM ...
WHERE id = '192 OR 1=1'
```

Abfrage ohne Ergebnis → Angriff schlägt fehl.

Quoting Umgehen

```
id = "192\' OR 1=1"
```

```
SELECT ... FROM ...
WHERE id = '192' OR 1=1'
```

Die Syntax ist invalid, der Angriff funktioniert nicht.
Wir müssen das nachgestellte Hochkomma entfernen.

Quoting Umgehen – 2ter Versuch

```
id = "192\' OR 1=1 --"
```

```
SELECT ... FROM ...
WHERE id = '192' OR 1=1 --'
```

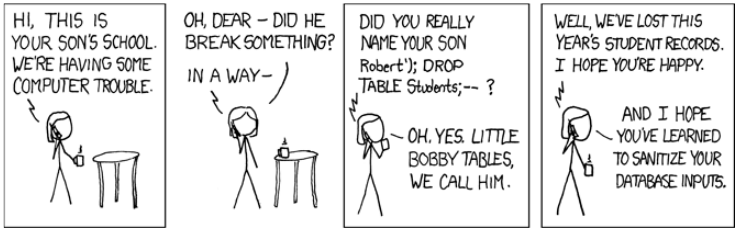
`--` ist ein SQL-Kommentar.

Achtung

Der Angreifer kann beliebigen Code ausführen.

Selbstversuch – Login mit SQL Injection

http://juice-shop.herokuapp.com



Source: xkcd.com

Weitere Kommandos

- (Administrative) Stored Procedures
- DDL Commands (DROP DATABASE , GRANT , ...)

Key Takeaway

Achtung

Daten in SQL Statements sind potentiell immer gefährlich.

Gegenmaßnahmen

- Erkennung (nicht gut genug).
  - Blacklisting: schwierig und fehleranfällig.
- Filterung (nicht gut genug).
  - mit regulären Ausdrücken möglich, aber fehleranfällig.
- Quoting / Escaping (gut genug für Legacy).
  - Vorher kanonisieren, sonst umgehbar mit char().
  - OWASP Enterprise Security API.

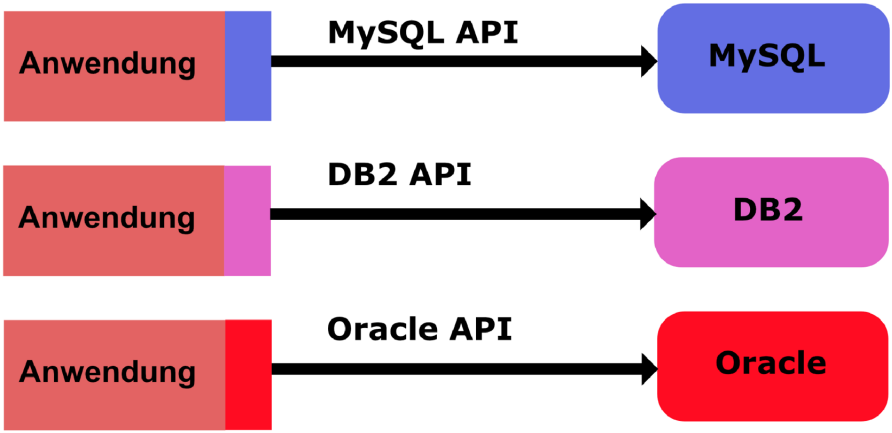
```
ESAPI.encoder.encodeForSQL(new OracleCodec(), param);
```

- Korrekt konstruierte Stored Procedures.

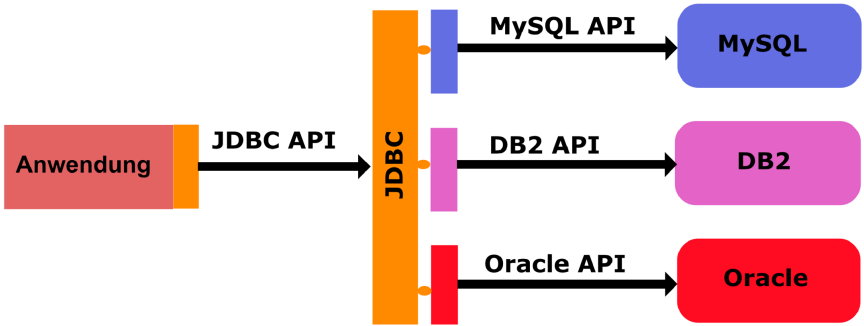
Gegenmaßnahmen

- **Prepared Statements** mit Bindings.
  - Trennung von Code und Daten.
  - Datentypen werden überprüft.
  - Implizites Quoting.
  - Bonus: es ist auch schneller.
- SQL Injection Prevention Cheat Sheet.

Datenbank APIs Ohne JDBC



Datenbank APIs Mit JDBC



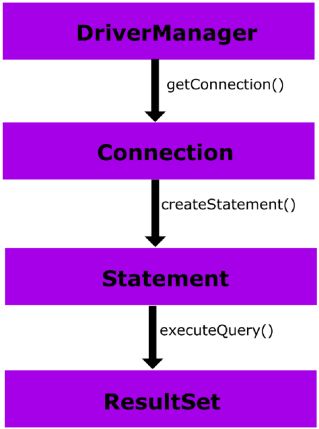
Database Access Interface für Java

- Abstrakt und Datenbankneutral.
- Low Level API: direkte Verwendung von SQL.
- Java Package `java.sql`.
- Entwickelt von Sun Microsystems.

Klassen

- **DriverManager:** Einstiegspunkt, Laden des Treibers.
- **Connection:** Datenbankverbindung.
- **Statement:** Ausführung von Anweisungen über eine Verbindung.
- **ResultSet:** Verwaltet die Ergebnisse einer Abfrage, Zugriff auf einzelne Spalten.

Ablauf


DriverManager
getConnection()
Connection
createStatement()
Statement
executeQuery()
ResultSet

Datenbankverbindungsablauf

- Herstellen einer Verbindung zur DB.
  - Angabe der Verbindungsinformationen.
  - Auswahl und dynamisches Laden des Treibers.
- Senden einer SQL-Anweisung.
  - Definition der Anweisung.
  - Zuweisung von Parametern.
- Verarbeitung der Abfrageergebnisse.
  - Navigation über die Ergebnisrelation.
  - Zugriff auf Spalten.

Datentypkonvertierung MariaDB

Type name	Description	Used for
TYPE_STRING	Zero ended string	char, varchar, text
TYPE_INT	4 bytes integer	int, mediumint, integer
TYPE_SHORT	2 bytes integer	smallint
TYPE_TINY	1 byte integer	tinyint
TYPE_BIGINT	8 bytes integer	bigint, longlong
TYPE_DOUBLE	8 bytes floating point	double, float, real
TYPE_DECIM	Numeric value	decimal, numeric, number
TYPE_DATE	4 bytes integer	date, datetime, time, timestamp, year

Source: https://mariadb.com/kb/en/connect-data-types/

- Dokumentation MariaDB Connector/J.
- Maven Dependency.

Verwendung von SQL-Anweisungen in Java

SQL-Anweisungen sollten aus Performance-Gründen nur sehr sparsam eingesetzt werden, wenn möglich mit dem Datenbankmanagementsystem DBMS arbeiten.
Cursor, Trigger, PLSQL, Exceptions, ...

Übersicht

Sometimes it is more convenient to use a PreparedStatement object for sending SQL statements to the database. This special type of statement is derived from the more general class, Statement, that you already know. If you want to execute a Statement object many times, it usually reduces execution time to use a PreparedStatement object instead.

The main feature of a PreparedStatement object is that, unlike a Statement object, it is given a SQL statement when it is created. The advantage to this is that in most cases, this SQL statement is sent to the DBMS right away, where it is compiled.

As a result, the PreparedStatement object contains not just a SQL statement, but a SQL statement that has been precompiled. This means that when the PreparedStatement is executed, the DBMS can just run the PreparedStatement SQL statement without having to compile it first. Although you can use PreparedStatement objects for SQL statements with no parameters, you probably use them most often for SQL statements that take parameters. The advantage of using SQL statements that take parameters is that you can use the same statement and supply it with different values each time you execute it. Examples of this are in the following sections.

Source: https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html

Vorteil von Prepared Statements

The most important advantage of prepared statements is that they help prevent SQL injection attacks.

Source: https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html

Beispiel

Die folgende Methode, *CoffeesTable.updateCoffeeSales*, speichert die Anzahl der in der aktuellen Woche verkauften Pfund Kaffee in der Spalte SALES für jede Kaffeesorte und aktualisiert die Gesamtzahl der verkauften Pfunde Kaffee in der Spalte TOTAL für jede Kaffeesorte:

Beispiel

```
public void updateCoffeeSales(HashMap<String, Integer> salesForWeek)
↪   throws SQLException {
    String updateString =
      "update COFFEES set SALES = ? where COF_NAME = ?";
    String updateStatement =
      "update COFFEES set TOTAL = TOTAL + ? where COF_NAME = ?";

    try (PreparedStatement updateSales =
    ↪   con.prepareStatement(updateString);
         PreparedStatement updateTotal =
         ↪   con.prepareStatement(updateStatement))

// ...
```

Gesamter Code: https://gist.github.com/x21L/28669a58947e5ff50d38a1407063fb6f

Erstellung eines PreparedStatement Objekts

```
String updateString = "update COFFEES " + "set SALES = ? where COF_NAME =
↪  ?";
// ...
PreparedStatement updateSales = con.prepareStatement(updateString);
```

Bereitstellung von Werten für PreparedStatement-Parameter

Bevor man PreparedStatement Objekte ausführen kann, muss man zuerst Werte anstelle der Fragezeichenplatzhalter (falls vorhanden) angeben. Dies geschieht durch den Aufruf einer der Setter-Methoden, die in der PreparedStatement Klasse definiert sind. Die folgenden Anweisungen liefern die beiden Fragezeichenplatzhalter im PreparedStatement updateSales:

```
updateSales.setInt(1, e.getValue().intValue());
updateSales.setString(2, e.getKey());
```

Achtung, man muss die entsprechende Set-Methode für den verwendeten Datentypen aufrufen.

Schleifen um Werte zu Setzten

Die Methode *CoffeesTable.updateCoffeeSales* verwendet eine for-each-Schleife, um wiederholt Werte in den PreparedStatement-Objekten updateSales und updateTotal zu setzen:

```
for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {
  updateSales.setInt(1, e.getValue().intValue());
  updateSales.setString(2, e.getKey());
  // ...
}
```

Schleifen um Werte zu Setzten

Die Methode *CoffeesTable.updateCoffeeSales* verwendet eine for-each-Schleife, um wiederholt Werte in den PreparedStatement-Objekten updateSales und updateTotal zu setzen:

```
for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {
  updateSales.setInt(1, e.getValue().intValue());
  updateSales.setString(2, e.getKey());
  // ...
}
```

Die Methode *CoffeesTable.updateCoffeeSales* hat ein Argument, HashMap. Jedes Element im HashMap-Argument enthält den Namen einer Kaffeesorte und die Anzahl der Pfunde dieser Kaffeesorte, die in der aktuellen Woche verkauft wurden. Die for-each-Schleife durchläuft jedes Element des HashMap-Arguments und setzt die entsprechenden Fragezeichen-Platzhalter in updateSales und updateTotal.

Ausführen der PreparedStatement Objekte

```
updateSales.setInt(1, e.getValue().intValue());
updateSales.setString(2, e.getKey());
updateSales.executeUpdate(); // <--

updateTotal.setInt(1, e.getValue().intValue());
updateTotal.setString(2, e.getKey());
updateTotal.executeUpdate(); // <--
con.commit();
```

Anmerkung: Zu Beginn von *CoffeesTable.updateCoffeeSales* wird der Auto-Commit-Modus auf false gesetzt: `con.setAutoCommit(false);`.

Wichtige Punkte der Datenbanksicherheit

- Escaping ' und ; ist gut, aber nicht ausreichend!
- Überprüfung aller Eingabedaten anhand einer Whitelist.
  - Setzten einer strikten Längenbeschränkung $\rightarrow$ SQL sind normalerweise (aber nicht immer) lange Zeichenketten.
  - Verifiziere welche Zeichen vorkommen könnten. Etwa Name mit ', wie O'Brian.
- Verifizierung aller übermittelten Datentypen.

Berechtigungen

- Limitierung der Datenbankberechtigungen.
  - DB selbst sollte immer ein separater Benutzer mit den geringsten Rechten sein.
  - Jede Applikation soll ihren eigenen Benutzer haben.
    - Und jede Anwendung die darauf zugreift soll ihren eigen User haben.
    - Beispiel: Backend $\rightarrow$ Schreibrechte, Frontend $\rightarrow$ Leserechte mit eigenen Views, mit den relevanten Daten.

Parameter

- Verwende parametrisierte Abfragen
  - Kreiere keine Queries mittels Konkatenieren.
  - Bereite (prepare) alle Abfragen vor und rufe sie mit den Benutzerinhalten als Parameter auf.
- **Verwende Stored Procedures**
  - Wie parametrisierte Abfragen, aber in der DB gespeichert.
- Indirekte Lösung: ORMs (Object Relational Mappers).
  - Achtung: Sicherstellen, dass Clients die Objekterstellung nicht manipulieren können.

Car Based SQL Injcetion