

SQL-инъекции как метод атаки на веб-ориентированные информационные системы

Болтовский Гавриил Александрович

Приамурский государственный университет им. Шолом-Алейхема

Студент

Фатеенков Данила Витальевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В статье рассматриваются SQL-инъекции как метод атаки на информационные системы. Описываются принципы работы инъекций, их появление в информационных системах, а также защита сайтов и других ИС от внедрения такого рода атак. Примеры представлены на PHP, а также применяются специальные модули и библиотеки для работы с СУБД MySQL для выбранного языка программирования.

Ключевые слова: MySQL, SQL-инъекции, информационная безопасность, PHP, подготовка SQL запросов

SQL injection as a method of attack on web-oriented information systems

Boltovsky Gavriil Alexandrovich

Sholom-Aleichem Priamursky State University

Student

Fateenkov Danila Vitalievich

Sholom-Aleichem Priamursky State University

Student

Abstract

The article discusses SQL injection as a method of attacking information systems. It describes the principles of injections, their appearance in information systems, as well as the protection of sites and other IS from the introduction of such attacks. Examples are presented in PHP, and special modules and libraries for working with DBMS MySQL for the selected programming language are applied.

Keywords: MySQL, SQL injection, information security, PHP, SQL query preparation

1. Введение

1.1 Актуальность

Информационная безопасность в настоящее время остаётся одним из основных направлений в IT сфере за счёт постоянно изменяющихся подходов к получению несанкционированного доступа к информации различных систем: начиная с прикладного ПО, работающего с локальными БД, заканчивая полноценными веб-ориентированными ИС, в которой могут быть представлены большие объём информации.

Одним из самых популярных основных видов атак на информационные системы в настоящее время являются SQL-инъекции, представляющие собой достаточно простой, но эффективный способ обхода ограничений в ИС и получения доступа к ограниченному для определённых лиц функционалу системы.

Методы защиты ИС от SQL-инъекций остаются актуальными в настоящее время за счёт постоянных изменений в подходе организации запросов к БД со стороны клиента и также за счёт нахождения угроз безопасности на уровне фреймворков, ORM модулей и самих языков программирования.

1.2 Обзор исследований

А.К. Починяев, А.А. Лобачев, А.А. Медведев и В.С. Чернова в своём исследовании рассмотрели различные способы защиты от SQL-инъекций, от наиболее продвинутых и до простых в своей реализации. Авторы сделали вывод о том, что должно использоваться для лучшей надёжности и безопасности веб-систем [1].

Г.М. Казанюк и В.В. Воронин рассмотрели в своей работе атаки на базы данных SQL с помощью инъекций [2]. Приведены варианты последствий и способы защиты от данного вида атак на информационные системы.

В.М. Митрофанов, М.А. Крибель, С.А. Фроленков и Л.О. Спицын разработали систему обучения искусственных нейронных сетей для осуществления обнаружения признаков компьютерных атак на основе языка программирования Python на информационно-коммуникационные системы в реальном времени на основе метода логической регрессии [3].

Г.В. Беликов, И.Д. Крылов и В.А. Селищев провели исследование SQL-инъекций и рассмотрели основные аспекты работы SQL-инъекций, опасность допуска таких уязвимостей и продемонстрировали пример внедрения инъекции [4].

Д.И. Якушев и Н.А. Вайберт рассмотрели вопрос защиты информации от SQL-инъекций, выявили основные причины появления таких уязвимостей, а также методы защиты от подобного типа атак [5].

1.3 Цель исследования

Цель – рассмотреть основные виды SQL-инъекций, их реализации и методы предотвращения.

2 Материалы и методы

Для решения поставленной задачи используется язык программирования PHP, а также ORM RedBeanPHP.

3 Результаты и обсуждения

SQL-инъекция – это вид атаки на веб-приложение, при котором злоумышленник внедряет вредоносный SQL-код в запросы к базе данных. Это позволяет атакующему взаимодействовать с базой данных напрямую, что может привести к утечке, изменению или удалению данных, а также к другим опасным последствиям.

Веб-приложения часто используют SQL-запросы для взаимодействия с базой данных. Например, при авторизации пользователя приложение может отправить такой запрос:

```
SELECT * FROM `user` WHERE `username` = 'Ik9YB6WdfA' AND  
`password` = 'KmKwtCxRJw';
```

Если приложение не проверяет и не обрабатывает входные данные должным образом, злоумышленник сможет подставить в поля ввода специально сформированные данные, которые изменяют логику SQL-запроса. Например:

1. В поле имени пользователя вводится: ' OR '1'='1
2. В поле пароля: любое значение.

Тогда запрос примет вид:

```
SELECT * FROM `user` WHERE `username` = '' OR '1'='1' AND  
`password` = 'KmKwtCxRJw';
```

Условие '1'='1' всегда истинно, поэтому запрос вернёт все записи из таблицы users, и злоумышленник может получить доступ к системе.

Описанный выше пример представляет собой классический вариант SQL-инъекций, также называемый Classic SQLi или также In-band SQL Injection. Главной отличительной особенностью такого варианта можно считать использование единого канала передачи данных между клиентом и сервером приложения, то есть атакующий отправляет вредоносный SQL-запрос через входные данные приложения (например, формы ввода, URL-параметры), и при этом результат атаки (данные или ошибки) возвращается через тот же интерфейс, что и обычные ответы приложения.

Поскольку результат атаки сразу виден, злоумышленник может быстро определить, уязвимо ли приложение.

Принцип работы In-band SQL Injection заключается в следующем:

1. Атакующий вводит в поля ввода специальные символы, такие как ', ", --, #, /*, чтобы проверить, как приложение обрабатывает входные данные.
2. Если приложение возвращает ошибки или неожиданные результаты, это указывает на возможную уязвимость.

3. Атакующий формирует вредоносный SQL-запрос, чтобы извлечь данные, изменить их или вызвать ошибки.

4. Результаты атаки (данные или ошибки) отображаются в интерфейсе приложения.

Помимо классических методов внедрения SQL-запросов в виде инъекций выделяют ещё 2 основных вида атак:

1. SQL инъекция, основанная на эксплуатации выводимых СУБД сообщений об ошибках (Error-based SQLi).

2. Слепая SQL инъекция (Blind SQLi).

Стоит разобрать каждый вид инъекций подробнее перед тем, как рассматривать примеры защиты ИС от данного рода атак.

Error-based SQL Injection – это вид SQL-инъекции, при котором злоумышленник намеренно вызывает ошибки в базе данных, чтобы получить информацию о её структуре или данных. Этот метод часто используется, когда приложение отображает сообщения об ошибках, возвращаемые СУБД (системой управления базами данных). Эти ошибки могут содержать ценную информацию, такую как имена таблиц, столбцов, версия базы данных и даже сами данные.

Отличительной особенностью данного вида инъекций считается уникальный подход к вызову ошибок со стороны СУБД для получения необходимой для атаки информации (версии и типа СУБД, структуры БД и др.).

Работает Error-based SQLi следующим образом:

1. Атакующий вводит в поля ввода специальные символы, такие как ', ", --, #, чтобы проверить, как приложение обрабатывает входные данные.

2. Если приложение возвращает ошибки базы данных, это указывает на возможную уязвимость.

3. Атакующий формирует вредоносный SQL-запрос, который вызывает ошибку и возвращает полезную информацию.

4. Сообщения об ошибках могут содержать имена таблиц, столбцов, версию базы данных или даже сами данные.

Предположим, приложение использует следующий запрос:

```
SELECT * FROM `user` WHERE `id` = '14';
```

Атакующий может ввести в поле id значение, отличное от целого числа: ' AND 1=CONVERT(int, (SELECT @@version)) --. И в таком случае запрос примет следующий вид:

```
SELECT * FROM `user` WHERE `id` = '' AND 1=CONVERT(int, (SELECT @@version)) --';
```

Наиболее опасным случаем Error-based SQLi можно считать вывод информации о структуре БД на сервере. Для этого достаточно представленный

выше запрос изменить, чтобы создать ошибку, которая будет выводить название таблицы в БД. Можно использовать функцию EXTRACTVALUE, которая работает с XML-данными и вызывает ошибку, если XML-синтаксис некорректен.: ' AND EXTRACTVALUE(1, CONCAT(0x3a, (SELECT TABLE_NAME FROM information_schema.tables WHERE TABLE_SCHEMA = DATABASE() LIMIT 1,1))) --.

Итоговый запрос будет выглядеть следующим образом:

```
SELECT * FROM `user` WHERE `id` = '' AND EXTRACTVALUE(1,
CONCAT(0x3a, (SELECT TABLE_NAME FROM information_schema.tables
WHERE TABLE_SCHEMA = DATABASE() LIMIT 1,1))) -- `password`='1241';
```

```
mysql> SELECT * FROM scary_stories.user WHERE id = '' AND EXTRACTVALUE(1, CONCAT(0x3a, (SELECT TABLE_NAME FROM information_schema.tables WHERE
TABLE_SCHEMA = DATABASE() LIMIT 1,1))) -- password='1241';
-> ;
ERROR 1105 (HY000): XPATH syntax error: ':category'
mysql> SELECT * FROM scary_stories.user WHERE id = '' AND EXTRACTVALUE(1, CONCAT(0x3a, (SELECT TABLE_NAME FROM information_schema.tables WHERE
TABLE_SCHEMA = DATABASE() LIMIT 2,1))) -- password='1241';
-> ;
ERROR 1105 (HY000): XPATH syntax error: ':commentary'
```

Рисунок 1. Пример выполнения Error-based SQLi запроса с выводом ошибки, в которой содержится название таблиц

Таким образом можно перебрать все таблицы, изменяя значение offset для параметра LIMIT (после превышения значения количества таблиц данный запрос начнёт возвращать пустой результат, что будет служить маркером о том, что были раскрыты все таблицы из БД).

Но главным недостатком такого вида SQL-инъекций можно считать тот факт, что не все языки программирования (или отдельные модули и библиотеки) напрямую отображают информацию о возникающей ошибке на стороне СУБД. Это не позволит программисту получить необходимую для дальнейшей работы информацию (см. рис. 2).

Войти в систему

Логин

' AND EXTRACTVALUE(1, CONCAT(0x3a, (SELECT TAE

Пароль

.....

Войти

Произошла ошибка во время входа в систему.
Попробуйте позже. PDOException: SQLSTATE[42000]:
Syntax error or access violation: 1064 You have an error in
your SQL syntax; check the manual that corresponds to your
MySQL server version for the right syntax to use near
'21356236236" at line 1 in C:\xampp\htdocs\login.php:55
Stack trace: #0 C:\xampp\htdocs\login.php(55): PDO->
>query('SELECT * FROM u...') #1
C:\xampp\htdocs\login.php(82): login() #2 {main}

© 2025 Umbrella Apps

Рисунок 2. Пример неинформативной ошибки на стороне сервера (язык программирования, который был использован для отправки запроса на сервер – PHP)

В случае, если стандартный способ и основанный на ошибках не подошёл, атакующий может предпринять попытку получения данных с помощью слепых SQL-инъекций.

Blind SQL Injection (слепая SQL-инъекция) – это тип атаки, при котором злоумышленник не может напрямую увидеть результат выполнения SQL-запроса (например, данные из базы данных или сообщения об ошибках). Вместо этого атакующий определяет результат запроса по косвенным признакам, таким как:

1. Изменение поведения приложения (например, наличие или отсутствие определённого контента на странице).

2. Время ответа сервера (например, задержка выполнения запроса).

3. HTTP-коды ответа (например, 200 OK или 500 Internal Server Error).

Blind SQL Injection используется, когда приложение не отображает данные из базы данных напрямую, но всё ещё уязвимо к SQL-инъекциям.

Существуют 2 основных вида слепых SQLi:

1. Boolean-based Blind SQL Injection: атакующий определяет результат запроса по логическому ответу (true/false). Например, приложение может показывать разный контент в зависимости от результата запроса.

2. Time-based Blind SQL Injection: атакующий определяет результат запроса по времени ответа сервера. Например, если запрос выполняется дольше обычного, это может указывать на истинность условия.

Часто данный вид инъекций применяется в контексте грубой силы («brute-force» атака), предлагает атакующему эффективный, но зачастую медленный метод получения необходимой информации из таблицы.

Предположим, приложение использует тот же запрос, что и в Error-based SQLi:

```
SELECT * FROM `user` WHERE `id` = '14';
```

Если атакующий не знает пароль для подключения, но при этом знает имя пользователя, он может попробовать его получить с помощью слепой SQLi. Для начала атакующему необходимо узнать длину пароля. Для этого достаточно перебрать возможные целые значения, пока результат проверки не будет возвращать истинное значение. Для этого вместо id пользователя необходимо указать условие сравнения длины пароля с некоторым целым числом: ' AND (SELECT LENGTH(password) FROM user WHERE username = 'admin') = 10 --.

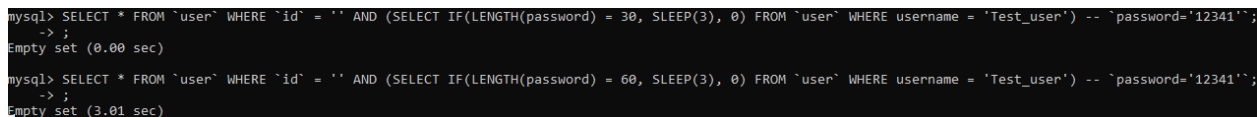
Итоговый запрос будет выглядеть следующим образом:

```
SELECT * FROM `user` WHERE `id` = '' AND (SELECT LENGTH(password) FROM user WHERE username = 'admin') = 60 --  
`password` = 12341;
```

Этот же запрос может быть переписан под принципы Time-based Blind SQLi: ' AND (SELECT IF(LENGTH(password) = 60, SLEEP(3), 0) FROM `user` WHERE username = 'admin') --.

Итоговый запрос будет выглядеть следующим образом:

```
SELECT * FROM `user` WHERE `id` = '' AND (SELECT IF(LENGTH(password) = 60, SLEEP(3), 0) FROM `user` WHERE username = 'admin') -- `password`='12341';
```



```
mysql> SELECT * FROM `user` WHERE `id` = '' AND (SELECT IF(LENGTH(password) = 30, SLEEP(3), 0) FROM `user` WHERE username = 'Test_user') -- `password`='12341';  
-> ;  
Empty set (0.00 sec)  
  
mysql> SELECT * FROM `user` WHERE `id` = '' AND (SELECT IF(LENGTH(password) = 60, SLEEP(3), 0) FROM `user` WHERE username = 'Test_user') -- `password`='12341';  
-> ;  
Empty set (3.01 sec)
```

Рисунок 3. Пример выполнения Time-based Blind SQLi для нахождения длины пароля пользователя

Таким же образом можно перебрать все символы пароля, а если брать во внимание факт того, что на сервере пароли могут храниться не в зашифрованном виде, то таким образом атакующий сможет получить доступ к учётной записи пользователя без каких-либо трудностей.

Главной проблемой для атакующего с помощью слепых SQL-инъекций может стать необходимость знания структуры базы данных для построения корректного запроса для будущей инъекции.

Рассмотрим пример PHP-функции входа в веб-ориентированную систему:

```
function login() {  
    $log = $_POST['login'];  
    $pass = $_POST['password'];  
  
    if (empty($log) || empty($pass)) {  
        $_SESSION['error'] = "Пожалуйста, заполните все  
поля.";  
        header("Location: login.php");  
        exit();  
    }  
    try {  
        $pdo = new  
PDO('mysql:host=localhost;dbname=system_inform', 'root',  
'root');  
        $pdo->setAttribute(PDO::ATTR_ERRMODE,  
PDO::ERRMODE_EXCEPTION);  
  
        $sql = "SELECT * FROM user WHERE username = '$log'  
AND password = '$pass'";  
        $stmt = $pdo->query($sql);  
        $user = $stmt->fetch(PDO::FETCH_ASSOC);  
  
        if (!$user) {
```

```
$_SESSION['error'] = 'Неправильно введён логин  
или пароль.';  
header("Location: login.php");  
exit();  
}  
} catch (Exception $e) {  
    $_SESSION['error'] = 'Произошла ошибка во время  
входа в систему. Попробуйте позже.' . $e;  
    header("Location: login.php");  
    exit();  
}  
header("Location: index.php");  
exit();  
}
```

В этой функции можно выделить ряд недостатков, которые создают угрозу для внедрения SQL-инъекции:

1. Параметры логина и пароля подставляются в явном виде в запрос на выборку пользователя, что создаёт серьёзную уязвимость в данной функции.

2. Также пароли в БД хранятся в незашифрованном виде. Такой вывод можно сделать по запросу к БД на получение информации о пользователе (не используется функция `password_verify` или ей аналогичные, но вместо этого сравнивается введённое пользователем значение с ячейкой в таблице).

3. Лог об ошибках выводится на страницу, что тоже может создать уязвимость для дальнейшей инъекции, основанной на ошибках в запросе.

Обойти данную функцию авторизации будет легко даже стандартным методом внедрения SQLi. Достаточно при входе в систему указать вместо логина часть запроса: `' OR 1=1 --`. Пароль при этом может быть любым:

```
SELECT * FROM user WHERE username = '' OR 1=1 -- AND password  
= ' KmKwtCxRJw ';
```

Несмотря на то, что функция также и выводит информацию об ошибке на стороне сервера пользователю, это будет в ряде случаев бесполезно (зависит от характера ошибки), так как PDO в PHP не выводит точный лог об ошибке помимо сообщения о том, что возникла ошибка в запросе.

Следующий SQL запрос, не трудно заметить, будет завершён с ошибкой, что может дать возможность атакующему детальнее изучить SQL-запрос (см. рис. 4):

```
SELECT * FROM user WHERE username = '' OR 1=CAST("test" AS  
INT) -- AND password = 'pass';
```

Для атакующего можно выделить следующие наиболее полезные ошибки при формировании SQL-инъекции:

1. Ошибка синтаксиса (раскрывает часть синтаксиса и указывает на конкретное место возникновения ошибки).

2. Ошибки преобразования типов данных (раскрывает тип данных, который ожидался, и значение, которое не удалось преобразовать, а также может содержать имена таблиц, столбцов или даже данные).

3. Ошибки доступа к таблицам или столбцам (раскрывает имена таблиц и столбцов, что помогает атакующему понять структуру базы данных).

4. Ошибки, связанные с ограничениями (раскрывает значения, которые уже существуют в базе данных).

5. Ошибки, связанные с правами доступа (раскрывает имена пользователей, хостов и баз данных).

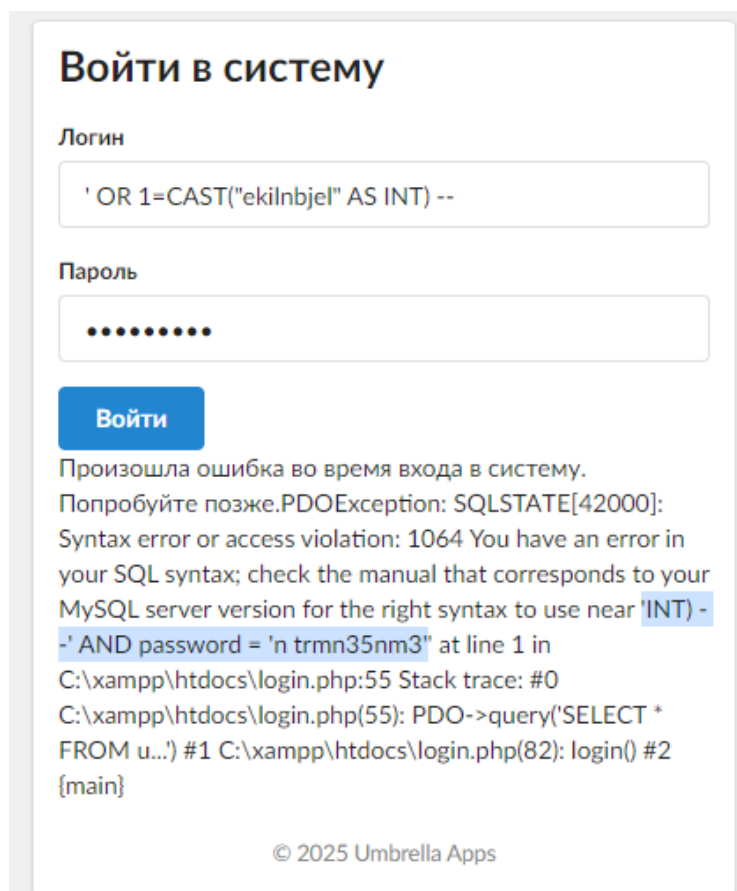


Рисунок 4. Более детальный лог об ошибке, сообщающий о месте возникновения ошибки в запросе

Слепые SQL-инъекции также имеют место быть в данном примере, но только для получения данных о логинах пользователей, так как пароли на сервере хранятся в зашифрованном виде.

Для защиты от SQL-инъекций можно использовать различные ORM (Object-Relational Mapping): на PHP это RedBeanPHP, Doctrine и другие. Большинство ORM заранее подготавливают запросы за счёт экранирования данных перед отправкой их на сервер, что исключает риск появления инъекций. Такие запросы ещё называются подготовленными (prepared statements)

В качестве решения проблемы функции, которая была представлена выше, логика входа в приложение была реализована с использованием ORM RedBeanPHP:

```
function login() {
    $log = $_POST['login'];
    $pass = $_POST['password'];
    if (empty($log) || empty($pass)) {
        $_SESSION['error'] = "Пожалуйста, заполните все поля.";
        header("Location: login.php");
        exit();
    }

    try {
        R::setup('mysql:host=localhost;dbname=system_inform',
            'root', 'root');
        $user = R::findOne('user', 'username = ?', [$log]);
        if (!$user || !password_verify($pass, $user->password))
        {
            $_SESSION['error'] = 'Неправильно введён логин или
            пароль.';
            header("Location: login.php");
        }
    }
    catch (Exception $e) {
        $_SESSION['error'] = 'Произошла ошибка во время входа в
        систему. Попробуйте позже.';
        header("Location: login.php");
        return;
    }
    header("Location: index.php");
}
```

В данном примере реализации функции входа в аккаунт были внесены следующие изменения:

1. Вместо стандартного PDO используется ORM RedBeanPHP, предоставляющий возможность экранирования данных в запросах. Если ранее запрос к БД был указан в явном виде, то теперь он задаётся в виде функции самой ORM, в которой уже предусмотрено преобразование данных перед отправкой:

```
$user = R::findOne('user', 'username = ?', [$log]);
```

2. Также пароли теперь задаются в зашифрованном виде и проверяются с помощью функции `password_verify`, что делает невозможным подбор пароля через слепые SQL-инъекции.

3. Вывод ошибок исключён и атакующий теперь не сможет узнать результат выполнения заранее неправильно составленного запроса.

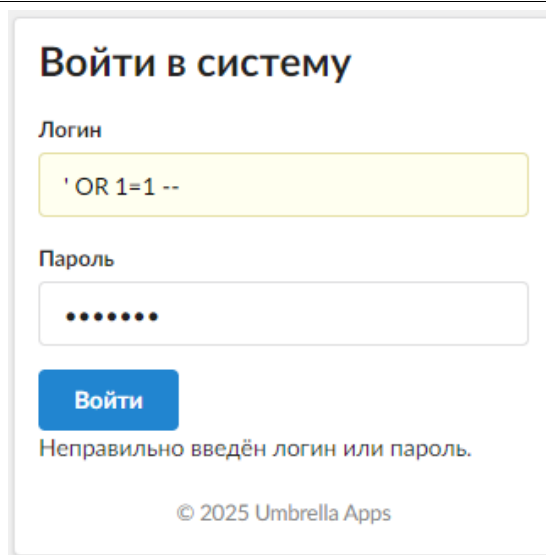


Рисунок 5. Результат внедрения защиты от SQL-инъекций в информационной системе

PHP также предоставляет возможность экранирования данных (предварительной подготовки запросов). Для подготовки запросов можно использовать функцию `prepare`, которая есть как в `mysqli`, так и в `PDO`.

Можно выделить следующие способы защиты от SQL-инъекций:

1. Использование подготовленных выражений (Prepared Statements).

Подготовленные выражения (Prepared Statements) – это механизм, при котором SQL-запрос и данные передаются отдельно. Это предотвращает возможность внедрения SQL-кода. В случае использования таких выражения SQL-запрос компилируется один раз и затем выполняется с различными параметрами. Гарантируется, что параметры передаются отдельно и автоматически экранируются.

2. Использование ORM (Object-Relational Mapping).

ORM-библиотеки (например, `SQLAlchemy` для Python, `Doctrine` для PHP) автоматически генерируют SQL-запросы и экранируют данные.

3. Валидация входных данных.

С помощью регулярных выражений или других механизмов можно проверять входные данные и при несоответствии данных указанному шаблону или правилам стоит возвращать ошибку клиенту. Также стоит явным образом приводить к тому типу данных, в котором предполагается хранение информации для последующего формирования запроса (например, если клиент хранит информацию об ID пользователя, то перед отправкой этого параметра нужно на стороне клиента преобразовать это значение в целочисленный тип данных [если в БД используются целочисленные первичные ключи]).

4. Ограничение прав доступа к базе данных.

Выше были рассмотрены примеры инъекций для получения доступа к ИС, но атакующий также может попытаться очистить таблицы или удалить базу данных. Это будет также легко, как и войти в систему, если ИС не

защищена. Для решения такой проблемы пользователь, который выполняет запросы в БД, должен быть ограничен в правах доступа. Все современные СУБД предоставляют механизмы регулирования прав доступа к БД и информации в ней.

5. Отключение вывода ошибок на production-серверах.

Отображение ошибок может создать риск возникновения уязвимости для создания Error-based SQL-инъекции. Такие инъекции можно исключить отключением вывода ошибок на экран пользователю различными методами (обычно это делается либо на стороне кода ИС, либо, если ИС представлена в виде сайта, в .htaccess).

SQL-инъекции можно внедрить в ИС на различных языках программирования. Например, в ИС, написанной на Python.

В Python для работы с MySQL можно использовать модуль `mysql.connector`, предоставляющий функционал по формированию и выполнению запросов к выбранной БД.

Ниже представлен пример SQL-запроса, в который также можно внедрить инъекцию и зайти в систему:

```
query = f"SELECT * FROM user WHERE username = '{username}'  
AND password = '{password}'"  
cursor.execute(query)
```

Для решения данной проблемы запрос также можно подготовить:

```
query = "SELECT * FROM user WHERE username = %s AND password  
= %s"  
cursor.execute(query, (username, password))
```

Также рекомендуется использовать различные ORM, такие как SQLAlchemy для автоматической подготовки запросов, экранирования данных и обеспечения безопасности при работе с базой данных.

4 Выводы

В данной работе были рассмотрены SQL-инъекции в контексте информационной безопасности как методы атаки на различные информационные системы (преимущественно веб-ориентированные). Были определены основные виды SQL-инъекций, их особенности и недостатки, а также базовые реализации на языке программирования PHP.

Также был продемонстрирован пример внедрения SQL-инъекций в запрос проверки данных пользователя (логин и пароль) на этапе авторизации в веб-ориентированной ИС. Код функции был представлен с ошибками, которые в дальнейшем были исправлены.

Также в данной работе были описаны способы защиты от SQL-инъекций, а именно: использование подготовленных выражений и ORM,

валидация входных данных, ограничение прав доступа к БД и отключение вывода ошибок на production-серверах для клиентской стороны.

Библиографический список

1. Починяев А.К. SQL-инъекции и способы защиты // Аспирант и соискатель. 2020. № 6 (120). С. 42-45.
2. Казанюк Г.М., Воронин В.В. Содержательный анализ технологии SQL-инъекций // В сборнике: Информационные технологии XXI века. Сборник научных трудов. Редакционная коллегия: ответственный редактор В.В. Воронин [и др.]. Хабаровск, 2020. С. 442-445.
3. Методика обнаружения атак, типа SQL-инъекция на основе алгоритмов искусственного интеллекта, методом задачи классификации / В.М. Митрофанов, М.А. Крибель, С.А. Фроленков, Л.О. Спицын // Научные технологии в космических исследованиях Земли. 2021. Т.13. № 5. С. 96-102.
4. Беликов Г.В., Крылов И.Д., Селищев В.А. SQL-инъекция как способ обхода авторизации // Известия Тульского государственного университета. Технические науки. 2021. № 12. С. 217-221.
5. Якушев Д.И., Вайберт Н.А. Методы защиты информации от SQL-инъекций // В сборнике: Информационная безопасность регионов России (ИБРР-2021). Материалы XII Санкт-Петербургской межрегиональной конференции. СПб., 2021. С. 65-66.