SQL Injection in Java mit JPA und Hibernate verhindern

entwickler.de – entwickler.de Deine Wissensplattform

[…]Weiterlesen…
Wirft man einen Blick auf die Top-10-Schwachstellen der OWASP [1], sind SQL Injections immer noch in einer prominenten Position zu finden. In diesem Artikel diskutieren wir verschiedene Möglichkeiten, wie SQL Injections effizient vermieden werden können.

Wenn Anwendungen auf Datenbanken zugreifen, bestehen immer wieder hohe Sicherheitsrisiken für die Applikation. Hat ein Angreifer die Möglichkeit, die Datenbankschicht einer Anwendung zu kapern, kann er zwischen mehreren Optionen wählen. Die Daten der gespeicherten Benutzer zu stehlen, um sie mit Spam zu überfluten, ist dabei nicht das schlimmste mögliche Szenario. Noch problematischer wäre es, wenn gespeicherte Zahlungsinformationen missbraucht würden. Eine weitere Variante eines SQL-Injection-Cyberangriffs ist der illegale Zugriff auf eingeschränkte kostenpflichtige Inhalte und/oder Dienste. Wie wir sehen, gibt es viele Gründe, sich um die Sicherheit von (Web-)Anwendungen zu kümmern.

Um eine gut funktionierende Prävention gegen SQL Injections etablieren zu können, müssen wir zunächst verstehen, wie ein solcher Angriff funktioniert und auf welche Punkte wir achten müssen. Kurz gesagt verhält es sich so: Jede Benutzerinteraktion, die die Eingabe ungefiltert in einer SQL-Abfrage verarbeitet, ist ein mögliches Angriffsziel. Die Dateneingabe kann so manipuliert werden, dass die übermittelte

SQL-Abfrage eine andere Logik enthält als das Original. Der folgende Code gibt eine gute Vorstellung davon, was möglich ist:

```
SELECT Username, Password, Role FROM User
  WHERE Username = 'John Doe' AND Password = 'S3cr3t';
SELECT Username, Password, Role FROM Users
  WHERE Username = 'John Doe'; --' AND Password='S3cr3t';
```

Die erste Anweisung zeigt die ursprüngliche Abfrage. Wird die Eingabe für die Variablen Benutzername und Passwort nicht gefiltert, entsteht das klassische Angriffsszenario. Die zweite Abfrage fügt für die Variable Benutzername einen String mit dem Benutzernamen *John Doe* ein und erweitert ihn um die Zeichen *; –*. Diese Anweisung umgeht die *UND*-Verzweigung und gibt in diesem Fall Zugriff auf das Log-in. Mit der Zeichensequenz *,;* schließen Sie die *WHERE*-Anweisung und mit *–* werden alle folgenden Zeichen auskommentiert. Theoretisch ist es möglich, zwischen diesen beiden Zeichenfolgen jeden gültigen SQL-Code auszuführen. Es lässt sich leicht ahnen, welcher Schabernack an dieser Stelle möglich ist.

Mein Plan ist natürlich nicht, zu verbreiten, welche SQL-Befehle die schlimmsten Folgen für das Opfer haben könnten. Bei diesem einfachen Beispiel gehe ich davon aus, dass die Botschaft klar angekommen ist. Wir müssen jede UI-Eingabevariable in unserer Anwendung vor Benutzermanipulation schützen. Auch dann, wenn sie nicht direkt für Datenbankabfragen verwendet werden. Um diese Variablen zu erkennen, ist es immer eine gute Idee, alle vorhandenen Eingabeformulare zu validieren. Doch moderne Anwendungen haben meist mehr als nur ein paar Eingabeformulare. Aus diesem Grund sage ich auch sehr eindringlich: Behalten Sie Ihre REST-Endpunkte im Auge. Oft sind deren Parameter auch mit SQL-Abfragen verbunden.

Security Afternoon

Durch einen stetigen Strom an Releases, neuen Features und spannenden Projekten rückt Security in der IT-Welt gerne einmal in den Hintergrund. Beim Security Afternoon rücken wir mit Michael Kaufmann und Inko Lorch einen ganzen Nachmittag lang die IT-Sicherheit in den Fokus und zeigen, warum es so wichtig ist, Anwendungssicherheit nicht als lästige Fleißaufgabe zu verstehen.

Deshalb sollte die Eingabevalidierung generell Teil des Sicherheitskonzepts sein. Annotationen aus der Spezifikation Bean Validation [2] sind für diesen Zweck sehr mächtig. Beispielsweise sorgt *@NotNull* als Annotation für das Datenfeld im Domänenobjekt dafür, dass das Objekt nur persistiert werden kann, wenn die Variable nicht leer ist. Um die Bean Validation Annotations in Ihrem Java-Projekt zu verwenden, müssen Sie lediglich eine kleine Bibliothek einbinden:

```
<dependency>
  <groupId>org.hibernate.validator</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>${version}</version>
</dependency>
```

Eventuell ist es notwendig, komplexere Datenstrukturen zu validieren. Mit regulären Ausdrücken haben Sie ein weiteres mächtiges Werkzeug an der Hand. Aber seien Sie vorsichtig: Es ist nicht so einfach, korrekt funktionierende RegEx zu schreiben. Schauen wir uns dazu ein kurzes Beispiel an (Listing 1).

Listing 1: Validierung durch reguläre Ausdrücke in Java

```
public static final String RGB_COLOR = "#[0-9a-fA-F]{3,3}([0-9a-fA-F]{3,3})?";

public boolean validate(String content, String regEx) {
  boolean test;
  if (content.matches(regEx)) {
    test = true;
  } else {
    test = false;
  }
  return test;
}

validate('#000', RGB_COLOR);
```

Die RegEx zur Erkennung des korrekten RGB-Farbschemas ist recht einfach. Gültige Eingaben sind *#ffF* oder *#000000*. Der Bereich umfasst die Zeichen *0-9* und zusätzlich noch Buchstaben *A-F*. Groß-/Kleinschreibung wird in unserem Beispiel nicht beachtet. Wenn Sie Ihre eigene RegEx entwickeln, müssen Sie bestehende Grenzen immer sehr gut im Auge behalten. Ein gutes Beispiel, um obere beziehungsweise untere Schranken zu verstehen, ist das 24-Stunden-Zeitformat. Typische Fehler sind ungültige Eingaben wie *23:60* oder *24:00*. Ein Blick auf die Anzeige der Digitaluhr zeigt für ein 24-Stunden-Format als untere Schranke *00:00* und als obere Schranke *23:59*, alles andere ist ungültig.

Die Methode *validate* vergleicht die Eingabezeichenfolge mit der RegEx. Wenn das Muster mit der Eingabe übereinstimmt, gibt die Methode *TRUE* zurück. Wenn Sie weitere Ideen zu Validatoren

in Java erhalten möchten, können Sie auch in meinem GitHub-Repository [3] nachsehen.

Zusammengefasst ist unsere erste Idee, um Benutzereingaben vor Missbrauch zu schützen, alle problematischen Zeichenfolgen herauszufiltern wie SQL-Kommentare und so weiter. Und solch eine Sperrliste ist auch nicht schlecht. Zumindest für den Anfang. Eine Blacklist weist aber einige Einschränkungen auf. Zunächst erhöht sich die Komplexität der Anwendung, da das Blockieren einzelner Zeichen wie –; und , manchmal unerwünschte Nebenwirkungen verursachen kann. Auch eine anwendungsweite Standardbegrenzung der Zeichen könnte Probleme bereiten. Stellen Sie sich vor, es gibt einen Textbereich für ein Blogsystem oder Ähnliches.

Das bedeutet, dass wir ein weiteres leistungsstarkes Konzept benötigen, um die Eingabe so zu filtern, dass unsere SQL-Abfrage nicht manipuliert werden kann. Um dieses Ziel zu erreichen, bietet der SQL-Standard eine sehr gute Lösung. SQL-Parameter sind Variablen innerhalb einer SQL-Abfrage, die als Inhalt und nicht als Anweisung interpretiert werden. Das ermöglicht es, große Texte entgegenzunehmen, ohne einige gefährliche Zeichen blockieren zu müssen. Schauen wir uns an, wie das mit einer PostgerSQL-Datenbank [4] funktioniert:

```
DECLARE user String;
SELECT * FROM login WHERE name = user;
```

Für den Fall, dass Sie den OR-Mapper Hibernate [5] verwenden, gibt es mit dem Java Persistence API (JPA) einen eleganteren Weg (Listing 2).

Listing 2: Hibernate-JPA-SQL-Parameter verwenden

```
String myUserInput;

@PersistenceContext
public EntityManager mainEntityManagerFactory;

CriteriaBuilder builder =
```

```
  mainEntityManagerFactory.getCriteriaBuilder();

CriteriaQuery<DomainObject> query =
  builder.createQuery(DomainObject.class);

// create Criteria
Root<ConfigurationDO> root =
  query.from(DomainObject.class);

//Criteria SQL Parameters
ParameterExpression<String> paramKey =
  builder.parameter(String.class);

query.where(builder.equal(root.get("name"), paramKey);

// wire queries together with parameters
TypedQuery<ConfigurationDO> result =
  mainEntityManagerFactory.createQuery(query);

result.setParameter(paramKey, myUserInput);
DomainObject entry = result.getSingleResult();
```

Listing 2 zeigt ein vollständiges Beispiel für Hibernate mit JPA und dem Criteria API. In der ersten Zeile wird die Variable für die Benutzereingabe deklariert. Die Kommentare in der Auflistung erklären sehr deutlich, wie es funktioniert. Wie Sie sehen können, ist das keine Raketenwissenschaft. Die Lösung hat neben der Verbesserung der Sicherheit von Webanwendungen einige weitere nette Vorteile. So wird kein einfaches SQL verwendet. Dadurch wird sichergestellt, dass jedes Datenbankverwaltungssystem, das von Hibernate unterstützt wird, durch diesen Code gesichert werden kann.

Die Nutzung sieht vielleicht etwas komplizierter aus als eine einfache Abfrage, aber der gewonnene Nutzen für Ihre Anwendung ist enorm. Andererseits gibt es natürlich einige zusätzliche Codezeilen. Aber die sind nicht so schwer zu verstehen, wie dieser Artikel gezeigt hat.

Marco Schulz studierte an der HS Merseburg Diplominformatik und twittert regelmäßig als @ElmarDott über alle möglichen technischen Themen. Seine Schwerpunkte sind hauptsächlich Build- und Konfigurationsmanagement, Softwarearchitekturen und Release-Management. Seit knapp 20 Jahren realisiert er in internationalen Projekten für namhafte Unternehmen umfangreiche Webapplikationen. Er ist freier Consultant/Trainer. Sein Wissen teilt er mit anderen Technikbegeisterten auf Konferenzen, wenn er nicht gerade wieder einmal an einem neuen Fachbeitrag schreibt.

Links & Literatur

[1] https://owasp.org

[2] https://beanvalidation.org

[3] https://github.com/ElmarDott/TP-CORE/blob/master/src/main/java/org/europa/together/utils/Validator.java

[4] https://www.postgresql.org/docs/9.1/plpgsql-declarations.html

[5] https://hibernate.org

[6] https://elmar-dott.com/courses/de/web-application-securiy

[7] Originalartikel: https://elmar-dott.com/articles/preventing-sql-injections-in-java/