SQL Injections

Lukas Ziefle
Hochschule Aalen

Zusammenfassung

SQL Injections sind leicht zu verhindern und dennoch sind sie seit über 20 Jahren präsent. Als SQL Injection wird das Einschleusen von Code in SQL Datenbankabfragen bezeichnet. Dies geschieht durch Benutzereingaben, die von Anwendungen nicht validiert werden und direkten Einfluss auf die Abfrage haben. Dadurch kann es einem Angreifer gelingen, sensible Daten aus der Datenbank auszulesen, Daten zu verändern oder zu löschen. SQL Injections können in verschiedene Kategorien unterteilt werden. Diese Arbeit gibt einen Überblick über die verschiedenen Arten und veranschaulicht diese anhand einer beispielhaften Anwendung, die in PHP geschrieben ist. Um SQL Injections zu verhindern können prepared Statements verwendet werden. Prepared Statements werden vor der der Benutzereingabe an die Datenbank geschickt und enthalten Parameter, die später von der Datenbank mit den Eingaben des Benutzers ersetzt werden. Diese Arbeit klärt den Leser über SQL Injections auf, sensibilisiert und bietet Lösungsansätze.

1 Einführung

SQL Injections (SQLi) gehören zu einer Gruppe von Sicherheitslücken, bei denen Code von außen in eine Anwendung eingeschleust wird. Obwohl SQLi bereits 1998 erstmals erwähnt wurden [1], gehört diese Gruppe laut OWASP noch immer zu dem am häufigsten verbreitetsten Sicherheitslücken [2]. Wenn eine Anwendung durch SQLi verwundbar ist, könnte ein Angreifer Zugriff auf Daten erlangen, die er eigentlich nicht einsehen kann, Daten verändern oder löschen [3]. Oftmals sind in einer Datenbank sensible Daten wie z. B. Benutzernamen und Passwörter gespeichert. Wenn ein Angreifer solche geheimen Daten ausliest, kann er diese weiterverwenden und im Namen anderer Benutzer auf dem System agieren. Je nach Konfiguration und Version der Datenbank kann es einem Angreifer auch möglich sein, Befehle auf dem Server auszuführen, der die Datenbank betreibt [4].

Als SQLi wird das Einschleusen von Code in SQL Befehle bezeichnet. Dies ist dann möglich, wenn eine (Web-)Anwendung Benutzereingaben nicht ausreichend validiert und diese Eingaben anschließend in einer Datenbankabfrage verwendet. Durch geschickte Eingaben kann ein Angreifer einen vordefinierten Befehl verändern oder erweitern und somit Datenbankabfragen direkt beeinflussen.

2 Arten von Benutzereingaben

Die in einer SQL Abfrage verwendeten Daten stammen oft von außerhalb der Anwendung. Am offensichtlichsten ist das bei direkten Benutzereingaben, wie z. B. die Eingabefelder für Benutzernamen und Passwort auf der Loginseite einer Webanwendung. Diese Eingaben werden über `GET` oder `POST` Anfragen an das Backend der Anwendung übermittelt und dort verarbeitet.

Die Webanwendung kann aber auch andere externe Daten verwenden. Um Benutzer zu tracken werden häufig Cookies verwendet. Dazu wird in einem Cookie z. B. eine eindeutige ID gespeichert, die bei einem erneuten Besuch der Webseite wieder ausgelesen werden kann. Um dem Benutzer dann z. B. personalisierte Inhalte anzeigen zu können, wird diese ID benutzt, um Daten über den Benutzer aus einer Datenbank auszulesen. Da Cookies auf dem Gerät der Benutzer gespeichert sind, können diese von einem Angreifer so bearbeiten werden, dass schädliche Datenbankabfragen auf dem Server ausgeführt werden.

Auch die Header einer HTTP Anfrage können von einem Angreifer beliebig bearbeitet werden, bevor die Anfrage tatsächlich abgeschickt wird. HTTP Header werden von Webanwendungen oft für statistische Zwecke gespeichert, um z. B. auswerten zu können, von welcher Seite ein Benutzer auf die eigene Anwendung verlinkt wurde oder welchen Browser der Nutzer verwendet. Fließen diese Werte ungefiltert in eine Datenbankabfrage ein, kann es auch hierüber zu SQLi kommen.

3 Arten von SQL Injections

SQLi können in verschiedene Arten unterteilt werden. Dieses Kapitel gibt einen Überblick und erklärt Anhand von Beispielen, wie die einzelnen Arten von Angreifern ausgenutzt werden können.

3.1 In-band SQL Injections

In-band bedeutet, dass der Angreifer seine Attacke ausführt und über die gleiche Verbindung eine Antwort von der Datenbank bekommt. Es gibt unterschiedliche Möglichkeiten, wie dies ausgenutzt werden kann. Um diese zu verdeutlichen, wird im Folgenden eine beispielhafte Webanwendung vorgestellt.

Die Beispielanwendung ist in PHP geschrieben und speichert die Daten in einer MariaDB. Sie bietet dem Benutzer Informationen über einige Programmiersprachen. Um Zugang zu diesen Informationen zu bekommen muss sich der Benutzer zunächst anmelden. Listing 1 zeigt die für SQLi anfällige Implementierung.

Listing 1: Anmeldung in der Webanwendung

```
<?php
$query = 'SELECT * FROM users WHERE username = "'
    . $_POST['username'] . '" AND password = "' .
    $_POST['password'] . '"';

$result = $mysqli->query($query);
if (!$result) {
    die('Error: ' . $mysqli->error);
}

if ($result->num_rows > 0) {
    loginSuccessful();
} else {
    loginFailed();
}
```

Der Benutzer gibt seinen Namen und sein Passwort ein und die Anmeldedaten werden per HTTP POST Methode an die Anwendung übergeben. Diese liest die Parameter aus und setzt den Benutzernamen und das Passwort an der passenden Stelle in der SQL Abfrage ein.

Wenn ein Benutzer sich mit dem Namen *live* und dem Passwort *evil* einloggt, wird der in Listing 2 gezeigte Befehl an die Datenbank geschickt.

Listing 2: SQL Abfrage bei der Anmeldung

```
SELECT * FROM users WHERE username = "live" AND
    password = "evil"
```

3.1.1 Tautologie

Eine Tautologie ist eine Aussage, die immer wahr ist. Bei der gezeigten Webanwendung kann ein Angreifer folgendes Passwort eintragen: `"␣OR␣1=1;--␣`. Daraus erzeugt die Anwendung die Datenbankabfrage in Listing 3.

Listing 3: SQL Abfrage mit Tautologie

```
SELECT * FROM users WHERE username = "admin" AND
    password = "" OR 1=1;-- "
```

Alles was nach den beiden – steht, wird von der MySQL Datenbank als Kommentar interpretiert und fließt somit nicht in die Abfrage mit ein. Da *1=1* immer wahr ist, gibt die Datenbank alle Benutzer an die Anwendung zurück. In der Anwendung wird nur überprüft, ob die Datenbank mindestens einen Benutzer zurückliefert. Da dies der Fall ist, wird der Angreifer eingeloggt.

3.1.2 Fehlerbasierte SQL Injections

Die Anwendung zeigt dem Benutzer wie in Listing 1 zu sehen, Fehler an, die bei Datenbankabfragen entstehen. Diese Fehlermeldungen können ausgenutzt werden, um Informationen über die Datenbank auszulesen. Hierzu kann ein Angreifer die Funktion *EXTRACTVALUE* der MariaDB verwenden. Diese Funktion erhält als ersten Parameter einen XML String und als zweiten Parameter einen XPATH Ausdruck und gibt den Teil des ersten Parameters zurück, der zum angegebenen Ausdruck passt [5]. Wenn der XPATH Ausdruck allerdings einen Fehler enthält, gibt die Funktion folgende Fehlermeldung zurück: *XPATH syntax error: 'XPATH Ausdruck'*.

Da der Fehler in der Webanwendung ausgegeben wird, kann eine Angreifer diese Schwachstelle mit folgender Eingabe als Benutzernamen ausnutzen: `"␣AND␣EXTRACTVALUE(" ",␣CONCAT(":",␣(SELECT␣table_name␣FROM␣information_schema .tables␣WHERE␣table_schema␣=␣database()␣LIMIT␣0,␣1)))--␣`. Die Datenbank führt dann die Abfrage aus Listing 4 aus.

Listing 4: Fehlerbasierte SQL Injection

```
SELECT * FROM users WHERE username = "" AND
    EXTRACTVALUE("", CONCAT(":", (SELECT
    table_name FROM information_schema.tables
    WHERE table_schema = database() LIMIT 0, 1)))
    -- " AND password = ""
```

In der Abfrage wird der Funktion *EXTRACTVALUE* als erster Parameter ein leerer String mitgegeben, dieser wird für den Angriff nicht benötigt. Im zweiten Parameter wird eine Abfrage konstruiert, die aus der Datenbank *information_schema* den *table_name* der Einträge der Tabelle *tables* abfragt, bei denen *table_schema* dem Namen der Datenbank entspricht, die für diese Abfrage verwendet wird. Anschließend wird mit dem *LIMIT* Operator ein Eintrag aus dem Ergebnis ausgewählt. Der Offset ist 0, somit wird der erste Eintrag ausgegeben. Das Ergebnis dieser Abfrage ist der Name der ersten Tabelle in der gerade verwendeten Datenbank. Der Name der Tabelle ist ein valider XPATH Ausdruck. Damit ein Fehler entsteht und der Name in der Fehlermeldung

angezeigt wird, wird mit *CONCAT* ein *:* vorangestellt.

Dem Angreifer wird folgender Fehler *Error: XPATH syntax error: ':users'* angezeigt. Damit weiß der Angreifer, dass die Datenbank, die für diese Abfrage verwendet wird, eine Tabelle *users* besitzt. Der Offset der Abfrage kann nun erhöht werden, um den nächsten Tabellennamen der Datenbank herauszufinden.

Mit derselben Methode kann ein Angreifer auch den Namen der verwendeten Datenbank oder die Spalten der Tabelle *users* auslesen.

3.1.3 UNION basierte SQL Injections

Im zweiten Teil der Beispielanwendung kann ein angemeldeter Benutzer über ein Eingabefeld Suchanfragen an die Datenbank stellen. Die Ergebnisse werden in einer Tabelle dargestellt. Listing 5 zeigt den PHP Code, der dies umsetzt.

Listing 5: Nach Programmiersprachen suchen

```
<?php
$query = 'SELECT name, description FROM languages
    WHERE description LIKE "%' . $_POST['query'] .
     '%"';

$result = $mysqli->query($query);

if ($result && $result->num_rows > 0) {
    while($row = $result->fetch_assoc()) {
        echo '<tr>';
        echo '<td>' . $row['name'] . '</td><td>' .
             $row['description'] . '</td>';
        echo '</tr>';
    }
}
```

Mit dem *UNION* Operator können die Ergebnisse mehrerer SQL Abfrage kombiniert werden. Dabei müssen alle *SELECT* Abfragen die gleiche Anzahl an Spalten haben und Datentypen der Spalten müssen ähnliche sein [6]. Wenn ein Angreifer Tabellennamen und deren Spalten kennt, z. B. durch einen Angriff wie in Unterunterabschnitt 3.1.2 beschrieben, oder durch raten, dann kann er durch einen UNION Angriff Daten aus diesen Tabellen lesen.

Im Beispiel aus Listing 5 kann der Angreifer folgenden Text in das Suchfeld eingeben: `"␣UNION␣SELECT␣username,␣password␣FROM␣users;--␣`. Dies resultiert in der Datenbankabfrage aus Listing 6.

Listing 6: SQL Abfrage bei der Suche

```
SELECT name, description FROM languages WHERE
    description LIKE "%" UNION SELECT username,
    password FROM users;-- %"
```

Wenn die Tabelle *users* mit den Spalten *username* und *password* in der Datenbank existiert, werden die Ergebnisse der zweiten Abfrage an die Ergebnisse der ersten Abfrage angehängt. Die Anwendung listet alle gespeicherten Benutzer unterhalb der Namen der Programmiersprachen und die Passwörter unter der Beschreibung der Programmiersprachen in einer Tabelle auf.

Auf diese Weise kann ein Angreifer beliebige Daten aus verschiedenen Tabellen auslesen, sofern er die Metadaten der Tabelle kennt.

3.2 Blind SQL Injections

Blind SQLi funktionieren ohne direkte Fehlermeldungen oder Ausgaben von Datenbankabfragen, wie in Unterabschnitt 3.1 beschrieben. Stattdessen muss ein Angreifer das Verhalten der Webanwendung für verschiedene Eingaben beobachten.

3.2.1 Booleanbasierte SQL Injections

Das Beispiel für booleanbasierte SQLi verwendet den PHP Code aus Listing 5. Es wird jedoch davon ausgegangen, dass eine *UNION* Attacke nicht mehr möglich ist. Der Angreifer bekommt auch keine Fehlermeldung von der Datenbank zu sehen. Die Benutzereingaben werden weiterhin nicht richtig von der Anwendung validiert, der Angreifer kann also weiterhin die Datenbankabfrage direkt beeinflussen.

Um eine booleanbasierte SQLi durchzuführen, muss ein Angreifer zunächst eine Eingabe konstruieren, die zu einer wahren Aussage führt und eine, die zu einer falschen Aussage führt. Für das Beispiel aus Listing 5 könnte eine Eingabe, die eine wahre Aussage erzeugt z. B. so aussehen: `"␣AND␣1=1;--␣` und eine falsche Aussage kann mit `"␣AND␣1=2;--␣` erzeugt werden. Anschließend muss der Angreifer einen Unterschied in der Ausgabe der Anwendung für diese beiden Eingaben ausmachen. In diesem Beispiel werden bei der wahren Abfrage alle Einträge der Datenbank in der Tabelle ausgegeben und bei der Falschen bleibt die Tabelle leer.

Schließlich kann der Angreifer die oben genannten einfachen Eingaben anpassen, um Informationen über die Datenbank zu erhalten. Um beispielsweise die Anzahl der Buchstaben des Namens der Datenbank herauszufinden, kann diese Eingabe verwendet werden: `"␣AND␣LENGTH(database())␣=␣10;--␣`. Die ergibt die in Listing 7 dargestellte Abfrage.

Listing 7: Booleanbasierte SQL Injection

```
SELECT name, description FROM languages WHERE
    description LIKE "%" AND LENGTH(database()) =
    10;-- %"
```

Wenn der Name der Datenbank 10 Zeichen hat, produziert die Eingabe eine wahre Aussage, somit wird dem Angreifer eine Tabelle mit allen Einträgen angezeigt. Mit der folgenden Eingabe kann der Angreifer Buchstaben für Buchstaben den Namen der Datenbank auslesen: `"␣AND␣SUBSTRING(database(),␣2,␣1)␣=␣"e";--␣`.

Die Funktion *SUBSTRING* gibt angefangen beim zweiten Buchstaben des Namens der Datenbank einen Buchstaben

zurück. Wenn dieser einem *e* entspricht, ist die Aussage wahr. Durch verändern des zweiten Parameters kann ein Angreifer jeden Buchstaben testen.

Da Buchstaben untereinander verglichen werden können, kann ein Angreifer das *"="* durch ein *">"* ersetzen und eine binäre Suche durchführen, um schneller Ergebnisse zu erzielen.

3.2.2 Zeitbasierte SQL Injections

Zeitbasierte SQLi werden anhand des Beispielcodes aus Listing 1 erklärt. Die Fehlermeldung wird nun nicht mehr ausgegeben, sondern in eine Logdatei gespeichert und dem Benutzer wird angezeigt, dass seine Anmeldedaten falsch sind. Dadurch hat ein Angreifer keine Möglichkeit mehr, durch Fehlermeldungen wie in Unterunterabschnitt 3.1.2 oder durch Unterschiede auf der Seite wie in Unterunterabschnitt 3.2.1 Daten auszulesen.

Ein Angreifer kann jedoch eine Datenbankabfrage provozieren, die wenn eine Bedingung erfüllt ist einige Sekunden *schläft* und wenn diese Bedingung nicht erfüllt ist direkt ein Ergebnis zurück gibt. Solch eine Abfrage kann z. B. mit der folgenden Eingabe als Benutzernamen konstruiert werden: `"␣AND␣(SELECT␣SLEEP(5)␣FROM␣DUAL␣WHERE␣SUBSTRING(database(),␣1,␣1)␣=␣"p");--␣"`. Dadurch ergibt sich die Datenbankabfrage aus Listing 8.

Listing 8: Zeitbasierte SQL Injection

```
SELECT * FROM users WHERE username = "" AND (
    SELECT SLEEP(5) FROM DUAL WHERE SUBSTRING(
    database(), 1, 1) = "a");-- "" AND password =
    ""
```

Der Angreifer verknüpft hier die eigentliche Abfrage mit einer zweiten. Dabei wird aus der Tabelle *DUAL* ausgewählt. Diese Tabelle existiert in MySQL-Datenbanken, um Ergebnisse von Funktionen wie *USER* oder *SYSDATE* auszuwählen. Wenn die Bedingung, dass der erste Buchstaben des Datenbanknamens ein *"a"* ist, zutrifft, wird die Abfrage fünf Sekunden warten. Wenn der erste Buchstabe des Namens ein anderer ist, wird die Abfrage ohne Verzögerung fortgeführt.

Dadurch ist es einem Angreifer möglich, beliebige Inhalte der Datenbank auszulesen. Dieser Vorgang ist jedoch sehr zeitintensiv, da pro Abfrage höchstens ein Buchstabe ausgelesen werden kann. Um den Vorgang zu beschleunigen, kann auch hier eine binäre Suche verwendet werden.

Listing 9 zeigt, wie ein Angreifer die Tabellennamen der aktuell verwendeten Datenbank auslesen kann.

Listing 9: Zeitbasierte SQL Injection

```
SELECT * FROM users WHERE username = "" AND (
    SELECT SLEEP(5) FROM DUAL WHERE SUBSTRING((
    SELECT table_name FROM information_schema.
    tables WHERE table_schema = database() LIMIT
    0, 1), 1, 1) > "j");-- "" AND password = ""
```

Durch den *LIMIT* Befehl wird der erste Tabellenname ausgewählt, durch erhöhen des Offsets kann der nächste Eintrag untersucht werden. Durch erhöhen des zweiten Parameters des Befehls *SUBSTRING* kann der nächste Buchstaben überprüft werden. Im Beispiel in Listing 9 wird die Abfrage um fünf Sekunden verzögert, wenn die erste Tabelle der Datenbank mit einem Buchstaben beginnt, der nach *"j"* im Alphabet kommt.

3.3 Out-of-band SQL Injections

Out-of-band SQLi unterscheiden sich von den ersten beiden Kategorien, da ein Angreifer Informationen nicht direkt ausgegeben bekommt, wie bei Unterabschnitt 3.1 und sich auch keine Informationen durch unterschiedliche Antworten erschließen lassen, wie in Unterabschnitt 3.2. Bei dieser Art von SQLi versucht der Angreifer Informationen aus der Datenbank auszulesen und diese an ein anderes, von ihm kontrolliertes System zu schicken. Dies kann über verschiedene Protokolle passieren. Es bietet sich aber an, DNS Abfragen zu verwenden, da diese in Produktivsystemen meist erlaubt sind, damit das System funktioniert [7].

Wie genau eine DNS Abfrage in SQL aussieht, hängt stark vom Datenbanksystem ab, das benutzt wird. Die Idee dahinter ist es, dass durch eine Datenbankabfrage Informationen ausgelesen werden und diese dann als Subdomain der eigentlichen Adresse vorangestellt werden. Ein Angreifer kann dann auf dem von ihm kontrollierten System in den Logs eine Anfrage erkennen, die die ausgelesenen Daten beinhaltet.

3.4 Second Order SQL Injections

Eine weitere Möglichkeit für einen Angreifer SQLi auszunutzen sind Second-Order SQLi. Hierbei wird die Benutzereingabe von der Anwendung validiert und anschließend in der Datenbank abgespeichert. Zu einem späteren Zeitpunkt werden diese Daten an einer anderen Stelle ohne Validierung für eine Datenbankabfrage verwendet [8]. Ziel des Angreifers ist es, die erste Eingabe so zu konstruieren, dass sie die zweite Datenbankabfrage zu seinen Gunsten verändert. Solche Schwachstellen sind für Angreifer schwerer zu finden als die in den vorigen Kapiteln beschriebenen Arten von SQLi.

Ein Beispiel für Second Order SQLi ist das Erstellen eines Benutzers mit kompromittiertem Benutzernamen und das spätere Ändern des Passworts. Der Angreifer registriert sich mit dem Benutzernamen *admin"–* . Die Webanwendung überprüft die Eingabe, wie es in Abschnitt 4 beschrieben wird. Anschließend nutzt der Angreifer die Funktion der Anwendung, sein Passwort zu ändern. Dazu schickt die Anwendung eine Abfrage wie in Listing 10 an die Datenbank, um zu überprüfen, ob der Benutzer sein altes Passwort korrekt eingegeben hat [9].

Anschließend verwendet die Anwendung das Ergebnis der Abfrage, um das Passwort des Benutzers zu aktualisieren, da

Listing 10: Abfragen des Benutzernamens

```
SELECT * FROM user WHERE username = "admin'--" AND
    password = "old_password"
```

davon ausgegangen wird, dass die Werte, die von der Datenbank kommen nicht von einem Angreifer kompromittiert sind. Der passende PHP Code ist in Listing 11 zu sehen.

Listing 11: Ändern des Passworts

```
$query = 'UPDATE users SET password = "' .
    $password_new . '" WHERE username = "' .
    $result['username'] . '"';
$result = $mysqli->query($query);
```

$username ist der Benutzername, der durch die erste SQL Abfrage von der Datenbank zurückgeliefert wurde. Da der Angreifer sich aber mit dem Benutzernamen *admin"–* registriert hat, ergibt sich die in Listing 12 gezeigte Datenbankabfrage.

Listing 12: SQL Abfrage zum Ändern des Passworts

```
UPDATE users SET password = "123456" WHERE
    username = "admin"-- "
```

Der Benutzernamen des Angreifers beendet die Abfrage durch die *"–* vorzeitig. Dadurch wird nicht das Passwort des Angreifers geändert, sondern das des Benutzers *admin*.

4 Verhindern von SQL Injections

SQLi können verhindert werden, indem die Anwendung sog. *prepared Statements* verwendet. Prepared Statement bedeutet, dass der Entwickler zuerst die komplette Datenbankabfrage definiert. Anstatt jedoch die Benutzereingaben in die Abfrage zu konkatenieren, müssen dafür *"?"* als Platzhalter verwendet werden. Anschließend werden die vorbereitete Abfrage und die Benutzereingaben an die Datenbank übergeben. Diese ersetzt die Platzhalter mit den Benutzereingaben und garantiert dadurch, dass die bereits festgelegte Struktur der Datenbankabfrage nicht mehr verändert wird. Auf diese Weise werden Benutzereingaben immer als solche behandelt und können die vom Entwickler festgelegte Struktur Abfrage nicht mehr verändern. Listing 13 zeigt, wie prepared Statements in PHP umgesetzt werden.

Listing 13: Prepared Statements in PHP

```
$query = 'SELECT * FROM users WHERE username = ?
    AND password = ?';
$stmt = $mysqli->prepare($query);
$stmt->bind_param('ss', $_POST['username'], $_POST
    ['password']);
$stmt->execute();
```

Zuerst wird die Datenbankabfrage mit Platzhaltern definiert und an die Datenbank geschickt, die das Statement vorbereitet. Anschließend werden die Parameter mit den Benutzereingaben verknüpft, dabei muss auch festgelegt werden, um was für einen Datentyp es sich bei den Variablen handelt. Das *"s"* steht dabei für einen String. Schließlich wird die Abfrage mit *execute* an die Datenbank geschickt. Dort werden die Parameter ersetzt und die Datenbank führt die Abfrage aus.

Prepared Statements sollten wann immer möglich verwendet werden. Jedoch lassen sich nicht alle Teile einer SQL Abfrage durch prepared Statements variabel gestalten. Für den Tabellenname, die Spalten, die Sortierreihenfolge und Argumente für den SQL Operator *IN* können keine Platzhalter verwendet werden, die später ersetzt werden. Um bei diesen Abfragen dennoch vor SQLi geschützt zu sein, sollten hier entweder keine Benutzereingaben einfließen, oder die Benutzereingaben müssen validiert werden. In der Anwendung sollte dazu eine Liste an zulässigen Eingaben definiert sein und bei jeder Eingabe eines Benutzers muss überprüft werden, ob dessen Eingabe in dieser Liste enthalten ist, oder nicht.

4.1 Weitere Sicherheitsmaßnahmen

Neben der stetigen Verwendung von prepared Statements sollten weitere Sicherheitsmaßnahmen getroffen werden. Der Datenbankbenutzer, mit dem die Anwendung auf die Datenbank zugreift, sollte nur die nötigsten Rechte haben. Wenn eine Anwendung z. B. nur Informationen aus einer Datenbank auslesen muss, benötigt der Benutzer keine Rechte, Daten zu verändern oder zu löschen. Der Benutzer sollte auch nur Rechte für die Tabellen bekommen, die er wirklich abfragen muss, und nicht für alle in der Datenbank enthaltenen Tabellen. Es gilt das Prinzip der geringsten Rechte [10].

5 Fazit

Die hier beschriebenen Arten von SQLi und Methoden, wie Angreifer derartige Sicherheitslücken ausnutzen können, bilden nicht die vollständige Breite an Möglichkeiten für SQLi ab. Laut OWASP sind SQLi limitiert durch das Geschick und die Fantasie eines Angreifers [11], wenn sie nicht wie in Abschnitt 4 beschrieben, verhindert werden.

Obwohl SQLi und Methoden, diese zu verhindern, schon lange bekannt sind, kommen sie auch heute immer noch vor [12] [13] [14]. Es ist schwer definitiv zu sagen warum das der Fall ist, es gibt viele Möglichkeiten und Ursachen. Zum einen müssen viele Projekte schnell fertiggestellt werden und die Entwickler stehen unter hohem Druck, dabei konzentrieren sie sich u. U. mehr auf Funktionalität als auf Sicherheit [15]. Zum anderen gibt es immer noch viele Tutorials im Internet, von denen Entwickler lernen, die aber anfällig für SQLi sind [16].

Literatur

[1] .:: Phrack Magazine ::. [Online; aufgerufen am 3. Juni 2020] http://phrack.org/issues/54/8.html.

[2] 2017 Top 10 | OWASP. [Online; aufgerufen am 8. Juni 2020] https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_Top_10.html.

[3] What is SQL Injection? Tutorial & Examples | Web Security Academy. [Online; aufgerufen am 3. Juni 2020] https://portswigger.net/web-security/sql-injection.

[4] PHP: SQL Injection - Manual. [Online; aufgerufen am 3. Juni 2020] https://www.php.net/manual/en/security.database.sql-injection.php.

[5] EXTRACTVALUE. [Online; aufgerufen am 10. Juni 2020] https://mariadb.com/kb/en/extractvalue.

[6] SQL UNION Operator. [Online; aufgerufen am 10. Juni 2020] https://www.w3schools.com/sql/sql_union.asp.

[7] What is Blind SQL Injection? Tutorial & Examples | Web Security Academy. [Online; aufgerufen am 12. Juni 2020] https://portswigger.net/web-security/sql-injection/blind.

[8] SQL injection (second order). [Online; aufgerufen am 9. Juni 2020] https://portswigger.net/kb/issues/00100210_sql-injection-second-order.

[9] Lu Yan, Xiaohong Li, Ruitao Feng, Zhiyong Feng, and Hu Jing. Detection method of the second-order sql injection in web applications. pages 156–157, 10 2013.

[10] SQL Injection Prevention. [Online; aufgerufen am 13. Juni 2020] https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html#least-privilege.

[11] SQL Injection | OWASP. [Online; aufgerufen am 12. Juni 2020] https://owasp.org/www-community/attacks/SQL_Injection.

[12] Zack Whittaker. Font sharing site DaFont has been hacked, exposing thousands of accounts. *ZDNet*, May 2017. [Online; aufgerufen am 13. Juni 2020] https://www.zdnet.com/article/font-sharing-site-dafont-hacked-thousands-of-accounts-stolen.

[13] Zack Whittaker. Epic's forums hacked again, with thousands of logins stolen. *ZDNet*, Aug 2016. [Online; aufgerufen am 13. Juni 2020] https://www.zdnet.com/article/epic-games-unreal-engine-forums-hacked-in-latest-data-breach.

[14] Qatar Bank Breach Lifts the Veil on Targeted Attack Strategies - The Trend Micro UK Blog, Apr 2016. [Online; aufgerufen am 13. Juni 2020] https://blog.trendmicro.co.uk/qatar-bank-breach-lifts-the-veil-on-targeted-attack-strategies.

[15] The History of SQL Injection, the Hack That Will Never Go Away, Nov 2015. [Online; aufgerufen am 13. Juni 2020] https://www.vice.com/en_us/article/aekzez/the-history-of-sql-injection-the-hack-that-will-never-go-away.

[16] Insert Data into MySQL Database - Tutorialspoint. [Online; aufgerufen am 13. Juni 2020] https://www.tutorialspoint.com/php/mysql_insert_php.htm.