SQL INJECTION IN PHP

HARMONYMOUS TEAM

Team Members : Raouf Baghejary – Faezeh RoeinFard

Professor : Dr.SobatiMoghadam

Progress Report NO.2

TABLE OF CONTENTS

CONTENTS

SQL INJECTION IN PHP

Preamble

SQL Injection is a technique for taking control of a database query and often results in a compromise of confidentiality. In some cases (e.g. if SELECT 'evil code here' INTO OUTFILE '/var/www/reverse_shell.php' succeeds) this can result in a complete server takeover.

Since code injection (which encompasses SQL, LDAP, OS Command, and XPath Injection techniques) has consistently remained on top of the OWASP Top Ten vulnerabilities, it's a popular topic for bloggers trying to get their feet wet in the application security field.

While more people sharing knowledge about application security is a good thing, unfortunately much of the advice circulating on the Internet (especially on ancient blog posts that rank high on search engines) is outdated, unintentionally misleading, and often dangerous.

HOW TO PREVENT SQL INJECTION (ALMOST) EVERY TIME, GUARANTEED*

Use Prepared Statements, also known as parametrized queries. For example:

```
1.  /**
2.   * Note: This code is provided for demonstration purposes.
3.   *      In general, you want to add some application logic to validate
4.   *      the incoming parameters. You do not need to escape anything.
5.   */
```

```
6. $stmt = $pdo->prepare('SELECT * FROM blog_posts WHERE YEAR(created) = ? AND
   MONTH(created) = ?');
7. if ($stmt->execute([$_GET['year'], $_GET['month']])) {
8.     $posts = $stmt->fetchAll(\PDO::FETCH_ASSOC);
9. }
```

Prepared Statements eliminate any possibility of SQL Injection in your web application. No matter what is passed into the $_GET variables here, the structure of the SQL query cannot be changed by an attacker (unless, of course, you have PDO::ATTR_EMULATE_PREPARES enabled, which means you're not truly using prepared statements).

Note: If you attempt to turn PDO::ATTR_EMULATE_PREPARES off, some versions of some database drivers might ignore you. To be extra cautious, explicitly set the character set in the DSN to one your application and database both use (e.g. UTF-8, which if you're using MySQL, is confusingly called utf8mb4).

Prepared Statements solve a fundamental problem of application security: They separate the data that is to be processed from the instructions that operate on said data by sending them in completely separate packets. This is the same fundamental problem that makes stack/heap overflows possible.

As long as you never concatenate user-provided or environment variables with the SQL statement (and make sure you aren't using emulated prepares) you can for all practical purposes cross SQL injection off your checklist forever.

IMPORTANT CAVEAT AND CLARIFICATION*

Prepared statements secure the interactions between your web application and your database server (if they're on separate machines, they should also be communicating over TLS). It's still possible that an attacker could store a payload in a field that could be dangerous in, for example, a stored procedure. We call this a higher-order SQL injection (the linked Stack Overflow answer refers to them as "second-order", but anything after the initial query is executed should be a target for analysis).

In this situation, our advice would be not to write stored procedures such that they create higher-order SQL injection points.

WHAT ABOUT SANITIZING INPUT?

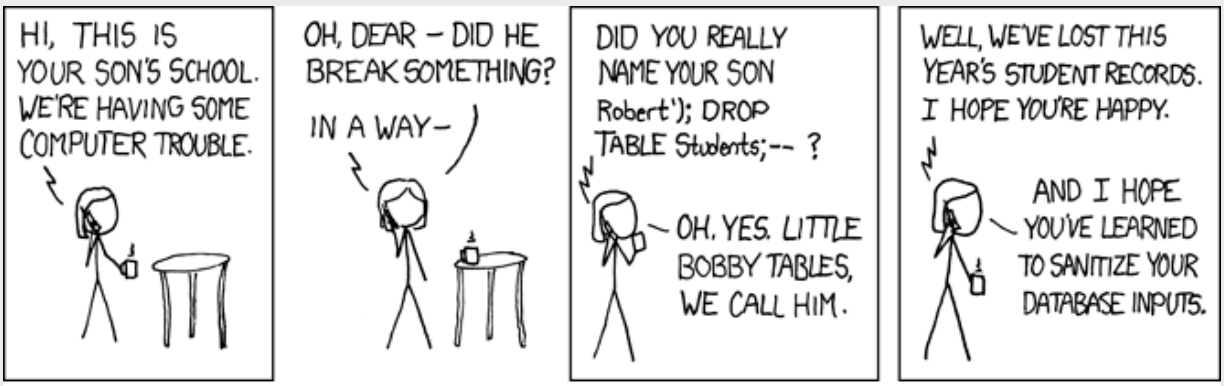


Image from XKCD https://xkcd.com/327/

Many people have seen this 2007 comic from XKCD about SQL Injection exploits. It's frequently cited or included in security conference talks, especially ones addressed to newcomers. The comic has done a lot of good raising awareness of the dangerous of user input in database queries, but its advice to sanitize your database inputs is, by a 2015 understanding of the issues involved, only a half-measure.

You're Likely Better Off Forgetting About Sanitizing Input (in Most Cases)

While it's possible to prevent attacks by rewriting the incoming data stream before you send it to your database driver, it's rife with dangerous nuance and obscure edge-cases. (Both links in the previous sentence are highly recommended.)

Unless you want to take the time to research and attain complete mastery over every Unicode format your application uses or accepts, you're better off not even trying to sanitize

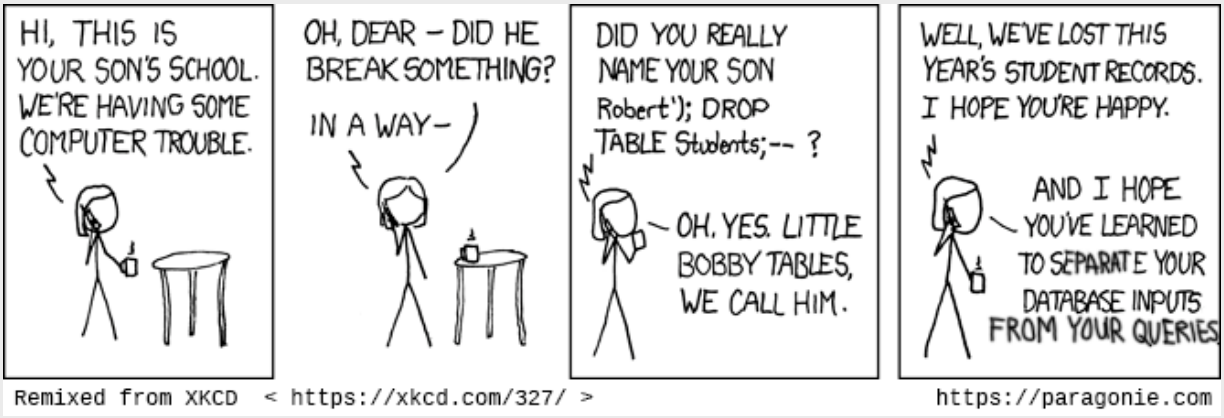
your inputs. Prepared statements are more effective at preventing SQL injection than escaping strings.

Furthermore, altering your incoming data stream can cause data corruption, especially if you are dealing with raw binary blobs (e.g. images or encrypted messages).

Prepared statements are easier and can guarantee SQL Injection prevention.

If user input never has the opportunity to alter the query string, it can never lead to code execution. Prepared statements completely separate code from data.

XKCD's author Randall Munroe is a smart cookie. If this comic were being written today, the hacker mom character probably would have said this instead:



Input Should Still Be Validated

Validation is not the same thing as sanitation. Prepared statements can prevent SQL Injection, but they cannot save you from bad data. For most cases, filter_var() is useful here.

```
$email = filter_var($_POST['email'], FILTER_VALIDATE_EMAIL);
if (empty($email)) {
    throw new \InvalidArgumentException('Invalid email address');
}
```

Note : filter_var() validates that the given email address string conforms to the RFC specification. It does not guarantee that there is an open inbox at that address, nor does it check that the domain name is registered. A valid email address is still not safe to use in raw queries, nor to display on a web page without filtering to prevent XSS attacks

---

WHAT ABOUT COLUMN AND TABLE IDENTIFIERS?

Since column and table identifiers are part of the query structure, you cannot parametrize them. Consequently, if the application you are developing requires a dynamic query structure where tables or columns are selected by the user, you should opt for a whitelist.

A whitelist is an application logic strategy that explicitly only allows a few accepted values and either rejects the rest or uses a sane default. Contrast it with a blacklist, which only forbids known-bad inputs. In most cases, whitelists are better for security than blacklists.

```
$qs = 'SELECT * FROM photos WHERE album = ?';
// Use switch-case for an explicit whitelist
switch ($_POST['orderby']) {
```

```
    case 'name':
    case 'exifdate':
    case 'uploaded':
      // These strings are trusted and expected
      $qs .= ' ORDER BY ' . $_POST['orderby'];
      if (!empty($_POST['asc'])) {
          $qs .= ' ASC';
      } else {
          $qs .= ' DESC';
      }
      break;
    default:
      // Some other value was passed. Let's just order by photo ID in descending order.
      $qs .= ' ORDER BY photoid DESC';
    }
$stmt = $db->prepare($qs);
if ($stmt->execute([$_POST['album_id']])) {
    $photos = $stmt->fetchAll(\PDO::FETCH_ASSOC);
}
```

If you're allowing the end user to provide the table and/or column names, because identifiers cannot be parameterized, you still must resort to escaping. In these situations, we recommend the following:

Don't: Just escape SQL meta characters (e.g. ')

Do: Filter out every character that isn't allowed.

The following code snippet will only allow table names that begin with an uppercase or lowercase letter, followed by any number of alphanumeric characters and underscores.

```
if (!preg_match('/^[A-Za-z][A-Za-z0-9_]*$/', $table)) {
    throw new AppSpecificSecurityException("Possible SQL injection attempt.");
}
// And now you can safely use it in a query:
$stmt = $pdo->prepare("SELECT * FROM {$table}");
if ($stmt->execute()) {
    $results = $stmt->fetchAll(PDO::FETCH_ASSOC);
}
```

WHAT IF USING PREPARED STATEMENTS SEEMS TOO CUMBERSOME?

The first time a developer encounters prepared statements, they can feel frustrated about the prospect of being forced to write a lot of redundant code (prepare, execute, fetch; prepare, execute, fetch; ad nauseam).

Thus, the team at Paragon Initiative Enterprises wrote a PHP library called EasyDB.

HOW TO USE EASYDB

There are two ways to start using EasyDB in your code:

1 - You can use EasyDB to wrap your existing PDO instances.

2 - If you're familiar with PDO constructors, you can pass the same arguments to \ParagonIE\EasyDB\Factory::create() instead.

```
// First method:
$pdo = new \PDO(
    'mysql;host=localhost;dbname=something',
    getenv('MYSQL_USERNAME'),
    getenv('MYSQL_PASSWORD')
);
$db = \ParagonIE\EasyDB\EasyDB($pdo);

// Second method:
$db = \ParagonIE\EasyDB\Factory::create(
    'mysql;host=localhost;dbname=something',
    getenv('MYSQL_USERNAME'),
    getenv('MYSQL_PASSWORD')
);
```

(The use of getenv() is best supplemented with a library such as phpdotenv.)

Once you have an EasyDB object, you can begin leveraging its simplified interface to quickly develop secure database-aware applications. Some examples include:

SAFE DATABASE QUERYING WITH PREPARED STATEMENTS :

```
/**
 * As mentioned previously, you should perform validation on all input.
 * Not necessarily for security reasons, but because well-designed software
 * validates all user-supplied input and informs them how to correct it.
 *
 * For the sake of easy auditing, you probably don't want to pass $_GET,
 * $_POST, other superglobals. Instead, validate and store the results
 * in a local variable.
 */
$data = $db->safeQuery(
    'SELECT * FROM transactions WHERE type = ? AND amount >= ? AND date >= ?',
    [
        $_POST['ttype'],
```

```
        $_POST['minimum'],
        $_POST['since']
    ]
);
```

SELECT MANY ROWS FROM A DATABASE TABLE

```
/**
 * Although safe from SQL injection, this example snippet does not
 * validate its input. In real applications, please check that any data
 * your script is given is valid.
 */
$rows = $db->run(
    'SELECT * FROM comments WHERE blogpostid = ? ORDER BY created ASC',
    $_GET['blogpostid']
);
foreach ($rows as $row) {
    $template_engine->render('comment', $row);
}
```

SELECT ONE ROW FROM A DATABASE TABLE

```
/**
 * Although safe from SQL injection, this example snippet does not
```

```
 * validate its input. In real applications, please check that any data
 * your script is given is valid.
 */

$userData = $db->row(
    "SELECT * FROM users WHERE userid = ?",
    $_GET['userid']
);
```

INSERT A NEW ROW INTO A DATABASE TABLE

```
/**
 * Although safe from SQL injection, this example snippet does not
 * validate its input. In real applications, please check that any data
 * your script is given is valid.
 */
$db->insert('comments', [
    'blogpostid' => $_POST['blogpost'],
    'userid' => $_SESSION['user'],
    'comment' => $_POST['body'],
    'parent' => isset($_POST['replyTo']) ? $_POST['replyTo'] : null
]);
```

NEW: DYNAMIC QUERIES WITH EASYSTATEMENT

If you're using EasyDB 1.2.0 or 2.2.0 (or newer), you can use the new EasyStatement API (provided by Woody Gilk) to generate dynamic queries.

```
$statement = EasyStatement::open()
    ->with('last_login IS NOT NULL');

if (strpos($_POST['search'], '@') !== false) {
    // Perform a username search
    $statement->orWith('username LIKE ?', '%' . $db-
>escapeLikeValue($_POST['search']) . '%');
} else {
    // Perform an email search
    $statement->orWith('email = ?', $_POST['search']);
}

// The statement can compile itself to a string with placeholders:
echo $statement; /* last_login IS NOT NULL OR username LIKE ? */

// All the values passed to the statement are captured and can be used for
querying:
```

```
$user = $db->single("SELECT * FROM users WHERE $statement", $statement->values());
```

THE EASYSTATEMENT API SUPPORTS VARIABLE ARGUMENTS (X IN (1, 2, 3)):

```
// Statements also handle translation for IN conditions with variable arguments,
// using a special ?* placeholder:
$roles = [1];
if ($_GET['with_managers']) {
    $roles[] = 2;
}

$statement = EasyStatement::open()->in('role IN (?*)', $roles);

// The ?* placeholder is replaced by the correct number of ? placeholders:
echo $statement; /* role IN (?, ?) */

// And the values will be unpacked accordingly:
print_r($statement->values()); /* [1, 2] */
```

FINALLY, WITH EASYSTATEMENT, YOU CAN ALSO GROUP CONDITIONS TOGETHER :

```
// Statements can also be grouped when necessary:
$statement = EasyStatement::open()
    ->group()
        ->with('subtotal > ?')
        ->andWith('taxes > ?')
    ->end()
    ->orGroup()
        ->with('cost > ?')
        ->andWith('cancelled = 1')
    ->end();

echo $statement;   /* (subtotal > ? AND taxes > ?) OR (cost > ? AND cancelled = 1) */
```

Despite the dynamic nature of the above queries, prepared statements are being used consistently.

ESCAPE AN IDENTIFIER (COLUMN/TABLE/VIEW NAMES) FOR DYNAMIC QUERIES

The new EasyStatement API should be preferred over doing this manually.

```
$whiteListOfColumnNames = ['username', 'email', 'last_name', 'first_name'];
$qs = 'SELECT * FROM some_table';
```

```
$and = false;
if (!empty($where)) {
    $qs .= ' WHERE ';
    foreach (\array_keys($where) as $column) {
        if (!\in_array($column, $whiteListOfColumnNames)) {
            continue;
        }
        if ($and) {
            $qs .= ' AND ';
        }
        $qs .= $db->escapeIdentifier($column).' = ?';
        $and = true;
    }
}
$qs .= ' ORDER BY rowid DESC';

// And then to fetch some data
$data = $db->run($qs, \array_values($where));
```

Caution: The escapeIdentifier() method is meant for this very specific use-case of escaping field and table names and should not be used for escaping user input.

CAN I USE EASYDB TO SATISFY BUSINESS NEEDS?

Yes. We have chosen to release EasyDB under a very permissive license (MIT) because we wish to promote the adoption of better security practices in the community at large. Feel free to use EasyDB in any of your projects, even commercial ones. You don't owe us anything.

SHOULD I USE EASYDB OVER AN ORM OR COMPONENT OF MY FRAMEWORK?

If you're already using tools that you're comfortable with that provide secure defaults (e.g. most modern PHP frameworks), don't drop them in favor of EasyDB. Easy doesn't mean "fits all use cases".

If you're using a CMS that doesn't follow secure best practices, you should solve the problem upstream by getting the CMS to adopt non-emulated prepared statements.

PHP- AND PDO-SPECIFIC RECOMMENDATIONS

If you are a PHP developer looking to get the most out of PDO, and you don't want to add EasyDB to your project, we recommend changing two of the default settings:

1 - Turn off emulated prepares. This ensures you get actual prepared statements.

2 - Set error mode to throw exceptions. This saves you from having to check the result of PDOStatement::execute() and makes your code less redundant .

```
$pdo = new PDO(/* Fill in the blank */);

$pdo->setAttribute(PDO::ATTR_EMULATE_PREPARES, false);

$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

Because PDO::ATTR_EMULATE_PREPARES is set to false, we're getting real prepared statements, and because we've set PDO::ATTR_ERRMORE to PDO::ERRMODE_EXCEPTION, instead of this...

```
$stmt = $pdo->prepare("SELECT * FROM foo WHERE first_name = ? AND last_name = ?");
if ($stmt->execute([$_GET['first_name'], $_GET['last_name']])) {
    $users = $stmt->fetchAll(PDO::FETCH_ASSOC);
} else {
    // Handle error here.
}
$args = [
    json_encode($_GET)
    (new DateTime())->format('Y-m-d H:i:s')
];
$insert = $pdo->prepare("INSERT INTO foo_log (params, time) VALUES (?, ?);");
if (!$insert->execute($args)) {
    // Handle error here.
}
```

...you can just write your code like this:

```
try {
    $stmt = $pdo->prepare("SELECT * FROM foo WHERE first_name = ? AND last_name = ?");
    $stmt->execute([$_GET['first_name'], $_GET['last_name']]);
    $users = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

```
    $args = [
        json_encode($_GET),
        (new DateTime())->format('Y-m-d H:i:s')
    ];
    $pdo->prepare("INSERT INTO foo_log (params, time) VALUES (?, ?);")
        ->execute($args);
} catch (PDOException $ex) {
    // Handle error here.
}
```

Better security, brevity, and better readability

References :
PARAGON, OWASP SECURITY, KaliBoys