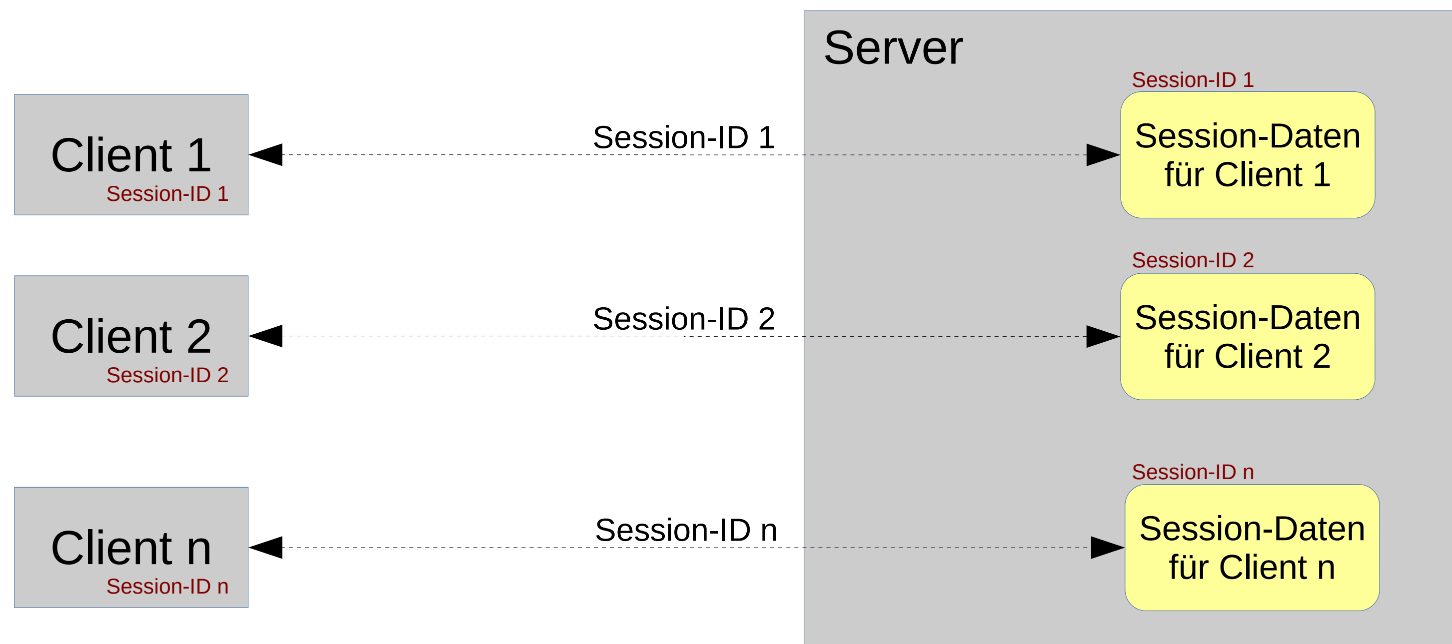
Sessions

- **Session-Daten** *(„Sitzungsdaten“)*
  - Client-spezifische Parameter, die nur im Server zugänglich sind
    - Parameter sind **nur im Server zugänglich**, nicht im Client
    - Parameter werden im Server mittelfristig unter einer **Session-ID** gespeichert
      - z.B. in einer **Datei** oder einer **Datenbank**
    - Die **Session-ID** muss vom Client bei (allen) Requests übermittelt werden



Sessions und Authentifizierungs-Tokens

- **Session-IDs in PHP**
  - PHP enthält bereits einen Mechanismus, der diese Verwaltung durchführt
    - Ordnet neuen Clients eine Session-ID zu
    - Bietet Interface, um bequem Server-interne Daten zur Session-ID zuzuordnen
    - Speichert diese Daten dauerhaft
      - z.B. in einer Datei unter „`/var/lib/php*`" auf dem Server
  - Dazu muss am Anfang die Funktion **`session_start()`** aufgerufen werden (→ php.net)
    - Dadurch wird ein Cookie „PHPSESSID" mit der generierten Session-ID gesetzt.
    - Zudem wird die superglobale Variable **$_SESSION** aktiviert
      - Hier kann man von PHP aus Werte zuweisen, die dauerhaft **im Server** gespeichert werden
      - d.h. auch beim nächsten Request zu dieser Sitzung (Cookie PHPSESSID) sind diese Werte verfügbar
      - Der Client kann diese Werte nicht manipulieren!

        Übungsfrage: Warum?

Sessions und Authentifizierungs-Tokens

- **Sessions nutzen – Beispiel: Anzahl Seitenaufrufe**
  - Session-Daten sind auch weitaus handlicher zu ändern als Cookies, da sie auch nach der ersten Ausgaben modifiziert werden können. (Frage: Warum?)

```
<?php session_start(); ?>

<!DOCTYPE html>
<html> <head> <!-- … --> </head> <body>

<?php
   // Wir zählen die Zahl der Aufrufe in dieser Session
   if (!isset($_SESSION['number_of_calls'])) {
      $_SESSION['number_of_calls'] = 0;
   } else {
      $_SESSION['number_of_calls']++;
   }
   // Die Änderung an der Variablen wird sofort dauerhaft wirksam
?>

<p>Sie haben
   <?php echo $_SESSION['number_of_calls']; ?>
   Aufrufe in dieser Session gemacht.
</body>
</html>
```

Sessions

- **Die Session-ID ...**
  - ist für jeden Client unterschiedlich
    - **Identifiziert** aus Sicht des Servers den Client eindeutig (über längere Zeit)
  - dient zur Zuordnung der im Server gespeicherten **Session-Daten** zur Sitzung (und somit zum Client)
    - Der Server kann viele solche Sessions mit verschiedenen Clients zugleich haben
  - muss vom Client bei Requests mitgeliefert werden
    - Meist als Cookie („**Session-Cookie**“)
    - Alle Parameterübergabeverfahren (GET, POST, Pfad) sind aber möglich
  - fungiert als **Authentifizierungs-Token**
    - Wenn ein Client sich **authentifiziert** hat (also z.B. gültige Login-Daten in einem **Login-Formular** eingegeben wurden), kann der Server die bewiesene Identität auch der **Session** zuordnen, die **Session-ID** wird zum Authentifzierungs-Token.
  - darf also nicht erratbar sein oder Dritten offenbart werden
    - Sonst könnte ein Angreifer eine fremde **Session** (und damit Identität) „**stehlen**“

Sessions und Authentifizierungs-Tokens

- **Cookies als Authentifizierungs-Tokens**
  - Session-Cookies sind Authentifizierungs-Tokens
    - Sie identifizieren den Client (Browser-Instanz) → Nutzer
    - Es kann ein Account/Benutzer zugeordnet werden, wenn dieser eingeloggt ist
    - Aber **auch ohne Login** kann man die Folge von Requests dem Client zuordnen (anonyme Session)
      - Beispiel: Warenkorb in einer Shopping-Plattform:
        Man kann Gegenstände in den Warenkorb legen, ohne eingeloggt zu sein
  - Wie sollte so ein Session-Cookie aussehen
    - (Schlechte) Idee: Benutzername
      - Problem: Manipulierbar (der Nutzer könnte einen anderen Benutzernamen erraten und das Cookie ändern)
      - Problem: Keine anonyme Session mit späterem Login möglich
    - Idee: Zufallszahl (ausreichend komplex, „**Session-ID**“)
      - Manipulationssicher, da andere Zufallszahl nicht erratbar
      - anonyme Sitzung möglich
      - Session-Daten müssen im Server unter der Session-ID gespeichert werden

Sessions und Authentifizierungs-Tokens

- **Login-Session setzen und nutzen**

```php
<?php
   session_start();  // Session wieder aufnehmen oder ggf. neu erzeugen

   function get_login($id = NULL, $password = NULL) {

      global $user_data, $user_id;
      $user_data = $user_id = NULL;
      $u = get_userdata($id);   // liefert assoz. Array u.a. mit Passwort

      if ($u && @$u['password'] == $password ) {
         // neuer Login erfolgreich
         $user_id   = $id;
         $user_data = $u;
         @$_SESSION['user_id'] = $id;  // User der Session zuordnen:
                                       // Zuweisung ist dauerhaft!
      }

      elseif ( @$_SESSION['user_id'] ) {
         // Bestehenden User-Login aus Session lesen:
         $user_id   = @$_SESSION['user_id'];
         $user_data = get_userdata($user_id);
      }

   }

   // ...
   get_login(@$_POST['name'], @$_POST['password']);
?>
```

Default: nicht eingeloggt

Hiermit neu eingeloggt

Oder zuvor bereits eingeloggt

Sessions und Authentifizierungs-Tokens

- **Login-Session setzen und nutzen** (Fortsetzung)
  - Die in den Session-Daten gespeicherten Daten gehen beim Ende der Session verloren …
    - Wenn das Cookie abläuft (Lebensdauer)
    - Wenn das Cookie (z.B. am Ende einer Browser-Sitzung) gelöscht wird
  - Die Eigenschaften des Cookies können gesteuert werden:
    - in der PHP-Konfigurationsdatei php.ini
      - z.B. session.cookie_lifetime int
        - Lebensdauer in Sekunden oder 0 (bis zum Ende der Browser-Sitzung – Default)
        - z.B. für 1 Stunde: „session.cookie_lifetime 3600“
    - oder dynamisch mit ...
      - ```
        void session_set_cookie_params (
               int $lifetime
            [, string $path
            [, string $domain
            [, bool $secure = false
            [, bool $httponly = false ]]]] )
        ```
      - vor session_start() aufrufen (Verständnisfrage: **Warum?**)
        ```

PHP-Schnittstelle zu MySQL

- **PHP-Schnittstelle zu MySQL**
  - Es gibt mehrere Prozedurale und OO-Schnittstellen
  - Wir benutzen `mysqli` in prozeduraler Form
    - Siehe http://php.net/manual/de/book.mysqli.php
    - Siehe http://php.net/manual/de/mysqli.quickstart.dual-interface.php
- **Erste Schritte ...**
  - Verbindung zum Server aufbauen:

```
$mysqli = mysqli_connect(  $mysql_server,
                           $mysql_user,
                           $mysql_password,
                           $mysql_database
                         );
```

  - Mit dem Objekt $mysqli kann man dann auf die DB zugreifen
    - Am Ende des PHP-Scripts sollte die Verbindung geschlossen werden

```
mysqli_close($mysqli);
```

oder

```
$mysqli->close();
```

PHP-Schnittstelle zu MySQL

- **Verbindungsaufbau als PHP-Datei**

```php
<?php
     // Zugangsdaten zur DB aus Datei lesen
     include('/home/lamp/.mysql_credentials');

     // Verbindung aufbauen
     $mysqli = mysqli_connect(
                           $mysql_server,
                           $mysql_user,
                           $mysql_password,
                           $mysql_database );

     /* hier mit $mysqli Queries stellen und verarbeiten */

     mysqli_close($mysqli);   // oder $mysqli->close();
?>
```

Importiert Inhalt der PHP-Datei hierher

  – Wir nehmen an, dass in der Require-Datei die benötigten Zugangsdaten liegen:

```php
<?php  // Zugangsdaten zur DB
     $mysql_server          = 'localhost';
     $mysql_user            = 'lamp';
     $mysql_password        = 'asdf3asdf3asdf';
     $mysql_database        = 'wikipedia_sql_example';
?>
```

Idee: Zentral konfiguriert + nicht im htdocs-Baum

PHP-Schnittstelle zu MySQL

- **Query stellen und Daten empfangen**

```
<?php
    $mysqli = mysqli_connect(...);

    // Query stellen …
    $res = mysqli_query($mysqli, "SELECT * FROM Student");

    // Datensätze abrufen und verarbeiten
    while ($row = mysqli_fetch_assoc($res)) {
        print "<pre>";
        print_r($row);
        print "</pre>";
    }

    mysqli_close($mysqli);
?>
```

  - Ergebnis:

```
Array
(
    [MatrNr] => 25403
    [Name] => Jonas
)

Array
(
    [MatrNr] => 26120
    [Name] => Fichte
)
```

```
Array
(
    [MatrNr] => 27103
    [Name] => Fauler
)

Array
(
    [MatrNr] => 27104
    [Name] => Peter
)
```

```
Array
(
    [MatrNr] => 27106
    [Name] => Neumann
)
```

PHP-Schnittstelle zu MySQL

- **Attributzugriff in Datensätzen aus Queries**

```
<?php
    $mysqli = mysqli_connect(...);

    // Query stellen …
    $res = mysqli_query($mysqli, "SELECT * FROM Student");

    // Datensätze abrufen und verarbeiten
    print "<table>\n<tr><th>MatrNr <th>Name";
    while ($row = mysqli_fetch_assoc($res)) {
        print "\n<tr>";
        print "<td>" . $row['MatrNr'];
        print "<td>" . $row['Name'];
    }
    print "\n</table>";
?>
```

  - Ergebnis:

MatrNr	Name
25403	Jonas
26120	Fichte
27103	Fauler
27104	Peter
27106	Neumann

PHP-Schnittstelle zu MySQL

- **Fehlerbehandlung: Verbindungsaufbau**
  - Fehler bei `mysqli_connect()`
    - Liefert False zurück
    - **`mysqli_connect_errno()`** liefert den Fehlercode
    - **`mysqli_connect_error()`** liefert die Fehlerbeschreibung

```
<?php
    $mysqli = mysqli_connect(...);
    if (!$mysqli) {
            print 'Connect Error ('
                    . mysqli_connect_errno() . ') '
                    . mysqli_connect_error() . "\n";
            exit(1);
    }

    // Hier ist die Verbindung in $mysqli erfolgreich hergestellt …

?>
```

PHP-Schnittstelle zu MySQL

- **Fehlerbehandlung: Queries**
  - Fehler bei `mysqli_query()`
    - Liefert False zurück
    - **`mysqli_errno($mysqli)`** liefert den Fehlercode
    - **`mysqli_error($mysqli)`** liefert die Fehlerbeschreibung

```
<?php
    $mysqli = mysqli_connect(...);
    if (!$mysqli) { /* … */ }

    $res = mysqli_query($mysqli, "SELECT * FROM Student");
    if (!$res) {
            print 'Query Error ('
                  . mysqli_errno($mysqli) . ') '
                  . mysqli_error($mysqli) . "\n";
            exit(1);
    }

    // Hier kann das Query-Ergebnis in $res verarbeitet werden
?>
```

PHP-Schnittstelle zu MySQL

- **Fehlerbehandlung: Warnungen bei Queries**
  - Warnungen bei `mysqli_query()`
    - Auch wenn kein Fehler gemeldet wird, können **relevante Probleme** aufgetreten sein, die MySQL nur als **Warnungen** behandelt.
    - `mysqli_warning_count($mysqli)` liefert die Anzahl
    - `mysqli_get_warnings( $mysqli)` Warnungen als interaktives Objekt

```php
<?php
    $mysqli = mysqli_connect(...);
    if (!$mysqli) { /* … */ }

    $res = mysqli_query($mysqli, "SELECT * FROM Student");
    if (!$res) { /* … */ }
    if (mysqli_warning_count($mysqli)) {
            $warn = mysqli_get_warnings($mysqli);
            do {
                print 'Query Warning ('
                      . $warn->errno   . ') '
                      . $warn->message . "\n";
            } while ($warn->next());
    }
    // Auch bei Warnungen kann das Query-Ergebnis verarbeitet werden
```

PHP-Schnittstelle zu MySQL

- **Fehlerbehandlung: Warnungen bei Queries**
  - Die gelieferten Warnungen entsprechen den Ausgaben des SQL-Kommandos „SHOW WARNINGS“
  - Beispiel: Beim Einfügen eines Datensatzes wird ein Wert nicht explizit angegeben, der NOT NULL ist und keinen DEFAULT hat.
    - z.B. Student.Name im obigen Beispiel

```
INSERT INTO Student (MatrNr) VALUES (34567);
  Query OK, 1 row affected, 1 warning (0.09 sec)
SHOW WARNINGS;
+---------+------+------------------------------------------+
| Level   | Code | Message                                  |
+---------+------+------------------------------------------+
| Warning | 1364 | Field 'Name' doesn't have a default value |
+---------+------+------------------------------------------+
SELECT * FROM Student;
+--------+---------+
| MatrNr | Name    |
+--------+---------+
|    ... | ...     |
|  27105 | Fauler  |
|  34567 |         |
+--------+---------+
```

PHP-Schnittstelle zu MySQL

- **Fehlerbehandlung und -Meldung allgemein**
  - Die obige Art der Fehlerausgabe ist **nur für Testsysteme** sinnvoll.
    - Der Anwender kann mir der Fehlermeldung meist nichts anfangen.
    - Der Administrator erfährt evtl. gar nichts von dem Problem.
    - Es werden durch die Fehlertexte evtl. Informationen an außenstehende geliefert, die für Angriffe genutzt werden können.
  - Produktivsysteme sollten dem Endanwender nur (freundlich) anzeigen, dass es überhaupt ein Problem gab.
    - Fehler sollten in ein Logdatei oder eine Log-Datenbank geschrieben werden.
    - Kritische Fehler sollten ggf. aktiv an den Administrator gemeldet werden.
      - Zentrale Systemüberwachung oder Email
    - **Zur Erinnerung**: *Fehlerbehandlung* im Kapitel PHP in Teil 1 der Vorlesung
      - `set_error_handler(my_callback_function, E_ALL);`
        - Eigene Behandlung von Fehlermeldungen
      - `error_log`('Etwas schlimmes ist passiert', 1, 'admin@my.domain');
        - Schreibt Log-Text und sendet Email

PHP-Schnittstelle zu MySQL

- **Queries mit Daten aus Parametern**
  - Das PHP-Script erzeugt einen Query-String, der von der Eingabe abhängt
    - Wurde kein Suchausdruck angegeben, werden alle Studenten angezeigt
    - Wurde ein Suchausdruck angegeben, werden alle Datensätze angezeigt, bei denen der Ausdruck im Namen vorkommt.

```
<?php
    $search = @$_GET['name'];
    // ...
    // Query stellen …
    $query = "SELECT * FROM Student";
    if ($search)
              $query .= "  WHERE Name  LIKE '%$search%'";
    $res = mysqli_query($mysqli, $query);
```

- Verständnisfrage: ist Ihnen klar, wie diese Suche funktioniert?
  - Überlegen Sie ggf. wie der Query-String aussieht und wie LIKE funktioniert

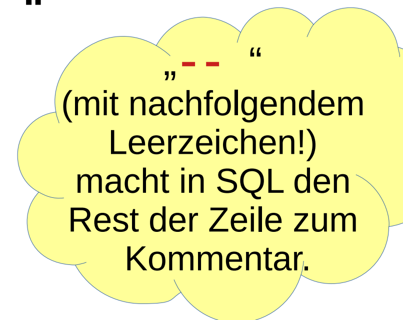
PHP-Schnittstelle zu MySQL

- **Queries mit Daten aus Parametern**

```
<?php $search = @$_GET['name']; ?>

<h1>Search for Student</h1>
<form action="" method=get>
     <input type=text     name=name      value="<?php print $search; ?>">
     <input type=submit   name=search    value="Search" >
</form>
<h1>Search Result
     <?php if ($search) print "for '$search'"; ?>
</h1>
<?php
     $mysqli = mysqli_connect(...);

     // Query stellen …
     $query = "SELECT * FROM Student";
     if ($search)
               $query .= "  WHERE Name  LIKE '%$search%'";
     $res = mysqli_query($mysqli, $query);

     // Datensätze abrufen und verarbeiten
     print "<table>\n<tr><th>MatrNr <th>Name";
     while ($row = mysqli_fetch_assoc($res)) {
         print "\n<tr>";
         print "<td>" . $row['Name'];
```

PHP-Schnittstelle zu MySQL

- **Queries mit Daten aus Parametern** (Angriffe)
  - Beispiel: Suchausdruck „au“
    - → SELECT * FROM Student WHERE Name LIKE '%au%'
      - Korrekt, liefert alle Studenten, deren Name „au“ enthält
  - Beispiel: Suchausdruck „'“ (das Apostroph-Zeichen)
    - → SELECT * FROM Student WHERE Name LIKE '%'%'
      - Der Query-String ist offensichtlich nicht syntaktisch korrekt
      - Es entsteht eine Fehlermeldung aus der Datenbank
  - Beispiel: Suchausdruck „' AND MatrNr > 27000 -- “
    - → SELECT * FROM Student WHERE Name LIKE '%' AND MatrNr > 27000 -- %'
      - Der String ist syntaktisch korrekt, bedeutet aber etwas anderes

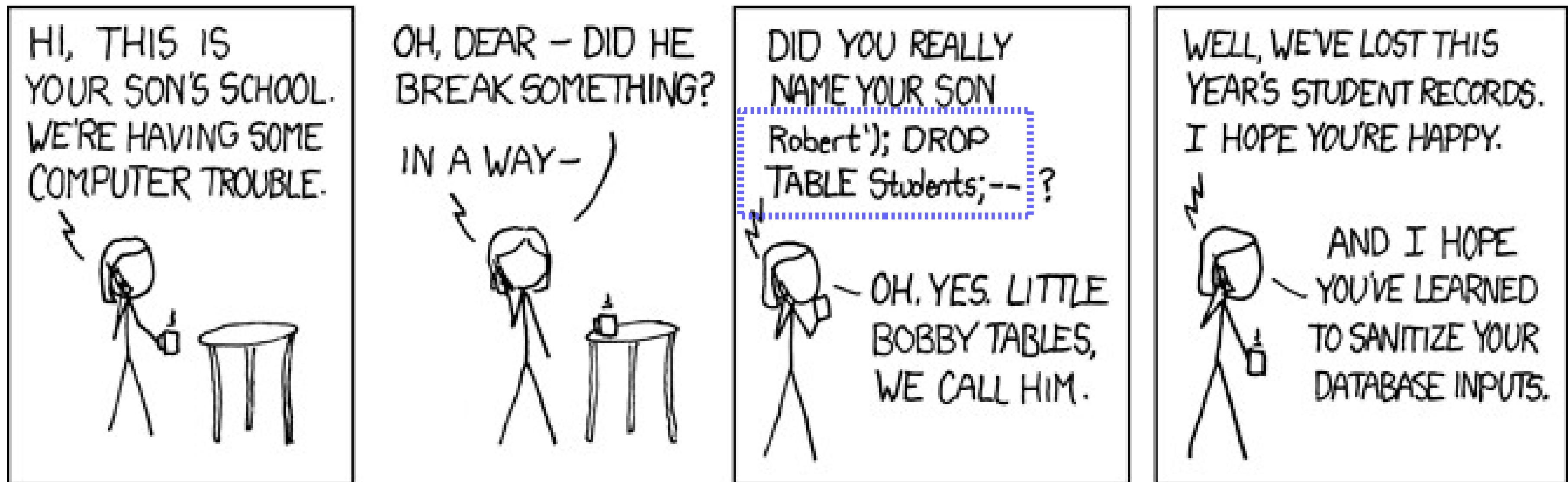


→ Ein Teil der **SQL-Instruktion** stammt **aus Nutzerdaten**!

- **Das nennt man SQL-Injection**

Injections

- **XKCD-Comic: „*Little Bobby Tables*“**



https://xkcd.com/327/

  - Übung:
    - Wie sah das gesamte SQL-Statement hier also (vermutlich) aus?
    - Was soll es bewirken?

Injections

- **SQL-Injection**
  - Bei einer SQL-Injection stammen nicht nur (wie beabsichtigt) Daten im Query-String, sondern auch **SQL-Statements**, **Bedingungen**, etc. aus unzuverlässigen Quellen (z.B. vom Endanwender)
  - Ursache: Bei der Bildung des Query-Strings werden SQL-Statements und Benutzer-Daten kombiniert.
    - Wenn bei der Zerlegung des entstandenen Strings eine andere Interpretation entsteht als ursprünglich beabsichtigt, können Benutzerdaten fälschlich als SQL-Statements interpretiert werden
    - Siehe auch http://de.wikipedia.org/wiki/SQL-Injection
- **Mögliche Folgen**
  - Syntaktische Fehler beim Datenbankzugriff
  - Manipulation von Auswahl-Bedingungen (WHERE-Bedingungen)
    - Dadurch kann man z.B. fremde Datensätze sehen
  - Einschleusung kompletter Datenbank-Anweisungen
    - Dadurch kann man volle Kontrolle über die Datenbank erlangen

Injections

- **Mögliche Folge: Ausweitung von Ergebnis-Menge**
  - Dadurch kann man z.B. fremde private Datensätze sehen
  - Beispiel:
    - Ein eingeloggter Student soll die von ihm selbst (MatrNr aus Session) belegten Vorlesungen nach Titeln durchsuchen können.

```
<?php
     $search = @$_GET['titel'];
     $matrnr = @$_SESSION['MatrNr'];  // MatrNr des eingeloggten Stud.
     // ...
     $query = "SELECT * FROM hört JOIN Vorlesung USING (VorlNr) WHERE";
     if ($search)
            $query .= " Titel LIKE '%$search%' AND";
     $query .= " MatrNr=$matnr";
```

    - Beabsichtigte Query (Suchausdruck „ET“):
      - `SELECT * FROM hört JOIN Vorlesung USING (VorlNr) WHERE Titel LIKE '%ET%' AND MatrNr=25403`
    - Query durch Injection-Suchausdruck „`' --` “
      - `SELECT * FROM hört JOIN Vorlesung USING (VorlNr) WHERE Titel LIKE '%' -- %' AND MatrNr=25403`

Liefert nur eigene Vorlesungen

Liefert Vorlesungen aller Studenten

Injections

- **Mögliche Folge: Einschleusen neuer SQL-Anweisungen**
  - Dadurch kann man beliebige Manipulationen in der Datenbank vornehmen (im Rahmen der Rechte des Benutzers)
  - Beispiel (zu Studenten-Suchfunktion von oben):
    - **Injection-Suchausdruck**

      „' ; DELETE FROM Professor -- '“

    - **Resultierende Anfrage:**

      - Löscht alle Professor-Datensätze
    - **Analog**: ändern von Passwörtern, exmatrikulieren von Studierenden, …
  - Dieses Beispiel funktioniert bei mysqli_query nicht, da hier keine **mehrteiligen SQL-Queries** erlaubt sind (→ dann Syntax-Fehler)

Injections

- **Gegenmaßnahmen zu SQL-Injections (1)**
  - Daten im Query-String vor Einbettung geeignet „escapen“
    - Vermeidet die Unterbrechung des umgebenden '…'-Strings
    - Funktion: **mysqli_real_escape_string** (mysqli $link , string $escapestr)
      - Behandelt NUL (ASCII 0), \n, \r, \ \, ' , ", Control-Z.
      - Wird auf jeden Parameter einzeln angewandt
    - Beispiel:

```
<?php
    $search = @$_GET['name'];
    // ...
    // Query stellen …
    $query = "SELECT * FROM Student";
    if ($search) {
            $xsearch = mysqli_real_escape_string($mysqli, $search)
            $query .= " WHERE Name  LIKE '%$xsearch%'";
    }
    $res = mysqli_query($mysqli, $query);
```

- Verständnisfrage: Kann ich den ganzen Query-String damit behandeln?

Injections

- **Gegenmaßnahmen zu SQL-Injections (2a)**
  - Idee: Daten nicht in Query-String einbetten, sondern **separat** an das DBMS übergeben
  - **Technik: Prepared Statements**
    - Ich formuliere zunächst den Query-String ohne Daten.
      - Er enthält überall ein „**?**“, wo Daten eingefügt werden sollen.
      - Beispiel: „`SELECT * FROM Student WHERE a=? AND b=?`“
      - Ziel: Ein „prepared Statement“
    - Erst in einem späteren Aufruf „binde“ ich Parameter an das prepared Statement
      - Beides wird getrennt an das DBMS übergeben und verarbeitet.
      - Es es gibt also keine Vermischung von Code (SQL-Statements) und Daten

Injections

- **Gegenmaßnahmen zu SQL-Injections (2b)**
  - Daten nicht in Query-String einbetten, sondern **explizit** übergeben
    - **Technik: Prepared Statements**
      - Query-String enthält überall ein „**?**“, wo Daten eingefügt werden sollen.
      - mit mysqli_prepare() wird ein Prepared Statement erzeugt
      - mit mysqli_stmt_bind_param() werden die Parameterwerte gebunden
      - mit mysqli_stmt_execute() wird der Query ausgeführt
      - mit mysqli_stmt_get_result() wird das Ergebnis geholt
    - Beispiel:

```
<?php
     $a = @$_GET['a'];         $b = @$_GET['b'];
     $stmt = mysqli_prepare($mysqli,
                      "SELECT * FROM Student WHERE a=? AND b=?");
     mysqli_stmt_bind_param($stmt, 'ss', $a, $b);
     mysqli_stmt_execute($stmt);
     $res = mysqli_stmt_get_result($stmt);

     while ($row = mysqli_fetch_assoc($res)) {
         // ...
```

    - Der 2. Parameter in mysqli_stmt_bind_param() gibt die Typen an
      - 's'=String, 'i'=integer, 'd'=Fließkommazahl, 'b'=BLOB

Injections

- **Gegenmaßnahmen zu SQL-Injections (3)**
  - Keine mehrteiligen SQL-Statements in Query-Strings zulassen
    - Dies ist bei mysqli_query ja gesichert (s.o.)
    - Es bietet aber keine vollständige Sicherheit
  - Filterung von Eingaben
    - z.B. bekannte Injection-Muster erkennen und ablehnen oder bereinigen
    - Problem: **False Positives**
      - Zulässige Eingaben werden abgelehnt oder verändert
    - Problem: **False Negatives**
      - Unbekannte (neue / modifizierte) Angriffe werden nicht sicher erkannt
- **SQL-Injections können sehr vielseitig ablaufen**
  - Siehe auch http://de.wikipedia.org/wiki/SQL-Injection
    - Überschreiben von Server-Dateien mit INTO OUTFILE
    - Überlasten des Servers, löschen von Tabellen, Ändern von Rechten, ...

Injections

- **Injections**
  - Allgemein bezeichnet man mit „**Injection**" jede Form von fremdbestimmten Daten, die als Code interpretiert werden
  - „**Code**" in diesem Sinne sind **alle Kontrollstrukturen**
    - also nicht nur klassische Programmiersprachen
    - **Beispiele:**
      - SQL-Statements,
      - Javascript-Code,
      - HTML-Tags und HTML-Tag-Parameter
  - „**fremdbestimmte Daten**" sind alle Daten, die von nicht vertrauenswürdiger Seite beeinflussbar sind
    - **Beispiele:**
      - Benutzereingaben (GET- oder POST-Parameter)
      - Cookies
      - HTML-Header
      - hochgeladene Dateien

Injections

- **Injections (Beispiele)**
  - Über einen POST-Parameter werden SQL-Fragmente übergeben, die eine Verfälschung der in der Folge erzeugten SQL-Queries verursachen
    - **SQL-Injection**
    - Gefahr: Fehler, Manipulation der Datenbank, Geheimnis-Enthüllung
    - Gegenmaßnamen (s.o.):
      - Parameter-Escaping
      - sichere Query-Erzeugung
      - Daten-Filterung (evtl. unsicher)
  - Das obige Script enthält eine solche Verwundbarkeit

Injections

- **Queries mit Daten aus Parametern**

```
<?php $search = @$_GET['name']; ?>

<h1>Search for Student</h1>
<form action="" method=get>
     <input type=text     name=name      value="<?php print $search; ?>">
     <input type=submit   name=search    value="Search" >
</form>
<h1>Search Result
     <?php if ($search) print "for '$search'"; ?>
</h1>
<?php
     $mysqli = mysqli_connect(...);

     // Query stellen …
     $query = "SELECT * FROM Student";
     if ($search)
               $query .= "  WHERE Name  LIKE '$search'";
     $res = mysqli_query($mysqli, $query);

     // Datensätze abrufen und verarbeiten
     print "<table>\n<tr><th>MatrNr <th>Name";
     while ($row = mysqli_fetch_assoc($res)) {
         print "\n<tr>";
         print "<td>" . $row['Name'];
```

Injections

- **Injections (Beispiele)**
  - Über einen GET-Parameter werden HTML-Fragmente übergeben, die eine Verfälschung der in der Folge erzeugten HTML-Seite verursachen.
  - Diese HTML-Fragmente können auch Javascript enthalten, die im Browser eines Nutzers ausgeführt werden
    - **HTML-Injection**
    - Javascript-Injection („**XSS**“ = **Cross-Site-Scripting**)
    - Gefahr: Fehlerhafte Darstellung, Manipulation der Benutzer-Webseite, Geheimnis-Enthüllung durch Javascript (z.B. gefälschte Passwort-Dialoge)
    - Gegenmaßnamen:
      - Parameter-Escaping (PHP-Funktion htmlspecialchars)
      - Daten-Filterung (z.B. PHP-Funktion strip_tags) (allgemein evtl. unsicher)
  - Das obige Script enthält zwei solcher Verwundbarkeiten

Injections

- **HTML-Injection, XSS**

```
<?php $search = @$_GET['name']; ?>

<h1>Search for Student</h1>
<form action="" method=get>
     <input type=text     name=name      value="<?php print $search; ?>">
     <input type=submit   name=search    value="Search">
</form>
<h1>Search Result
     <?php if ($search) print "for '$search'"; ?>
</h1>
<?php
     $mysqli = mysqli_connect(...);

     // Query stellen …
     $query = "SELECT * FROM Student";
     if ($search)
               $query .= "  WHERE Name  LIKE '$search'";
     $res = mysqli_query($mysqli, $query);

     // Datensätze abrufen und verarbeiten
     print "<table>\n<tr><th>MatrNr <th>Name";
     while ($row = mysqli_fetch_assoc($res)) {
         print "\n<tr>";
         print "<td>" . $row['Name'];
```

Injections

- **HTML-Injection, XSS (Demo)**
  - **Beispiel**: Demo-XSS-Injection String für Such-Feld:

    ```
    " onclick="alert('Hallo')
    ```

    Liefert auf dem verwundbaren Input-Feld:

    ```
    <input type="text" name="name" value="" onclick="alert('Hallo')">
    ```

    Öffnet also beim Klick auf das Text-Eingabefeld einen Alert.

  - ***Übung***
    - Überlegen Sie sich einen Injection-String, durch den Sie das Action-Feld des Formulars auf einen beliebige URL setzen können.
      (Zwei JS-Funktionsaufrufe genügen.)

  - **Angriffsszenario** (bei nicht geschützten Servern)
    - Sie ergänzen die Login-URL einer Bank mit einem GET-Parameter, durch den die Login-Daten an einen fremden Server geschickt werden.
      Dann schicken Sie die URL per gefälschter Email an einen Kunden mit der Aufforderung, sich „*ganz dringend*“ darüber einzuloggen.

Injections

- **Persistente / Reflektierte Injections**
  - Injections müssen nicht immer unmittelbar durch einen Query erfolgen
  - Sie können auch über Daten Erfolgen, die zunächst auf dem Server dauerhaft („**persistent**“) gespeichert werden und erst bei einer späteren Abfrage einen Effekt haben
    - Sie werden sozusagen vom Server „reflektiert“
  - Beispiel:
    - In der Datenbank liegt ein Fragment für eine Javascript-Injection (XSS)
    - In der Datenbank liegt ein Fragment für eine SQL-Injection
      - Verständnisfrage: Wie kann das sein? Sie wurde doch escapt beim Einfügen?
    - Über eine Upload-Funktion wird ein PHP-Script hochgeladen, dass vom Server beim späteren Zugriff ausgeführt wird (Script-Injection)
      - Gegenmaßnahme: Im Upload-Bereich immer aktives Scripting deaktivieren!
  - Das obige Script enthält eine solche Verwundbarkeit

Injections

- **Persistente HTML-Injection, XSS**

```
<?php $search = @$_GET['name']; ?>

<h1>Search for Student</h1>
<form action="" method=get>
     <input type=text     name=name      value="<?php print $search; ?>">
     <input type=submit   name=search    value="Search" >
</form>
<h1>Search Result
     <?php if ($search) print "for '$search'"; ?>
</h1>
<?php
     $mysqli = mysqli_connect(...);

     // Query stellen …
     $query = "SELECT * FROM Student";
     if ($search)
               $query .= "  WHERE Name  LIKE '$search'";
     $res = mysqli_query($mysqli, $query);

     // Datensätze abrufen und verarbeiten
     print "<table>\n<tr><th>MatrNr <th>Name";
     while ($row = mysqli_fetch_assoc($res)) {
         print "\n<tr>";
         print "<td>" . $row['Name'];
```

Injections

- **Persistente / Reflektierte Injections (Demo)**
  - **Beispiel**: Demo-DB-XSS-Injection String für Studenten-Name:

```
INSERT INTO Student (Name) VALUES
                ('<script>alert("DB")</script>');
```

Öffnet beim Auflisten des Studenten in jeder Such-Ausgabe einen Alert.

  - ***Übung***
    - Die DB-Textfelder sind evtl. sehr kurz für komplexeren JS-Code (Student.Name hier 64 Zeichen). Wie könnte man das umgehen?
    - Können Sie sich ein Szenario vorstellen, in dem man ausdrücklich Daten aus der Datenbank ungeschützt ausgeben will?
  - **Angriffsszenario** (bei nicht geschützten Servern)
    - Sie schreiben in ein Forum einen Kommentar. Auf der Einstiegsseite des Forums werden die neuesten Kommentare (ungeschützt) angezeigt.
      Der JS-Code im Kommentar sendet jeweils das Session-Cookie als GET-Parameter an einen fremden Server.

Injections

- **Abgesichertes Beispiel** → mysqli_04_search_safe Quelle + Ausführbar

```
<!DOCTYPE html>
<html>
<head></head>
<body>
<?php $search = @$_GET['name']; ?>

<h1>Search for Student</h1>
<form action="" method=get>
        <input type=text     name=name      value="<?php print htmlspecialchars($search); ?>">
        <input type=submit   name=search    value="Search" >
</form>

<h1>Search Result
        <?php if ($search) print "for '".htmlspecialchars($search)."'"; ?>
</h1>
<?php
        require('/home/lamp/.mysql_credentials');
        $mysqli = mysqli_connect($mysql_server, $mysql_user, $mysql_password, $mysql_database);
        if (!$mysqli) {
                        echo "Failed to connect to MySQL: " . mysqli_connect_error();
                        exit(1);
        }
        $query = "SELECT * FROM Student";
        if ($search) {
                        $xsearch = mysqli_real_escape_string($mysqli, $search);
                        $query .= " WHERE Name LIKE '%$xsearch%'";
                        // $query .= " OR MatrNr LIKE '%$xsearch%'";
        }
        print '<div style="margin: 1em 0; padding: 0 1em; border: thin solid grey;">';
        print "<pre>" . htmlspecialchars($query) . "</pre>";
        $res = mysqli_query($mysqli, $query);
        if (!$res) {
                        echo "Failed to Query MySQL: " . mysqli_error($mysqli);
                        exit(1);
        }
        print "</div>";
        print "<table border=1>\n<tr><th>MatrNr <th>Name";
        while ($row = mysqli_fetch_assoc($res)) {
                        print "\n<tr>";
                        print "<td>" . htmlspecialchars($row['MatrNr']);
                        print "<td>" . htmlspecialchars($row['Name']);
        }
        print "\n</table>";
        mysqli_close($mysqli);
?>
</body>
</html>
```