# Практика 4. Разработка безопасного программного обеспечения

## Использование статического анализатора кода для обеспечения безопасности

## Лабораторная работа

**Задача**

1. Необходимо найти участок кода, содержащий инъекцию SQL кода в задании Blind Sql Injection на сайте dvwa.local с использованием статического анализатора кода (Можно использовать [официальный ресурс](https://github.com/digininja/DVWA) или виртуальную машину Web Security Dojo)
2. Проанализировать код и сделать кодревью, указав слабые места

<?php

if( isset( $\_GET[ 'Submit' ] ) ) {

// Get input

$id = $\_GET[ 'id' ];

// Check database

$getid = "SELECT first\_name, last\_name FROM users WHERE user\_id = '$id';";

$result = mysqli\_query($GLOBALS["\_\_\_mysqli\_ston"], $getid ); // Removed 'or die' to suppress mysql errors

// Get results

$num = @mysqli\_num\_rows( $result ); // The '@' character suppresses errors

if( $num > 0 ) {

// Feedback for end user

$html .= '<pre>User ID exists in the database.</pre>';

}

else {

// User wasn't found, so the page wasn't!

header( $\_SERVER[ 'SERVER\_PROTOCOL' ] . ' 404 Not Found' );

// Feedback for end user

$html .= '<pre>User ID is MISSING from the database.</pre>';

}

((is\_null($\_\_\_mysqli\_res = mysqli\_close($GLOBALS["\_\_\_mysqli\_ston"]))) ? false : $\_\_\_mysqli\_res);

}

?>

1. Разработать свою систему вывода информации об объекте на любом языке, исключающий возможность инъекции SQL кода. Возможно исправление участка кода из dvwa.local  
   Требования к системе авторизации
   * Система вывода информации об объекте должна использовать запросы GET с параметрами, аналогичными из задания Blind SQL injection dvwa

dvwa.local/vulnerabilities/sqli/?username=USER&password=PASS&user\_token=TOKEN&Login=Login

1. Использовать sqlmap для нахождения уязвимости в веб-ресурсе
2. Использовать Burp для нахождения уязвимости в веб-ресурсе

## Требования для лабораторной работы

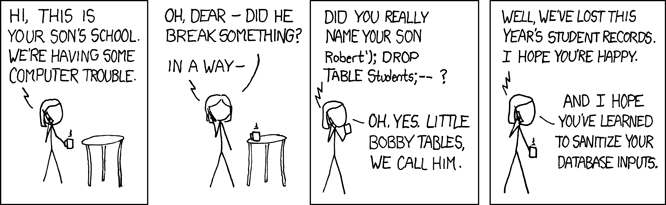
1. Виртуальная машина Web Security Dojo  
   1а. или сервер с установленным программным обеспчением [dvwa](https://github.com/digininja/DVWA" \t "_blank)
2. Редактор кода для языков PHP и выбранного Вами языка для написания программы перебора пароля для формы аутентификации.
3. Облачная версия [https://sonarcloud.io](https://sonarcloud.io/) или установленная на компьютере

* Для работы можно установить [Orace VM Virtualbox](https://www.virtualbox.org/wiki/Downloads" \t "_blank) и расширения для него [VirtualBox Extension Pack](https://download.virtualbox.org/virtualbox/6.1.14/Oracle_VM_VirtualBox_Extension_Pack-6.1.14.vbox-extpack" \t "_blank)
* Скачать последний архив с виртуальной машиной [Web Security Dojo](https://sourceforge.net/projects/websecuritydojo/" \t "_blank)
* [Установка и настройка SonarQube с использованием Docker](https://itsecforu.ru/2020/04/07/🐳-анализ-кода-с-помощью-sonarqube-на-docker/)

## Литература

1. [Burp](https://portswigger.net/burp)
2. [Использование Burp](https://systemweakness.com/sql-injection-basics-with-burp-suite-e07551bc5172)
3. [SQLmap](https://sqlmap.org/)
4. [Использование sqlmap](https://defcon.ru/web-security/4544/)
5. [Статический анализ кода //Wikipedia](https://ru.wikipedia.org/wiki/Статический_анализ_кода)
6. [A1:2017-Injection](https://owasp.org/www-project-top-ten/2017/A1_2017-Injection.html)
7. [Broken authentifaction and session management, стр.60 //OWASP CodeReview](https://owasp.org/www-pdf-archive/OWASP_AlphaRelease_CodeReviewGuide2.0.pdf)
8. [SQL Injection //OWASP Community](https://owasp.org/www-community/attacks/SQL_Injection)
9. [SQL Injection Prevention Cheat Sheet¶ //OWASP](https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html)
10. [Уязвимости из OWASP Top 10. A1: 2017 – Injections (Часть 1) //Хабр](https://habr.com/ru/company/gaz-is/blog/419387/)
11. [Защита от SQL-инъекций в PHP и MySQL //Хабр](https://habr.com/ru/post/148701/)
12. [Устраняем уязвимости: как защитить сайт от SQL-инъекции //](https://skillbox.ru/media/code/kak_zashchitit_sayt_ot_sql_inektsii/https://securitytutorials.co.uk/brute-forcing-web-logins-with-dvwa/)
13. [Контролируем качество кода с помощью платформы SonarQube //Хабр](https://habr.com/ru/company/pvs-studio/blog/315422/)
14. [SonarQube | SonarScanner : a step by step guide //swlh@medium](https://medium.com/swlh/sonarqube-sonarscanner-setup-1a633b654828)
15. [Bobby Tables: A guide to preventing SQL injection //bobby-tables.com](https://bobby-tables.com/)
16. [Sql Injection Myths and Fallacies //Karwin Software Solutions LLC](https://www.slideshare.net/billkarwin/sql-injection-myths-and-fallacies)

## Теоретическая часть



Правила, соблюдение которых гарантирует нас от инъекций

* данные подставляем в запрос только через плейсхолдеры
* идентификаторы и ключевые слова подставляем только из белого списка, прописанного в нашем коде.

Всего два пункта.  
Разумеется, практическая реализация этих правил нуждается в более подробном освещении.  
Но у этого списка есть большое достоинство — он точный и исчерпывающий. В отличие от укоренившихся в массовом сознании правил «прогонять пользовательский ввод через mysql\_real\_escape\_string» или «всегда использовать подготовленные выражения», мой набор правил не является катастрофическим заблуждением (как первое) или неполным (как второе).

Но вперёд, читатель — перейдём уже к подробному разбору.

Плейсхолдеры — подстановка данных

В принципе, тут всё просто: любые данные должны попадать в запрос не напрямую, а через некоего представителя, подстановочное выражение.  
Запрос пишется в таком, например, виде,

SELECT \* FROM table WHERE id > ? LIMIT ?

а данные добавляются и обрабатываются отдельно.  
Но чем это лучше «обычного эскейпинга»? А всем:

* Во-первых, «эскейпинг» вообще не имеет никакого отношения к безопасности. Представьте себе!
* Во-вторых, код становится банально короче. Никаких тебе mysql\_real\_escape\_string(), ни даже intval() — вся обработка скрыта внутри.
* В-третьих, код становится проще. Не нужно запоминать различные правила для форматирования разных частей запроса
* В-четвёртых, использование плейсхолдеров (при условии, что они корректно обрабатываются) гарантирует нас от инъекций через данные. «Это же очевидно!» — скажете вы — «стоило ли городить отдельный пункт?» Стоило. Тот же эскейпинг защиту не гарантирует. Плюс сокращение правил форматирования до одного — перепутать что-либо тут будет затруднительно
* В-пятых — и в самых главных — мы обрабатываем данные ровно там, где нужно! Это очень важный момент, который многие не понимают. В классических «похапешных» учебниках форматирование данных для SQL разбросано по всему коду. А в старых версиях PHP оно и вовсе начиналось ещё даже до начала выполнения кода — что совсем уже ни в какие ворота не лезет! Такая ситуация приводит к тому, что одни данные форматируются дважды, другие — только наполовину, а третьи — и вовсе ни разу или совсем не так, как нужно, без малейшей пользы. При этом отформатированные для SQL данные вдруг оказываются в HTML или куках, что тоже не прибавляет радости пользователям и разработчикам.

Поэтому самым лучшим вариантом будет форматировать данные непосредственно перед исполнением запроса — таким образом мы всегда будем уверены в том, что данные форматируются корректно, это делается только один раз, и отформатированные данные попадут строго по назначению — в БД и никуда больше.  
И вот как раз целям такой — своевременной, безопасной и корректной — обработки данных и служат плейсхолдеры, давая нам гарантию безопасности, в то же время упрощая код.

Примеры использования:

Из мануала:

$stmt = $dbh->prepare("SELECT \* FROM REGISTRY where name LIKE ?");

$stmt->execute(array("%$\_GET[name]%"));

$data = $stmt->fetchAll();

Пример того, к чему стоит стремиться:

$ban = $db->getRow("SELECT 1 FROM ban WHERE ip = inet\_aton(s:)", $ip);

Как видите, совсем несложно, а так же — при умелом использовании — выходит гораздо короче, чем составлять запрос руками. Вы всё ещё хотите писать по-старинке?

Важное замечание: Разумеется, подстановка данных через плейсхолдеры должна производиться всегда, вне зависимости от источника данных или каких бы то ни было других условий.

### Идентификаторы и ключевые слова — белые списки

Подавляющее большинство статей, посвящённых инъекциям, совершенно упускают этот момент из виду. Но реальность такова, что в ней мы сталкиваемся с необходимостью подставлять в запрос не только данные, но и другие элементы — идентификаторы (имена полей и таблиц) и даже элементы синтаксиса, ключевые слова. Пусть даже такие незначительные, как DESC или AND, но требования к безопасности таких подстановок всё равно должны быть не менее строгими!

Разберем довольно банальный случай.

У нас есть база товаров, которая выводится пользователю в виде HTML таблицы. Пользователь может сортировать эту таблицу по одному из полей, в любом направлении.  
То есть, как минимум, со стороны пользователя к нам приходит имя колонки и направление сортировки.

Подставлять их в запрос напрямую — гарантированная инъекция. Привычные методы форматирования здесь не помогут. Подготовленные выражения ни с идентификаторами, ни с ключевыми словами не приведут ни к чему, кроме сообщения об ошибке.  
Единственное решение — белый список.  
Это, разумеется, не бином ньютона, и многие разработчики легко по ходу дела реализуют эту парадигму на ходу, впервые столкнувшись с необходимостью подстановки имени поля в запрос. Тем не менее, статья о защите от инъекций без этого правила будет неполной, а сама защита — дырявой.

Суть метода заключается в том, что все возможные варианты выбора должны быть жёстко прописаны в нашем коде, и в запрос должны попадать только они, на основании пользовательского ввода.

Пример применения:

$order = isset($\_GET['order']) ? $\_GET['order'] : ''; // просто для полноты кода

$sort = isset($\_GET['sort']) ? $\_GET['sort'] : '';

$allowed = array("name", "price", "qty"); //перечисляем варианты

$key = array\_search($sort,$allowed); // ищем среди них переданный параметр

$orderby = $allowed[$key]; //выбираем найденный (или, за счёт приведения типов - первый) элемент.

$order = ($order == 'DESC') ? 'DESC' : 'ASC'; // определяем направление сортировки

$query = "SELECT \* FROM `table` ORDER BY $orderby $order"; //запрос 100% безопасен

Раньше я предполагал, что для идентификаторов достаточно плейсхолдера. Но со временем пришло понимание недостатков этого метода:

* во-первых, в случае неверного имени поля запрос вызовет ошибку. А ошибки — это всегда плохо
* во-вторых, и гораздо более важных: если подставлять имена полей без предварительной фильтрации, автоматом, только эскейпя их, можно получить инъекцию другого рода — ведь пользователь тогда может вписать в те имена полей, которые ему изменять не положено! Скажем, если мы формируем SQL запрос автоматически на базе массива $\_POST (при этом правильно форматируя имена полей!), то хакер при регистрации добавляет в форму поле admin со значением «1» и становится админом

Так что теперь я использую оба метода:  
Сначала получаю идентификатор из белого списка.  
А потом добавляю его через плейсхолдер — просто чтобы не заниматься форматированием вручную. И для единообразия. В этом случае последняя строчка будет выглядеть, как

$query = "SELECT \* FROM `table` ORDER BY n: $order";

В принципе, этой информации достаточно, чтобы начать писать полностью безопасные запросы. Но, как водится, в реальной жизни случаются разные нюансы, да и разобраться с механизмом плейсхолдеров хотелось бы поподробнее.

### Работа с плейсхолдерами

Для начала нужно понимать, что существует два варианта реализации плейсхолдеров — серверный и клиентский:

* В первом случае запрос так и уходит на сервер с плейсхолдерами, а данные отправляются отдельно от него. По-английски называется native prepared statements — «родные» подготовленные выражения — то есть, обработка плейсхолдеров осуществляется самой СУБД, на сервере. Для краткости я буду использовать наименование «серверные плейсхолдеры».
* Во втором случае данные форматируются и подставляются в строку запроса на место плейсхолдеров прямо на клиенте, формируя классический SQL запрос, который затем уходит в базу обычным порядком.

Каждый из способов имеет свои достоинства и недостатки, которые мы рассмотрим ниже.  
Следует также помнить, что PDO сидит на этих двух стульях одновременно — по умолчанию работает по второму варианту, лишь эмулируя первый. Эту функциональность можно отключить, заставив PDO отправлять на сервер данные отдельно от запроса.

$dbh->setAttribute( PDO::ATTR\_EMULATE\_PREPARES, false );

Но поскольку даже эмуляция делается без участия программиста, ниже мы будем рассматривать PDO как представителя серверных плейсхолдеров.

### Серверные плейсхолдеры

Для начала давайте сформулируем — почему вообще возможны инъекции?  
По факту, SQL запрос представляет собой программу. Полноценную программу — с операторами, переменными и строковыми литералами. Проблема же состоит в том, что мы эту программу собираем динамически, на ходу. В отличие от наших PHP скриптов, которые написаны раз и навсегда, и не изменяются на основе поступающих данных, SQL запрос каждый раз динамически формируется заново. И, как следствие, неверно отформатированные данные могут испортить запрос, или даже поменять его, подставив непредусмотренные нами операторы. Собственно, именно в этом и состоит суть инъекций.

Что же нам предлагает серверная обработка плейсхолдеров?  
Очень простую вещь: мы вносим в нашу программу такое понятие, как… переменные! Да-да, плейсхолдер — это обычная переменная, которая жёстко прописана в нашем SQL-«скрипте» и не меняется в зависимости от данных. А сами данные едут на сервер отдельно от запроса, и никогда с ним не пересекаются. Только после того, как запрос будет интерпретирован, данные будет использованы уже непосредственно на этапе исполнения.  
На практике это выглядит так: при вызове prepare() наш запрос едет на сервер прямо в таком виде — с плейсхолдерами/переменными, сервер его парсит и сигнализирует — «всё окей, готов принимать данные» (ну, или сообщает об ошибке). А затем, при выполнении execute(), на сервер едут уже данные (причём не в текстовом виде, а в бинарном пакете, по структуре напоминающем тот, в котором возвращается результат запроса), и участвуют уже непосредственно в выполнении.

В теории звучит очень заманчиво.  
Однако на практике, к сожалению, в имеющихся библиотеках для работы с Mysql в PHP реализация работы с подготовленными выражениями ещё очень далека от идеала.  
Достаточно привести такие, к примеру, факты:

* Такая банальная возможность, как получение строки результата в массив (аналог mysql\_fetch\_array()), для запросов, использующих подготовленные выражения, была добавлена в mysqli только в версии в 5.3(!). До этой версии библиотека была практически неюзабельна без адовых извращений.
* То же самое касается задания кодировки соединения в PDO — это стало возможным только в той же 5.3. Во всех предыдущих версиях задать кодировку соединения в PDO было попросту невозможно (что — справедливости ради следует признать — не является большой проблемой при выключенном режиме совместимости, но — дыра-с!).
* mysqli в некоторых случаях пытается зарезервировать столько памяти, сколько максимально вмещает поле в БД — осторожнее с mediumtext-ами!  
  Эти факты говорят нам о том, что обе библиотеки до сих пор довольно сырые, и мы можем ожидать от них отсутствия и другого необходимого функционала.

Перечислим основные недостатки имеющихся библиотек

* многословность
* отсутствие некоторых полезных плейсхолдеров
* невозможность получить классический SQL запрос для целей отладки
* возможные проблемы с производительностью\* (этот вопрос будет рассмотрен ниже)

и разберём их подробнее

### Многословность.

Возьмём такую, к примеру, востребованную операцию, как получение всех строк результата запроса в двумерный массив. В mysqli до сих пор нет такой функции. Или, скажем, привязывать переменные к плейсхолдерам там можно только отдельной функцией.  
В итоге, на получение данных одного запроса в массив у нас уйдёт минимум девять(!) строк кода:

$data = array();

$query = "SELECT Name, Population, Continent FROM Country WHERE Continent=? ORDER BY Name LIMIT 1";

$stmt->prepare($query);

$stmt->bind\_param("s", $continent);

$stmt->execute();

$result = $stmt->get\_result();

while ($row = $result->fetch\_array(MYSQLI\_NUM)) {

$data[] = $row;

}

Причём большая часть этого кода не несёт никакой смысловой нагрузки, является абсолютно одинаковой для всех выполняемых запросов и при этом повторяется множество раз в сотнях скриптов.  
при том, что для получения данных вполне достаточно всего двух строк:

$query = "SELECT Name, Population, Continent FROM Country WHERE Continent=? ORDER BY Name LIMIT 1";

$data = $db->getAll($query,$continent);

У PDO с этим немного лучше — там догадались разрешить передавать массив в execute() и есть метод fetchAll(). Но всё равно приходится писать множество ненужного однообразного кода для самых простых операций.  
В частности — вернёмся к теме нашей статьи — биндинг. Вот есть у меня переменная $\_GET[‘id’]; я хочу привязать её к плейсхолдеру. Отлично, я могу это сделать прямо в execute()… но только сделав её массивом. Зачем? Неужели сам драйвер не может сделать это за меня?

### Недостаточность функциионала

Ещё одна проблема, которая уже поднималась в комментариях к недавней статье — оператор IN(). Сделать для него подстановки — задача весьма нетривиальная. Хотя казалось бы, именно для таких случаев плейсхолдеры-то как раз и придуманы:

$conts = array('Europe','Africa','Asia','North America');

$query = "SELECT \* FROM Country WHERE Continent IN(?) ORDER BY Name LIMIT 1";

$data = $db->getAll($query,$conts);

Вместо десятков строк хитровыдуманного кода.  
Какой вывод мы можем из этого сделать? Надо допиливать.

### Невозможность получить SQL запрос

Кто-то скажет, что ему это не нужно, кто-то изворачивается и пишет запрос руками или использует хитрый софт. Будем уважать мнение этих людей, но факт остаётся фактом — функционал вывода готового запроса удобен для отладки, а серверные плейсхолдеры его не позволяют.

### Производительность

Обычно апологеты серверных подготовленных выражений напирают на тот факт, что парсинг запроса делается только один раз.

К сожалению, для веб-приложения это не работает. Та копия скрипта, которая выполнила prepare(), выполняет execute() для этого запроса ровно 1 раз и благополучно умирает. А новая копия заново делает prepare. В итоге выходит больше работы там, где хотели сэкономить.

В комментариях к недавней статье мне указали на ещё один потенциальный выигрыш в скорости — кэширование плана запроса. И ведь действительно, даже если мы делаем prepare для каждого запроса, то БД может закэшировать один и тот же запрос для разных данных. И получить план исполнения без парсинга, а простым сравнением строк!

Увы, я не настолько силён во внутреннем устройстве MySQL, чтобы подтвердить или опревергнуть как наличие самого механизма, так и его практическую эффективность.  
При этом, насколько мне известно, под большими реальными нагрузками серверные подготовленные выражения проигрывают по скорости стандартным SQL запросам.  
В любом случае, тема открытая и ещё ждёт своего исследователя. В конце концов, версии СУБД растут и результаты тестов устаревают.

В общем, мы выяснили, что средства, которые предоставляет нам СУБД и драйверы для работы с ней, оказались не так хороши, какими представлялись в рекламе. И возникает вопрос — а можем ли мы реализовать работу с плейсходерами самостоятельно? И возникает ответ — можем!

### Самостоятельная реализация плейсхолдеров

Вообще, на самом деле никто не мешает нам улучшить юзабилити имеющихся библиотек, не заморочиваясь на плейсхолдеры. Скажем, написать обёртку над PDO, которая реализует недостающий функционал, используя при этом собственные плейсхолдеры PDO (или mysqli, как это сделано в библиотеке, опубликованной год назад на Хабре).

Но у нас есть несколько причин рассмотреть самодельные плейсхолдеры:  
Во-первых, как мы уже убедились, имеющегося в стандартных библиотеках набора плейсхолдеров явно недостаточно.  
Во-вторых, серверные плейсхолдеры по каким-либо причинам могут нам не подойти.  
В-третьих, на примере самостоятельной обработки плейсхолдеров мы рассмотрим нюансы корректного форматирования SQL запросов.

Принципы форматирования различных элементов SQL запроса

Реализовать самодельные плейсхолдеры совсем несложно. Примитивный парсер уже встроен в PHP. Единственное, что нам нужно — это научиться отличать различные элементы запроса. Но это очень важный момент, на котором стоит остановиться поподробнее.

Поскольку правила форматирования зависят от типа элемента, нам надо, во-первых, чётко понимать, какой именно элемент запроса мы в него подставляем. А во-вторых, нам надо как-то сообщить эту информацию обработчику плейсходеров.

Для начала давайте определимся, из каких элементов вообще может состоять запрос?  
Возьмём, для примера, такой SQL:

INSERT INTO `db`.`table` as `t1` VALUES('string',1,1.5,NOW());

В нём можно выделить три основные группы элементов:  
собственно элементы языка SQL — операторы, встроенные функции, переменные и пр.  
идентификаторы (имена баз данных, таблиц и полей)  
литералы (данные, подставленные непосредственно в запрос) нескольких разных типов.  
Нас будут интересовать последние два пункта, поскольку только они требуют специального форматирования.

Теперь подумаем, как сообщить информацию о типе подставляемых данных нашему обработчику. Имеющиеся решения делают это криво (что уже не должно вас удивлять). Варианта тут два: либо тип приходится задавать, вызывая функцию биндинга (что сразу в разы усложняет код), либо не задавать его вовсе, как делает PDO, если передать данные прямиком в execute(). Но совсем без типа обойтись нельзя, и поэтому все переданные в execute данные PDO трактует, как строки. Что приводит к забавным последствиям если PDO работает в режиме совместимости: при попытке передать параметры для LIMIT-а в execute(), PDO вылетит с сообщением об ошибке, в котором можно разглядеть кавычки, которыми честная библиотека обрамила оффсет и лимит!

В общем, нужно другое решение. И оно есть! Но о нём мы поговорим ниже, а пока рассмотрим правила форматирования.

### Форматирование идентификаторов

Вообще, правила именования идентификаторов довольно обширные. Но учитывая, что для безопасности мы пользуемся белыми списками, а плейсхолдер — только для форматирования, то достаточно будет этих двух:  
идентификатор должен быть заключён в обратные одинарные кавычки (backticks)  
если такая кавычка встречается в имени — она должна быть экранирована удвоением.

function escapeIdent($value)

{

return "`".str\_replace("`","``",$value)."`";

}

Лирическое отступление:  
Нужно или не нужно форматировать идентификаторы? Ведь в большинстве случаев это не требуется?  
Если запрос пишется руками, то необходимость можно определить на месте: работает запрос — можно не форматировать; вылетает с ошибкой на идентификаторе — надо форматировать.  
Если же мы используем плейсхолдер — то есть, добавляем идентификатор в запрос динамически — форматировать надо обязательно, поскольку мы не знаем, какое имя поля будет подставлено в запрос, и, следовательно, требуется ли ему форматирование, или нет. Значит, будем форматировать всё.

Собственно, если говорить об использовании плейсхолдеров в целом, именно последовательное, без исключений, применение правил форматирования и позволяет говорить о гарантированной защите от инъекций и — что немаловажно — ошибок. Ведь форматирование в первую очередь делается для того, чтобы обеспечить синтаксическую корректность запроса. А защита от инъекций является всего лишь побочным эффектом.  
Поэтому я предпочитаю употреблять термин «форматирование», а не «эскейпинг»

### Форматирование строковых литералов

Самая избитая, казалось бы, тема. Но, как показали обсуждения недавних статей, многие до сих пор путаются если не в понимании, то как минимум — в формулировках. Итак, давайте сформулируем правила форматирования строк в SQL.

* строка должна быть заключена в кавычки (одинарные или двойные, но поскольку двойные могут быть использованы для идентификаторов, лучше всегда использовать одинарные)
* в строке должны быть экранированы спецсимволы по списку. Для этого API предоставляет специальную функцию. Для корректной работы этой функции должна быть правильно задана кодировка соединения

при этом

* эти правила всегда должны применяться вместе, а не только какое-то одно
* ни одно из них не должно применяться к любым другим данным кроме строк

Вроде бы, простые правила? Но просто поразительно, сколько людей им не следует.  
и даже в документации по PHP, в статье про mysql\_real\_escape\_string() написана дичь: «Если не пользоваться этой функцией, то запрос становится уязвимым для взлома с помощью SQL-инъекций. » — как будто если её использовать для чисел или идентификаторов, то это хоть чему-то поможет!

А вот разработчиков PDO можно похвалить — они поступили совершенно логично, выполняя оба правила вместе: функция PDO::quote() делает не половину дела, а всё целиком — эскейпит строку и заключает её в кавычки. Поступим так же и мы:

function escapeString($value)

{

return "'".mysqli\_real\_escape\_string($this->connect,$value)."'";

}

Остаётся только добавить, что «правильно задать кодировку» можно только функциями mysqli\_set\_charset / mysql\_set\_charset, а в PDO — в DSN.

### Форматирование чисел

Теоретически, в большинстве случаев числа можно форматировать как строки, и тогда задача сведётся к предыдущей. Но есть три проблемы  
режим STRICT MODE в mysql, который, будучи включённым, в некоторых случаях будет выдавать ошибки при попытке выдать строку за число  
оператор LIMIT, в котором использование строк не предусмотрено вовсе  
замечания специалистов по mysql о том, что тип литерала очень важен, и играет роль при планировании и исполнении запроса. Сам я в этом не силён, и буду рад подробному комментарию.  
В общем, числа форматировать тоже будем. Тем более, что с помощью плейсхолдеров это не представляет проблемы.  
Проблему представляет другое — недостаточноая разрядность встроенных механизмов приведения типов в PHP. Поэтому если вам надо работать с целочисленными значениями, большими чем PHP\_INT\_MAX, используйте регулярки для валидации. Ну а в обычном случае можно использовать intval().

Числа же с десятичной точкой желательно проверять только регуляркой, поскольку в PHP нет типа данных, аналогичного супер-полезному типу DECIMAL в MySQL.

### «Ленивые плейсхолдеры»

Итак, приступим к реализации.  
Самый первый вариант, который приходит в голову — это sprintf(). Собственно, подстановочные выражения этого семейства функций и есть настоящие плейсхолдеры. Причём типизованные плейсхолдеры! То есть, то, как именно будут обрабатываться данные, определяет сам плейсхолдер. Это, на самом деле, не такая уж незначительная вещь — ни авторы PDO, ни авторы mysqli так до неё и не додумались. И указать тип подставляемых данных в этих библиотеках можно только вызвав специальную функцию.

В общем, sprintf снимет с нас заботу о любых типах данных кроме строк. Разумеется, sprintf никак не форматирует строки, а выдаёт их как есть. Но нас это не устраивает — у нас есть жёсткие правила форматирования. И это именно правила, а не правило. Как мы помним, эскейпинг — всего лишь пол-дела.  
Но поскольку такую ответственную работу, как добавление кавычек, мы не можем отдать на откуп программисту, наш код должен это делать сам. Тупо строковой заменой. Это неидеальное решение, но для учебного кода подойдёт.

Кроме этого мы будем эскейпить все передаваемые в функцию данные. Да, это неправильно. Но вреда от этого не будет, но зато мы сохраним главное преимущество sprintf() — тот факт, что она будет парсить строку за нас.  
Почему не будет вреда? Ведь только что я рассказывал, что эскейпить что-либо кроме строк — бесполезно и опасно! Да, если мы больше никак не обрабатываем остальные данные. Но поскольку оставшимися вариантами плейсхолдеров являются только числовые, инъекция через них не пройдёт.  
В итоге у нас получилась функция

function query(){

$query = array\_shift($args);

$query = str\_replace("%s","'%s'",$query);

foreach ($args as $key => $val) {

$args[$key] = mysql\_real\_escape\_string($val);

}

$query = vsprintf($query, $args);

if (!$query) return FALSE;

$res = mysql\_query($query) or trigger\_error("db: ".mysql\_error()." in ".$query);

return $res;

}

которая неплохо справляется со своими обязанностями по подстановке данных в запрос.  
причём с защитой от дурака: если мы перепутаем плейсхолдер, и поставим, к примеру, %s вместо %d в операторе LIMIT, то получим ошибку запроса на этапе разработки, но не инъекцию (как это произошло бы в случае эскейпинга).

Разумеется, реализация на основе printf() не лишена недостатков — нaдо следить, чтобы не было лайков, прописанных прямо в запросе, плюс всего три типа плейсхолдеров.  
Но для тех, кто ну ооочень не любит ООП, приведу здесь функцию, которую я написал много лет назад и которую можно просто скопировать в конфигурационный файл и начать пользоваться, получив безопасный способ выполнения запросов, ко всему ещё и сокращающий код получения данных в 3-4 раза.  
показать код

примеры использования:

$name = dbget(1,"SELECT name FROM users WHERE id=%d",$\_GET['id']); // если нам нужно только имя

$user = dbget(2,"SELECT \* FROM users WHERE id=%d",$\_GET['id']); // если нужна вся строка

$sql = "SELECT \* FROM news WHERE title LIKE %s LIMIT %d,%d";

$news = dbget(3,$sql,"%$\_GET[search]%",$start,$per\_page); //если нужен массив

Несмотря на некоторую неуклюжесть, эта функция дорога моему сердцу, поскольку написана в строгом следовании известным принципам: и если KISS ей можно поставить в упрёк, то применение её делает код очень и очень DRY.

Но все мы понимаем, что это тупиковая ветвь. И нам нужен полноценный класс.  
Увы, объём статьи уже и превысил все разумные размеры, и создание класса надо выносить в отдельный пост. Остановлюсь здесь лишь на нескольких вопросах.

Во-первых, класс наш будет наследником вышеприведённой функции, и служить высокой миссии избавления кода от повторений. Поэтому помимо поддержки плейхолдеров, класс будет предоставлять набор функций-хелперов для получения из БД информации сразу в нужном формате — скаляра, одномерного массива, двумерного массива, индексированного по полю одномерного массива, индексированного по полю двумерного массива.

Во-вторых, для нашего класса мы используем поистине блистательную идею типизованных плейсхолдеров. А заодно поживимся у PDO плейсхолдерами именованными.  
Пусть наш плейсхолдер имеет вид

[a-z]:[a-z]\*

Например,

i:

или

s:name

В первом случае это будет анонимный плейсхолдер, а во втором — именованный.  
первая буква задаёт тип, двоеточие отличает плейсхолдер от других элементов строки, а имя — опционально.

В-третьих, долгожданный плейсхолдер для оператора IN() (для строк)

function createIN($data)

{

if (!is\_array($data))

{

throw new E\_DB\_MySQL\_parser("Value for a: type placeholder should be array.");

}

if (!$data)

{

throw new E\_DB\_MySQL\_parser("Empty array for a: type placeholder.");

}

$query = $comma = '';

foreach ($data as $key => $value)

{

$query .= $comma.$this->escapeString($value);

$comma = ",";

}

return $query;

}

и пара замечаний по нему.  
Как можно видеть, парсер выбрасывает исключение на пустой массив. Почему?  
Потому что mysql ругнётся на пустой IN(), и выполнять запрос всё равно бесполезно.  
Где-то я видел остроумный вариант с implode и кавычками по краям, но так делать нельзя: пустая строка — вполне себе честное значение, но искать её имеет смысл только если она явно была передана в массиве.

DmitryKoterov в своей библиотеке dbSimple эту проблему решает довольно красиво — весь блок с оператором IN заключается в фигурные скобки, и, в случае пустого массива, изымается из запроса целиком. Я не уверен в правильности этого решения, но оно, по крайней мере, весьма остроумно.

Update:  
Благодаря идее, которую подали zerkms и david\_mz, оказалось, что проблема с пустым массивом решается очень просто!

IN(NULL)

не выдаёт ошибку и всегда возвращает FALSE — идеальное представление для пустого массива.  
То есть, вместо него надо просто подставлять NULL:

function createIN($data)

{

if (!is\_array($data))

{

throw new E\_DB\_MySQL\_parser("Value for a: type placeholder should be array.");

}

if (!$data)

{

return 'NULL';

}

$query = $comma = '';

foreach ($data as $key => $value)

{

$query .= $comma.$this->escapeString($value);

$comma = ",";

}

return $query;

}

Первое же исключение я решил оставить, исходя из таких соображений:  
Теоретически, мы могли бы превратить скаляр в массив и дальше обрабатывать обычным порядком. Но такие скрытые преобразования типов, хоть и привычны для языка, чреваты логическими ошибками.  
В итоге, если массив формируется внутри нашего кода, то эта проверка поможет нам отловить все ошибки формирования на этапе разработки. Ну а если массив приходит извне, то тут тем более нужно кидать исключение.  
</update>

Cам же я остановился на варианте с предварительной валидацией массива. В этом мне помогает ещё один метод класса — parse(), который парсит строку с плейсхолдерами, подставляет переданные параметры и выдаёт готовый SQL запрос… или его часть(ведь, в отличие от случая с серверными плейсхолдерами, мы легко можем пропарсить произвольный кусок запроса!). И если первое нам пригодится для отладки, то второе — для случаев, подобных нашему:

if (is\_array($array) and $array) {

$sql .= $db->parse(" AND type IN(a:)",$array);

}

Так же этот метод можно применять для составления многоэтажных условных WHERE:

$w = array();

$where = '';

if (!empty($\_GET['type'])) $w[] = $db->parse("type = s:", $\_GET['type']);

if (!empty($\_GET['rooms'])) $w[] = $db->parse("rooms IN (a:)",$\_GET['rooms']);

if (!empty($\_GET['max\_price'])) $w[] = $db->parse("price <= i:", $\_GET['max\_price']);

if (count($w)) $where = "WHERE ".implode(' AND ',$w);

$data = $db->getArr("SELECT \* FROM table $where LIMIT i:,i:",$start,$per\_page);

Вышеупомянутую билиотеку DbSimple, кстати, стоит отметить отдельным пунктом, как пример готовой реализации большей части изложенных здесь идей (в частности, трансляция самописных плейсхолдеров в родные, типизованные плейсхолдеры на 100500 типов и многое другое), отдельно отметив тот факт, что написана она была чуть ли не 10 лет назад, но по какой-то причине не получила широкого распространения.

### Сеанс разоблачения суеверий

Поговорим немного о заблуждениях.  
Я не знаю другой такой темы в веб-разработке, которая обросла бы подобным количеством заблуждений и суеверий. Разве что тема шаблонизации, хехе.  
Судя по всему, механизм здесь такой: с азами защиты от инъекций мы знакомимся очень рано, в самом начале. И заученные правила запоминаются в качестве основ, незыблемых истин. И в дальнейшем мы уж к рассмотрению вопроса не возвращаемся. А надо бы.  
Плюс толковой информации в интернете как всегда мало, а самому тестировать и разбираться лень. В итоге, наряду с уже существующими суевериями, в интернет вбрасываются (и подхватываются массами!) новые, не менее непрофессиональные. Попробуем разобраться с некоторыми из них

### Функции mysql\_\* уже давно deprecated в PHP

Грустная тема.  
История о том, как лень и невежество хоронят прекрасную библиотеку.

* Во-первых, не «функции», а расширение целиком.
* Во-вторых, не deprecated, а discouraged, причём не давно, а совсем недавно. Тоже не сахар, но надо быть точнее в формулировках.
* В-третьих, никаких реальных причин отказываться от этой прекрасной и стабильной библиотеки нету. И единственная причина, по которой её хотят исключить из языка — не нашлось мейнтейнера, который бы хотел ей заниматься. Ну и плюс негативное общественное мнение, которое сложилось только из-за того, что общество само же не научилось правильно с ней обращаться. Однако против лома не попрёшь, и большие красные предупреждения в мануале говорят нам о том, что с расширения mysql надо срочно переезжать.

Но в этой фразе есть еще один нюанс, очень важный. Из разряда вещей, которые не понимают очень многие начинающие разработчики.  
Если говорить именно о «функциях mysql\_», то в коде приложения их действительно быть не должно! Равно как и функций mysqli\_, pdo\_\* или любых других обращений к «голому» API. Прошу отметить — речь идёт о прикладном коде, а не о коде вообще: просто все обращения к функциям API должны быть упакованы в библиотеки, и в коде приложения нужно обращаться уже к библиотечным функциям. А использование всех этих mysql(i)\_query(), и mysql(i)\_fetch\_array() для исполнения конкретных запросов говорит, увы, о непрофессионализме.

Причина этого, в сущности, очень простая: функции API ужасно избыточны. Для того, чтобы получить строку данных из БД, надо написать десяток строк. И это не учитывая таких необходимых вещей, как обработка ошибок, логирование и профайлинг запросов и многого другого. В итоге, если мы пользуемся функциями API прямо в коде, то он получается, с одной стороны — ужасно избыточным, а с другой… недостаточно функциональным!  
Авторы PDO попытались сделать свою библиотеку более юзабельной, но, на мой взгляд, недостаточно. И обращения к PDO надо точно так же инкапсулировать в методы собственного класса.  
Ещё один аргумент — сейчас приходится менять mysql\_\* на что-то другое. В единственном файле библиотеки сделать это куда проще, чем в десятках проектов.

### Серебряной пули нет

Как видите — есть. Очень простой набор правил, который несложно реализовать и соблюдать.  
Надо сказать, что у меня тоже не сразу получилось. Сначала список не был исчерпывающим, потом — компактным.

### Эскейпинг

Здесь целый ворох суеверий. Я надеюсь, что после приведенных выше разъяснений вопросов уже не осталось, но, на всякий случай:

* «проблема с функцией mysql\_real\_escape\_string в том, что ей вообще пользуются» — это ерунда. Единственная проблема функции mysql\_real\_escape\_string() только в том, что очень многие люди не понимают, для чего она нужна
* «эскейпинг нужен для входящих данных» — разумеется, это не так. эскейпинг нужен для строк
* «эскейпинг делает „вредные“ символы „безопасными“» — это полная ерунда. Никаких «вредных» символов не бывает. Грубо говоря, эскейпинг всего лишь экранирует ограничители строк, чтобы они, встретившись в тексте, не разбили строку — вот и всё
* «эскейпинг в принципе имеет какое-то отношение к защите от инъекций» — ну, вы поняли уже? Эскейпинг нужен для форматирования строк. Причём любых строк, а не только потенциально опасных. А защита обеспечивается уже в качестве побочного эффекта
* «при использовании подготовленных выражений эскейпингом занимается база» — здесь, проблема, скорее в терминологии. С одной стороны, всё верно — серверные плейсхолдеры обрабатываются базой. Но она не эскейпит данные — это просто не нужно, поскольку данные попросту не попадают в SQL запрос. Забавным исключением здесь является PDO, которая по умолчанию работает в режиме совместимости — то есть, не использует серверные плейсхолдеры, а подставляет данные в запрос сама. Но делает PDO это не на сервере. Плюс, как мы уже знаем, слово «эскейпинг» не является синонимом безопасности. Так что PDO занимается не «эскейпингом», а корректным форматированием данных.

### Проверка входящих данных на «опасные» слова и символы

Такие как «UNION», одинарная кавычка и пр. Очень смешной способ защиты. Я, когда вижу на каком-нибудь сайте автора, всерьёз предлагающего очередной её вариант, всегда хочу его спросить — а что было бы, если бы если сам сайт пользовался предлагаемой защитой? Наш горе-изобретатель просто не смог бы опубликовать свой пост, как содержащий те самые «опасные» «стоп-слова».

### Универсальная функция защиты данных

Тоже очень распространённое заблуждение среди начинающих пхпшников. «Поставим на входе универсальную функцию, которая будет убивать все „вредные символы“ в поступающих данных и таким образом обезопасим себя от любых инъекций!» — думают они.  
Но как мы уже выяснили, «вредных» символов не бывает. Есть только служебные символы, которые имеют определённое значение только в определённом контексте. И совершенно безопасны во всех других случаях.  
Также мы выяснили, что в первую очередь нам нужна не безопасность, а корректное форматирование данных. А сделать это можно только тогда, когда мы уже знаем — в каком контексте они будут использоваться. На входе в скрипт мы этого ещё, увы, не знаем. Яркий пример такого подхода представляет собой печально известная директива magic\_quotes, ныне, к счастью, изъятая из языка. Обрабатывая совсем не те данные, которые нужно, она создавала иллюзию безопасности, защищая только строки (и только попадающие в скрипт извне).  
Так что, пожалуйста, не повторяйте этих ошибок — не форматируйте данные для SQL сразу при попадании их в скрипт.

### Кодировки

Тема кодировок относится только к ручному форматированию строк. При обработке плейсхолдеров на сервере никакой проблемы не представляет.  
Тема тут старая, «это все придумал Шифлетт в восемнадцатом году» и тогда же Альшанецкий показал, что mysql\_real\_escape\_string() тоже не помогает(!)  
Вся фишка в том, что вопреки сказанному в документации, mysql\_real\_escape\_string() не «принимает во внимание кодировку соединения», если только её специально не пнуть. По умолчанию же текущая кодировка — latin1, для которой никакие хитрые настройки не требуются, и mysql\_real\_escape\_string() действует не умнее своей туповатой родственницы mysql\_escape\_string().  
При этом в 2006 году средства сообщить функции mysql\_real\_escape\_string() текущую кодировку в расширении mysql просто не было. С тех пор, впрочем, прогресс сильно ушёл вперёд и средство появилось — функция mysql\_set\_charset(). Которая и должна использоваться для выставления клиентской кодировки вместо запроса SET NAMES.

В любом случае, инъекция возможна только для некоторых экзотический кодировок. Все однобайтные кодировки и UTF-8 безопасны и так (следствием чего является тот факт, что для них годится и пресловутая addslashes()).

Я не поленился в свое время проверить, как это всё работает. Убедился — работает ровно так, как предсказано.

Характерная деталь, о которой я упоминал ранее:  
В PDO до недавнего времени вообще нельзя было выставить кодировку соединения. Функции, аналогичной mysql\_set\_charset() в PDO нету, а в DSN до версии 5.3 был только муляж параметра charset, который ошибок не выдавал, но и никакую кодировку не выставлял.  
Ничего, в сущности, особенного, кроме возможности троллить персонажей, рассказывающих о том, что PDO защищает от всего.

### «Уязвимость в LIKE»

Уязвимость в LIKE бывает только одна: применение этого оператора не по назначению.  
Почему-то всякие составители статей ужасно любят пугать новичков возможностью подставить в выражение метасимвол. При этом никто не пишет, что если LIKE используется, то метасимвол в любом случае должен быть. Иначе в LIKE нет смысла. То есть, правильным средством борьбы с этой уязвимостью будет не экранирование символов % и \_, а неиспользование этой функции там, где нас интересует точное совпадение — для проверки пароля, например. Вот и всё.