НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

«До захисту допущено»
В.о. завідувача кафедри

_____ М.В.Грайворонський
(підпис)
“___”_____2019 р.

Дипломна робота

на здобуття ступеня бакалавра

з напряму підготовки 6.170101 «Безпека інформаційних і комунікаційних систем»

на тему: Проксуючий фаєрвол для захисту від SQL ін’єкцій

Виконав (-ла): студент (-ка) _4_ курсу, групи ___ФБ-51_____
(шифр групи)

_____Трифонов Іван_____
(прізвище, ім’я, по батькові) (підпис)

Керівник: доцент кафедри ІБ, к.т.н. Барановський О.М._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант _____ _____ _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент_____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ - 2019 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

Рівень вищої освіти – перший (бакалаврський)

Напрям підготовки 6.170101 «Безпека інформаційних і комунікаційних систем»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ М.В.Грайворонський
(підпис)
«___»_____2019 р.

ЗАВДАННЯ
на дипломну роботу студенту

_____ Ратенок Денис _____
(прізвище, ім'я, по батькові)

1. Тема роботи Безпека Open Source наприкладі пакетного менеджера NPM _____ ,
науковий керівник роботи доцент к.т.н. Литвинова Т.В. _____
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «____ » 2019 р. № _____

2. Термін подання студентом роботи 10 червня 2019 р.

3. Вихідні дані до роботи _____
_____
_____

4. Зміст роботи_____
_____
_____
_____
_____

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо) _____
_____
_____
_____

6. Дата видачі завдання _____

РЕФЕРАТ

Робота обсягом 89 сторінок, містить 13 ілюстрацій, 9 таблиць та 10 літературних посилань.

Метою даної роботи є написання програмного засобу (фаєрвола), призначеного для захисту СКБД від такого типу атак, як SQL-ін'єкції. Фаєрвол повинен бути універсальним, а саме працювати з СКБД: MsSQL Server, MySQL, MariaDB, PostgreSQL. Фаєрвол повинен забезпечувати захист від атак типу впровадження SQL-коду, мати можливість аналізу пакетів, виявлення ін'єкцій і блокування небезпечних запитів.

Об’єктом роботи є модель фаєрволу для захисту від SQL ін’єкцій.

Головними вимогами до системи є універсальність, масштабованість, розширюваність предметної області та здатність працювати у реальному часі. Робота включає аналіз результатів практичного дослідження.

Ключові слова: SQLi, WAF, СКБД, проксі-сервер.

ABSTRACT

Thesis consists of 89 pages, contains 13 illustrations, 9 tables and 10 literary references.

The purpose of thesis is to create software (firewall) to secure databases from SQL injections. Firewall should be universal, should work with MSSQL Server, MySQL, MariaDB, PostgreSQL DBMS. Firewall should secure from attacks that insert SQL code, also it should be able to analyze packets, detect injections and block malicious queries.

The object of the research is the firewall model for SQL injection prevention.

The main requirements of the system are versatility, scalability, the expansion of substantive zone and the ability to functionate in real time. Thesis includes result analysis, obtained from practical research.

Keywords: SQLi, WAF, DBMS, proxy-server.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

SQL	structured query language
WAF	web application firewall
XSS	cross-site scripting
DDoS	distributed denial of service
ODBC	open database connectivity
JDBC	java database connectivity
HPP	http parameter pollution
HPF	http parameter fragmentation
CPD	cross domain policies
FFM	fast flux monitor
SDK	software development kit
CDN	content delivery network

ВСТУП

Ін'єкційні атаки відносяться до широкого класу векторів атак. Під час ін'єкційної атаки зловмисник постачає довільний код до програми. Цей код обробляється інтерпретатором як частина команди або запиту, що у свою чергу, може призвести до зміни у логіці роботи програми.

Ін'єкції є одними з найдавніших і найнебезпечніших атак, спрямованих на веб-застосунки. Вони можуть призвести до крадіжки даних, втрати даних, втрати цілісності даних, відмови в обслуговуванні, а також повної компрометації системи. Основною причиною для впровадження ін'єкцій, як правило, є недостатня перевірка вводу користувача. Зазвичай атаки типу SQL ін'єкції розглядають щодо web-додатків, однак до даної уразливості схильні будь-які клієнт-серверні та серверні програми, що працюють з системами управління базами даних.

SQL-ін'єкції (SQLi) – один із найпоширеніших типів атак даного класу. Незважаючи на те, що перші змістовні згадки датуються 1998 роком, даний тип ефективно використовує вразливості сучасних веб-застосунків. Зміст атаки полягає у використанні вразливостей веб-застосунку, котрий формує SQL-запит із користувацького вводу. Вдала SQL-ін’єкція може призвести до отримання доступу до конфіденційної інформації неавторизованими користувачами, зміни даних у відповідних базах, виконання операцій привілейованих користувачів (таких як зупинка роботи СКБД (системи керування базою даних)), відновлення застарілого змісту файлу в файловій системі СКБД.

Наслідки вдалої ін’єкції щодо інформації:

1) зазвичай бази даних містять інформацію, котра потребує захисту, тому втрата її конфіденційності є дуже поширеною проблемою при наявності вразливостей до ін’єкцій;

2) у разі використання некоректних алгоритмів перевірки вводу користувача, а саме, логіну та паролю, постає проблема автентифікації стороннього користувача, котрий попередньо не мав інформації щодо доступу;

3) у разі зберігання інформації щодо авторизації користувачів у базі даних, постає проблема можливої зміни цієї інформації потенційним зловмисником при успішній ін'єкції;

4) потенційне порушення цілісності даних у випадку доступу до їх зчитування, модифікації даних або їх видалення.

Актуальність

На даний час великі організації все більше використовують різноманітні додатки - системи банківського обслуговування, офіційні сайти компаній, портали державних послуг, електронні торгові майданчики, системи управління ресурсами підприємств і так далі. Всілякі спеціалізовані додатки на основі клієнтського програмного забезпечення (ПО) повсюдно замінюються web-версіями і хмарними сервісами.

Згідно статистики відкритого проекту забезпечення безпеки веб-застосунків OWASP - ін'єкції є небезпекою номер 1 у даній сфері - і це поважна причина. Крім того, ін'єкції – добре вивчений клас атак, а це означає, що існують безкоштовні та надійні засоби, котрі дозволяють навіть недосвіченому зловмиснику впроваджувати даний тип атак.

Згідно звіту команди із дослідження загроз веб-застосункам Akamai за 2018 рік, Akamai`s Kona Site Defender web-application firewall (WAF) відслідкував 8.5 мільйонів SQL-ін'єкцій на 2 тисячі унікальних веб-застосунків лише за 7 днів.

За перший квартал цього ж року було проведено близько 92 мільйонів ін'єкцій, що склало 46% від загальної кількості атак на веб-застосунки.

Найбільша кількість атак прийшлась на веб-застосунки, пов’язані із фінансовою діяльністю, понад ніж у 2,5 рази більше атак, в порівнянні з іншими. Відсоток атак на http ресурси – 91,5%, https – 8,5%

Завдання

Завданнями даної роботи є:

- вивчення особливостей впровадження операторів SQL (SQL-ін'єкції);
- дослідження методів виявлення аномалій в SQL-запитах до баз даних;
- вивчення методів захисту від такого даного атак;
- дослідження особливостей операторів в таких популярних СКБД як: MsSQL Server, MySQL, MariaDB, PostgreSQL.

Метою даної роботи є написання програмного засобу (фаєрвола), призначеного для захисту СКБД від такого типу атак, як SQL-ін'єкції. Фаєрвол повинен бути універсальним, а саме працювати з СКБД: MsSQL Server, MySQL, MariaDB, PostgreSQL. Фаєрвол повинен забезпечувати захист від атак типу впровадження SQL-коду, мати можливість аналізу пакетів, виявлення ін'єкцій і блокування небезпечних запитів.

1 АНАЛІЗ ПРОБЛЕМИ SQL-ІНЄКЦІЙ В ВЕБ-ЗАСТОСУНКАХ

В даному розділі будуть розглянуті основні типи SQL ін’єкцій, їх особливості реалізації щодо різних СКБД, порівняння наявних коммерційних фаєрволів, а також сучасні методи обходу WAF.

1.1 Типи SQL ін’єкцій

Впровадження операторів SQL (SQL injection) - це різновид атак, спрямованих на несанкціоноване отримання, редагування та видалення інформації з системи управління базами даних, а також, в деяких випадках, на придбання повного контролю як над сервером СКБД, так і над його операційною системою. Ін'єкції проводяться через web-сервери, що конструюють запити до серверів СКБД на основі інформації, введеної користувачем. Якщо інформація, яка приходить від клієнта погано фільтрується, то зловмисник може змінити запит до сервера СКБД, що відправляється додатком. При цьому даний запит буде виконуватися з тим же рівнем привілеїв, що і web-додаток.

У більшості випадків здійснити SQL-ін'єкцію можна тільки тоді, коли при спробі впровадження операторів SQL сервер бази даних відповідає помилками на некоректно складений запит. Дані помилки можна назвати "помилками першого роду". Проте, повідомлення про помилки бази даних не завжди очевидні. Однак розробники дуже часто помиляються, тому для доказу успішності SQL-ін'єкції зловмисники шукають помилки у всіх можливих місцях. Перше, що робить атакуючий - шукає в повідомленні про помилку фрази на кшталт "ODBC", "SQL Server", "Syntax" і так далі.

Але буває так, що розробники приховують повідомлення про помилки від сервера бази даних. Замість них кодом програми генеруються власні

помилки, назвемо їх "помилками другого роду", що говорить про кращому стилі програмування. Більш детальна інформація про характер помилки може знаходитися в, так званому, прихованому Input (приховані поля html-сторінки), коментарях і так далі. Бувають такі веб-додатки, які видають повідомлення про помилку, не маючи абсолютно ніякої інформації про неї в тілі http-відповіді, однак така інформація міститься в заголовку. Взагалі в багатьох веб-додатках є такі функції, вбудовані в них для налагодження і тестування, але їх забувають відключити або видалити під час релізу.

Описаними вище помилками успішно користуються зловмисники, використовуючи прийоми і методи, що розглядаються далі.

Зміна вхідних параметрів шляхом додавання в них конструкцій мови SQL

В даному випадку мова йде про впровадження SQL-коду в запити, які використовують параметр id, тобто числовий тип, в якості вхідного параметра. Наприклад, розглянемо таку url адресу:

```
http: //localhost/test_1? book_id = 1
```

Можна припустити, що буде згенеровано такий запит до СКБД:

```
"SELECT * FROM Books WHERE Book_ID = id"
```

В даному запиті Books - деяка таблиця книг, Book_ID - унікальний номер книги в таблиці Books, id - параметр, що отримується сервером. Очевидно, що якщо на сервер переданий параметр id, що дорівнює, наприклад, 80, то виконається наступний SQL-запит:

```
"SELECT * FROM Books WHERE Book_ID = 100"
```

Однак якщо передати на сервер в якості параметра id рядок "-1 OR 1 = 1", то вийде наступний SQL-запит:

```
"SELECT * FROM Books WHERE Book_ID = -1 OR 1 = 1"
```

В результаті виконання цього SQL-запиту сервер виведе всі наявні в базі книги, так як Book_ID = -1 не вірно, за замовчуванням, умова, а умова 1 = 1 - вірно завжди. Підводячи підсумок, стає очевидним, що зміна вхідних параметрів за допомогою SQL-ін'єкцій змінює логіку всього SQL-запиту.

SQL ін’єкція в строкові параметри

Даний метод SQL-ін'єкцій схожий на попередній. Перше, що можна розглянути в даній ін'єкції - це обхід авторизації. Припустимо, що код веб-додатки виглядає так:

```
SQLQuery = "SELECT Username FROM Users WHERE Username = '" &
strUsername & " 'AND Password ='" & strPassword & " '"
strAuthCheck = GetQueryResult (SQLQuery)
If strAuthCheck = "" Then
 boolAuthenticated = False
Else
 boolAuthenticated = True
End If
```

Отже, ось що відбувається, коли користувач відправляє логін і пароль. Запит буде йти через таблицю users, щоб перевірити, чи збігаються надані користувачем дані з даними, що містяться в цій таблиці. Якщо такі дані знайдені, то ім'я користувача буде зберігатися у змінній strAuthCheck, яка вказує, що користувач повинен бути аутентифікований. Якщо таких даних немає, то змінна strAuthCheck буде порожній, і користувач не зможе пройти перевірку справжності.

Іншим випадком є впровадження коду в оператор LIKE. Припустимо, у нас є запит, в результаті виконання якого, користувач отримує вибірку статей. Стаття потрапляє до вибірки, якщо в її назві є необхідна слово. Тоді SQL-запит буде мати вигляд:

```
"SELECT Book_ID, Book _Date, Book _Caption, Book _Text,
Book _ID_Author FROM Books WHERE Book _Caption
LIKE ( '% Search_Text%') "
```

В даному запиті Books - деяка таблиця статей, Book_ID - унікальний номер книги, Book_Date - дата публікації книги, Book_Caption - опис книги, Book_Text - текст книги, Book_ID_Author - унікальний номер автора книги в таблиці Books, Search_Text - переданий параметр.

Використання UNION

Більшість веб-додатків використовують динамічний контент, витягнутий з бази за допомогою даного оператора, для побудови сторінок. У більшості випадків частина запиту, з якою проводяться маніпуляції, знаходиться в умови оператора WHERE. Є можливість змінити запит так, щоб він повертав записи, не призначені даними запитом. Робиться це за допомогою оператора UNION, який дозволяє використовувати кілька операторів SELECT в одному запиті. Це виглядає приблизно так:

```
"SELECT Company FROM Shippers WHERE 1 = 1 UNION ALL SELECT Company
FROM Customers WHERE 1 = 1"
```

Цей запит поверне записи для першого і другого запиту разом. Оператор ALL необхідний для того, щоб уникнути SELECT DISTINCT, тобто виведення тільки одного запиту.

Як можна помітити, цей запит майже ідентичний запиту, описаному вище, однак, в даному випадку параметр City укладений в круглі дужки, тому рядок ін'єкції також повинна їх утримувати. Змінимо попередню ін'єкцію наступним чином:

```
" ') UNION SELECT OtherField FROM OtherTable WHERE (' '='"
```

Таким чином, на сервер відправиться наступний запит, який синтаксично буде правильним:

```
"SELECT Name, Surname, Title, FROM Employees WHERE (City = '')
UNION SELECT OtherField From OtherTable WHERE ( '' = '')"
```

Екранування хвоста запиту

Цей тип атак з використанням SQL-коду можна зустріти в разі, якщо запит має складну структуру, до якої досить складно або зовсім неможливо застосувати UNION. Наприклад, розглянемо запит:

```
"SELECT Author FROM Journals WHERE Author_ID = id AND

Author_Name LIKE ( 'J%') "
```

В даному запиті Journals деяка таблиця журналів, Author_ID - унікальний номер учасника журналу, Author_Name - ім'я автора журналу, id - переданий параметр. Сервер створює вибірку імен авторів по переданому параметру id, тільки в тому випадку, якщо ім'я автора починається з букви "J". В даному випадку зловмисникові досить складно впровадити в запит UNION, тому використовується екранування запиту. А саме передається таке значення параметра -1 UNION SELECT usr_name, password FROM admin / *.

Тоді запит набуває такого вигляду:

```
"SELECT Author FROM Journals WHERE Author_ID = -1 UNION

SELECT usr_name, password FROM admin / * AND Author_Name
```

```
LIKE ( 'J%') "
```

Результатом виконання цього запиту буде вибірка імен користувачів і паролів з таблиці admin, так як автора з унікальним номером -1 швидше за все не існує, а частина запиту, що обмежує вибірку по імені автора, зловмисник закоментувавши. Таким чином частина запиту AND Author_Name LIKE ('J%') не впливає на виконання. Залежно від типу СКБД екранування запиту здійснюється різними символами коментування.

Розщеплення SQL запиту

Даний тип атаки на увазі виконання декількох SQL-запитів, замість запланованого одного. Для цього зловмисник використовує символ «;» (крапка з комою). Однак не всі версії SQL підтримують таку можливість. Припустимо у нас є запит:

```
"SELECT * FROM Journals WHERE Journal_ID = id"
```

В даному запиті Journals - деяка таблиця з журналами, Journal_ID - унікальний номер журналу в таблиці Journals, id - параметр, що отримується сервером. Можна спробувати замість звичайного значення параметра передати, наприклад, таке значення:

```
"80; INSERT INTO admin (usr_name, password)
VALUES ( 'AdmName', '1234') "
Тоді запит матиме такий вигляд:
"SELECT * FROM Journals WHERE Journal_ID = 80; INSERT INTO
admin (usr_name, password) VALUES ( 'AdmName', '4320') "
 В цьому випадку в одному запиті будуть виконані 2 команди:
"SELECT * FROM Journals WHERE Journal_ID = 80"
"INSERT INTO admin (usr_name, password) VALUES
```

```
( 'AdmName', '4320') "
```

У підсумку в таблицю admin буде додано запис про нового адміністратора.

1.2 Особливості SQL ін’єкцій для різних СКБД (PostgreSQL, MySQL, MsSQL Server, MariaDB)

Звичайно, всі розглянуті СКБД управляються за допомогою мови SQL. Однак вони мають різні діалекти. Внаслідок цього, то, що справедливо для однієї СКБД може не працювати для іншої. У таблиці 1.1 наведені подібності та відмінності розглянутих СКБД.

Таблиця 1.1 - Схожість і відмінності СКБД

	PostgreSQL	MSSQL	MySQL	MariaDB
Коментарі	i / *	i / *	i / ** / i #	i / ** / i #
об'єднання запитів	union	union	union для v.3>	union
розщеплення запитів	;	;	немає	немає

У таблиці зображені тільки ті оператори, які можуть мати цінність при проведенні атаки типу впровадження SQL коду. Як ми можемо бачити, додавання коментаря не скрізь однакове. Так, наприклад, в СКБД PostgreSQL і MSSQL вони ідентичні, але в MySQL і MariaDB є додаткові способи коментування.

Об'єднання запитів за допомогою оператора UNION присутнє скрізь, проте, в MySQL версії 3 і нижче такий оператор був відсутній, внаслідок чого неможливо було провести SQL-ін'єкцію.

Розщеплення запиту можна зробити тільки в СКБД PostgreSQL і MSSQL. В інших розглянутих СКБД така можливість відсутня.

1.3 Аналіз існуючих контролів захисту

Якщо ж аналізувати імплементацію фаєрволу як окремої стистеми протидії – слід розглянути вже існуючі контролі захисту. За допомогою цього ми зможемо зробити вірні висновки про те, як повинен функціонувати фаєрвол для баз даних на основі переваг і недоліків розглянутих рішень.

Різні СКБД мають різні клієнт-серверні протоколи і діалекти мови SQL, тому використання одного фаєрвола для різних баз даних не представляється можливим.

Існуюючі фаєрволи для баз даних, такі як: MySQL Enterprise Firewall, DataArmor, Oracle Firewall і т.д. Розглянемо порівняльну таблицю 1.2 даних типів захисту:

Таблиця 1.2 - Порівняльна таблиця фаєрволів для баз даних

	MySQL Enterprise Firewall	Oracle Firewall	Data Armor
прослуховування трафіку	присутнє	присутнє	присутнє
Блокування підозрілих запитів	присутнє	присутнє	присутнє
Навантаження на базу даних	є	немає	немає

Продовження таблиці 1.2 - Порівняльна таблиця фаєрволів для баз даних

Підтримка декількох баз даних	немає	За умови наявності додаткового пакета	За умови наявності додаткового пакета
вартість	$ 2000 - $ 6000.	$ 6000. + підтримка $ 1000 в рік	За додаткову плату, але вартість дізнатися не вдалося

Як видно з таблиці всі представлені засоби захисту баз даних мають:

- прослуховування трафіку, тобто збереження всіх, хто йшов запитів в лог файл. В даному режимі не відбувається аналіз SQL-запитів.
- Блокування підозрілих запитів. В даному режимі аналізуються всі запити. Безпечні записуються в лог файл і пропускаються далі на сервер БД. Запити, які містять ін'єкції також пишуться в лог, позначаються як підозрілі і блокуються.

Однак не всі працюють незалежно від бази даних. Так, наприклад, MySQL Enterprise Firewall вбудовується в кожен екземпляр СУБД MySQL. У цьому випадку стає ясно, що база даних все-одно приймає SQL-запит, передає його фаєрвол для аналізу і тільки після цього приймає рішення на видачу результатів користувачеві. На дані дії витрачаються ресурси бази даних.

Наступний критерій - підтримка декількох баз даних. В MySQL Enterprise Firewall така можливість відсутня. Для двох інших фаєрволів варто розглядати цей критерій з вартістю цих продуктів. Вся справа в тому, що

підтримка кожної нової СУБД потребують покупці додаткових пакетів, придатних для них.

Отже, як ми бачимо, кожне з цих коштів має свої недоліки, такі як додаткове навантаження на базу даних, неможливість застосування для інших баз даних і вартість придбання та обслуговування.

1.4 Сучасні технології обходу WAF

WAF, як правило, розгортаються в певному вигляді проксі-сервера лише перед веб-додатками, тому вони не бачать весь трафік в наших мережах. Відстежуючи трафік до того, як він досягне веб-додатку, WAF можуть аналізувати запити, перш ніж передати їх. Саме це дає їм таку перевагу перед IPS. Оскільки IPS призначені для опитування всього мережевого трафіку, вони не можуть аналізувати прикладний рівень так само ретельно, як WAF.

Проте WAF недосконалі, тому слід розглянути методи, котрі дозволяють нам реалізувати обхід фаєрволів.

Обхід WAF: SQL Injection - Метод нормалізації

Приклад Номер (1) уразливості у функції запиту Normalization.

Наступний запит не дозволяє нікому проводити атаку

```
/? id = 1 + union + select + 1,2,3 / *
```

Якщо у WAF існує відповідна уразливість, цей запит буде успішно виконано

```
 /? id = 1 / * union * / union / * select * / select + 1,2,3 / *
```

Після обробки WAF запит стане

```
index.php? id = 1 / * uni x на * / union / * sel x ect * / select
+ 1,2,3 / *
```

Даний приклад працює в разі очищення небезпечного трафіку, а не у разі блокування всього запиту або джерела атаки.

Використання забруднення параметрів HTTP (HPP)

Наступний запит не дозволяє нікому проводити атаку

```
/? id = 1, виберіть + 1,2,3 + від + користувачів + де + id = 1--
```

Цей запит буде успішно виконано з використанням ГЕС

```
/? id = 1; виберіть + 1 & id = 2,3 + від + користувачів + де +
id = 1--
```

Успішне проведення атаки HPP в обхід WAF залежить від середовища нападу програми.

Використання забруднення параметрів HTTP (HPP)

Уразливий код:

```
SQL = "вибір ключа з таблиці де id =" + Request.QueryString
("id")
```

Цей запит успішно виконаний з використанням техніки ГЕС

```
/? id = 1 / ** / union / * & id = * / select / * & id = * / pwd
/ * & id = * / from / * & id = * / users
```

Запит SQL стає ключем вибору з таблиці, де

```
id = 1 / ** / union / *, * / select / *, * / pwd / *, * / від
користувачів / *, * / користувачів
```

ByPassing WAF: SQL Injection - HPF за допомогою фрагментації параметрів HTTP (HPF)

Приклад з вразливим кодом:

```
Запит ("виберіть * з таблиці, де a =". $ _ GET ['a']. "I b =". $ _ GET ['b']);

Запит ("виберіть * з таблиці, де a =". $ _ GET ['a']. "I b =". $ _ GET ['b']. "Limit". $ _ GET ['c']);
```

Наступний запит не дозволяє нікому проводити атаку

```
/? a = 1 + union + select + 1,2 / *
```

Ці запити можуть бути успішно виконані за допомогою HPF

```
/? a = 1 + union / * & b = * / select + 1,2

/? a = 1 + union / * & b = * / select + 1, pass / * & c = * / від + users--
```

Запити SQL стають

```
виберіть * з таблиці, де a = 1 union / * i b = * / select 1,2

виберіть * з таблиці, де a = 1 union / * i b = * / select 1, pass / * limit * / від користувачів.
```

Обхід WAF: сліпий ввід SQL з використанням логічних запитів AND / OR

Наступні запити дозволяють здійснити успішну атаку для багатьох WAF

```
/? id = 1 + АБО + 0x50 = 0x50

/? id = 1 + i + ascii (нижній (середній ((виберіть + pwd + від + користувачів + ліміт + 1,1), 1,1))) = 74
```

Знаки заперечення та нерівності (! =, <>, <,>) Можна використовувати замість рівня рівності - це дивно, але багато WAF пропускають це!

Стає можливим скористатися вразливістю методом сліпого SQL Injection шляхом заміни функцій SQL, які потрапляють до підписів WAF з їх синонімами.

substring () -> mid (), substr ()

ascii () -> hex (),

бенчмарк bin () () -> sleep ()

Широкий вибір логічних запитів.

i 1

або 1

i 1 = 1

i 2 <3

i 'a' = 'a'

i 'a' <> 'b'

i char (32) = ''

i 3 <= 2

i 5 <=> 4

i 5 <=> 5

i 5 є нульовими

або 5 не є нульовими

....

Відомо:

substring ((виберіть 'password'), 1,1) = 0x70

substr ((виберіть 'пароль'), 1,1) = 0x70

mid ((виберіть 'password'), 1,1) = 0x70

Нове:

strcmp (ліворуч ('password', 1), 0x69) = 1

strcmp (ліворуч ('password', 1), 0x70) = 0

strcmp (ліворуч ('password', 1), 0x71) = -1

STRCMP (expr1, expr2) повертає 0, якщо рядки однакові, -1, якщо перший, аргумент менше, ніж другий, і 1 інакше.

Обхід підпису WAF

Наступний запит надходить до підпису WAF

```
/? id = 1 + union + (виберіть + 1,2 + від + користувачів)
```

Але іноді використовувані сигнатури можна обійти

```
/? id = 1 + union + (виберіть + 'xz'from + xxx)

/? id = (1) союз (виберіть (1), середину (хеш, 1,32) від
(користувачів))

/? id = 1 + union + (select'1 ', concat (логін, hash) від +
користувачів)

/? id = (1) union ((((((виберіть (1), hex (хеш) від
(користувачів)))))))

/? id = (1) або (0x50 = 0x50)
```

Висновки до розділу 1

В даному розділі були розглянуті основні типи SQL ін’єкцій, їх особливості реалізації щодо різних СКБД, порівняння наявних коммерційних фаєрволів, а також сучасні методи обходу WAF.

В процесі аналізу існуючих контролів захисту та методів їх обходу було виявлено, що наявні методи реалізації фаєрволів – недосконалі, звідки випливає актуальна проблема пошуку методів захисту від існуючих вразливостей.

2 МЕХАНІЗМИ ВИЯВЛЕННЯ ТА ЗАПОБІГАННЯ SQL ІН’ЄКЦІЯМ

В даному розділі будуть розглянуті та проаналізовані додаткові та експериментальні методи виявлення SQL ін’єкцій, створені порівняльні таблиці щодо методів та засобів виявлення та запобігання SQL ін’єкціям, а також проаналізовані технології проксі http-запитів.

2.1 Аналіз додаткових методів виявлення SQL ін’єкцій

Для того щоб уникнути атак, пов'язаних з впровадженням зловмисних SQL-кодів в запити, розробники прагнуть створити додаток, який буде працювати виключно з параметризованих запитами, вони намагаються обмежити додатком доступ до даних сервера за допомогою збережених процедур, тобто розробити додаток, що не має можливості створювати власні SQL команди. При використанні збережених процедур програма має дозволи, необхідні тільки для виконання цих процедур, але немає доступу до базових таблиць. Тому, навіть якщо код є схильним до SQL-ін'єкцій, у зловмисників не вийде отримати доступ до даних, так як у програми не вистачає прав для доступу і маніпулювання таблицями. Крім цього збережені процедури мають вбудовані функції для перевірки типу переданих додатком параметрів.

Однак в реальному житті створення подібних додатків важко піддається реалізації, особливо при роботі з системами, які були створені раніше. Складнощі виникають і при реалізації програми в змішаних середовищах, а також, якщо команда розробників не може домовитися про найбільш вигідному вирішенні між собою. У зв'язку з цим існує набір правил, яких слід дотримуватися розробникам при створенні компонентів баз даних для підтримки різних типів додатків. Незважаючи на те, що не кожне з цих правил може бути застосовано для кожної програми, все-таки слід придивитися до

всіх вимог. Розглянемо докладніше ці методи боротьби з впровадженням SQL-коду.

Фільтрація строкових параметрів

Розглянемо приклад коду (на мові Паскаль), що генерує SQL-запит:

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

В даному випадку, щоб зробити SQL-ін'єкції неможливими, потрібно виконати процедуру заміни символів, а саме екранувати спецсимволи. При виконанні цієї процедури в параметрі замінюють:

- "(Лапки) на \"
- '(Апостроф) на \'
- \ (Зворотну косу риску) на \\

Або можна відкинути спецзнаки наступним чином:

```
$ String = preg_replace ( "/ [^ a-zA-Z0-9] / i", "", $ _POST [ 'string']);
```

Фільтрація цілочисельних параметрів

Розглянемо приклад коду, що генерує інший SQL-запит:

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

У ситуації, коли передається параметр, що має числовий тип, заміна спецсимволов і переміщення параметру в лапки не допоможе, тому що найчастіше числові параметри передаються без лапок. Тому в таких випадках розробники можуть перевірити тип параметра, а саме, уточнити чи є параметр

числом. Тоді, якщо параметр не число, то запит не повинен виконуватися зовсім.

У багатьох мовах програмування існують стандартні функції для конвертації рядок з цілим значенням в цілочисельне значення.

Застосуємо процедуру перевірки параметра і розглянемо виконання SQL-запиту, поданого вище:

```
id_int: = StrToInt (id);

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

Якщо параметр id не є цілочисельним, то виникне помилка, і функція StrToInt викличе виключення EConvertError. У обробнику цієї виключення можна буде задати висновок повідомлення про помилку. Завдяки подвійному перетворенню, реакція на числа в шістнадцятковій системі числення буде коректною.

Усічення вхідних параметрів

Одним з видів захисту може стати скорочення довжини рядка для вхідних параметрів запиту. З прикладів, розібраних вище видно, що для порушення логіки запиту необхідно ввести досить довгих рядків, мінімальна довжина впроваджуваної рядки в наших прикладах становить 8 символів (1 OR 1 = 1). Тому, якщо заздалегідь відомо, що довжина переданого параметра не може перевищувати якогось певного значення, то можна обрізати рядок за допомогою спеціальних функцій. Як приклад розглянемо запит з переданим параметром id. Якщо відомо, що його значення не може перевищувати 999, то можна відсікти вхідний рядок до 3-х символів, тоді отримаємо:

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

В даному випадку LeftStr - процедура усічення переданого рядка.

Використання параметризованих запитів

Даний вид захисту може бути реалізований розробниками, завдяки можливості відправки параметризованих запитів серверами баз даних. В даному випадку передаються зовнішні параметри на сервер окремо від SQL-запиту або автоматично екрануються клієнтської бібліотекою.

Розглянемо назви реалізацій даної функції захисту для різних мов програмування:

- на Delphi - властивість TQuery.Params;
- на Perl - через DBI :: quote або DBI :: prepare;
- на Java - через клас PreparedStatement;
- на C # - властивість SqlCommand.Parameters;
- на PHP - MySQLi (при роботі з MySQL), PDO.
- на Parser - мова сама запобігає атаці подібного роду.

Використання принципу найменших привілеїв при наданні доступу до баз даних

Цей спосіб захисту слід використовувати, навіть якщо ви допускаєте доступ до застосунку тільки через збережені процедури. Кожному обліковому запису бази даних повинні бути призначені мінімальні права, необхідні для доступу до даних. Слід прагнути того, щоб, надаючи додатку право

виконувати процедури, заборонявся доступ до таблиць даних. Якщо використання збережених процедур стає неможливим, слід надати можливість виконання прямих SQL-запитів через облікові записи з найменшими привілеями. Тобто у цього облікового запису повинні бути чітко обмежені кордони доступу до читання / додаванню / зміни / видалення даних в таблицях, а також позначено до яких саме таблиць є доступ. Слід уникати створення облікового запису адміністратора в додатку. Якщо дотримуватися принципу найменших привілеїв, то навіть у випадку виявлення зловмисником вразливості, він зможе нанести менші встрати.

Blind SQL Injection

Виявлення і профілактика є складними завданнями. Проте, якщо є правильне розуміння про типи SQL ін’єкцій, тоді легше запобігти нападу. Для запобігання сучасним SQL Injection Attack завжди рекомендується використовувати підготовлені шаблоним, котрі фіксовані і не можуть бути змінені користувачем веб-сайту або веб-application. Такі методи, як mafic_quotes() і add_slashes() не можуть захистити веб-додаток або веб-сайт від SQL Injection Attack. Тому сдід обговорити різні методи для виявлення і запобігання сучасного SQL Injection.

Існує досить багато науково-дослідних робіт, котрі описують сліпі SQL-ін'єкцій, де наводяться різні системи виявлення та запобігання методам впровадження даного типу атак. Blind SQL Injection важко виявити і запобігти ним, але дослідники були інформовані про Blind SQL Injection з минулих багатьох років. Найбільш популярний метод AMNESIA, який dghjdfl;e’nmcz для аналізу, моніторингу та нейтралізації атак SQL-ін'єкцій. Інструмент застосовується лише для захисту Java додатків і використовує під час виконання метод monitoring.Komiya, котрий являється кращим для

запобігання BlindSQL Injection. Розробники рекомендують використовувати алгоритми машинного начання для того, щоб поліпшити виявлення та запобігання сліпих SQL Injection. Також вони отримали результати і підтвердили, що запобігання та виявлення було краще, ніж ри використанні чистого SQLCheck та AMNESIA.

Fast Flux

Основні атаки даного типу направлені на вразливості систем захисту на клієнтській стороні. Із Fast Flux SQL Injection зіткнулися в Університеті штату Індіана, США і навіть системи захисту ФБР були стурбовані цим типом атак. Кращий спосіб захисту від даної атаки - зробити сервери більш безпечними. Швидкий потік може бути захищений з використанням методу, за допомогою якого точка URL для хітів доставки Javascript може потрапляє в чорний список, якщо вони визначені в стилі Fast Flux. Практичною реалізацією такого методу є Fast Flux Monitor (FFM), який може виявити, а також класифікувати Fast Flux Service Networks за лічені хвилини для використання як активного і пасивного моніторингу DNS, який доповнює довгострокове спостереження за FFSNs. Після того, як Fast Flux мережа класифікована, ми можемо використовувати SQL Injection метод для того, щоб зупинити SQL-ін'єкцію, за допомогою додаткового моніторингу. Впровадження таких технологій представляється у вигляді навідних заходів для забезпечення методів подальшого кодування. Проте зловмисники стають все розумнішими і до винайдення нових способів захисту долучились навіть дослідники із контррозвідки ООН. Вони дійшли висновків у полі дослідження та виявлення швидких мереж потоку і SQL-ін'єкції і запропонували технологію Expert Systems. Дана технологія впроваджували методи машинного навчання, які можуть бути використані для виявлення. Серед багатьох методів машинного

навчання, вони роблять акцент на C5.0 - класифікатор і нативний байєсівський класифікатор для визначення Fast Flux SQL Injection Attack. Запобігання Fast Flux дійсно складне завдання, і багато дослідників знаходять правильні методи, щоб протистояти даному типу атак.

SQLi XSS

Ardilla Tool – засіб для виявлення SQL + XSS атак. Має два режими для перевірки на наявність небезпеки: strict, Lenient. При використанні strict режиму застосовуються методи ідентифікації та запобігання ін’єкціям.

Технологія роботи засобу полягає, у використанні методу детекції існуючих методів захисту системи – Taint Base. Данний метод аналізує статичні вже впроваджені техніки, і якщо попередні вимоги до застосування системи згідно наведеного вище методу не виконані, Ardilla доповнює систему необхідними методиками за допомогою фільтрів заповнення і так званої ‘санітарної’ обробки. SQL ін’єкції часто використовують в поєднанні із XSS і впроваджують за допомогою Java Script. Аналогічні проекти не запроваджують такого ж рівня безпеки, адже для повного захисту системи, першочергво потрібно захиститися саме від XSS, лише пізніше впроваджувати методики захисту від SQL, чим і займається Ardilla.

Механізм для обробки обох етапів впровадження захисту являє собою так званий котроль потоку виконання. Його суть полягає у захисті від SQL ін’єкцій, котрі сховані в тілі XSS атаки. Використовуєтсья так званий автомат кінцевого стану для аналізу клієнтського java script коду. Даний автомат запобігає просоченню шкідливого коду в тілі скріпта до бази даних.

Іншим прикладом реалізації методик захисту можеслугувати Noxes tool. Проте даний засіб, як вже обговорювалось вище, не запроваджує такого ж

рівня безпеки, так як у ньому наявна вразливість до html тегів, які зловмисник може застосовувати для впровадження шкідливого коду, замість самих скріптів. Серед плюсів даного засобу можна виділити перевірку https запиту і запобігання модифікації http заголовку, а також наявність функціоналу контролю куків.

SQLi DNS

Методи захисту від поєднання даних типів атак полягають у реалізації технології розподілення. По-перше, розпізнається DNS Hijacking, а по-друге, впроваджуються самі методи захисту від ін'єкцій. DNS Hijacking можна запобігти, у разі відмови у завантаженні, як приклад, безкоштовного програмного забезпечення із веб-сайтів, так як дані продукти мають досить багато вразливостей. Також слід виділити DNS підміну. Даний метод намагаєтсья перехопити налаштування роутера клієнта. Заради впровадження запобіжних заходів був представлений фільтр, котрий слідкує за потоком виконання DNS – LWCSS (lightweight client side shield).

SQLi Cross Domain Policies

Методи запобігання данному типу атак полягають у реалізації коректного впровадження політик захисту та зменшення використання тк званих засобів, котрі мають субдоменні вразливості. Наразі існує метод DEMACRO. Він використовує статичні та динамічні аналізатори коду впроваджених Cross Domain Policies. Перевагою данного методу являєтсья те, що він не потребує машинного навчання для нормативного аналізу. Сутність методу полягає у виявленні спроби атаки та деавторизації нападника у системі.

SQLi DDoS

DDoS-атаки добре розуміються фахівцями з безпеки, але, незважаючи на те, що дана темп - добре обговорювана, є деякі лазівки, які зловмисник використовує для атаки на систему. Тому дослідники винайши так звану технологію кластерного аналізу. Вона допомагає виявити атаку і може легко визначити тип атаки на систему. Після фази виявлення, йде фаза пом'якшення, в котрій відбувається порівння типів атак, причому слід відмітити, що випадок SQLI DDoS атак широко поширений. В даний час покращення методу полягає поеріщенні методів імплементації для подальшого використання, адже основними обєктами захисту є веб-сервери, веб-програми та веб-сайти від SQL DDoS.

SQLI Insufficient Authentication

Для того, щоб захиститися від SQLI, не отримуючи проблеми автентифікації, адміністратор може використовувати методику криптографічних функцій Hash, що використовуються для захисту від SQLI + Insufficient Authentication. У цьому методі додаються два додаткові атрибути, які є хеш-функцями для поля ім'я користувача та пароля. Хеш-функції автоматично генеруються з використанням алгоритмів хеш. Тепер, коли клієнт вводить ім'я користувача і пароль, то хеш-функція генерується і передається на сторону сервера для перевірки. Все, що відбувається тут, зберігається і переноситься у зашифрованому вигляді. Якщо ім'я користувача та пароль однакові щодо тих, котрі зберігаються в базі даних, в порівнянні із її хешем функції, то існує лише незначний шанс на вторгнення до бази даних.

Аналіз методів виявлення та запобігання

На даному етапі слід проаналізувати методи виявлення та запобігання, представлені в попердніх частинах.

У таблиці 2.1 показано, які методики використовується для виявлення та запобігання використання сучасним SQL Injection Attacks. Вони допоможуть дослідникам і фахівцям з безпеки вжити відповідних заходів або використовувати зазначені методи для вирішення проблем, що виникли всередині організації внаслідок нападу. Описані тут техніки можуть бути використані для розробки системи з додатковою функціональністю для захисту системи від будь-якого виду цих сучасних нападів, які відповідають SQL ін'єкціям та швидкому потоку SQL атаки.

У таблиці 2.2 аналізуються засоби виявлення та профілактики. Наведені різні засоби для виявлення та попередження від сучасних атак. Ці засоби - готові продукти, а деякі - з відкритим кодом, які можна завантажити з Інтернету. Більшість цих засобів була розроблена для цілей дослідження, але завдяки своїм значним перевагам вони використовуються в комерційних секторах. Дана таблиця слугує загальним оглядом того, який з цих засобів може використовуватися для конкретного типу атак. За нашими У ході опитування ми спостерігаємо, що Noxeus і SQLMap є останніми і мають кращий механізм запобігання та виявлення.

Використання різних символів:

(*) - використовується як для виявлення, так і для попередження.

(o) - використовується тільки для механізму виявлення.

(+) - використовується для відображення тільки відповідних профілактичних заходів.

(x) - зображують, що методи або засоби не відповідають сучасним SQL Injection (з точки зору запобігання та виявлення).

(p) - зображує неповноту, що означає після застосування конкретного методу повинен бути застосований інший метод для досягнення повноти виявлення та профілактики.

Таблиця 2.1 - Аналіз методів виявлення та запобігання

S.No.	Technique	Blind SQL Injection	FF_SQLI	SQLI_XSS	SQLI_DNS	SQLI_CDP	SQLI_DDoS	SQLI_In_Authen
1.	Crypto Graphical Hash Functions	p	x	x	x	x	x	+
2.	Dynamic Cookies Rewriting	x	x	+	p	x	x	p
3.	Execution Flow Mechanism	x	x	*	x	x	x	x
4.	Static Code Analysis	o	x	o	x	*	x	x
5.	Dynamic Data Tainting	x	x	*	x	x	x	x
6.	Run Time Monitoring	p	+	*	x	x	x	x
7.	Machine Learning	x	o	*	x	x	o	x

Таблиця 2.2 - Аналіз засобів виявлення та запобігання

S.No.	Tools	Blind SQL Injection	FF_SQLI	SQLI_XSS	SQLI_DNS	SQLI_CDP	SQLI_DDoS	SQLI_In_Authen
1.	Ardilla Tool	x	x	o	x	o	x	x
2.	Noxes	x	x	p	x	+	x	x
3.	Session Shield	x	x	*	p	x	x	p
4.	AMNESIA	*	x	p	x	x	x	x
5.	SQLMap	o	x	o	p	x	o	o
6.	Fast Flux Monitor	x	o	x	o	x	x	x

2.2 Аналіз технології проксі HTTP-запитів

Проксі-сервер діє як шлюз між вами та Інтернетом. Це сервер-посередник, який відокремлює кінцевих користувачів від веб-сайтів, які вони переглядають. Проксі-сервери надають різні рівні функціональності, безпеки та конфіденційності залежно від випадку використання, потреб або політики компанії.

Сучасні проксі-сервери роблять набагато більше, ніж пересилання веб-запитів, все в ім'я безпеки даних і продуктивності мережі. Проксі-сервери функціонують як брандмауер і веб-фільтр, надають спільні мережні підключення та кешують дані, щоб прискорити загальні запити. Хороший проксі-сервер захищає користувачів і внутрішню мережу від поганих речей, які живуть у Інтернеті. Нарешті, проксі-сервери можуть забезпечити високий рівень конфіденційності.

HTTP-проксі-сервер є проксі-сервером, який обробляє запити HTTP (S). HTTP-проксі-сервер надає додатковий рівень безпеки, оскільки реальні веб-сервери не можуть бути безпосередньо досягнуті, так як проходять лише запити на певні IP-адреси та порти. Баланс навантаження є гарним показником для веб-проксі. Виходячи з вищевикладеного, можна зробити висновок, що проксі-сервер HTTP (S) діє як посередник між вами та веб-сайтом, який ви відвідуєте, простими словами. В Інтернеті існує досить багато HTTP(S) проксі-серверів.

Також слід додати, що HTTP-проксі-сервер - це сервер, який використовується для обробки запитів HTTP, які використовують формат IP: порт. Запит, зроблений клієнтом, передається у вигляді повної URL-адреси. Після передачі цієї URL-адреси проксі-сервер робить запит більш конкретним, щоб отримувати лише найрелевантніші результати пошуку. Основний механізм підтримки анонімності користувача зберігається так, що не ви, а веб-проксі-

сервер підключається до сайту. Це діє як щит між вами та веб-сайтом, до якого ви звертаєтеся, тим самим не дозволяючи нікому шукати вашу діяльність в Інтернеті.

Тунелювання передає дані та протокол приватної мережі через загальнодоступну мережу шляхом інкапсуляції даних. HTTP тунелювання використовує протокол більш високого рівня (HTTP) для транспортування протоколу нижчого рівня (TCP).

Найбільш поширеною формою тунелювання HTTP є стандартизований метод HTTP CONNECT. У цьому механізмі клієнт просить проксі-сервер HTTP перенаправити з'єднання TCP до потрібного пункту призначення. Потім сервер переходить до підключення від імені клієнта. Після встановлення з'єднання із сервером, проксі-сервер продовжує передавати потік TCP до та з клієнта. Тільки початковий запит на з'єднання - HTTP - після цього сервер просто проксує встановлене з'єднання TCP.

Варто зазначити, що тунель HTTP також може бути реалізований, використовуючи тільки звичайні методи HTTP як POST, GET, PUT і DELETE. Це схоже на підхід, що використовується у двонаправлених потоках над синхронним HTTP.

SOCKS — мережевий протокол, який дозволяє клієнт-серверним додаткам прозоро використовувати сервіси за міжмережевими екранами (фаєрволами). SOCKS — це скорочення від «SOCKet Secure».

Клієнти за міжмережевим екраном, що потребують доступ до зовнішніх серверів, замість цього можуть з'єднуватися з SOCKS проксі-сервером. Такий проксі-сервер контролює права клієнта для доступу до зовнішніх ресурсів і передає запит до сервера. SOCKS може використовуватися і протилежним способом, дозволяючи зовнішнім клієнтам з'єднуватися з серверами за міжмережевим екраном (брандмауером).

На відміну від HTTP проксі-серверів, SOCKS передає всі дані від клієнта, нічого не додаючи від себе, тобто з точки зору кінцевого сервера, SOCKS проксі є звичайним клієнтом. SOCKS більш універсальний - не залежить від конкретних протоколів рівня додатків (7-го рівня моделі OSI) і базується на стандарті TCP/IP - протоколі 4-го рівня. Зате HTTP проксі кешує дані і може ретельніше фільтрувати вміст переданих даних.

SOCKS 4 призначений для роботи через фаєрвол без аутентифікації для додатків типу клієнт-сервер, що працюють за протоколом TCP, таких, як TELNET, FTP і таких популярних протоколів обміну інформацією, як HTTP, WAIS і GOPHER. По суті, SOCKS-сервер можна розглядати як міжмережевий екран, що підтримує протокол SOCKS.

Типовий запит SOCKS 4 виглядає наступним чином (кожне поле - один байт):

Запит Клієнта до SOCKS-Серверу:

Поле 1: номер версії SOCKS, 1 байт (повинен бути 0x04 для цієї версії)

Поле 2: код команди, 1 байт:

0x01 = установка TCP / IP з'єднання

0x02 = призначення TCP/IP порту (binding)

Поле 3: номер порту, 2 байти

Поле 4: IP-адреса, 4 байти

Поле 5: ID користувача, рядки змінної довжини, завершується null-байтом (0x00)

Відповідь Сервера SOCKS-Клієнту:

Поле 1: null-байт

Поле 2: код відповіді, 1 байт:

0x5a = запит наданий

0x5b = запит відхилений чи помилковий0x5c = запит не вдався, бо не запущений identd (або не доступний з сервера)

0x5d = запит не вдався, оскільки клієнтський identd не може підтвердити ідентифікатор користувача в запиті

Поле 3: 2 довільних байта, повинні бути проігноровані

Поле 4: 4 довільних байта, повинні бути проігноровані

SOCKS 5 розширює модель SOCKS 4, додаючи до неї підтримку UDP, забезпечення універсальних схем строгої аутентифікації і розширює методи адресації, додаючи підтримку доменних імен і адрес IPv6. Початкова установка зв'язку тепер складається з наступного:

- Клієнт підключається, і посилає запит, який включає перелік підтримуваних методів аутентифікації
- Сервер вибирає з них один (чи надсилає відповідь про невдачу запиту, якщо жоден із запропонованих методів недоступний)
- В залежності від обраного методу, між клієнтом і сервером може пройти деяка кількість повідомлень
- Клієнт посилає запит на з'єднання, аналогічно SOCKS 4
- Сервер відповідає, аналогічно SOCKS 4

Методи аутентифікації пронумеровані таким чином:

0x00 - аутентифікація не вимагається

0x01 - GSSAPI

0x02 - ім'я користувача / пароль

0x03-0x7F - зарезервовано IANA

0x80-0xFE - зарезервовано для методів приватного використання

Початковий запит від клієнта:

Поле 1: номер версії SOCKS (повинен бути 0x05 для цієї версії)

Поле 2: кількість підтримуваних методів аутентифікації, 1 байт

Поле 3: номери методи аутентифікації, змінна довжина, 1 байт для кожного підтримуваного методу

Сервер повідомляє про свій вибір:

Поле 1: Версія SOCKS, 1 байт (0x05 для цієї версії)

Поле 2: обраний метод аутентифікації, 1 байт, або 0xFF, якщо не було запропоновано прийнятного методу.

Подальша ідентифікація залежить від обраного методу.

Запит клієнта:

Поле 1: номер версії SOCKS (повинен бути 0x05 для цієї версії)

Поле 2: код команди, 1 байт.

Висновки до розділу 2

У даному розділі були досліджені додаткові методи виявлення та запобігання SQL ін’єкціям – типу атак, котрий панує у сфері веб-застосунків. Також було обговорено поєднання ін’єкцій із іншими типами атак для поглибленого розуміння методів та засобів захисту. Для коректної розробки системи потрібне чітке розуміння сутності наведених атак і усвідомлення наявності методів та засобів виявлення та запобігання для зменшення прогнозованої шкоди від потенційних дій зловмисника на систему.

Дослідивши механізми виявлення та запобігання SQL ін’єкціям та змішаним типам атак, приходимо до висновку, що наявні засоби захисту не в повній мірі забезпечують захист від наведених типів атак і потребують подальшого доопрацювання.

У ході аналізу мережевих протоколів було виявлено, що ідеальним для реалізації проксуючого фаєрволу та його подальшого підняття на проксі-сервері вирішенням, є SOCKS 4.

3 МОДЕЛЬ ПРОКСУЮЧОГО ФАЄРВОЛУ ДЛЯ ЗАХИСТУ СУБД ВІД SQL ІН'ЄКЦІЙ

В даному розділі будуть розглянуті та проаналізовані архітектура програмного комплексу та технологія розробки практичної реалізації роботи.

3.1 Архітектура програмного комплексу

Головною небезпекою в комунікації користувача і бази даних є той факт, що спілкування відбувається безпосередньо, як показано на рисунку 3.1

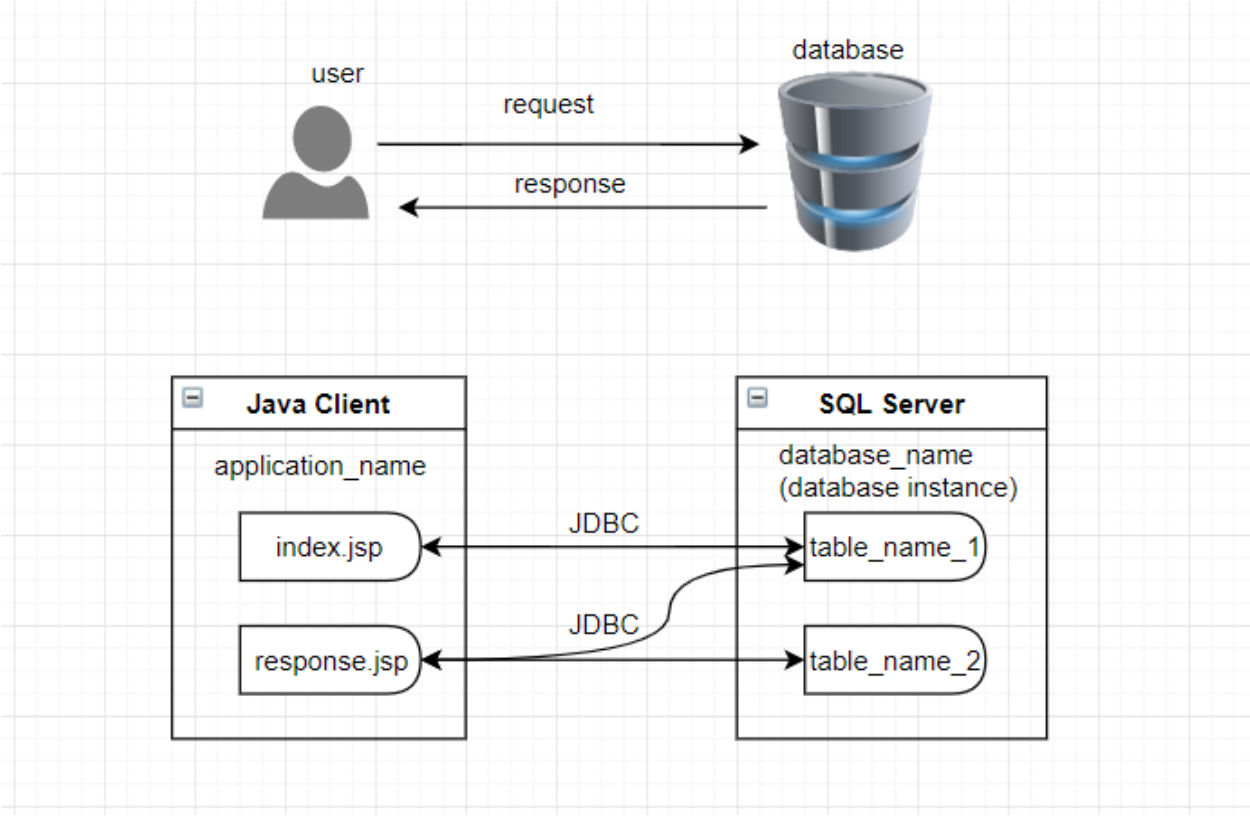

Рисунок 3.1 - Спілкування користувача і бази даних

За умови поганої реалізації програми та перевірки вхідних параметрів користувача, ризик успішного здійснення атаки типу SQL-ін'єкції дуже високий.

Далі буде описано, як розробляється засіб, котрий братиме участь в спілкуванні користувача і бази даних.

При запуску програма повинна прослуховувати трафік між клієнтом і сервером бази даних і записувати його в лог-файл, або аналізувати запити клієнта до сервера бази даних і блокувати їх, якщо вони будуть містити ознаки SQL-ін'єкції.

При старті програми користувач повинен буде вказати, яка база даних буде використовуватися, порт, який клієнт буде використовувати для зв'язку, ip-адреса і порт бази даних, режим, в якому буде працювати фаєрвол.

Фаєрвол буде реалізовано у вигляді проксі-сервера, тобто пропускати через себе весь клієнт-серверний трафік.

Для простого прослуховування запитів читаємо пакет клієнта, після чого дістаємо з пакету SQL-рядок і записуємо його в лог файл. По завершенню цих процесів пакет відправляється до сервера СУБД. Далі очікуємо наступний пакет.

Для аналізу і блокування запитів читаємо пакет клієнта, дістаємо з нього SQL-рядок. Даний рядок аналізується на предмет вмісту SQL-ін'єкції. Якщо така виявляється, то пакет не відправляється до сервера, при цьому даний SQL-рядок також записується в лог файл. Після чекаємо наступний пакет від клієнта.

Алгоритм роботи програми для захисту баз даних від SQL-ін'єкцій у вигляді блок-схеми зображений на рисунку 3.2

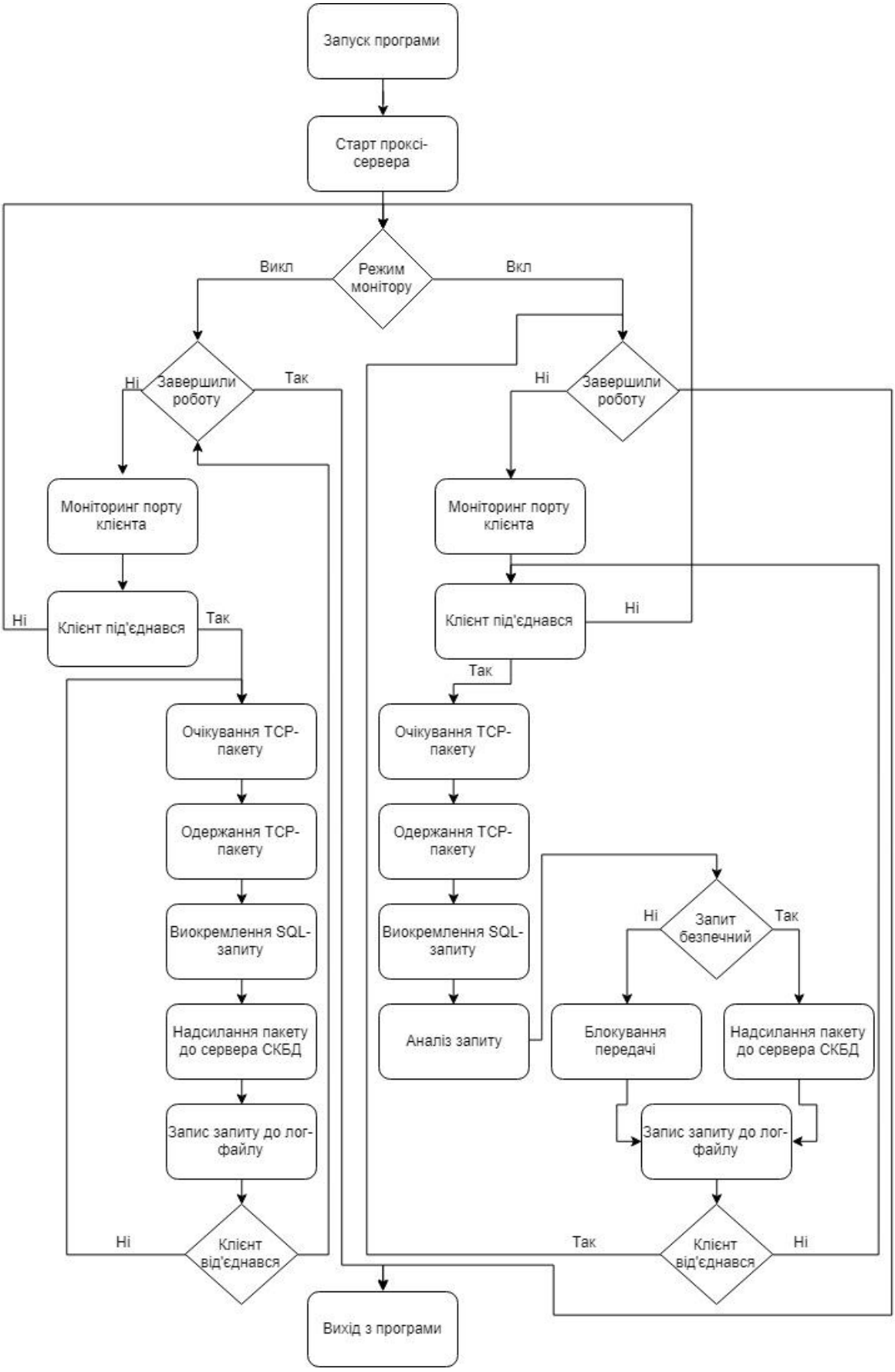

Рисунок 3.2 - Алгоритм роботи фаєрвола

Після того як був проведений аналіз існуючих рішень та виявлено їх недоліки, сформулюємо основні вимоги до програми:

- має бути реалізовано прослуховування трафіку і його запис в лог-файл;
- повинна бути реалізована можливість аналізу SQL-запитів приходять від клієнта;
- повинна бути реалізована блокування запитів, що містять ін'єкцію і їх запис в лог-файл;
- не повинна бути присутнім навантаження на базу даних;
- повинна бути реалізована робота з такими СУБД як: MsSQL Server, MySQL, MariaDB, PostgreSQL.

3.2 Вибір технологій розробки

Виробники баз даних, такі як Microsoft ® і Oracle ® , впроваджують свої системи баз даних за допомогою технологій, які змінюються залежно від потреб клієнтів, потреб ринку та інших факторів. Програмні додатки, написані на популярних мовах програмування, такі як C, C ++ і Java ® , потребують способу спілкування з цими базами даних. Відкрита база даних (ODBC) і Java Database Connectivity (JDBC) - це стандарти для драйверів, які дозволяють програмістам писати агностичні програми. ODBC і JDBC забезпечують набір правил, рекомендованих для ефективної комунікації з базою даних. Постачальник бази даних відповідає за впровадження та забезпечення драйверів, які дотримуються цих правил.

ODBC - це стандартний інтерфейс Microsoft Windows ®, що дозволяє здійснювати зв'язок між системами управління базами даних і програмами, які зазвичай написані на мові C або C ++.

JDBC - це стандартний інтерфейс, що дозволяє здійснювати зв'язок між системами управління базами даних та додатками, написаними на Oracle Java.

Залежно від середовища і того, що планується в реалізації, вирішуєтсья, який драйвер використовувати: драйвер ODBC або драйвер JDBC.

Розглянемо випадки застосування драйверів.

Використовується ODBC у наступних випадках і за для:

- Найшвидших показників імпорту та експорту даних;
- Імпорту та експорту даних, що потребують пам'яті;
- Всіх функцій, крім runstoredprocedure функції.

Використовувати JDBC у наступних випадках і за для:

- Незалежності від платформи, що дозволяє працювати з будь-якою операційною системою (включаючи Mac і Linux ®), версії драйвера, або розрядності;
- Доступу до всіх функцій Toolbox Database.

Дана програма буде написана на мові JAVA. Мова JAVA в даний час одна з найбільш використовуваних і сучасних мов програмування. Перевагою даної мови є наявність великої кількості безкоштовних інструментів і докладних відомостей, що дає можливість реалізовувати будь-які проекти.

Так само слід сказати, що програми на Java транслюються в байт-код Java, який виконується віртуальною машиною Java (JVM) - програмою, обробній байтовий код і передавальної інструкції обладнанню як інтерпретатор.

Java використовує те, що називається JDBC (Java Database Connectivity) для підключення до баз даних. JDBC (Java DataBase Connectivity - з'єднання з базами даних на Java) призначений для взаємодії Java-додатка з різними системами управління базами даних (СКБД). Увесь рух в JDBC заснований на драйверах які вказуються спеціально описаним URL.

В основі JDBC лежить концепція так званих драйверів, що дозволяють отримувати з'єднання з базою даних по спеціально описаному URL. Драйвери можуть завантажуватись динамічно (під час роботи програми). Завантажившись, драйвер сам реєструє себе й викликається автоматично, коли програма вимагає URL, що містить протокол, за який драйвер «відповідає».

З'єднання з базою даних описується класом, що реалізує інтерфейс java.sql.Connection. Маючи з'єднання з базою даних, можна створювати об'єкти типу Statement, використовувані для здійснення запитів до бази даних на мові SQL.

Існують такі види типів Statement, що відрізняються своїм призначенням:

- java.sql.Statement — Statement загального призначення;
- java.sql.PreparedStatement — Statement, що служить для здійснення запитів, котрі містять підставні параметри (позначаються символом '?' у тілі запиту);
- java.sql.CallableStatement — Statement, призначений для виклику збережених процедур.
- java.sql.ResultSet дозволяє легко обробляти результати запитів.

В ході аналізу технологій розробки для реалізації практичної частини були виявлені наступні переваги JDBC в порівнянні з аналогами:

- Легкість розробки: розробник може не знати специфіки бази даних, з якою працює;
- Код не змінюється, якщо компанія переходить на іншу базу даних;

- Не треба встановлювати громіздку клієнтську програму;
- До будь-якої бази можна під'єднатись через легко описуваний URL.

Висновки до розділу 3

В даному розділі була побудована модель та описана структура практичної реалізації майбутнього фаєрволу, а також були висунуті основні вимоги щодо самої програми і методу її реалізації.

Аналізуючи вибір технології розрбки, було досліджено, що перевагою подібного способу виконання програми є повна незалежність байт-коду від операційної системи і устаткування, що дозволяє виконувати Java-додатки на будь-якому пристрої, для якого існує відповідна віртуальна машина.

Також були проаналізовані драйвери для коректного зв’язку середовища розробки та баз даних. Враховуючи широкий спектр баз даних, для котрих розробляється практична реалізація і кількість операційних систем, на котрих вони базуються – прийнято рішення використовувати JDBC інтерфейс.

4 ПРАКТИЧНА РЕАЛІЗАЦІЯ І ТЕСТУВАННЯ ФАЄРВОЛА ДЛЯ ЗАХИСТУ СУБД ВІД SQL ІН'ЄКЦІЙ

В даному розділі будуть розглянуті та розроблені практична реалізація проксі-сервера, алгоритм розбору пакету клієнта, а також буде проведений аналіз ефективності виявлення SQL інєкцій.

4.1 Практична реалізація проксі-сервера

Програма буде написана на мові JAVA. Для реалізації проксі-сервера знадобляться наступні класи: java.net.Socket і java.net.ServerSocket, котрі є стандартними класами JAVA SDK.

Щоб приймати пакети від клієнта і відправляти йому відповідь сервера СУБД, створюється сокет за допомогою класу ServerSocket наступним чином:

```
ServerSocket server = new ServerSocket (clientport, 100, InetAddress.getByName (clienthost));
```

Тут clienthost - ip-адреса, clientport - порт, на які будуть приходити пакети від клієнта і куди будуть відправлятися відповіді сервера.

Сокет для спілкування із сервером реалізується за допомогою класу Socket:

```
Socket server = new Socket (SERVER_URL, SERVER_PORT);
```

Тут SERVER_URL - ip-адреса бази даних, SERVER_PORT - порт бази даних, на який будуть приходити пакети від сервера і йти пакети до сервера.

Проксі-сервер приймає багато клієнтів, а не одного, тому він реалізується багатопотоковим за допомогою класу Thread і його методу run ().

Для прийняття даних від клієнта реалізується потік введення InputStream:

```
final InputStream inFromClient = sClient.getInputStream ();
```

Метод getInputStream () повертає вхідний потік від сокета, до якого приєднався клієнт.

Дані клієнта посилаються, використовуючи потік виведення OutputStream:

```
final OutputStream outToClient = sClient.getOutputStream ();
```

Метод getOutputStream () повертає вихідний потік від сокета, до якого приєднався клієнт.

За аналогією визначаються такі ж потоки для сервера бази даних:

```
final InputStream inFromServer = server.getInputStream ();
final OutputStream outToServer = server.getOutputStream ();
```

У підсумку, спілкування користувача бази даних і клієнта відбуватиметься через даний проксі-сервер, як показано на рисунку 4.1, де запити надіслані користувачем будуть аналізуватися на предмет небезпечних або недозволених включень.

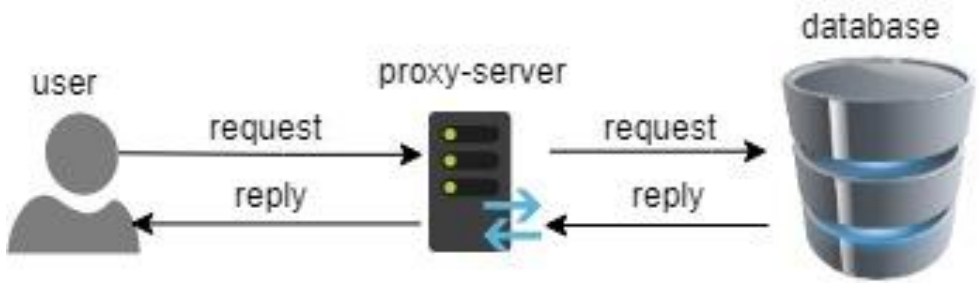

Рисунок 4.1 - Спілкування користувача і БД через проксі-сервер

4.2 Алгоритм розбору пакету клієнта

Щоб проаналізувати SQL-запит, надісланий клієнтом, розибрається пакет, відправлений ним, до бази даних. Для розбору пакету були створені 3 класи: MsSQLPackParser, MySQLPackParser і PGSQLPackParser. Як можна помітити, в цьому списку немає класу, який би розбирав пакети від MariaDB. Вся справа в тому, що клієнт-серверний протокол цієї бази даних такої ж, як у MySQL, тому для цих двох баз даних буде використовуватися однаковий клас MySQLPackParser.

Розгляд пакету баз даних MySQL і MariaDB. Перший пакет, який відсилається клієнтом серверу, є автентифікацією. Щоб однозначно ідентифікувати даний пакет, потрібно подивитися на байти з 17 по 32, вони повинні бути рівні нулю. Наступний крок - отримуємо ім'я користувача. В даному протоколі рядок автентифікації починається з 37 байта. Ім'я користувача закінчується байтом 0x0.

Після цього клієнтом надсилаються пакети зі службовою інформацією. Вони нам нецікаві, тому ми відразу пропускаємо їх далі.

Далі йдуть пакети з SQL-запитами. Перший байт дорівнює довжині SQL-рядки в запиті. Запит щоразу починається з 6-го байта, тому, володіючи цією інформацією, ми можемо прочитати з пакета запит. [7]

```
int query_length = packet [0];
j = 5;
while (j! = (query_length + 4)) {
 query + = (char) packet [j];
 j ++;
}
```

Розгляд пакету бази даних PostgreSQL. Перший пакет, який відсилається клієнтом серверу, так само є автентифікацією. Перевірити це можна наступним чином: з 9 по 13 байти повинні містити значення відповідно 0x75, 0x73, 0x65, 0x72 і 0x0, що в перекладі на символи дає рядок "user.". Наступний крок - отримуємо ім'я користувача. В даному протоколі рядок ім'я користувача починається з 13 байта, і так само, як і в попередньому прикладі закінчується байтом 0x0.

Розгляд пакету з SQL-запитами. Їх можна відрізнити від інших пакетів по першому байту. Якщо перший байт дорівнює 0x50 або 0x51, то цей пакет містить запит. Після першого байта йдуть чотири байти рівні довжині SQL-рядки. Наведемо їх до типу int:

```
int query_length = ((0xFF & b [0]) << 24) | ((0xFF & b [1]) << 16) |
((0xFF & b [2]) << 8) | (0xFF & b [3]);
```

Тут потрібно звернути увагу на те, що в даному протоколі довжина SQL-рядка включає в себе і ці 4 байта.

З 7 байта починається сам запит, тому читаємо його так само, як і в попередньому протоколі.

Розгляд пакету MsSQL Server. У даній СУБД використовується TDS протокол (Tabular Data Stream Protocol). Перший пакет - це пакет аутентифікації. Його однозначно ідентифікують, подивившись перший байт пакету, він повинен бути рівний 0x10.

Отримується пакет автентифікації. У 51 і 52 байтах знаходиться позиція початку імені користувача. У 53 і 54 байтах міститься довжина імені користувача. Отже, ми можемо його витягти.

Розгляд пакету, який містить SQL-запит. Його так само відрізняємо за першим байтом, він повинен бути рівним 0x1. В даному протоколі SQL-рядок

починається з 9 байта. Довжина SQL-рядки міститься в 3 і 4 байтах. При цьому всі значущі байти розділені байтом 0x0. Запит дістаємо в такий спосіб:

```
a [0] = packet [2];

a [1] = packet [3];

int length = ((0xFF & a [0]) << 8) | (0xFF & a [1]);

int j = 8;


while (j <length) {

 if (packet [j]! = 0x0)

 temp + = (char) packet [j];

 j ++;

}
```

У підсумку, після того, як дістається рядок, що містить SQL-запит, він аналізується на предмет небезпечних або заборонених конструкцій.

4.3 Реалізація аналізу SQL-запитів

Для реалізації аналізу SQL-запитів на наявність небезпечних або заборонених конструкцій, за основу була взята політика безпеки «по білому списку», тобто потрібно буде пропускати тільки ті запити, конструкції яких будуть дозволені адміністратором.

У зв'язку з викладеним вище, розробляється словник, що містить маски дозволених запитів. На рисунку 4.2 представлений файл dictionary.txt з розміткою, необхідною для читання словника з файлу.

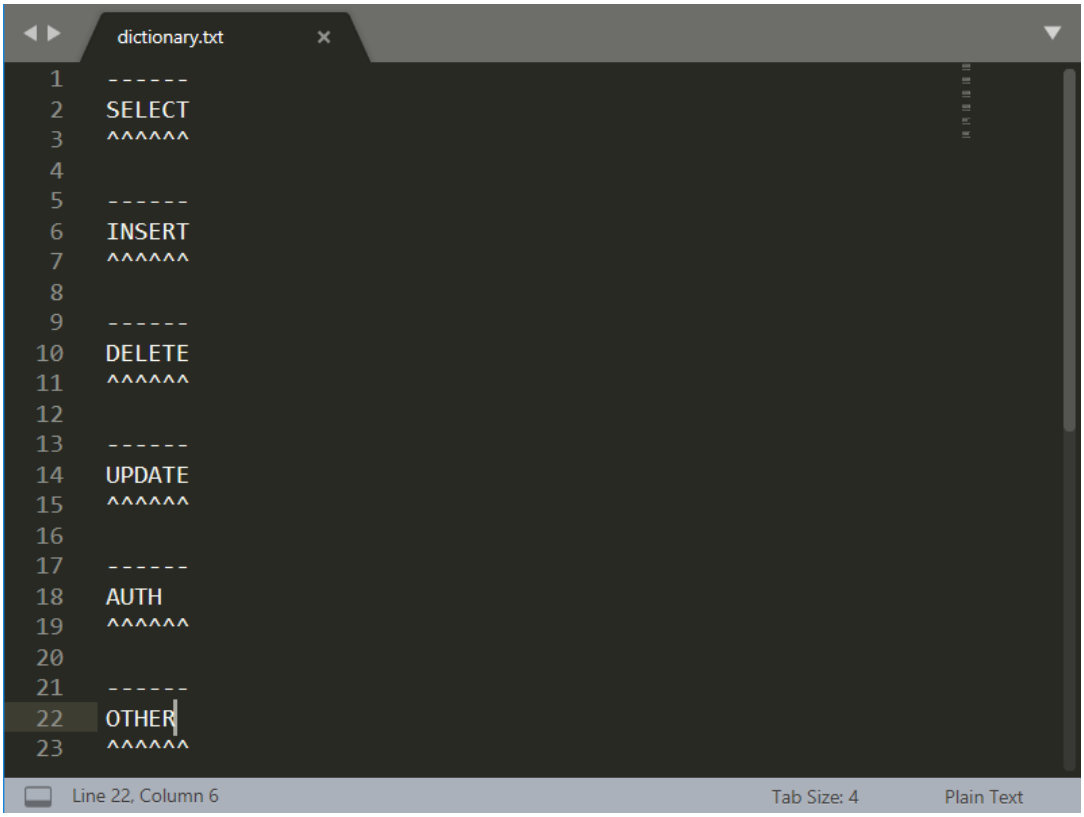

Рисунок 4.2 - Файл dictionary.txt з необхідною розміткою

Для зчитування даного словника був створений клас Dictionary.java. В даному класі були реалізовані:

- метод isDictLoad () - ініціює зчитування словника, повертає true, якщо словник успішно прочитаний, в іншому випадку повертає false,
- метод sortToArrays - розбирає отриманий словник в масиви запитів, що відповідають їхнім типом,
- get-методи для отримання масивів певного типу.

Розглянемо синтаксис словника. Для коректного зчитування всі типи запитів обрамляються рядком "______" зверху і рядком "^^^^^^" знизу. Так само варто відзначити, що для програми важливий порядок проходження типів запитів розглянутих на рисунку 6, тому міняти їх місцями не дозволяється.

Перехід до правил позначення параметрів в масках SQL-запитів. Так, якщо в запиті не має значення довжина і тип параметра, адміністратор повинен використовувати символ "*", і перевірка за цими критеріями не буде проведена.

Наступними позначеннями будуть символи " '" і "$". Ці символи використовуються спільно і позначають початок строкового параметра. При цьому адміністратору слід записувати таку кількість символів "$", яку максимальну кількість символів може міститися в рядку. Завершувати маску даного параметра слід так само - символ " '", з якого він починався.

Коли адміністратору неважливо, якої довжини строковий параметр буде надіслано користувачем, він може використовувати наступну комбінацію символів - " '$ *'". В даному випадку додатковий символ "*" вказує на те, що довжина строкового параметра не важлива.

Якщо ж параметр повинен бути чисельним, то в цьому випадку адміністратору слід вказувати символ "#". Цей символ повідомляє програмі, що параметр є числом, довжиною 1. За аналогією зі строковим параметром, слід записувати таку кількість символів "#", яку максимальну кількість символів може міститися в числі.

У разі якщо число може бути будь-якої довжини, потрібно використовувати наступну комбінацію символів: "# *". Тут додатковий символ "*" вказує на те, що довжина числового елемента не важлива.

Якщо адміністратор вирішить додати маски запитів таких типів, при написанні яких необхідне використання круглих дужок, то всі параметри повинні оформлятися так, як описано вище, а також повинні відокремлюватися комами без пробілів. Після символу "(" і перед символом ")" теж не повинно бути пробілів.

Оператори повинні оформлятися без змін, регістр не важливий. В кінці маски запиту ставити символ "; " не потрібно. Приклад заповнення словника наведено на рисунку 4.3

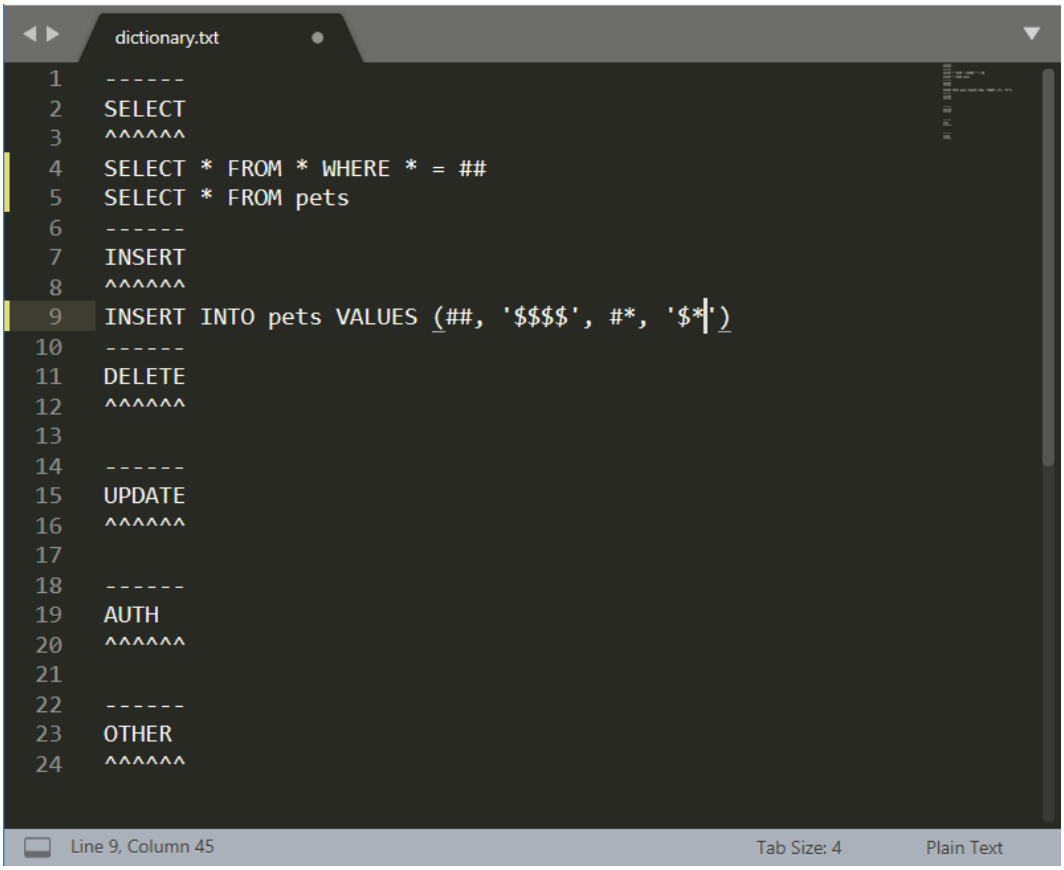

Рисунок 4.3 - Приклад заповнення словника

Реалізація класу CheckQuery.java, який використовує даний словник для аналізу надійшов запиту. В даному класі містяться такі методи:

- checker () - головний метод класу, розрізає кожну маску зі словника і запит на слова, додає ці слова до відповідних масиви і передає їх в наступний метод, в кінці повертає результат перевірки запиту користувача,
- compare () - метод перевіряє кожне слово, чи є воно оператором, строковим або чисельним параметром і передає маску даної частини запиту і частина запиту відповідним методам,

- checkStringValue () - метод перевіряє строковий параметр на відповідність масці строкового параметра,
- checkIntegerValue () - метод перевіряє чисельний параметр на відповідність масці числового елемента, також перевіряє чи є параметр із запиту числом, використовуючи наступний метод,
- checkCharIsDigit () - метод перевіряє кожен символ числа із запиту на предмет його приналежності числах,
- checkParenthesis () - метод перевіряє параметри, укладені в круглі дужки, використовуючи методи checkStringValue () і checkIntegerValue ().

Таким чином, реалізація програми для захисту баз даних від атак типу SQL-ін'єкції успішно реалізовано. На рисунках 4.4 та 4.5 зображено, що буде відбуватися при чистому і дозволеному запиті користувача, і що буде, якщо конструкція запиту не буде відповідати конструкції, що міститься в словнику відповідно.

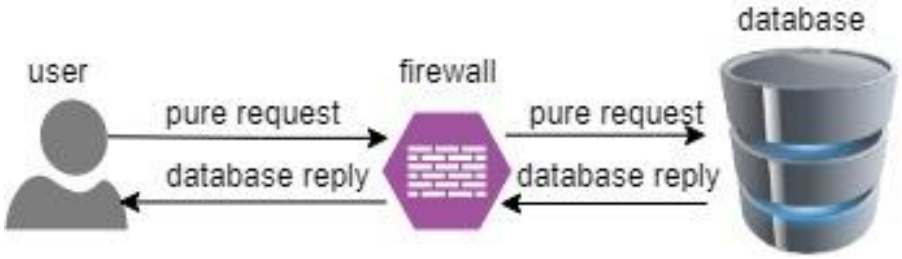

Рисунок 4.4 - Спілкування при легітимному запиті користувача

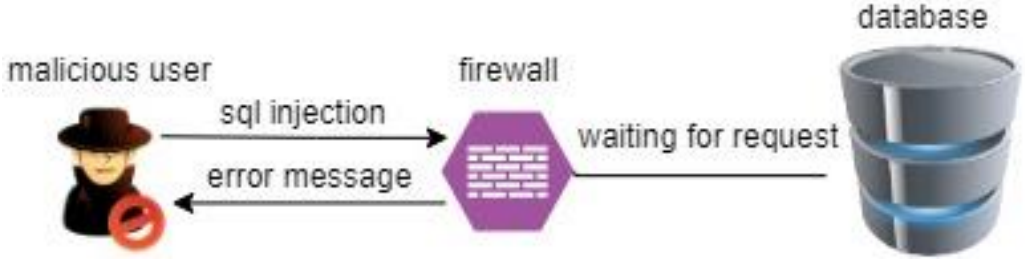

Рисунок 4.5 - Спроба отримати доступ при нелегітиму запиті

4.4 Аналіз ефективності фаєрвола

Тестування фаєрвола для баз даних за двома критеріями:

- якість фільтрації SQL-запитів,
- швидкість виконання SQL-запитів.

Слід пояснити, в якій конфігурації мережі будуть проходити тестування (рисунок 4.6).

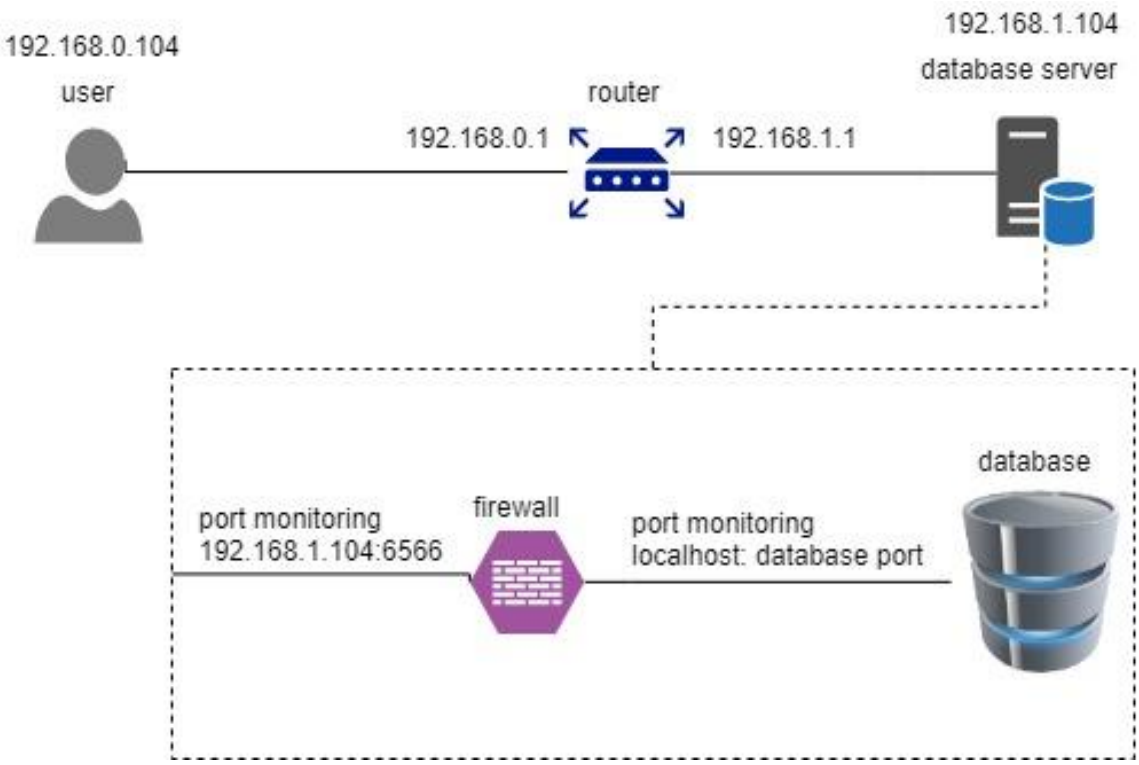

Рисунок 4.6 - Конфігурація мережі при проведенні тестування

Як можна помітити тестування будуть відбуватися в локальній мережі. Користувач буде звертатися до сервера баз даних через маршрутизатор. На сервері баз даних буде працювати фаєрвол, який слухає клієнта за адресою 192.168.1.104:6566, фільтрує SQL-запити і приймає рішення, чи посилати запит за адресою localhost: port, де port - порт, який слухає база даних. Приступимо до тестувань.

Тестування на якість фільтрації SQL-запитів

Для тестування фаєрвола в кожній з розглянутих баз даних були створені таблиці 4.1 і 4.2, відповідно pets (тварини) і owners (власники).

Таблиця 4.1 - Таблиця pets

id	name	weight
1	Barsik	2
2	Bobik	10
3	Murka	3
4	Tuzik	15

Таблиця 4.2 - Таблиця owners

id	name	petname	phone
1	Petya	Bobik	375637
2	Vasya	Murka	234987
3	Olya	Tuzik	164528
4	Sasha	Barsik	253749

Припустимо, адміністратор надав користувачам право доступу до таблиць вище, але дозволив їм оперувати з таблиці 4.1 іменами тварин (стовпець name) і стовпцем id, а з таблиці 4.2 тільки іменами власників (стовпець name) і іменами тварин (стовпець petname). При цьому, при вибірці за умовою "WHERE petname =« не шукати ім'я власника, якщо ім'я тварини більше п'яти символів. Інші операції з даними таблицями будуть заборонені. Заповнимо словник, керуючись даними правилами (рисунок 4.7).

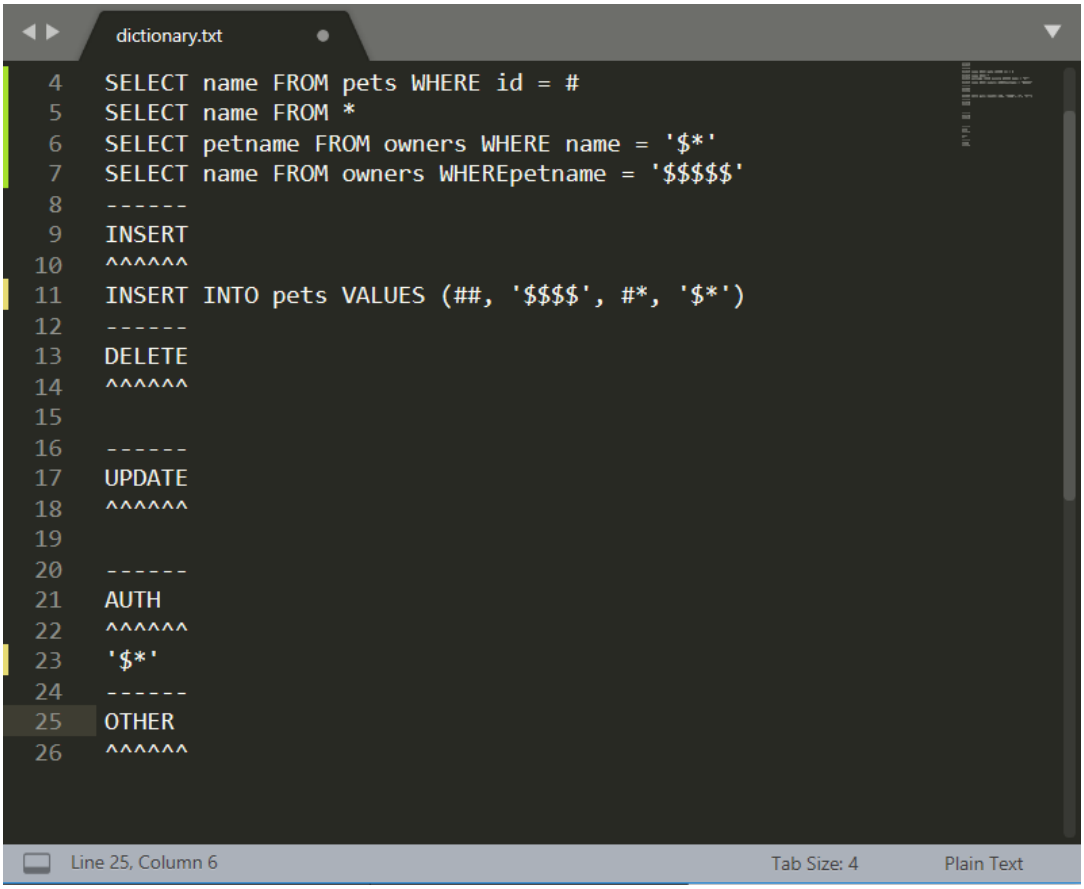

Рисунок 4.7 - Приклад заповнення словника

Для того щоб протестувати фаєрвол пишеться додаток-клієнт на мові JAVA. Додаток має підключатися до баз даних і виконувати запит. Створюється кілька класів для підключення. Для підключення до БД використовується jdbc драйвери, які підходять до них.

Однак є одна особливість, для того, щоб клієнт спілкувався з БД через фаєрвол, в url рядку програми слід змінити порт бази даних на порт, який прослуховує фаєрвол. Наприклад, маємо url рядок підключення:

```
String url = "jdbc: mysql: //192.168.1.104: 3306 / firewall";
```

В даному випадку 3306 - це порт, який слухає база даних. Щоб пустити трафік через фаєрвол, потрібно змінити порт, наприклад, на 6566. Рядок буде виглядати так:

```
String url = "jdbc: mysql: // 192.168.1.104:6566/firewall";
```

Тепер запускається фаєрвол, вводиться хост localhost і порт бази даних 3306 і слухається адреса 192.168.1.104:6566.

Створюються запити, які свідомо будуть легітимними, і проводиться спробва їх виконання:

1. "SELECT name FROM pets WHERE id = 3" - Результат виконання: Murka;
2. "SELECT name FROM pets WHERE id = 1" - Результат виконання: Barsik;
3. "SELECT name FROM pets" - Результат виконання: Barsik, Bobik, Murka, Tusik;
4. "SELECT name FROM owners" - Результат виконання: Petya, Vasya, Olya, Sasha;
5. "SELECT petname FROM owners WHERE name = 'Olya'" - Результат виконання: Tuzik;
6. "SELECT petname FROM owners WHERE name = 'Petya'" - Результат виконання: Bobik;
7. "SELECT name FROM owners WHERE petname = 'Tuzik'" - Результат виконання: Olya;
8. "SELECT name FROM owners WHERE petname = 'Bobik'" - Результат виконання: Petya;

Всі запити, відповідні маскам запитів, проходять через фаєрвол успішно. Тепер створюємо запити, маски яких не містяться в словнику, і спостерігаємо результат їх виконання:

1. "SELECT weight FROM pets" - Результат виконання: ERROR: Bad request;
2. "INSERT INTO pets VALUES (5, 'Stepan', 'Sharik', '385647')" - Результат виконання: ERROR: Bad request;
3. "SELECT * FROM owners" - Результат виконання: ERROR: Bad request;
4. "SELECT * FROM pets" - Результат виконання: ERROR: Bad request;
5. "SELECT phone FROM owners WHERE name = 'Vasya'" - Результат виконання: ERROR: Bad request;
6. "DROP TABLE pets" - Результат виконання: ERROR: Bad request;

7. "DELETE name FROM pets WHERE id = 2" - Результат виконання: ERROR: Bad request;
8. "SELECT name FROM owners WHERE petname = 'Barsik'" - Результат виконання: ERROR: Bad request;

Слід розглянути запит номер 8. У словнику існує схожа маска для нього - SELECT name FROM owners WHERE petname = '$$$$$'. Справа в тому, що в масці максимальна довжина строкового параметра дорівнює 5, а довжина імені Barsik дорівнює 6, тому запит не пройшов перевірку. Маски, що містяться в словнику, не підходять до жодного запиту вище, тому фаєрвол повертає користувачеві повідомлення "ERROR: Bad request".

Для інших типів команд справедливі ці ж правила. Можна зробити висновок, що фаєрвол працює добре і пропускає тільки ті запити, маски яких містяться в словнику. Перейдемо до наступного тестування.

Тестування на час виконання SQL-запитів

В ході даного тестування для кожного типу SQL-запитів буде створено по кілька різних їх варіантів. Для кожного варіанта буде вироблено 25 замірів часу виконання, щоб обчислити більш точне середнє значення.

Для виміру часу виконання запитів будемо використовувати стандартний метод мови JAVA - System.nanoTime ().

Наприклад, для запиту SELECT код буде виглядати так:

```
st = System.nanoTime ();
ResultSet rs = stmt.executeQuery ( "SELECT name FROM pets");
end = System.nanoTime ();
```

```
System.out.printf ( "Elapsed%, 9.3f ms \ n", (end - st) /1_000_000.0);
Thread.sleep (2000);
```

Іншими словами, метод виконання запиту обрамляється в методи, які повертають системний час. Після цього ми виводимо різницю між початковим і кінцевим значеннями часу, наведену до виду мілісекунд. Далі програма призупиняється на 2 секунди, для чистоти результату.

Наступний крок, створення запитів. Приклади створених запитів для кожного типу:

SELECT:

```
1. (f / d) запит:"SELECT name FROM pets, owners WHERE id = 3";
2. (f / d) запит: "SELECT * FROM pets";
3 (f / d) запит: "SELECT name FROM pets WHERE id = 1 and weight = 2";
4 (f / d) запит: "SELECT name FROM pets WHERE id = 2 or weight = 3".
```

INSERT:

```
1. (f / d) запит: "INSERT INTO pets VALUES (5," Bars ", 13)";
2. (f / d) запит: "INSERT INTO owners VALUES (5," Alex "," Bars "," ")";
3. (f / d) запит: "INSERT INTO pets, owners VALUES (6," Lola ")";
4. (f / d) запит: "INSERT INTO owners VALUES (6," Jacob ")".
```

DELETE:

```
1 (f / d) запит: "DELETE FROM pets WHERE id = 3";
2 (f / d) запит: "DELETE FROM owners WHERE id = 2";
3 (f / d) запит: "DELETE FROM pets WHERE id = 1 and weight = 2";
4 (f / d) запит: "DELETE FROM owners WHERE id = 1 or name =" Vasya "".
```

Створення маски запитів і запис їх в словник фаєрвола. Зміст словника представлено на рисунку 4.8

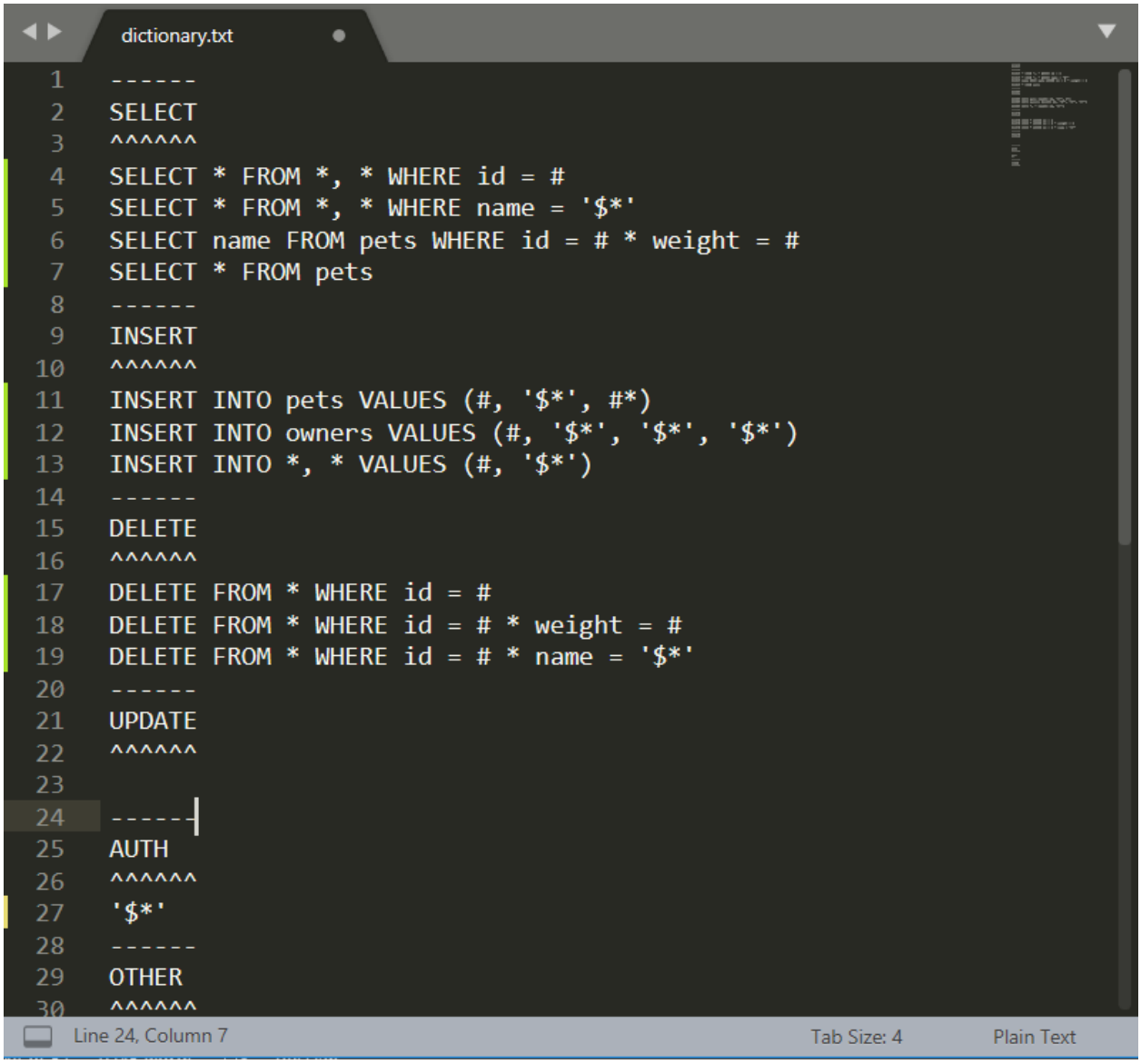

Рисунок 4.8 - Зміст словника для тестування запитів

Всі підготовчі етапи пройдені, тепер йде перехід до вимірювання часу виконання запитів. По черзі виконуються запити до різних баз даних безпосередньо і через фаєрвол по 25 разів. Середні узагальнені результати часу виконання представлені в таблицях 4.3, 4.4, 4.5.

Таблиця 4.3 - Час виконання SELECT запитів

	Час виконання запиту в мілісекундах																							
Q	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1d	4,533	1,969	2,159	1,932	1,964	1,796	1,898	1,911	2,064	1,805	1,889	1,951	1,911	1,838	2,406	1,946	1,977	1,95	1,975	1,963	1,957	1,87	1,917	1,943	2,29
1f	5,370	2,276	2,541	2,288	2,295	2,301	2,421	2,318	2,248	2,129	2,164	2,346	2,167	2,26	2,996	2,316	2,688	2,281	2,29	2,334	2,395	4,117	2,546	2,639	2,435
2d	3,896	1,841	1,936	1,827	1,877	1,808	1,773	2,016	1,814	1,856	1,698	1,736	1,821	1,742	3,947	1,931	2,374	1,93	1,907	1,819	1,766	1,723	1,661	2,094	1,947
2f	4,406	2,444	2,394	2,429	2,47	2,555	2,379	2,249	2,317	2,381	2,268	2,437	3,087	2,518	3,576	2,491	2,493	2,35	2,344	2,523	2,631	2,247	2,460	2,510	2,512
3d	4,206	1,814	1,763	1,813	1,914	1,793	1,867	1,78	1,808	1,742	1,761	1,89	1,718	1,689	4,748	1,973	2,143	2,117	1,906	1,816	1,834	1,903	1,846	1,94	1,959
3f	5,726	3,47	2,815	3,704	2,66	2,693	2,747	2,666	2,672	2,632	4,038	2,627	2,624	2,486	4,207	2,737	3,767	3,573	2,816	2,916	2,027	1,639	3,258	2,072	1,871
4d	4,09	1,907	2	1,922	1,959	1,933	1,854	1,839	1,91	1,931	1,969	1,981	1,935	1,928	3,148	2,078	1,956	1,95	1,91	1,867	1,875	1,872	1,856	1,831	1,873
4f	5,774	2,864	2,702	2,767	2,815	2,72	2,766	2,827	4,004	2,986	3,502	2,802	4,172	2,812	2,758	3,228	2,734	2,665	2,807	4,702	2,653	3,693	2,865	2,73	2,65

Примітка: в стовпці «Q» 1-4 варіанти запитів, приставка d - з'єднання з базою даних безпосередньо, приставка f - з'єднання з базою даних через фаєрвол.

Таблиця 4.4 - Час виконання INSERT запитів

	Час виконання запиту в мілісекундах																							
Q	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1d	3,187	1,781	1,963	1,842	1,716	1,79	1,802	1,736	1,775	1,754	1,661	1,886	1,825	1,751	3,958	1,814	1,84	1,823	1,74	1,702	1,759	1,799	1,851	1,865	1,886
1f	4,68	2,388	2,668	2,466	2,54	2,319	2,326	2,406	2,4	2,415	2,467	2,531	2,296	2,403	3,054	2,504	2,42	2,324	2,478	2,476	2,406	2,384	2,411	2,542	2,408
2d	3,023	1,906	1,874	1,825	1,781	1,848	1,785	1,803	1,846	1,898	1,796	1,893	1,89	1,833	3,106	1,914	1,781	1,822	1,873	1.829	1,788	1,92	1,809	1,83	1,919
2f	4,276	2,203	2,195	2,196	2,196	2,233	2,17	2,231	2,132	2,022	2,257	2,147	2,368	2,371	4,506	2,562	2,649	2,299	2,342	2,431	2,518	2,329	2,445	3,467	2,326
3d	4,628	1,839	2,064	1,809	1,85	1,776	1,717	1,749	1,866	1,834	1,902	1,862	1,765	1,702	3,405	1,907	1,828	1,853	1,738	1,776	1,809	1,719	1,832	1,859	1,862
3f	4,605	2,493	2,47	2,382	2,59	2,463	2,489	2,434	3,368	2,472	2,354	2,895	2,551	2,354	4,677	2,636	3,141	2,256	2,325	2,303	2,287	2,289	2,159	2,451	2,919
4d	4,628	1,839	2,064	1,809	1,85	1,776	1,717	1,749	1,866	1,834	1,902	1,862	1,765	1,702	3,405	1,907	1,828	1,853	1,738	1,776	1,809	1,719	1,832	1,859	1,862
4f	5,633	2,685	2,438	2,344	2,338	2,43	2,271	2,525	2,25	2,339	2,389	2,326	2,178	2,482	5,373	2,446	2,172	2,466	2,327	2,486	2,309	2,276	2,199	2,322	2,442

Примітка: в стовпці «Q» 1-4 варіанти запитів, приставка d - з'єднання з базою даних безпосередньо, приставка f - з'єднання з базою даних через фаєрвол.

Таблиця 4.5 - Час виконання DELETE запитів

	Час виконання запиту в мілісекундах																							
Q	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1d	1,896	0,786	0,791	0,754	0,793	0,789	0,662	0,746	0,787	0,646	0,635	0,855	0,752	0,622	1,748	0,848	0,971	0,818	1,125	0,772	0,68	0,771	0,67	0,749	0,774
1f	2,824	1,233	1,35	1.504	1,29	1,147	1,042	1,315	1,305	1,212	1,11	1,172	1,922	1,224	2,346	1,285	1,335	2,022	1,712	1,332	1,303	1,517	1,249	1,356	1,288
2d	1,784	0,804	0,809	0,788	0,788	0,751	0,611	0,848	0,786	0,649	0,569	0,73	0,814	0,738	1,167	0,876	0,757	0,783	0,767	0,853	0,828	0,916	0,886	0,873	0,82
2f	3,007	1,2	1,633	1,203	1,207	1,171	1,159	1,186	1,174	1,126	1,207	1,533	1,238	1,173	3,421	1,468	1,362	1,373	1,37	1,296	1,306	1,288	1,546	2,034	1,44
3d	2,707	0,837	0,874	0,801	0,836	0,753	0,762	0,83	0,759	0,727	0,65	0,741	1,169	0,82	2,316	0,837	0,962	0,863	0,84	0,909	0,941	0,861	0,814	0,883	0,89
3f	3,780	1,288	1,514	1,197	1,237	1,159	1,26	1,317	1,183	1,136	1,191	1,553	1,187	1,076	2,422	2,111	2,87	1,453	1,213	1,321	1,214	1,252	1,191	2,261	1,438
4d	2,977	0,877	0,808	0,793	0,897	0,787	0,885	1,024	0,854	0,808	0,689	0,817	0,793	0,711	1,348	1,048	0,783	0,814	0,708	0,752	0,751	0,781	0,714	0,875	0,805
4f	3,488	1,167	1,27	1,181	1,209	1,885	1,24	1,204	1,38	1,292	1,376	1,339	1,343	1,701	2,868	1,239	1,383	1,73	1,296	1,293	1,376	1,571	1,289	1,505	1,627

Примітка: в стовпці «Q» 1-4 варіанти запитів, приставка d - з'єднання з базою даних безпосередньо, приставка f - з'єднання з базою даних через фаєрвол.

Зображення графіків (рисунки 4.9, 4.10, 4.11) кожного типу запитів, використовуючи середнє арифметичне значення отриманих середніх узагальнених результатів часу виконання.

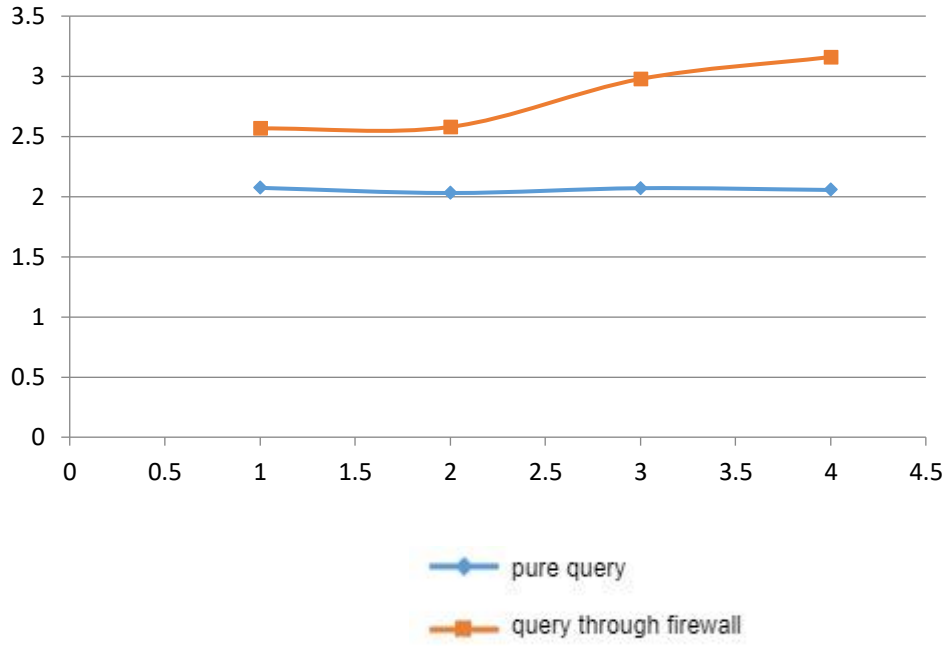

Рисунок 4.9 - Графік середніх значень варіантів запитів типу SELECT. Вісь Y - мілісекунди, вісь X - номер запиту

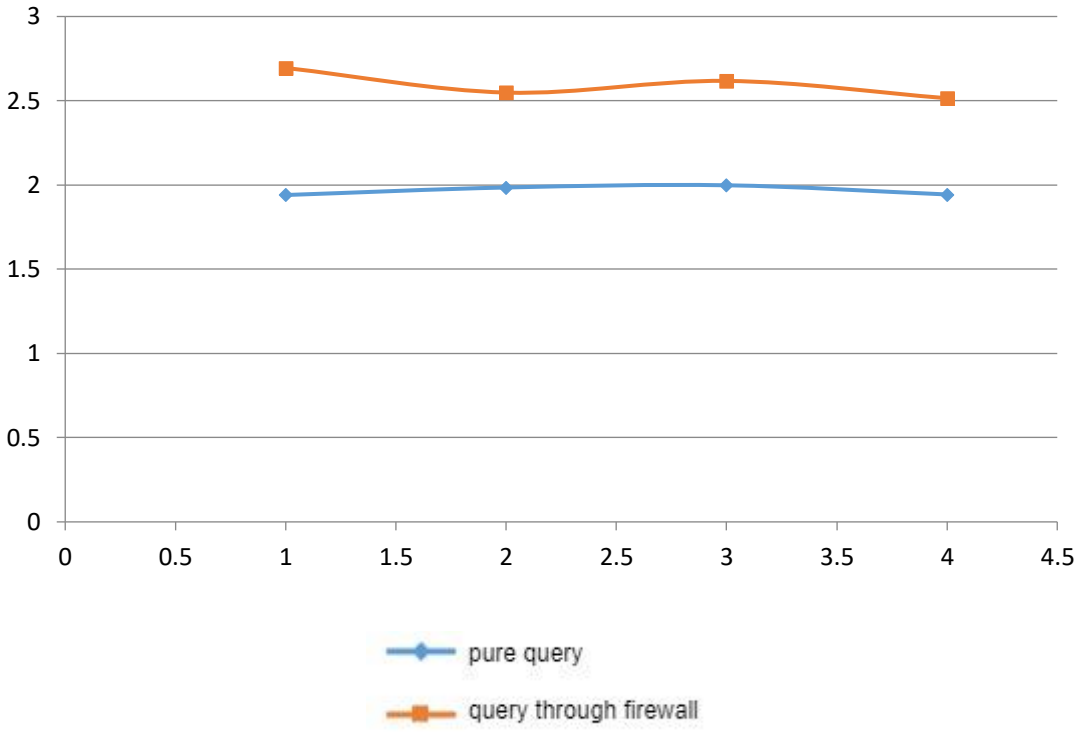

Рисунок 4.10 - Графік середніх значень варіантів запитів типу INSERT. Вісь Y - мілісекунди, вісь X - номер запиту

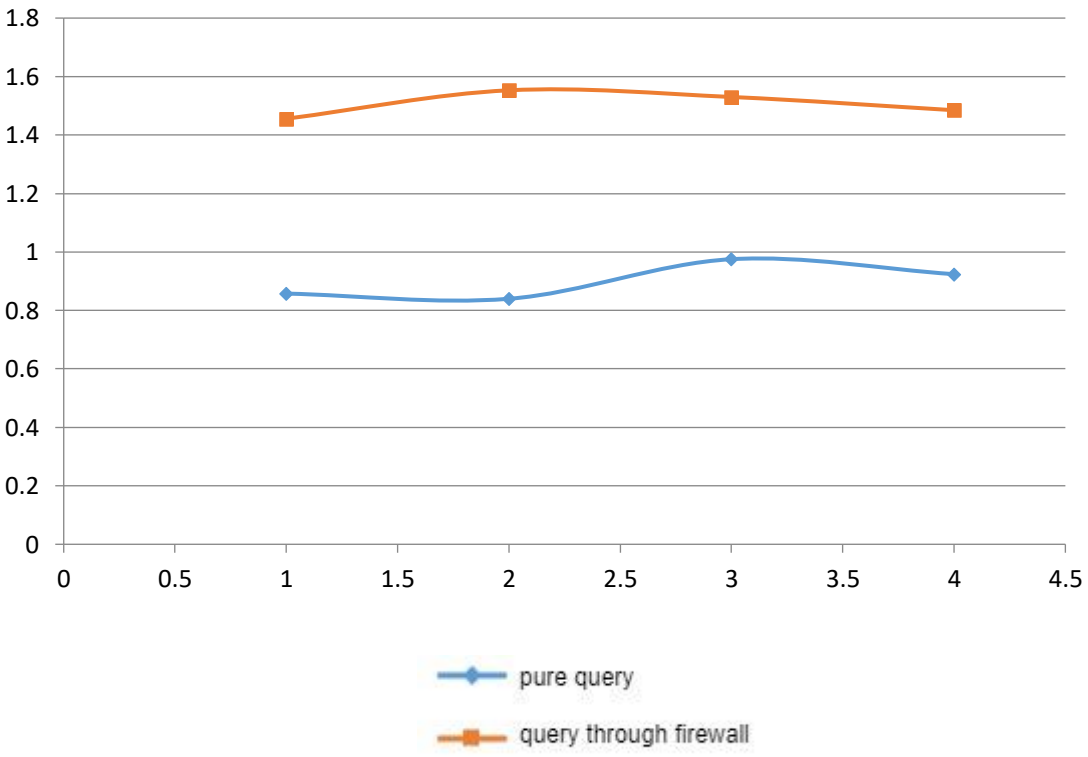

Рисунок 4.11 - Графік середніх значень варіантів запитів типу DELETE. Вісь Y - мілісекунди, вісь X - номер запиту

Запити, які виконуються через фаєрвол, відбуваються повільніше, ніж безпосередньо. Це обумовлюється аналізом SQL-запитів. Однак ця різниця не велика. При проведенні тестування для запитів:

- SELECT різниця виконання для 1, 2, 3, 4 запитів становить 0.49388, 0.54924, 0.908 і 1.10496 мілісекунд відповідно, що в середньому становить 0.76402 мілісекунди.
- INSERT різниця виконання для 1, 2, 3, 4 запитів становить 0.75428, 0.56352, 0.6198 і 0.57116 мілісекунд відповідно, що в середньому становить 0.62719 мілісекунди.
- DELETE різниця виконання для 1, 2, 3, 4 запитів становить 0.5982, 0.71316, 0.5548 і 0.56088 мілісекунд відповідно, що в середньому становить 0.60676 мілісекунди.

Виходячи зі сказаного вище, середня різниця часу виконання запитів дорівнює 0,66599 мілісекунди.

Висновки до розділу 4

В процесі виконання роботи були в повній мірі вирішені поставлені завдання:

- проаналізовані особливості впровадження операторів SQL (SQL-ін'єкції),
- проаналізовані методи виявлення аномалій в SQL-запитах до баз даних,
- проаналізовані методи захисту від такого типу атак,
- проаналізовані особливості операторів в таких популярних СУБД як: MsSQL Server, MySQL, MariaDB, PostgreSQL.

Також була досягнута мета роботи, а саме реалізовано програмний фаєрвол для таких СУБД як: MsSQL Server, MySQL, MariaDB, PostgreSQL.

У підсумку, фаєрвол успішно справляється із своїм завданням, при цьому навантаження на базу даних - незначне. Час виконання запитів збільшується мінімально.

ВИСНОВКИ

У першому розділі були розглянуті основні типи SQL ін’єкцій, їх особливості реалізації щодо різних СКБД, порівняння наявних коммерційних фаєрволів, а також сучасні методи обходу WAF. Також було виявлено, що наявні методи реалізації фаєрволів – недосконалі, звідки випливає актуальна проблема пошуку методів захисту від існуючих вразливостей.

У другому розділі були досліджені додаткові методи виявлення та запобігання SQL ін’єкціям. Також було обговорено поєднання ін’єкцій із іншими типами атак для поглибленого розуміння методів та засобів захисту. У результаті досліджень було зроблено висновок, що наявні засоби захисту не в повній мірі забезпечують захист від наведених типів атак і потребують подальшого доопрацювання. У ході аналізу мережевих протоколів - виявлено, що ідеальним для реалізації проксуючого фаєрволу та його подальшого підняття на проксі-сервері вирішенням, є SOCKS 4.

У третьому розділі була побудована модель та описана структура практичної реалізації майбутнього фаєрволу, а також були висунуті основні вимоги щодо самої програми і методу її реалізації. Аналізуючи вибір технології розрбки, було досліджено, що перевагою подібного способу виконання програми є повна незалежність байт-коду від операційної системи і устаткування. Також були проаналізовані драйвери для коректного зв’язку середовища розробки та баз даних. Враховуючи широкий спектр баз даних, для котрих розробляється практична реалізація і кількість операційних систем, на котрих вони базуються було прийнято рішення використовувати JDBC інтерфейс.

У четвертому розділі були в повній мірі вирішені завдання аналізу особливостей впровадження операторів SQL (SQL-ін'єкції), аналізу методів виявлення аномалій в SQL-запитах до баз даних, аналізу методів захисту від

такого типу атак, а також аналізу особливостей операторів в СУБД. Також, у даному розділі, була досягнута мета роботи.

У підсумку, фаєрвол успішно справляється із своїм завданням, при цьому навантаження на базу даних - незначне. Час виконання запитів збільшується мінімально.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. AWS Documantation [Електронний ресурс]. - Режим доступу URL: https://docs.aws.amazon.com/index.html?nc2=h_ql_doc
2. Free CDN to Speed Up and Secure WebSite [Електронний ресурс]. - Режим доступу URL: https://geekflare.com/free-cdn-list/
3. OWASP CheatSheetSeries [Електронний ресурс]. - Режим доступу URL: https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.md
4. SQL Injections Top Attack Statistics [Електронний ресурс]. - Режим доступу URL: https://www.darkreading.com/risk/sql-injections-top-attack-statistics/d/d-id/1132988
5. Best Practices to prevent SQL-injections [Електронний ресурс]. - Режим доступу URL: https://tableplus.io/blog/2018/08/best-practices-to-prevent-sql-injection-attacks.html
6. Akamai Report. The State of the Internet [Електронний ресурс]. - Режим доступу URL: https://www.akamai.com/us/en/resources/our-thinking/state-of-the-internet-report/global-state-of-the-internet-security-ddos-attack-reports.jsp
7. Cloudflare Business Plan [Електронний ресурс]. - Режим доступу URL: https://www.cloudflare.com/plans/business/
8. Cloud Web Application Firewall [Електронний ресурс]. - Режим доступу URL: https://www.cloudflare.com/waf/
9. Libinjection [Електронний ресурс]. - Режим доступу URL: https://github.com/client9/libinjection
10. SQL-injections: vulnerabilities and how to prevent attacks [Електронний ресурс]. - Режим доступу URL: https://www.veracode.com/security/sql-injection

ДОДАТКИ

ДОДАТОК А

Main.java

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
public class Main {
    public static void main(String[] args) {
        Main m = new Main();
        m.testDatabase();
    }
    private void testDatabase() {
        try {
            String url = "jdbc:mysql://localhost:6666/test";
            String login = "root";
            String password = "00000000";
            Connection con = DriverManager.getConnection(url, login, password);
            try {
                Statement stmt = con.createStatement();
                System.out.println("OK");
                stmt.executeUpdate("INSERT INTO pet VALUE ('BUD')");
                //ResultSet rs = stmt.executeQuery("INSERT INTO pet VALUE
('BUD')");
                //while (rs.next()) {
```

```
                //    String str = rs.getString("name") + ":" + rs.getString(1);
                //    System.out.println("Contact:" + str);
               // }
               // rs.close();
                stmt.close();
            } finally {
                con.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

DanjerousSQLInserts.java

```
public class DangerousSQLInserts {
    public int getSelectWeight(String query){
        String[] extraDangerousCharacters = { "--", "#" , ";"};
        String[] dangerousCharacters = {"<", ">", "*", "/" };
        String[] extraDangerousStrings = { "UNION", "ALL", "DROP", "INSERT",
"DELETE", "TRUNCATE", "UPDATE", "ALTER", "EXEC"};
        String[] dangerousStrings = {"NULL", "COUNT", "LIKE", "INTO"};
        int weight = 0;
        for (String edc : extraDangerousCharacters)
        {
            if (query.contains(edc))
                weight += 200;
        }
        for (String dc : dangerousCharacters)
        {
```

```
            if (query.contains(dc))
                weight += 35;
        }
        for (String eds : extraDangerousStrings)
        {
            if (query.contains(eds))
                weight += 200;
        }
        for (String ds : dangerousStrings)
        {
            if (query.contains(ds))
                weight += 100;
        }
        return weight;
    }
    public int getInsertWeight(String query){
        String[] extraDangerousCharacters = { "--", "#" , ";"};
        String[] dangerousCharacters = {"=", "<", ">", "*", "/"};
        String[] extraDangerousStrings = { "UNION", "ALL", "DROP", "SELECT",
"DELETE", "TRUNCATE", "UPDATE", "ALTER", "EXEC"};
        String[] dangerousStrings = {"NULL", "COUNT", "LIKE"};
        int weight = 0;
        for (String edc : extraDangerousCharacters)
        {
            if (query.contains(edc))
                weight += 200;
        }
        for (String dc : dangerousCharacters)
        {
            if (query.contains(dc))
                weight += 35;
        }
        for (String eds : extraDangerousStrings)
        {
            if (query.contains(eds))
```

```
                weight += 200;
        }
        for (String ds : dangerousStrings)
        {
            if (query.contains(ds))
                weight += 100;

        return weight;
    }
    public int getDeleteWeight(String query){
        String[] extraDangerousCharacters = { "--", "#" , ";"};
        String[] dangerousCharacters = {"=", "<", ">", "*", "/"};
        String[] extraDangerousStrings = { "UNION", "ALL", "DROP", "SELECT",
"INSERT", "TRUNCATE", "UPDATE", "ALTER", "EXEC"};
        String[] dangerousStrings = {"NULL", "COUNT", "LIKE", "VALUES", "INTO"};
        int weight = 0;
        for (String edc : extraDangerousCharacters)
        {
            if (query.contains(edc))
                weight += 200;
        }
        for (String dc : dangerousCharacters)
        {
            if (query.contains(dc))
                weight += 35;
        }
        for (String eds : extraDangerousStrings)
        {
            if (query.contains(eds))
                weight += 200;
        }
        for (String ds : dangerousStrings)
        {
            if (query.contains(ds))
                weight += 100;
```

```
        }
        return weight;
    }
    public int getUpdateWeight(String query){
        String[] extraDangerousCharacters = { "--", "#" , ";"};
        String[] dangerousCharacters = {"<", ">"};
        String[] extraDangerousStrings = { "UNION", "ALL", "DROP", "SELECT",
"INSERT", "TRUNCATE", "DELETE", "ALTER", "EXEC"};
        String[] dangerousStrings = {"NULL", "COUNT", "LIKE", "VALUES", "INTO"};
        int weight = 0;
        for (String edc : extraDangerousCharacters)
        {
            if (query.contains(edc))
                weight += 200;
        }
        for (String dc : dangerousCharacters)
        {
            if (query.contains(dc))
                weight += 35;
        }
        for (String eds : extraDangerousStrings)
        {
            if (query.contains(eds))
                weight += 200;
        }
        for (String ds : dangerousStrings)
        {
            if (query.contains(ds))
                weight += 100;
        }
        return weight;
    }
    public int getOtherWeight(String query){
        String[] extraDangerousStrings = {"DROP", "TRUNCATE", "ALTER", "EXEC"};
        int weight = 0;
```

```
        for (String eds : extraDangerousStrings)
        {
            if (query.contains(eds))
                weight += 300;
        }
        return weight;
    }
    public int getAuthWeight(String query){
        String[] extraDangerousStrings = {"OR", "=", "'", "1", "-"};
        int weight = 0;
        for (String eds : extraDangerousStrings)
        {
            if (query.contains(eds))
                weight += 300;
        }
        return weight;
    }
}
```

ДОДАТОК Б

MSSQLParser.java

```
public class MsSQLPackParser {
    private byte[] packet;
    public String getQuery(byte[] pack){
        String query;
        this.packet = pack;

        if (pack[0] == 0x10){
            query = "auth|" + authQuery();
            return query;
        } else if (pack[0] == 0x1){
```

```
        query = query();
        return query;
    }else{
        return "Service packet";
    }
}
public String authQuery (){
    String auth = "";
    byte[] b = new byte[2];
    b[0] = packet[40];
    b[1] = packet[41];
    int usrnamePos = ((0xFF & b[0]) << 8) | (0xFF & b[1]);
    b[0] = packet[42];
    b[1] = packet[43];
    int usrnameLen = ((0xFF & b[0]) << 8) | (0xFF & b[1]);
    for (int i = usrnamePos; i < usrnameLen; i++){
        auth += (char) packet[i];
    }
    return auth;
}
public String query (){
    String query = "";
    String temp = "";
    String query_type = "";
    int j = 30;

    if (packet[j] == 0xA) {
        j++;
        while (packet[j] != 0xA) {
            if (packet[j] != 0x0)
                temp += (char) packet[j];
            j++;
        }
    }
    j = 0;
```

```
        while (temp.charAt(j) != '\u0020') {
            query_type += temp.charAt(j);
            j++;
        }
        query += query_type + "|" + temp;
        return query;
    }
}
```

MySQLParser.java

```
public class MySQLPackParser {
    private byte[] packet;
    private boolean firstPacket = true;
    private boolean secondPacket = true;
    private boolean thirdPacket = true;
    public String getQuery(byte[] pack){
        String query;
        this.packet = pack;
        if (firstPacket){
            query = "auth|" + authQuery();
            firstPacket = false;
            return query;
        } else if (secondPacket){
            secondPacket = false;
            return "Service packet";
        } else if (thirdPacket){
            thirdPacket = false;
            return "Service packet";
        } else {
            query = query();
            return query;
        }
    }
```

```
    private String authQuery (){
        String auth = "";
        int i = 36;

        while (packet[i] != '\u0000'){
            auth += (char) packet[i];
            i++;
        }
        return auth;
    }
    private String query (){
        String query = "";
        String query_type = "";
        int query_length = packet[0];
        int j = 5;
        while (packet[j] != '\u0020') {
            query_type += (char) packet[j];
            j++;
        }
        query += query_type + "|";
        j = 5;
        while (j != (query_length + 5)){
            query += (char) packet[j];
            j++;
        }
        return query;
    }
}
```

PostgreSQLParser.java

```
public class PGSQLPackParser {
    private byte[] packet;
    private boolean firstPacket = true;
```

```
public String getQuery(byte[] pack){
    String query;
    this.packet = pack;
    if (firstPacket){
        query = "auth|" + authQuery();
        firstPacket = false;
        return query;
    } else if (pack[0] == '\u0051'){
        query = query();
        return query;
    }else{
        return "Service packet";
    }
}
public String authQuery (){
    String auth = "";
    int i = 13;
    while (packet[i] != '\u0000'){
        auth += (char) packet[i];
    }
    return auth;
}
public String query (){
    String query = "";
    String query_type = "";
    byte[] b = new byte[4];
    for (int i = 0; i < b.length; i++){
        b[i] = packet[i + 1];
    }
    int query_length = ((0xFF & b[0]) << 24) | ((0xFF & b[1]) << 16) |
            ((0xFF & b[2]) << 8) | (0xFF & b[3]);
    query_length -=4;
    int j = 5;
    while (packet[j] != '\u0020') {
        query_type += (char) packet[j];
```

```
            j++;
        }
        query += query_type + "|";
        j = 5;
        while (j != query_length){
            query += (char) packet[j];
            j++;
        }
        return query;
    }
}
```