PHP/MySQL: Mehr Sicherheit und erhöhte Performance durch MySQLi und Prepared Statements

Tobias Kölligan

Zahlreiche Webanwendungen basieren auf der sehr beliebten Scriptsprache PHP, welche sich besonders entwicklerfreundlich mit MySQL kombinieren lässt. Diese populäre Kombination sorgt aber auch dafür, dass Sicherheitslücken diesbezüglich verstärkt ausgenutzt werden. Allen voran seien hier MySQL-Injections genannt.

IN DIESEM ARTIKEL ERFAHREN SIE…

- Sie erfahren, wie Sie die neue MySQLi-Technologie einsetzen können, sowie welche Vorteile dadurch entstehen und welche sicherheitsrelevanten Features diese bietet.

WAS SIE VORHER WISSEN SOLLTEN…

- Sie sollten sich mit PHP/MySQL grundlegend auskennen und ein Grundverständnis für Datenbanken und OO-Programmierung mitbringen.

Im nachfolgenden Artikel geht es um die Vermeidung solcher Injections und die gleichzeitige Performancesteigerung der Webanwendung durch die Nutzung sogenannter Prepared Statements.

Zuerst eine kleine Einführung bezüglich (My)SQL-Injections. Es handelt sich dabei um eine Art des Angriffs, bei dem der Angreifer versucht, eine nur unzureichend gesicherte SQL-Query zu manipulieren und somit eigene SQL-Befehle abzusetzen. Ein sehr einfaches Beispiel wäre die nachfolgende SELECT-Abfrage:

```
mysql_query(„SELECT id, vorname, nachname FROM kunden
                    WHERE id = „ . $_GET[,id‘]);
```

In der oben stehenden Query wird ein Parameter (`$_GET[,id‘]`) ungefiltert an die SELECT-Abfrage weitergegeben. Ein böswilliger Benutzer der Webseite könnte nun die Variable `$_GET[,id‘]` nicht mit einer Ganzzahl (wie gewünscht) befüllen, sondern mit jedem beliebigen String, eben auch mit weiteren SQL Befehlen. So wäre es zum Beispiel sehr unpraktisch, wenn in der Variable `$_GET[,id‘]` folgendes stünde:

```
0; DROP TABLE kunden;
```

Denn dann würden zwei SQL-Querys ausgeführt, ein SELECT und ein DROP TABLE, wobei letzteres auf keinen Fall gewünscht ist. Natürlich müsste ein Angreifer zuerst die Bezeichnung der entsprechenden Tabelle herausbekommen. Oftmals ist dies jedoch leichter als gedacht: entweder verwendet die Webseite ein CMS, bei dem die Tabellenbezeichnungen bekannt sind oder es werden verschiedene Bezeichnungen durchprobiert, hier helfen zu großzügig gestaltete Fehlermeldungen auf dem Produktivsystem oftmals weiter.

Schutz vor derartigen Angriffen bietet nur eine Säuberung der entsprechenden Variablen via `mysql_real_escape_string()`oder analogen Befehlen, dieses sollte getreu dem Motto „Never trust user data“ immer geschehen!

Neben dem oben dargestellten sehr simplen Beispiel sind in der Realität auch wesentlich komplexere Angriffsszenarien möglich. So können Passwortabfragen oder Sicherheitssperren umgangen werden oder massenhaft Benutzerdaten (inkl. Passwörtern) entwendet werden. Seit PHP 5 sowie MySQL 4.1[1] gibt es jedoch einen sicheren und performanten Schutz vor dieser Art des Angriffs.

MySQLi versus MySQL

`MySQLi`[2] ist eine neue Schnittstelle (API) zu einer MySQL Datenbank ab Version 4.1, welche in PHP ab Version 5 standardmäßig integriert ist. Ein kurzer Blick auf die Ausgabe von `phpinfo()` zeigt Ihnen, ob die entsprechende Extension (ext/mysqli) auf Ihrem Webserver installiert und aktiviert ist. Falls nicht, kann diese durch ein erneutes Kompilieren der PHP-Quellen hinzugefügt werden, dabei Hilft einem oft auch der Webhoster weiter, sofern man ein Shared-Hosting-System nutzt, auf dem mehrere Kunden aufgesetzt sind.

Im Gegensatz zu der alten MySQL-Extension bietet die neue Variante zahlreiche verbesserte Funktionen (das ‚i‘

steht für *improved*). Dazu zählen vor allem die Unterstützung von Prepared Statements und zahlreiche Verbesserungen im Kern der API, welche die Performance drastisch erhöhen können. Und eben genau diese Prepared Statements verschaffen der Webanwendung wesentlich mehr Sicherheit und Performance. Darauf soll im Folgenden nun genauer eingegangen werden.

Hinweis: MySQLi bietet darüber hinaus auch noch zahlreiche weitere Neuerungen, zum Beispiel die Unterstützung von Transaktionen, welche vor allem im Enterprise-Umfeld unerlässlich sind. Dieses Feature ist jedoch auch von der gewählten Storage Engine (MyISAM, InnoDB, etc.) der MySQL-Tabelle abhängig und wird in diesem Artikel nicht weiter betrachtet. Nähere Informationen dazu sind auf php.net zu finden.

Prepared Statements

Doch worum geht es bei den sogenannten Prepared Statements konkret? Es handelt sich dabei um SQL-Templates inklusive Platzhalter, die in der Datenbank hinterlegt sind. Dies bietet vor allem drei wesentliche Vorteile. Erstens muss der Datenbankserver das Template (und damit den größten Teil der Query) nur einmal interpretieren, zweites kann die Schnittstelle Daten wesentlich schneller transportieren, da nur noch die Platzhalter ersetzt werden müssen und drittens fällt die Möglichkeit von SQL-Injections vollständig weg. Doch warum entfällt dieser Angriffsvektor?

Da das Template fest in der Datenbank hinterlegt ist, kann auch ein Angreifer keine Änderungen daran vornehmen. Die übergebenen Parameter werden selbstständig von der Datenbank passend in die Query eingefügt. Somit erreicht man durch die Verwendung von Prepared Statements nicht nur eine höhere Performance (vor allem bei wiederholten Abfragen), sondern auch eine wesentlich besser gesicherte Webanwendung.

Konkret sieht der Ablauf bei der Benutzung von Prepared Statements so aus:

- Eine SQL-Query wird mit Platzhaltern versehen an den Server gesendet.
- Dieser interpretiert und validiert die empfangene Query.
  - Interpretieren: Ist die Query syntaktisch korrekt?
  - Validieren: Ergibt die Query Sinn, sind zum Beispiel alle referenzierten Tabellen vorhanden?
- Anschließend wird die Query in einem speziellen Puffer gespeichert.
- Als Rückgabe gibt der Server eine Referenz auf die gepufferte Query zurück, damit kann die Query von PHP aus wieder aufgerufen werden.
- Wenn eine Query ausgeführt werden soll, wird diese Referenz inklusive der Werte für die Platzhalter an den Server geschickt.
- Der Server setzt die Parameter anstelle der Platzhalter ein und führt die Query aus.

An dem Ablauf ist der Vorteil dieser Methode erkennbar. Wesentliche Schritte einer Abfrage müssen nur ein einziges Mal erfolgen. Dazu gehört vor allem das Interpretieren und Validieren der Query, sowie das Senden der gesamten Abfrage an den Server. Beides geschieht lediglich einmal, sofern eine Query dem Server noch nicht bekannt ist. Bei jeder weiteren Abfrage werden lediglich eine Referenz auf die gepufferte Query sowie die eigentlichen Nutzdaten selber übermittelt. Dies reduziert die Menge der zu sendenden Bytes erheblich, was wiederum die Geschwindigkeit erhöht und die Netzwerkkommunikation schlank hält. Vergleichbar mit einem Ajax-Request an ein PHP-Script, bei dem dann ebenfalls nur die relevanten Teile einer Webseite neu aufgebaut werden müssen. Die Idee ist hier eine ähnliche.

Anwendungsbeispiel - Bound Parameter Prepared Statements

Im nachfolgenden Beispiel wird die OO-Syntax der PHP-Befehle verwendet. Es gibt jedoch auch zu jedem Befehl ein prozedurales Pendent, am Ende des Artikels gibt es ein kurzes Beispiel in der prozeduralen Variante. Die Befehle können aber auch problemlos auf php.net nachgeschlagen werden.

Grundsätzlich sehen die neuen Befehle kaum anders aus, als die alten zuvor. Lediglich die Syntax ist jetzt objektorientiert und die Anzahl der Befehle und Möglichkeiten hat sich leicht erhöht.

Ein einfaches Beispiel für ein Prepared Statement mit Platzhaltern könnte wie folgt aussehen:

```
// Datenbankverbindung aufbauen
$db = new mysqli(‚localhost‘, ‚user‘, ‚passwort‘,
                  ‚datenbank‘);
// Query vorbereiten
$stmt = $db->prepare(„INSERT INTO kunden VALUES (?, ?,
                  ?)“);
// Platzhalter befüllen (1x Integer und 2x String)
$stmt->bind_param(‚iss‘, $pId, $pVorname, $pNachname);
// Query ausführen
$stmt->execute();
// Statement schließen
$stmt->close();
// Datenbankverbindung schließen
$db->close();
```

Dieses sehr einfache Beispiel eines sogenannten „Bound Parameter Prepared Statements“ verdeutlicht die Arbeitsweise mit Prepared Statements. Nach dem obligatorischen Verbindungsaufbau wird der Datenbank via $db->prepare eine SQL-Query bekannt gemacht, diese wird dann intern in der Datenbank gespeichert (zuvor interpretiert und validiert) und bei erneutem Aufruf direkt fertig interpretiert hervorgeholt. Die Fragezeichen innerhalb der SQL-Query dienen als Platzhalter für Werte, die

im nächsten Schritt über $stmt->bind_param hinzugefügt werden. Hierbei steht der erste Parameter des Methodenaufrufs (‚iss') für die jeweiligen Datentypen der nachfolgenden Variablen. Das ‚i' steht dabei für eine Ganzzahl und das ‚s' für eine Zeichenkette. Nach erfolgreicher Ausführung der Query, wird das Statement geschlossen und die Datenbankverbindung getrennt.

Neben ‚i' und ‚s' sind auch noch zwei weitere Datentypen definiert :

Datentyp	Beschreibung
i	Ganzzahl (Integer)
d	Fließkommazahlen (Double / Float)
b	Binäre Daten (BLOB)
s	Zeichenketten, Sonstiges (String)

An diesem Beispiel sieht man deutlich den Sicherheitsgewinn durch die Verwendung von Prepared Statements. Der Datenbankserver kennt bereits die komplette Query als Template, bevor überhaupt PHP-Variablen mit der Query in Berührung kommen. Bei dem anschließenden Ersetzen der Platzhalter durch Variableninhalte werden konkrete Typen mitgegeben. Dadurch weiß der Server wie er die Werte zu interpretieren hat. SQL-Injections haben damit keine Chance mehr.

An diesem Beispiel sieht man deutlich den Sicherheitsgewinn durch die Verwendung von Prepared Statements. Der Datenbankserver kennt bereits die komplette Query als Template, bevor überhaupt PHP-Variablen mit der Query in Berührung kommen. Bei dem anschließenden Ersetzen der Platzhalter durch Variableninhalte werden konkrete Typen mitgegeben. Dadurch weiß der Server wie er die Werte zu interpretieren hat. SQL-Injections haben damit keine Chance mehr.

Anwendungsbeispiel - Bound Result Prepared Statements

Bei den sogenannten „Bound Result Prepared Statements" wird die Rückgabe einer SELECT-Abfrage in definierten Variablen gespeichert.

```
$db = new mysqli(‚localhost', ‚user', ‚passwort',
                    ‚datenbank');
if ($stmt = $db->prepare(„SELECT id, vorname, nachname
                    FROM kunden")) {
    $stmt->execute();
    $stmt->bind_result($rId, $rVorname, $rNachname);
    while ($stmt->fetch()) {
        printf(„%u -> %s, %s\n", $rId, $rVorname,
                    $rNachname);
    }
    $stmt->close();
}
$db->close();
```

Wie zuvor auch bei dem Bound Parameter Prepared Statement, wird eine SQL-Abfrage in der Datenbank hinterlegt und das Ergebnis durch `$stmt->bind_result` festgelegten PHP-Variablen zugewiesen. Die Geschwindigkeit einer solchen Abfrage ist wesentlich höher als die einer gewöhnlichen MySQL Abfrage über das alter Interface (ext/mysql). Da die SELECT-Abfrage vom Datenbankserver nur einmal interpretiert und validiert werden muss, spart man sich ab dem zweiten Aufruf eine Menge an Zeit, es muss lediglich die Information übertragen werden, welche (bereits gepufferte) Query ausgeführt werden soll, diese muss weder neu interpretiert noch validiert werden. Bei einigen hundert SELECT-Abfragen fällt eine solche Optimierung schon deutlich ins Gewicht. Ebenso spielt die intern optimierte MySQLi-Schnittstelle eine wesentliche Rolle in Sachen Geschwindigkeitszuwachs.

Möchte man nur bestimmte Zeilen in einer Tabelle selektieren, so kombiniert man einfach Bound Parameter und Bound Result Prepared Statements miteinander.

$db = new mysqli(‚localhost', ‚user', ‚passwort', ‚datenbank');

```
if ($stmt = $db->prepare(„SELECT id, vorname, nachname
                    FROM kunden WHERE id = ?")) {
    $stmt->bind_param(‚i', $pId);
    $stmt->execute();
    $stmt->bind_result($rId, $rVorname, $rNachname);
    $stmt->fetch();
    printf(„%u -> %s, %s\n", $rId, $rVorname,
                    $rNachname);
    $stmt->close();
}
$db->close();
```

Somit erhält man eine sichere und performante Schnittstelle zur Datenbank.

Anwendungsbeispiel – Prozedurale Variante

Die prozedurale unterscheidet sich von der objektorientierten Variante nur durch eine andere Befehlssyntax (logischerweise), die Funktionen sind die gleichen und die Vorteile somit natürlich auch.

```
$db = mysqli_connect(„localhost", „user", „passwort",
                    „datenbank");
if ($stmt = mysqli_prepare($db, „SELECT id, vorname,
                    nachname FROM kunden WHERE id = ?")) {
    mysqli_stmt_bind_param($stmt, „i", $pId);
     mysqli_stmt_execute($stmt);
     mysqli_stmt_bind_result($stmt, $rId, $rVorname,
                    $rNachname);
     mysqli_stmt_fetch($stmt);
     printf(„%u -> %s, %s\n", $rId, $rVorname,
                    $rNachname);
```

```
    mysqli_stmt_close($stmt);
}
mysqli_close($db);
```

Wie man an diesem Beispiel sehen kann, ist die Syntax zwar anders, die nötigen Schritte sind aber die gleichen. Hier muss man also lediglich entscheiden welche Struktur besser in die eigene Applikation passt. Basiert die Software noch auf PHP 4 und damit auf prozeduralen Strukturen, so bietet sich auch hier die prozedurale Variante der Datenbankzugriffe an. Denkt man hingegen an ein baldiges Update auf objektorientierte Programmierung oder hat bereits eine objektorientierte Applikation vor sich, so bietet sich natürlich das OO-Interface von MySQLi an.

Fazit

Betrachtet man die nicht wesentlich geänderte Syntax der PHP-Befehle und sieht auf der anderen Seite den enormen Gewinn an Sicherheit und Performance, so muss man eigentlich nicht lange überlegen um die Vorteile und Stärken der MySQLi-API zu erkennen.

- SQL-Injections sind unmöglich.
- Die Geschwindigkeit ist wesentlich höher, vor allem bei Massenabfragen.
- Der Quellcode ist besser strukturiert und die Abfragen sind besser lesbar.

In der heutigen Zeit ist es unerlässlich sich vor Angriffen zu schützen. Gerade auf Webseiten ist dies ein wichtiges Thema. Wer die Gefahr von SQL-Injections noch nicht erkannt hat, läuft große Gefahr Opfer einer solchen Attacke zu werden. Auch schon vor MySQLi war es natürlich möglich, sich durch korrekte Programmierung vor solchen Angriffen zu schützen. Jedoch musste man immer darauf achten, die Variablen welche durch den User gefüllt werden können, vor einer SQL-Abfrage zu säubern. Mit Hilfe der Prepared Statements nimmt einem die Datenbank einen Teil dieser Arbeit ab, bzw. sorgt dafür, dass bei vergessener Säuberung einer Variablen trotzdem keine SQL-Injection möglich ist.

Dies ist logischerweise kein Freibrief die Säuberung gänzlich zu unterlassen. Man sollte auch weiterhin zu Beginn einer Aktion jede Variable überprüfen und gegebenenfalls säubern, um spätere Probleme zu vermeiden. Das Wissen, dass eine Injection unmöglich ist, beruhigt jedoch das Entwicklergewissen.

TOBIAS KÖLLIGAN

Der Author ist IT-Consultant und beschäftigt sich seit vielen Jahren schwerpunktmäßig mit der Softwareentwicklung im Enterprise-Bereich (Java, Oracle und PHP/MySQL). Er ist außerdem Betreiber der Software-Suchmaschine alternative-zu.de sowie des Webmasterportals webmaster-eye.de.