Окончательное решение проблемы SQL инъекций.

Роман Шевченко
http://phpfaq.ru/

Teaser

- Всё, что вы делаете для защиты от инъекций - ерунда
- Всё, что вы знаете о prepared statements – ерунда
- Все способы инъекций, о которых вы слышали (blind, second order, time-delay, etc...) - полная ерунда
- Разное

Диалектика

В теории всё хорошо

На практике почему-то всегда дурно пахнущий код

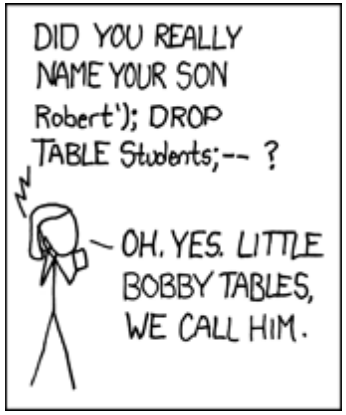
Причины я вижу две

- применение правильных и хороших методов далеко не так просто, как кажется. И столкнувшись с проблемами, разработчик плюёт, и делает опять по-старинке.
- Они работают далеко не всегда

Часть первая. “Кто виноват или Надо ли защищаться от инъекций?”

Несмотря на провокационность вопроса, это не троллинг и не шутка. Однозначный ответ – **нет**.

Давайте посмотрим, почему:




```
$name  = "Bobby';DROP TABLE users; -- ";
$query = "SELECT * FROM users WHERE name='$name'";
```

получаем

```
SELECT * FROM users WHERE name='Bobby';
DROP TABLE users; -- '
```

FAIL!

```
$name  = "Д'Артаньян";
$query = INSERT INTO users (name) VALUES ('$name');
```

Получаем

```
INSERT INTO users (name) VALUES ('Д'Артаньян');
```

FAIL!

Видимо, дело не только в защите от инъекций?

```
$id = NULL;
...
$sql = "SELECT * FROM table WHERE id=$id";
// SELECT * FROM table WHERE id=
$order = "order";
$sql = "SELECT * FROM t ORDER BY $order ";
```

FAIL!

Видимо, дело не только в пользовательском вводе?

Корректно отформатированные запросы

```
SELECT * FROM users WHERE name='Bobby\';
DROP TABLE users; -- '

INSERT INTO users (name) VALUES ('Д\'Артаньян');

SELECT * FROM t ORDER BY `order`

SELECT * FROM table WHERE id = NULL
```

SQL запрос — это программа

Формат важен в любом языке, будь то SQL или PHP

```
$s = fread("$f", 7); // epic fail
```

Инъекция - всего лишь следствие некорректно отформатированного запроса.

«blind», «time-delay», «second order»… тысячи их.

Всё вышеперечисленное – способы **эксплуатировать** инъекцию.

В то время как способ **совершить** инъекцию - ровно один: нарушить целостность запроса.

Как только мы найдем способ **всегда** корректно форматировать динамически формируемые элементы запроса, вся эта шушера превратится в тыкву.

«Хорошо зафиксированный пациент в наркозе не нуждается»

(с) один анестезиолог.

Форматируйте ваши запросы корректно, и никакая инъекция вам никогда не будет страшна.

Часть вторая:
Что делать? или Правила корректного составления запросов

1. Форматирование должно быть **полным**
2. Форматирование должно быть **адекватным**
3. Форматирование должно быть **обязательным**
4. Форматирование должно выполняться **как можно ближе к выполнению запроса**

Правило №1:
Форматирование должно быть полным

(иначе полным будет пушистый зверёк, который к нам придёт)

Примеры неполного форматирования

```
$id  = $mysqli->real_escape_string($id);
$sql = "SELECT * FROM table WHERE id = $id";

$sql = "SELECT * FROM `$table`";
```

mysql(i)_real_escape _string()

НЕ ИМЕЕТ ни малейшего отношения

- К SQL инъекциям
- К защите от чего бы то ни было

Имеет отношение

- К форматированию строк
- Даже для этого форматирования недостаточна

Проблема с этой функцией в том, что она делает только половину работы. Если бы изначально вместо нее был аналог PDO::quote, которая делает **полное** форматирование (**и** добавление кавычек, **и** экранирование), то никаких нареканий к ней никогда бы не было. Но у quote() есть другая проблема, о которой – чуть дальше.

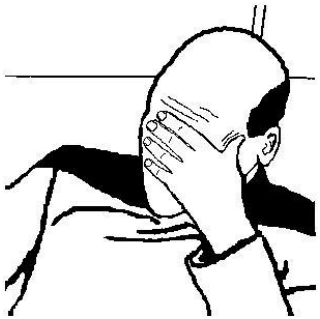
Правило № 2:
Форматирование должно быть адекватным

```
UPDATE table SET
field = 'text', num = num + 1, ip = INET_ATON(?)
WHERE id = :id AND status IN ('open','active')
LIMIT 1
```

- Идентификатор
- Строка
- число
- Плейсхолдер
- Ключевое слово, оператор, функция

Типичный пример неадекватного форматирования

```
$table = $mysqli->real_escape_string($table);
$sql = "SELECT * FROM `$table`"
```



Из комментариев к PDO::quote() (как испортить идеальную функцию)

```
While rewriting some application to PDO, remember that PDO->quote is adding the quotes
around the whole value! Compared to mysql_real_escape_string which is quoting only the
dangerous characters.
<?php
$value = "hello's world";
echo mysql_real_escape_string($value);
// hello''s world
echo $dbh->quote($value);
// 'hello''s world'
// This second quoting will break your old SQL statements which includes the quotes
    already. Workaround is to remove first and last character
echo substr($dbh->quote($value), 1, -1);
// hello''s world
?>
```

Комментарий провисел почти год и получил 8 плюсов. После того, как я устроил скандал, он был удалён.

Правила форматирования

1. Строки
 * могут быть добавлены через native prepared statement (работает не всегда)

или

 * должны быть заключены в кавычки
 * спецсимаолы (грубо говоря - те же самые кавычки) должны быть экранированы
 * плюс должна быть установлена правильная кодировка клиентской библиотеки.

2. Числа
 * могут быть добавлены через native prepared statement (работает не всегда)

или

 * должны быть отформатированы так, чтобы содержать только цифры, знак и - для соответствующего типа - точку

3. идентификаторы
 * должны быть заключены в обратные кавычки
 * спецсимволы (грубо говоря - те же самые обратные кавычки) должны быть экранированы

4. Операторы и ключевые слова
 * тут нет специальных правил форматирования, за исключением того, что они должны быть законными операторами и ключевыми словами

Правило №3: Форматирование должно быть ОБЯЗАТЕЛЬНЫМ

Если для константы, для элемента запроса, прописанного в скрипте, форматирование может быть опциональным, то для динамических, *переменных* частей запроса, оно должно быть **обязательным**.

Форматирование должно быть ОБЯЗАТЕЛЬНЫМ

Мы должны **гарантировать** обязательность форматирования

При этом обязательность должна быть **простой в использовании.** Иначе никто пользоваться не будет.

Правило №4: Форматирование выполняется КАК МОЖНО БЛИЖЕ к выполнению запроса

- Заранее мы не можем знать, для какого элемента запроса предназначены данные (нарушение правила № 2)
- При значительном отдалении форматирования от исполнения запроса, это форматирование может… потеряться (Нарушение правила № 3)
- Появляется соблазн совместить форматирование с валидацией входящих данных.

DO NOT WANT

- Stored procedures (с целью защиты от инъекций)
- Заведение разных пользователей на чтение и запись
- Фильтрация разных «опасных» символов или их сочетаний.
- Валидация пользовательского ввода (с целью защиты от инъекций)

Часть третья. “Как нам со всей этой фигней взлететь, или Типизованные плесхолдеры.”

Следует различать понятия

- “Родных” подготовленных выражений, поддерживаемых на уровне сервера БД. Данные отправляются на сервер отдельно от запроса
- И идею подготовленного выражения в целом, когда данные представлены в запросе неким представителем, плейсхолдером, который при окончательной обработке заменяется на актуальные данные

Что обычно известно про native prepared statements?

- Что они «быстрее»
- Что они «безопаснее», (продразумевается, что другие методы должной безопасности не обеспечивают)
- Что они «экранируют данные за нас» (с ударением на «экранируют»)

Из всего этого важным является только «за нас»

В чем на самом деле состоят преимущества подготовленных выражений

Они делают форматирование

1. Полным
2. Адекватным
3. Обязательным
4. Максимально близким к отправке запроса

И причем всё это - вне зависимости от желания, состояния или способностей программиста.

SQL генерируется одинаковый но подходы принципиально разные

```
$email = PDO::quote($email); // BAD
$sql = 'SELECT * FROM users WHERE email='.$email;
$res = $pdo->query($sql);

$sql = 'SELECT * FROM users WHERE email=?';
$res = $pdo->prepare($sql); // GOOD
$res->execute(array($email));
```

Первый вариант легко может привести **к нарушению правила N4**

Плюс два второстепенных, но весьма приятных бонуса:

- prepared statement форматирует только значение, которое попадает в запрос, а не исходную переменную, которую потом можно использовать, неиспорченную, где-то ещё - вывести на экран, например.
- prepared statement может сделать наш код фантастически коротким, выполняя все операции по форматированию внутри.

Prepared statements не поддерживаются

- для идентификаторов
- для массивов в операторе IN()
- для массивов в запросах INSERT/UPDATE

IN() statement.
Что нам предлагает PDO

```
$ids = array(1,2,3);
$in  = str_repeat('?,', count($arr) - 1) . '?';
$sql = "SELECT * FROM t WHERE c IN ($in) AND cat=?";
$stm = $db->prepare($sql);
$ids[] = $category;
$stm->execute($ids);
$data = $stm->fetchAll();
```

INSERT/UPDATE
Что нам предлагает PDO

```
function pdoSet($fields, &$values, $source = array()) {
  $set = '';
  $values = array();
  if (!$source) $source = &$_POST;
  foreach ($fields as $field) {
    if (isset($source[$field])) {
      $set.="`".str_replace("`","``",$field)."`". "=:$field, ";
      $values[$field] = $source[$field];
    }
  }
  return substr($set, 0, -2);
}
$allowed = array("name","surname","email"); // allowed fields
$sql = "INSERT INTO users SET ".pdoSet($fields,$values);
$stm = $dbh->prepare($sql);
$stm->execute($values);
```

Секрет корректного форматирования (и 100% защиты от инъекций)

Библиотека для работы с БД, будь то DAL или квери билдер, должна понимать плейсхолдеры для ЛЮБЫХ типов данных, которые могут попасть в запрос.

Существующие способы привязки
не годятся

```
//PDO "lzay" binding (не можем указать тип)
$db->execute(array($var1,$var2,$var3,$var4));
// mysqli: (не можем передать массив)
$db->bind_param("issi",$var1,$var2,$var3,$var4);
// PDO standard: (слишком длинно)
$sth->bindParam(1, $var1, PDO::PARAM_INT);
$sth->bindParam(2, $var2, PDO::PARAM_STR);
$sth->bindParam(3, $var3, PDO::PARAM_STR);
$sth->bindParam(4, $var4, PDO::PARAM_INT);
```

Решение

Древнее, как… эпоха Юникс!

Маркировать плейсхолдер типом:

```
$str = sprintf("Hello %s, you have %d lives left",
                $name, $lives);
```

Встречайте: типизованный плейсхолдер

```
$ids = array(1,2,3);
$sql = "SELECT * FROM t WHERE c IN (?a) AND cat=?i";
$data = $db->getAll($sql, $ids, $category);

$sql = "SELECT * FROM t ORDER BY ?n";
$data = $db->getAll($sql, $order);
```

Попинаем немного конкурентов

Многие квери билдеры имеют в своём составе новенькую японскую лесопилку:

```
$db->insert($table, $data);
$db->update($table,$data,$where);
$db->delete($where);
```

И тут мы им подсовываем ядрёную сибирскую рельсу:

```
INSERT INGORE
INSERT ... ON DUPLICATE UPDATE
DELETE FROM ... JOIN
```

«Хрррррр…» - сказал новенькая японская лесопилка.

Либо пишем всякие хелперы для хелперов(!), либо возвращаемся к разбитому корыту - ручной сборке запросов.

Типизованные плейсхолдеры спешат на помощь

```
$sql = "INSERT IGNORE INTO ?n SET ?u";
$db->query($sql,$table,$data);

$sql = "INSERT INTO stats
SET pid=?i, dt=CURDATE(), ?u
ON DUPLICATE KEY UPDATE ?u";
```

- Читаемо
- Безопасно
- Универсально

На самом деле - не конкуренты

- Любой квери билдер только приобретёт от поддержки типизованных плейсхолдеров.
- В скором времени поддержка типизованных полейсхолдеров будет добавлена в Giny - форк Yii, разрабатываемый нашей командой.
- Таким образом, станет удобнее работа с квери билдером Yii
- Но, самое главное, удобной и безопасной станет работа с запросами, которые пишутся без использования квери билдера.

Идея не нова

но до сих пор не получила широкого распространения

- Первая библиотека с поддержкой типизованных плейсхолдеров появилась ещё в 2006 году - это dbSimple Дмитрия Котерова
- Есть так же и другие - например, goDB Олега Григорьева, и ещё около десятка разработок.
- Но по какой-то причине до сих пор массового перехода не типизованные плейсхолдеры не произошло.
- Новое всегда с трудом пробивает себе дорогу. Именно поэтому и нужны такие презентации.

ВАЖНО!
Белые списки

К сожалению, подготовленные выражения работают не всегда.

В некоторых случаях нам нужна проверка по белым спискам.

- Идентификаторов
- Ключевых слов

Допустим, у нас есть поле user_rigts в таблице users, а запрос на вставку формируется динамически, на основе имен и значений полей в HTML форме. Поле user_rights легко сфабриковать. Поэтому динамически формируемые идентификаторы всегда надо проверять по списку, прописанному в скрипте. Брать структуру из базы **НЕДОСТАТОЧНО**.

Примеры реализации белых списков:

```
$allowed = array("name","price","qty");
$key = array_search($_GET['field'], $allowed));
if ($key === false) {
    throw new Exception('Wrong field name');
}
$field = $allowed[$key];
$query   = "SELECT $field FROM t";

$dir = $_GET['dir'] == 'DESC' ? 'DESC' : 'ASC';
$sql = "SELECT * FROM t ORDER BY field $dir";
```

Разное

Страшилки дядюшки Шифлетта про страшные инъекции через неверно указанную кодировку.

- Всё это касается только очень экзотических кодировок. Если бы utf8 была уязвима – мы бы из инъекций не вылезали.
- Поскольку сама по себе функция mysql(i)_real_escape_string() не работает. Ей надо специально сказать кодировку:

```
mysql(i)_set_charset()
charset=GBK; в DSN для PDO
```

Выводы

1. Не нужно защищаться от инъекций, **нужно корректно форматировать элементы запроса,** попадающие в него динамически
2. Форматирование должно быть **полным, адекватным, обязательным**, и **как можно ближе** к выполнению запроса.
3. Обеспечить соблюдение вышеперечисленных могут только **типизованные плейсхолдеры.**
4. Для помещаемых в запрос динамически **ключевых слов** и **идентификаторов** необходимо проверять их по **белым спискам.**

Спасибо за внимание

Класс, реализующий описанную функциональность

http://www.phpfaq.ru/safemysql

Текст доклада

http://www.phpfaq.ru/sql_injection

Слайды

http://www.phpfaq.ru/misc/sql_injection_dc2013.pdf

Coming soon

yii fork с поддержкой типизованных плейсхолдеров

https://github.com/Sotmarket/qiny