электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

 # 10, октябрь 2015

УДК 004.056.5

SQL-инъекции

Колесникова К. И., *студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»

Кошкарева Ю. И., *студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»

Научный руководитель: Гапанюк Ю.Е., к.т.н., доцент
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»
gapyu@bmstu.ru

Введение в SQL-инъекции

SQL-инъекции (англ. SQL-injection) – «один из распространённых способов взлома сайтов и программ, работающих с базами данных, основанный на внедрении в запрос произвольного SQL-кода».[1]

В зависимости от вида используемой СУБД и условий внедрения SQL-кода, SQL-инъекции могут позволить взломщику выполнить любой запрос к базе данных, к примеру, прочитать содержимое таблиц, удалить, изменить или добавить данные, приобрести возможность чтения и/или записи локальных файлов, а также выполнения произвольных команд на атакуемом сервере. В худшем случае внедренный код позволит злоумышленнику получить полный контроль над системой, используя множество уязвимостей в базе данных сервера и системные утилиты операционной системы.

Иначе говоря, SQL-инъекции позволяют злоумышленникам получить несанкционированный доступ к данным.

Попробуем упростить представление о применении SQL-кода, используя простой пример из повседневной жизни:

Предположим, что мать оставляет записку отцу с просьбой дать Ване, одному из сыновей, 200 рублей. Представим, как бы выглядел текст оставленной записки, в виде SQL-запроса.

ДОСТАНЬ ИЗ кошелька 200 РУБЛЕЙ И ДАЙ ИХ Ване

Так как мать оставила записку на видном месте, чтобы ее увидел отец, то второй сын Саша получил доступ к тексту записки. Саша дописал в записке «ИЛИ Саше» и SQL-запрос стал выглядеть иначе:

ДОСТАНЬ ИЗ кошелька 200 РУБЛЕЙ И ДАЙ ИХ Ване ИЛИ Саше

Отец, увидев записку, решил отдать деньги Саше, так как Ване давал их вчера. Вот простой пример применения SQL-инъекций в жизни. В данном примере Саша и является взломщиком, который, благодаря изменению SQL-запроса, добился нужных ему результатов.

Классификация SQL-атак

Рассмотрим различные типы атак, которые применяют SQL-инъекции. Иллюстрация представлена на рис. 1.

Рис. 1. Классификация SQL-атак

1) Изменение входных параметров путём добавления в них конструкций языка SQL.

В данном случае речь идет о внедрении SQL-кода в запросы, которые используют параметр id в качестве входного параметра.

Например, рассмотрим такой SQL-запрос:

*SELECT * FROM Articles WHERE Article_ID = id,*

В данном запросе *Articles* – некоторая таблица статей, *Article_ID* – уникальный номер статьи в таблице *Articles*, *id* – параметр, получаемый сервером.

Очевидно, что если на сервер передан параметр *id*, равный, например, 80, то

выполнится следующий SQL-запрос:

*SELECT * FROM Articles WHERE Article_ID = 80*

Однако, если передать на сервер в качестве параметра *id* строку *-1 OR 1=1*, то получится следующий SQL-запрос:

*SELECT * FROM Articles WHERE Article_ID= -1 OR 1=1*

В результате выполнения этого SQL-запроса сервер выведет все имеющиеся в базе статьи, так как *Article_ID= -1* не верное, по умолчанию, условие, а условие *1=1* – верно всегда. Подводя итог, становится очевидным, что изменение входных параметров с помощью SQL-инъекций изменяет логику всего SQL-запроса.

2) Внедрение SQL-кода в строковые параметры запроса.

Данный тип SQL-инъекций похож на предыдущий. Предположим, у нас есть запрос, в результате выполнения которого, пользователь получает выборку статей. Статья попадает в выборку, если в ее названии есть необходимое слово. Тогда SQL-запрос будет иметь вид:

SELECT Article_ID, Article_Date, Article_Caption, Article_Text, Article_ID_Author FROM Articles WHERE Article_Caption LIKE('%Search_Text%'),

В данном запросе *Articles* – некоторая таблица статей, *Article_ID* – уникальный номер статьи, *Article_Date* – дата публикации статьи, *Article_Caption* – описание статьи, *Article_Text* – текст статьи, *Article_ID_Author* – уникальный номер автора статьи в таблице *Articles*, *Search_Text* – передаваемый параметр.

Очевидно, что если на сервер передан параметр *Search_Text*, равный, например, *Internet*, то выполнится следующий SQL-запрос:

SELECT Article_ID, Article_Date, Article_Caption, Article_Text, Article_ID_Author FROM Articles WHERE Article_Caption LIKE('%Internet%')

Однако злоумышленник может добавить символ кавычки в параметр *Search_Text*, тогда логика SQL-запроса изменится. Рассмотрим, например, вариант, когда на сервер передано следующее значение параметра *')+and+(Article_ID_Author ='1*. Тогда сервер будет выполнять SQL-запрос:

SELECT Article_ID, Article_Date, Article_Caption, Article_Text, Article_ID_Author FROM Articles WHERE Article_Caption LIKE('%')+and+(Article_ID_Author ='1%')

Таким образом, после выполнения данного запроса, мы получим уже не выборку статей с заданным параметром *Search_Text*, а выборку статей определенного автора. Становится очевидным, что суть SQL-запроса полностью изменена.

3) Использование UNION

«В языке SQL ключевое слово UNION применяется для объединения результатов двух SQL-запросов в единую таблицу, состоящую из схожих строк. Оба запроса должны возвращать одинаковое число столбцов и совместимые типы данных в соответствующих столбцах».[2] Использование данной функции позволяет злоумышленнику получить доступ к закрытым данным.

Рассмотрим следующий SQL-запрос:

SELECT Article_ID, Article_ Header, Article_Text, Article_Author FROM Articles WHERE Article_ID= id;

В данном запросе *Articles* – некоторая таблица статей, *Article_ID* – уникальный номер статьи, *Article_ Header* – заголовок статьи, *Article_Text* – текст статьи, *Article_Author* – имя автора статьи в таблице *Articles*, *id* – передаваемый параметр.

Очевидно, что если на сервер передан параметр *id*, равный, например, 80, то будут получены данные статьи с уникальным номером 80: ее номер, заголовок, текст, имя автора статьи.

Однако, если злоумышленник воспользуется ключевым словом UNION, то он может передать на сервер следующую конструкцию *-1 UNION SELECT 1, username, password, 1 FROM admin*

Тогда SQL-запрос примет вид:

SELECT Article_ID, Article_ Header, Article_Text, Article_Author FROM Articles WHERE Article_ID=-1 UNION SELECT 1, username, password, 1 FROM admin

Скорее всего статьи с уникальным номером -1 не существует, поэтому в выборку попадут только записи из таблицы *admin*. Таким образом злоумышленник получает доступ к именам и паролям пользователей.

4) Экранирование хвоста запроса

Это тип атак с использованием SQL-кода можно встретить в случае, если запрос имеет сложную структуру, к которой достаточно сложно или вовсе невозможно применить UNION. Например, рассмотрим запрос:

SELECT Author FROM Articles WHERE Author_ID=id AND Author_Name LIKE ('A%');

В данном запросе *Articles* некоторая таблица статей, *Author_ID* – уникальный номер автора статьи, *Author_Name* – имя автора статьи, *id* – передаваемый параметр.

Сервер создает выборку имен авторов по передаваемому параметру id, только в том случае, если имя автора начинается с буквы А. В данном случае злоумышленнику достаточно сложно внедрить в запрос UNION, поэтому используется экранирование

запроса. А именно передается следующее значение параметра *-1 UNION SELECT username, password FROM admin/**

Тогда запрос приобретает следующий вид:

SELECT Author FROM Articles WHERE Author_ID=-1 UNION SELECT username, password FROM admin / AND Author_Name LIKE ('A%');*

Результатом выполнения этого запроса будет выборка имен пользователей и паролей из таблицы admin, так как автора с уникальным номером -1 скорее всего не существует, а часть запроса, ограничивающую выборку по имени автора, злоумышленник закомментировал. Таким образом часть запроса *AND Author_Name LIKE ('A%')* не влияет на выполнение.

В зависимости от типа СУБД экранирование запроса осуществляется различными символами комментирования.

5) Расщепление SQL-запроса

Данный тип атаки подразумевает выполнение нескольких SQL-запросов, вместо запланированного одного. Для этого злоумышленник использует символ «**;**» (*точка с запятой*). Однако не все версии SQL поддерживают такую возможность.

Предположим у нас есть запрос:

*SELECT * FROM Articles WHERE Article_ID = id**;***

В данном запросе *Articles* – некоторая таблица со статьями, *Article_ID* – уникальный номер статьи в таблице *Articles*, *id* – параметр, получаемый сервером.

Если вместо обычного значения параметра передать, например, следующее значение

80; INSERT INTO admin (username, password) VALUES ('NewName', '1234'); то в одном запросе будут выполнены 2 команды

*SELECT * FROM Articles WHERE Article_ID = 80**;***

INSERT INTO admin (username, password) VALUES ('NewName', '1234');

В таблицу admin будет добавлена запись о новом администраторе.

Обнаружение злоумышленниками уязвимых SQL-кодов

Для того чтобы найти SQL-запросы, которые могут быть подвержены атакам, злоумышленники изучают поведение серверов в зависимости от различных манипуляций с кодом запросов. Целью атакующих является обнаружение так называемого аномального поведения сервера. Ниже представлен список параметров, наиболее часто используемых для изучения реакции сервера:

- Данные, передаваемые через методы POST и GET
- Значения [HTTP-Cookie]
- HTTP_REFERER (для скриптов)
- AUTH_USER и AUTH_PASSWORD (при использовании аутентификации).

Обычно манипуляции основываются на подстановке в параметры запроса символов одинарной или двойной обратной кавычки.

К аномальному поведению относят любое поведение, «при котором страницы, получаемые до и после подстановки кавычек, различаются (и при этом не выведена страница о неверном формате параметров)».[1]

Примерами аномального поведения могут считаться:

- вывод сообщения о различных ошибках;
- отсутствие вообще какого-либо сообщения (т.е. не выводится совсем ничего, хотя страница отображается). Однако здесь стоит отличать случаи, кода сообщения не выводятся в силу специфики разметки страницы, такие ошибки отражаются в HTML-коде страницы.

Методы борьбы с внедрением SQL-кодов

Для того чтобы избежать атак, связанных с внедрением враждебных SQL-кодов в запросы, разработчики стремятся создать приложение, которое будет работать исключительно с параметризированными запросами, они стараются ограничить приложению доступ к данным сервера с помощью хранимых процедур, то есть разработать приложение, не имеющее возможности создавать собственные SQL команды. При использовании хранимых процедур приложение имеет разрешения, необходимые только для выполнения этих процедур, у него нет доступа к базовым таблицам. Поэтому, даже если код является подверженным SQL-инъекциям, у злоумышленников не получится получить доступ к данным, так как у приложения не хватает прав для доступа и манипулирования таблицами. Помимо этого хранимые процедуры имеют встроенные функции для проверки типа передаваемых приложению параметров.

Однако в реальной жизни создание подобных приложений трудно реализуемо, особенно при работе с системами, которые были созданы ранее. Сложности возникают и при реализации приложения в смешанных средах, а также, если команда разработчиков не может договорится о наиболее выгодном решении между собой. В связи с этим создан набор правил, которых следует придерживаться разработчикам при создании компонентов

баз данных для поддержки различных типов приложений. Несмотря на то, что не каждое из этих правил применимо для каждого приложения, все-таки следует присмотреться ко всем требованиям. На рис. 2 представлена графическая иллюстрация этих требований.

Рис. 2. Методы борьбы с внедрением SQL-кодов

1) Фильтрация строковых параметров

Рассмотрим пример кода (на языке Паскаль), генерирующий SQL-запрос:

*statement := 'SELECT * FROM users WHERE name = ''' + userName + ''';';*

В данном случае, чтобы сделать SQL-инъекции невозможными, нужно выполнить процедуру замены символов, а именно экранировать спецсимволы. При выполнении этой процедуры в параметре заменяют:

- " (кавычки) на \"
- ' (апостроф) на \'
- \ (обратную косую черту) на \\

Рассмотрим блок-схему подобной процедуры. Данная подпрограмма будет получать на входе строку, а на выходе выдавать уже строку в кавычках и с заменёнными спецсимволами.

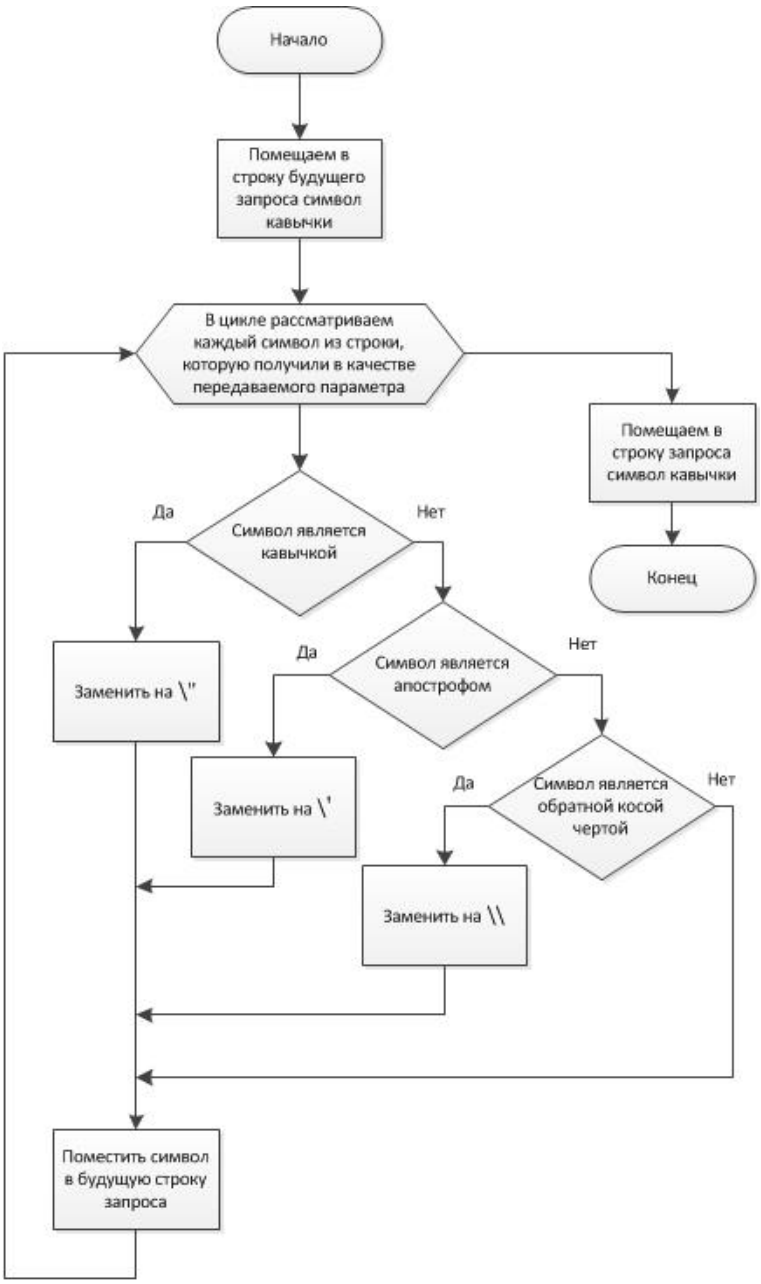

Рис. 3. Блок схема процесса экранизации символов

2) Фильтрация целочисленных параметров

Рассмотрим пример кода (на языке Паскаль), генерирующий другой SQL-запрос:

*statement := 'SELECT * FROM users WHERE id = ' + id + ';';*

В ситуации, когда передаваемый параметр имеет числовой тип, замена спецсимволов и помещение строки в кавычки не поможет, так как чаще всего числовые параметры передаются без кавычек. Поэтому в таких случаях разработчики могут проверить тип параметра, а именно, уточнить является ли параметр числом. Тогда если параметр не число, то запрос не должен выполняться вовсе.

Во многих языках программирования существуют стандартные функции для

конвертации строку с целым значением в целочисленное значение.

Применим процедуру проверки параметра (на языке Delphi) и рассмотрим выполнение SQL-запроса, представленного выше:

id_int := StrToInt(id);

*statement := 'SELECT * FROM users WHERE id = ' + IntToStr(id_int) + ';';*

Если параметр *id* не является целочисленным, то возникнет ошибка, и функция *StrToInt* вызовет исключение *EConvertError*. В обработчике этого исключения можно будет задать вывод сообщения об ошибке. Благодаря двойному преобразованию, реакция на числа в шестнадцатеричной системе счисления будет корректной.

3) Усечение входных параметров

Одним из видов защиты может стать сокращение длины строки для входных параметров запроса. Из примеров, разобранных выше видно, что для нарушения логики запроса необходим ввод достаточно длинных строк, минимальная длина внедряемой строки в наших примерах составляет 8 символов (*1 OR 1=1*). Поэтому, если заранее известно, что длина передаваемого параметра не может превышать какого-то определенного значения, то можно обрезать строку с помощью специальных функций.

В качестве примера рассмотрим запрос с передаваемым параметром id. Если известно, что его значение не может превышать 999, то можно усечь входную строку до 3-х символов, тогда получим:

*statement := 'SELECT * FROM users WHERE id = ' + LeftStr(id, 4) + ';';*

В данном случае *LeftStr* – процедура усечения передаваемой строки, она стандартна для языка Паскаль.

4) Функция по автоматическому определению зарезервированных SQL-слов в тексте и блокировки IP-адреса.

Одной из мер безопасности, применяемых для защиты от SQL-инъекций, являются специальные функции. Они подсчитывают количество использования специально зарезервированных SQL-слов в запросе. Если это количество превышает какое-то заранее определенное значение, то IP-адрес, с которого выполнялись данные запросы, заносится в список опасных и навсегда блокируется. Далее функция возвращает значение *TRUE*, и затем прекращается работа скрипта.

Данная функция вызывается еще до обращения к базе данных с полученными из формы параметрами, чтобы, например, проверить все полученные данные в цикле.

5) Использование параметризованных запросов

Данный вид защиты может быть реализован разработчиками, благодаря

возможности отправки параметризированных запросов серверами баз данных. В данном случае передаваемые внешние параметры передаются на сервер отдельно от SQL-запроса или автоматически экранируются клиентской библиотекой.

Рассмотрим реализацию данной функции защиты для разных языков программирования:

- на Delphi — свойство TQuery.Params;
- на Perl — через DBI::quote или DBI::prepare;
- на Java — через класс PreparedStatement;
- на C# — свойство SqlCommand.Parameters;
- на PHP — MySQLi (при работе с MySQL), PDO.

на Parser — язык сам предотвращает атаки подобного рода.

6) Использование принципа наименьших привилегий при предоставлении доступа к базам данных

Этот способ защиты следует использовать, даже если вы допускаете доступ приложению только через хранимые процедуры. Каждой учетной записи базы данных должны быть назначены минимальные права, необходимые для доступа к данным. Следует стремится к тому, чтобы, предоставляя приложению право выполнять процедуры, запрещался доступ к таблицам данных. Если использование хранимых процедур становится невозможным, следует предоставить возможность выполнения прямых SQL-запросов через учетные записи с наименьшими привилегиями. То есть у данной учетной записи должны быть четко ограничены границы доступа к чтению/добавлению/изменению/удалению данных в таблицах, а также обозначено к каким именно таблицам есть доступ. Следует избегать создания учетной записи администратора в приложении. Если соблюдать принцип наименьших привилегий, то даже в случае обнаружения злоумышленником уязвимого кода, он сможет нанести мало вреда.

7) Использование тестирования и мониторинга для защиты от SQL-инъекций.

Даже если разработчики выполнили все рекомендации, перед запуском кода базы данных следует выполнить необходимые проверки. Обязательно нужно проверить код специальными методами для выявления уязвимости к SQL-инъекциям. На данном этапе следует попробовать подвергнуть код SQL-инъекции. После запуска базы данных лог-файлы и другие устройства слежения должны быть использованы для мониторинга баз данных для любых признаков SQL-инъекции.

Таким образом задача надежной защиты информации от несанкционированного доступа является одной из самых распространенных и не решенных полностью до настоящего времени проблем. Рассмотренные методы борьбы с SQL-инъекциями помогут совершенствовать разработчикам свои программы, увеличить надежность хранения информации и уменьшить вероятность подверженности подобным атакам.

Список литературы

1. Статья «Внедрение SQL-кода». Режим доступа: https://ru.wikipedia.org/wiki/%D0%92%D0%BD%D0%B5%D0%B4%D1%80%D0%B5%D0%BD%D0%B8%D0%B5_SQL-%D0%BA%D0%BE%D0%B4%D0%B0 (дата обращения 22.03.15).
2. Статья «Union (SQL)». Режим доступа: https://ru.wikipedia.org/wiki/Union_%28SQL%29 (дата обращения 01.04.15).
3. Евтеев Д.А. SQL Injection от А до Я Режим доступа: http://www.ptsecurity.ru/download/PT-devteev-Advanced-SQL-Injection.pdf (дата обращения 25.03.15).
4. Статья «MSSQL: Практика предотвращения SQL-injection» Режим доступа: https://comnote.wordpress.com/category/microsoft-sql-server/ (дата обращения 25.03.15).
5. Статья «SQL injection для начинающих» Режим доступа: http://habrahabr.ru/post/148151/ (дата обращения 22.03.15)
6. Медведев Н.В., Быков А.Ю., Гришин Г.А. Противодействие неавторизованному доступу к серверу Web-приложений с использованием механизма подложных SQL-запросов // Вестник МГТУ им. Н.Э. Баумана. Сер. Приборостроение. 2006. № 3. Режим доступа: http://vestnikprib.bmstu.ru/catalog/it/hidden/329.html (дата обращения 22.03.15).
7. Постников В.М., Гребенников Н.А. Технология обработки запросов пользователей в СУБД ORACLE // Вестник МГТУ им. Н.Э. Баумана. Сер. Приборостроение. 2001. № 2. Режим доступа: http://vestnikprib.bmstu.ru/catalog/it/hidden/485.html (дата обращения 25.03.15).
8. Самохвалов Э.Н., Ревунков Г.И., Гапанюк Ю.Е. Генерация исходного кода программного обеспечения на основе многоуровнего набора правил // Вестник МГТУ им. Н.Э. Баумана. Сер. Приборостроение. 2014. № 5. Режим доступа: http://vestnikprib.bmstu.ru/catalog/it/hidden/527.html (дата обращения 22.03.15).