Sicherheitsüberlegungen

Beim Umgang mit Daten und Datenbanken müssen Sicherheitsüberlegungen von Beginn an bei der Anwendungsentwicklung berücksichtigt werden. Bei Applikationen, die Schnittstellen zum Internet haben, müssen solche Überlegungen eine noch zentralere Rolle spielen. Webapplikationen haben immer wieder schwerwiegende Sicherheitslücken, die dazu führen, dass sensitive Daten in falsche Hände oder die Öffentlichkeit gelangen.

Die OWASP listet in Ihrer "Top-Ten" die am häufigsten vorkommenden Fehler bei der Anwendungsentwicklung auf, auf Platz 1 befinden sich sogenannte „Injections“ - diese Fehlerklasse betrachten wir im folgenden etwas genauer.

"Injection flaws"

> „Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.“

Weiterführende Informationen

Eine Injection-Sicherheitslücke entsteht vereinfacht gesagt immer dann, wenn man (Benutzer-)Eingaben ohne weitere Validierung übernimmt und an weitere Befehle oder in weitere Verarbeitungsschritte weiterreicht. Das passiert sehr leicht, da man beim Programmieren für gewöhnlich nicht darüber nachdenkt, welche unsinnigen oder gar bösartigen Eingaben Benutzer möglicherweise machen, sondern meist darauf konzentriert ist, die erwarteten Eingaben effektiv weiterzuverarbeiten.

Grundregel der Webentwicklung: Vertraue keinem Datum, das dir ein Benutzer gibt. **Auf keinen Fall sollte man in einem produktiven System Benutzereingaben direkt in SQL Statements übernehmen.**[1)]

SQL-Injections

Da wir hier vor allem mit Datenbanken und Datenbankabfragen arbeiten wollen, spielen sogenannte SQL-Injections eine große Rolle in unserem Setting. Dabei erwarten wir beispielsweise einen *Namen* als Eingabe eines Formularfelds, nach dem wir in der Datenbank suchen wollen. Ein bösartiger Angreifer kann aber versuchen, im Namens-Eingabefeld SQL Befehle mit zu übergeben, die dann in unserem Programm in die Datenbankabfrage eingebaut werden. dadurch können Daten abgefragt werden, auf die der Benutzer vielleicht gar keinen Zugriff haben sollte[2)] oder Daten gelöscht oder manipuliert werden.

Eine **schlechte Idee** ist es also, das naheliegende zu tun:

```
// id wird in einem Formular vom Benutzer erfragt
```

```php
if(isset($_POST['id'])) {
   $id = $_POST['id'];
} else {
   die(" Es muss eine Datensatz ID angegeben werden!");
}

echo "Datensatz mit der ID $id: <br>";
$sql = "SELECT * FROM schueler WHERE id = $id";
$rows = $pdo->query($sql) ;
foreach ($rows as $row) {
   echo $row['id'] . " " . $row['vorname']." ".$row['nachname']."<br />";
}
```

Dies funktioniert zwar, ist aber anfällig für SQL Injections. Ein Angreifer kann über den POST-Parameter die SQL-Abfrage manipulieren und weiteren SQL-Code einschleusen.

Gibt der Anwender nämlich ins Formularfeld beispielsweise folgendes ein: `1 OR id > 1`

Werden alle Datensätze ausgegeben, denn an die Datenbank wird die Abfrage `SELECT * FROM schueler WHERE id = OR id > 1` geschickt.

(Quelle: https://xkcd.com/327/, Lizenz Creative Commons Attribution-NonCommercial 2.5 License.

Prepared Statements

Um SQL-Injections zu unterbinden, sollte man prepared Statements verwenden. In dem Moment, in dem ihr Daten von Benutzern an die Datenbank übergebt, solltet ihr stets auf prepared Statements zurückgreifen.

```php
// id wird in einem Formular vom Benutzer erfragt
if(isset($_POST['id'])) {
   $id = $_POST['id'];
} else {
   die(" Es muss eine Datensatz ID angegeben werden!");
}

echo "Datensatz mit der ID $id: <br>";

$statement = $pdo->prepare("SELECT * FROM schueler WHERE id = ?");
$statement->execute(array($id));

foreach ($row = $statement->fetch()) {
   echo $row['id'] . " " . $row['vorname']." ".$row['nachname']."<br />";
}
```

Blacklisting/Whitelisting

Ebenfalls effektiv ist es, Parameterwerte mit Blacklisting oder Whitelisting zu überprüfen und die Verarbeitung abzubrechen, wenn der Eingabewert nicht den geforderten Kriterien entspricht. Dabei sind reguläre Ausdrücke ein wertvolles Werkzeug.

- Beim Blacklisting kann man beispielsweise festlegen, dass die Verarbeitung abgebrochen wird, wenn bestimmte Zeichen in der Eingabe enthalten sind. Beispiel: „Wenn eimn =, ein `;` < oder > im Namensfeld übergeben werden, wird die Verarbeitung abgebrochen.“
- Beim Whilelistinh legt man ein Muster fest, dem die Eingabe entsprechen muss um zur Weiterverarbeitung zugelassen zu werden. Beispiel. „Im Feld `Postleitzahl` muss eine Eingabe übergeben, die aus fünf Ziffern besteht.“

Dieses Vorgehen nennt man „Input Sanitization“ und wie man das auf mannigfaltige Art vollständig verkacken kann, kann man sehr gut an der Luca-App betrachten.

Andere Angriffsvektoren

Die Mitigation andrer Angriffsvektoren - wie z.B. Authentifikation und Rechtemanagement können wir an DokuWiki abgegeben, beispielsweise indem wir unser Plugin nur auf Seiten einbauen, auf die Angemeldete (DokuWiki-)Benutzer mit den entsprechenden Rechten zugreifen dürfen. Nun ist aber zu beachten, dass unsere Applikation durch Sicherheitslücken im DokuWiki-System beeinträchtigt werden kann: Hat DokuWiki eine Sicherheitslücke in seinem Authentifikations- und Rechtemanagement, ist unsere WebApp davon möglicherweise auch betroffen.

Auch bei der Implementation muss man hier große Sorgfalt walten lassen, um unerwünschte Nebeneffekte zu vermeiden, wenn man beispielsweise aus Rechten, die ein Benutzer in DokuWiki hat Rechte innerhalb der WebApp ableiten möchte.

1)

https://www.ionos.de/digitalguide/server/sicherheit/sql-injection-grundlagen-und-schutzmassnahmen/

2)

Womöglich die gesamte Datenbank, mit gespeicherten Paswort-Hashes und Kreditkartennummern - wem das bekannt vorkommt...