Хмельницький національний університет
Факультет інформаційних технологій
Кафедра кібербезпеки

КВАЛІФІКАЦІЙНА РОБОТА

Галки Артура Олексійовича

на здобуття ступеня вищої освіти Бакалавра

Система захисту конфіденційної інформації в СКБД MS SQL від sql-атак

Галузь знань 12 – Інформаційні технології
Спеціальність 125 – Кібербезпека
Освітня програма Кібербезпека

Шифр КРБКБ. 2101113.21.01 ПЗ

Виконав студент 4 курсу група КБ-21-1 Артур ГАЛКА
Керівник канд. техн. наук, доцент Володимир ДЖУЛІЙ
Нормоконтролер старший викладач Сергій МОСТОВИЙ

До захисту допускаю:
Завідувач кафедри кібербезпеки Юрій КЛЬОЦ
9 06 2025 р.

Хмельницький 2025

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Кібербезпеки

Рівень вищої освіти Бакалавр

Галузь знань 12 – Інформаційні технології

Спеціальність 125 – Кібербезпека

Освітня програма Кібербезпека

ЗАТВЕРДЖУЮ

Завідувач кафедри кібербезпеки

Юрій КЛЬОЦ

15 лютого 2025 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Галці Артуру Олексійовича

1 Тема роботи Система захисту конфіденційної інформації в СКБД MS SQL від sql-атак

Керівник роботи Джулій Володимир Миколайович

Затверджено наказом ректора університету від 07 лютого 2025 № 23

2 Строк подання студентом кваліфікаційної роботи на кафедру

3 Вихідні дані до роботи Система розробляється як навчально-дослідний прогамний комплекс для виявлення та запобігання SQL-ін'єкціям у веб-додатках, з можливістю демонстрації вразливостей, їх експлуатації та методів захисту

4 Зміст пояснювальної записки (перелік питань, які потрібно розробити)
Огляд сучасних загроз інформаційній безпеці, пов'язаних в SQL-ін'єкціями; класифікація типів SQL-ін'єкцій та їх механізмів; аналіз конфіденційної інформації як об'єкта захисту в СКБД; опис проєкту системи для виявлення та запобігання SQL-ін'єкцій; моделювання атак та оцінка ефективності впровадження засобів захисту.

5 Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Копія графічної частини
Код програмного забезпечення

6 Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7 Дата видачі завдання 16 лютого 2024 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
Вибір і затвердження теми кваліфікаційної роботи	Січень	
Ознайомлення з предметною областю	Січень	
Дослідження існуючих рішень	Лютий	
Постановка задачі	Лютий	
Визначення загальних принципів рішення задачі	Лютий	
Деталізація принципів рішення задачі	Березень	
Впровадження системи	Березень	
Апробація проєктних рішень	Березень	
Оформлення пояснювальної записки згідно вимог	Квітень	
Оформлення графічної частини	Квітень	
Захист КР	Травень	

Студент _____ Артур ГАЛКА

Керівник кваліфікаційної роботи _____ Володимир ДЖУЛІЙ

АНОТАЦІЯ

Тема кваліфікаційної проекту: «Система захисту конфіденційної інформації в СКБД MS SQL від sql-атак»

Автор роботи: Галка Артур Олексійович

Керівник роботи: канд. техн. наук, доц. Джулій Володимир Миколайович

Загальний обсяг роботи: 72 сторінки., 2 додатки, 17 рисунків, 45 посилань.

Ключові слова: СКБД, SQL injection, Blockchain, Rate limiter, Smart contract, транзакція, Key Vault, Secrets manager, Docker, Ganache

Метою даного дипломного проекту є розробка надійної та сучасної системи захисту конфіденційної інформації в базах даних MsSql від sql-ін'єкцій. Основні цілі дипломного проекту включають розробку програмного забезпечення, яке здатне виявляти потенційно шкідливі sql запити, та обробляти їх належним чином, який надає змогу беззаперечного доказу факту спроби здійснення sql-ін'єкції. Окрім основного програмного засобу, передбачається розробку ряду допоміжних утиліт, та формування загальних рекомендацій, які допоможуть усунути вразливість.

Розроблене програмне забезпечення необхідно розгорнути, зробити загальнодоступним, а також протестувати, шляхом проведення спроб всіх видів sql-ін'єкцій.

25.05.2025 _____

ANNOTATION

Theme of qualification work: Confidential Information Protection System in MS SQL DBMS from SQL Attacks

Author: Halka Artur Oleksiiovych

Mentor: Ph.D in Technical Sciences, Associate Professor Dzhulii Volodymyr Mykolaiovych

Total Length: 66 pages, 2 appendices, 17 figures, 45 links.

Keywords: DBMS, SQL injection, Blockchain, Rate limiter, Smart contract, transaction, Key Vault, Secrets manager, Docker, Ganache

The aim of this diploma project is to develop a reliable and modern system for protecting confidential information in MS SQL databases from SQL injections. The main objectives of the project include the development of software capable of detecting potentially harmful SQL queries and handling them appropriately, providing undeniable proof of SQL injection attempts.

In addition to the main software solution, the project also envisions the development of several auxiliary utilities and the formation of general recommendations to help eliminate vulnerabilities.

The developed software must be deployed, made publicly accessible, and tested through attempts to perform all known types of SQL injection attacks.

25.05.2025 _____

ЗМІСТ

Вступ 7
1 Аналіз загроз та захист інформації в базах даних 8
1.1 Аналіз та дослідження механізмів шифрування в MS SQL Server 8
1.2 Цілісність, доступність, конфіденційність інформації в базах даних 11
1.3 Засоби та інструменти захисту конфіденційної інформації в СКБД MS SQL від sql-атак 23
1.4 Постановка задачі 27
2 Проєктування системи захисту інформації в базах даних sql від SQL-атак 28
2.1 Кібератаки на конфіденційність інформації в базах даних та її захист 28
2.2 Проєктування плейсхолдерів як основний засіб захисту даних від SQL-ін'єкцій 33
2.3 Інтеграція блокчейн-технології для підвищення ефективності захисту даних від sql-ін'єкцій 37
2.4 Висновки 47
3 Розробка програмного забезпечення для виявлення SQL-ін'єкцій у додатках . 48
3.1 Розробка варіантів використання програмного забезпечення системи захисту даних від sql-атак 48
3.2 Розробка логічної та фізичної структури програмного забезпечення системи захисту даних від sql-атак 50
3.3 Розгортання та тестування програмного забезпечення системи виявлення SQL ін'єкцій у додатках 57
3.4 Висновки 66
Висновки 67
Список джерел посилань 68
Додаток А Копія графічної части 73
Додаток Б Код програмного забезпече 76

Зм.	Арк.	№докум.	Підпис	Дата	КРБКБ.2101113.21.01 ПЗ		
Виконав		Галка А.О.		25.06.25	Система захисту конфіденційної інформації в СКБД MS SQL від sql-атак Пояснювальна записка	Літера	Аркуш 6 / Аркушів 72
Перевір.		Джулій В.М.		25.06.25			
Н.контр.		Мостовий С.В		25.06.25		ХНУ, КБ-21-1	
Затвер.		Кльоц Ю.П.		25.06.25			

ВСТУП

Одним із найцінніших ресурсів сучасного суспільства є інформація. Захист конфіденційної інформації є життєво важливим у сфері цифрових технологій. Оскільки витоки даних і кібератаки стають все більш частими та складними, зростає і пріорітет захисти персональних і корпоративних даних.

При розробці будь-якого програмного продукту, який обробляє конфіденційні дані, завжди існує ризик втрати даних або несанкціонованого доступу. Навіть технологічні гіганти з передовими заходами безпеки вразливі. Кібератаки часто спрямовані на бази даних, оскільки вони використовуються для зберігання, обробки та керування даними. Приклад такої вразливості стався в липні 2012 року, коли Yahoo став жертвою атаки, яка призвела до викрадення понад 400 000 паролів і особистих даних користувачів. Подібним чином у серпні 2011 року мережа розробників Nokia була зламана, розкривши конфіденційну інформацію користувачів. В обох інцидентах для використання недоліків безпеки в системі використовувалися різні методи впровадження SQL-ін’єкцій.

SQL-ін’єкція залишається одним із найпоширеніших і небезпечних векторів атак, здатних обійти традиційні механізми безпеки. Цей тип уразливості дозволяє зловмисникам виконувати довільний код SQL, уможливлюючи будь-які дії, від доступу та маніпулювання конфіденційними даними, до виконання довільних скриптів на пристрої, де розташований сервер. Наприклад, якщо зловмисник тримає доступ до бази та скомпроментує свої права, він зможе отримати доступ до процедури «xp_cmdshell», за допомогою якої він зможе працювати з командним рядком пристрою. Використовуючи цю команду, зловмисник може отримати доступ до RDP (Remote Desktop Protocol), та за допомогою нього віддалено підключитись до пристрою.

Метою цієї дипломної роботи є аналіз наявних загроз і ризиків, пов’язаних з атаками SQL-ін’єкцій, аналіз існуючих механізмів захисту, доступних для MS SQL Server, і розробка практичних рекомендацій щодо впровадження заходів безпеки для захисту від цих типів атак.

1 АНАЛІЗ ЗАГРОЗ ТА ЗАХИСТ ІНФОРМАЦІЇ В БАЗАХ ДАНИХ

1.1 Аналіз та дослідження механізмів шифрування в MS SQL Server

СКБД MS SQL — це система керування реляційними базами даних, яка широко використовується організаціями для зберігання та управління даними веб-додатків. Розуміння архітектури та функціональності MS SQL є критичним для ефективного вирішення проблем безпеки.

MS SQL працює як надійна система для обробки та зберігання великих обсягів даних. Вона підтримує мову структурованих запитів (SQL) для управління та запиту даних, що є основою для багатьох веб-додатків. Також SQL надає широкий спектр функцій для захисту даних, оптимізації продуктивності та масштабованості.

Для захисту даних можна використати шифрування даних; захист на основі ролей, використовуючи стратегію «найменшого доступу», до користувач має мінімальні права доступу, які необхідні для його роботи; проведення аудитів. Також можна використати Transparent Data Encryption. Це технологія, що дозволяє шифрування даних файлів, логи та резервні копії. Принцип роботи технології Transparent Data Encryption полягає у тому, що при збережені інформації автоматично відбувається процес шифрування, а в момент, коли відбувається доступ до неї – інформація розшифровується. [1] Це корисно в випадках, якщо зловмисник отримав доступ до файлів чи резервних копій, оскільки він всеодно не зможе зчитати цю інформацію, оскільки не володіє ключем шифрування, який генерується в момент установки та налаштування MsSql сервера. Алгоритм шифрування даних та генерації ключа шифрування зображені на рисунку 1.1.

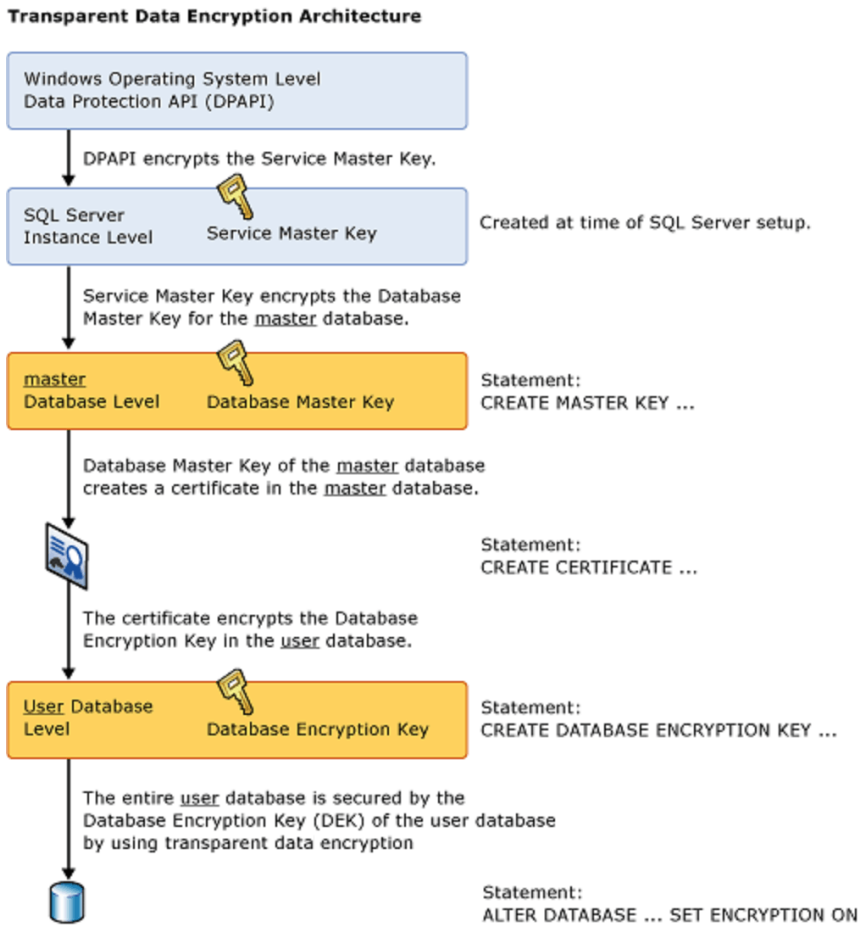


Рисунок 1.1 – Архітектура Transparent Data Encryption

Для шифрування конфіденційної інформації, такої як номера та паролі кредитних карток, адреси проживання, номери телефону і тд, можна використати Always Encrypted . Це дозволяє нам зашифрувати певний стовпець в таблиці і забезпечує поділ між тими, хто володіє даними, і може їх переглядати, і тими, хто керує даними, але не має доступу до них. Прикладом тих, хто керує даними, але не має досутпу до їх перегляду можуть бути локальні адміністратори баз даних, оператори хмарних баз даних, чи інші неавторизовані користувачі з високим рівнем доступу. Хоча Always Encrypted і забезпечує захищеність конфіденційних даних, він також має певні недоліки, наприклад, неможливість виконувати операції сортування, фільтрування, групування та операцію Distinct з шифрованими даними. [2]

Шифруванню не підлягають стовпці з наступними типами:

– xml;

– timestamp;

– rowversion;
– image (Blob);
– ntext;
– sql_variant;
– hierarchyid;
– geography;
– geometry, alias;
– user-defined types.

Щоб частково уникнути обмежень, які висуває Always Encrypted, існує Always Encrypted with secure enclaves. Використання цієї технології дає нам можливість виконувати певні обчислення над текстовими даними всередині безпечного анклаву на стороні сервера. Безпечний анклав – це захищена пам'ять в процесі Database Engine. Інші процеси на пристрої (включаючи інші DE процеси) бачать безпечний анклав як «непрозору» ділянку, та не мають можливість переглянути чи перезаписати дані всередині анклаву. Навіть використовуючи дебагер, потрапити в межі безпечного анклаву неможливо. Це робить його надійним середовищем виконання, яке може отримувати доступ до криптографічних ключів чи конфіденційної інформації користувачів у відкритому вигляді, без ризику бути втраченими чи перехопленими [3]. Описаний алгоритм роботи можемо розглянути на рисунку 1.2.

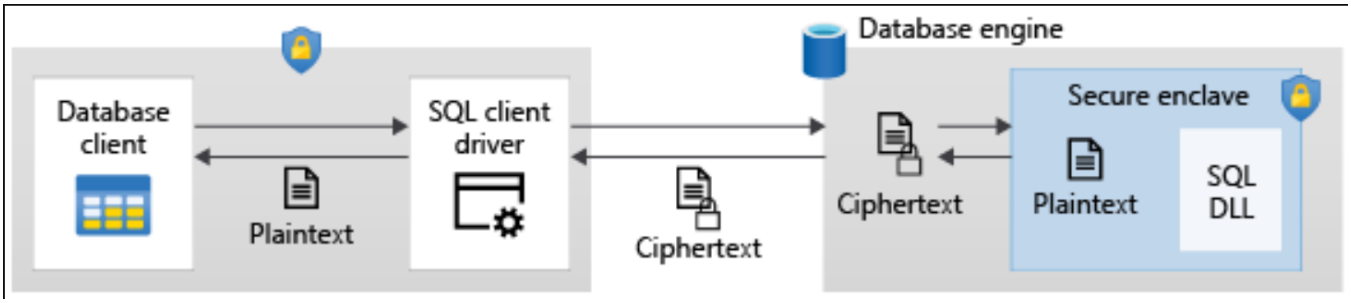


Рисунок 1.2 – Алгоритм роботи Always Encrypted з безпечним анклавом

MS SQL можна інсталювати на Windows, Linux, розгорнути в Docker контейнері, або розгорнути в Azure Virtual Machine [4].

1.2 Цілісність, доступність, конфіденційність інформації в базах даних

Конфіденційність передбачає в собі обмеження доступу до даних виключно уповноваженим персоналом. Прості користувачі чи посторонні особи не повині мати доступ до всієї інформаційної бази. Для доступу до інформаційної бази користувачі повинні пройти процес автентифікації. Це можна зробити за допомогою введення логіна та пароля або інших методів ідентифікації, таких як використання ID-картки, двофакторної автентифікації (2FA), біометричних даних (відбиток пальця, розпізнавання обличчя) або одноразових кодів. Ні в якому разі не можна повідомляти свій логін чи передавати ключ-ідентифікатор сторонім особам, чи навіть своїм колегам. В разі використання логіна та пароля як способу ідентифікації, варто дотримуватись політики паролів. Найбільш поширеними рекомендаціями щодо створення надійного паролю є:

- Довжина повинна бути від 8 до 12 символів;
- містить символи верхнього регістру;
- містить символи нижнього регістру;
- містить цифри;
- містить спеціальні символи;
- не використовується в інших системах;
- не повинен бути іменем, прізвищем, назвою компанії, логіном або будь-яким іншим справжнім словом, яке можна знайти в словнику;
- не допускається використання повторюваних символів чи послідовностей символів; [5]

Надійний пароль це лише половина захисту, так як навіть надійний пароль лише збільшує час, який потребується для взлому такого паролю. Для досягнення максимальнох безпеки, варто використовувати двохфакторну автентифікацію. [6] Таким чином, навіть ввівши правильний логін та пароль, ми не зможемо ввійти в свій аккаунт, поки повторно не підтвердимо свою особу. Підтвердиит свою особу можна через СМС, через push-сповіщення (за умови наявності додатку,

наприклад Authenticator), через голосове підтвердження, через одноразові коди чи спеціальні носії.

Цілісність інформації є критично важливою незалежно від сфери діяльності, в якій ця інформація використовується. Забезпечення цілісності інформації означає, що інформація не була спотворена, скомпроментована, та зберігається саме в тому вигляді, в якому була записана, тобто має бути забезпечено незмінність та достовірність даних протягом всього їх життєвого циклу. Спотворення інформації може спричинити значні збитки як грошові, так і в цілому для репутації компанії. Як приклад, зловмисник може скомпоментувати та підмінити інформацію про кредитну карту одного з клієнтів в базі даних на свій рахунок, та таким чином перенаправити всі платежі собі. Також цілісність даних може бути порушена внаслідок помилки працівника. Доволі часто зустрічається ситуація, коли при написанні SQL-запитів, розробник випадково забуває додати умову WHERE під час виконання операції UPDATE. Таким чином, замість того, щоб оновити лише необхідні записи, оновлюються всі записи в таблиці. Якщо таке відбудеться на продакшен сервері, то уникнути збитків буде неможливо. Щоб уникнути ситуацій, коли розробник може випадково спричинити проблеми з цілісністю даних, використовуються окремі сервери, які містять копію даних, але при цьому призначені лише для тестування. Навіть якщо дані будуть спотворені в тестовому середовищі, це не спричинить жодних проблем. [7]

Найкращим рішенням для вирішення проблеми цілісності інформації є створення резервних копій. Резервні копії дають можливість «відкотити» зміни в базі до періоду, коли було проведено резервне копіювання. Таким чином, навіть якщо цілісність інформації було порушено, ми маємо можливість це виправити. Можна виокремити найважливіші типи резервних копій:

- Copy-only backup;
- full backup;
- differential backup;
- transaction log backups;
- tail-log backup;

– full file backup;

– partial backup;

Copy-only backup – створює копію бази, не впливаючи на ланцюжок наявних копій. Цей тип копіювання не змінює точку останього резервного копіювання. Це може бути корисно, якщо копію бази терміново необхідно передати іншій команді, чи перенести на інший пристрій. Також copy-only може бути застосований для копіювання журналу транзакцій. Аналогічно, створюється повна копія журналу транзакцій, не впливаючи на точку останього резервного копіювання та без змін в журнал транзакцій; [8]

Full backup – створює повну копію бази даних. Зазвичай таке копіювання виконується лише один раз, при створені першого копіювання, оскільки створювати повну копію щоразу занадто затратно по ресурсам. Повна копія включає в себе всі дані, схему бази та лог-журнали. Повна копія є основою для подальших копій; [9]

Differential backup – часткове резервне копіювання. Цей тип резервного копіювання захоплює всі зміни з моменту останнього повного копіювання. Часткове резервне копіювання створюється значно швидше, ніж повне, але при при цьому відновлення даних займе довше часу, оскільки відновлення буде відбуватись на основі двох копій (повної, та часткової). Розмір та швидкість відновлення даних, використовуючи часткові копії, залежить від об’єму даних, які було змінено від моменту останього повного резервного копіювання бази даних. Інтервали між частковими копіюваннями необхідно обирати, зважаючи на щоденний обсяг даних; [10]

Transaction log backups – резервне копіювання журналу транзакцій. Таке резервне копіювання може бути створеним лише за умови наявності повного резервного копіювання. За умови наявності повного резервного копіювання, ми можемо створити резервне копіювання журналу транзакцій в будь-який момент, за умови, що копіювання журналу транзакцій вже не в процесі створення. Копію журналу транзакцій варто створювати часто, наприклад раз в декілька годин. Копія журналу транзацій містить в собі дані, від моменту попереднього бекапу

журналу транзакцій (вийнятком є перше створення, при якому початковою точкою буде повна копія бази); [11]

Tail-log backup – фіксує будь-які записи журналу транзакцій, для яких ще не створено резервну копію. Це робиться, аби запобігти втраті даних та зберегти ланцюжок наявних копій незмінним. Таке резервне копіювання необхідно виконувати, перед відновленням бази, аби переконатись, що не буде втрачено жодних транзакцій. Tail-log backup має такі опції, як NORECOVERY та CONTINUE_AFTER_ERROR. NORECOVERY переводить базу в стан відновлення та гарантує, що база не зміниться після резервного копіювання журналу. CONTINUE_AFTER_ERROR варто використовувати лише в випадко, коли створюється резервна копія пошкодженої бази; [12]

Full file backup – резервна копія одного або групи файлів. Створення резервних копій та процес відновлення не залежить від інших резеврних копій та є автономним. Якщо хоча б один файл з файлової групи перебуває в офлайн режимі, то копію файлової групи буде неможливо створити. Перевагою цього типу копіювання є те, що ми можемо відновити лише пошкоджені файли, замість того, щоб переписувати абсолютно всі, що значно збільшить швидкість відновлення в разі пошкодження файлів; [13].

Partial backup – часткове копіювання, призначене для використання в рамках простої моделі відновлення, щоб підвищити гнучкість резервного копіювання дуже великих баз даних, які містять одну або кілька груп файлів лише для читання. Часткові резервні копії корисні, коли потрібно виключити файлові групи, доступні лише для читання. Часткова резервна копія нагадує повну резервну копію бази даних, але часткова резервна копія не містить усіх файлових груп. Натомість для бази даних для читання й запису часткова резервна копія містить дані в основній групі файлів, кожній групі файлів для читання й запису та, за бажанням, один або кілька файлів, доступних лише для читання. Часткова резервна копія бази даних, доступної лише для читання, містить лише основну файлову групу; [14]

Для ефективного використання резервних копій, варто також розглянути наявні моделі відновлення. Моделі відновлення призначені для контролю та підтримання журналу транзакцій. Загалом існує три моделі відновлення:

– simple (проста);

– full (повна);

– bulk logged (масовий);

Simple – не вимагає наявність копій журналів транзакцій. Простий режим автоматично очищає журнал транзакцій, тому немає потреби в окремих резервних копіях журналу, тому операції, які потребують резервного копіювання журналу транзакцій, не підтримуються простою моделлю відновлення. Також через відсутність журналів логування, у разі збою, зміни, що були внесені після останьої резервної (повної або часткової) копії є незахищеними та будуть втраченими. Окрім цього, використовуючи просту модель відновлення, неможливо відкотити зміни до певного періоду в часі. Можливо відкотити лише до моменту останнього резервного копіювання;

Full – вимагає наявність копій журналів транзакцій. Оскільки журнали логування наявні, то зникає проблема з операціями, яка описана в Simple model. У разі збою роботи втрата даних не загрожує, за вийнятком ситуацій, коли пошкоджено хвіст журналу (tail-log). В такому разі буде втрачено дані починаючи з останього резервного копіювання журналу транзакцій. Використовуючи повну модель відновлення, можливо відкотити зміни до певного періоду в часі, якщо на цей момент було завершено всі процеси резервного копіювання;

Bulk logged – вимагає наявність копій журналів транзакцій. Ця модель є доповненням до повної моделі відновлення та дозволяє виконувати ефективні та швидкі операції масового копіювання. Ця модель оптимізує розмір журналу транзакцій, шляхом мінімального логування для більшості масових операцій. Якщо журнал транзакцій пошкоджений або масові операції відбувалися після останньої резервної копії журналу, зміни після цієї резервної копії будуть втраченими. Окрім цього, відновлення даних до певного періоду в часі не

підтримується. Можливе відновлення тільки до моменту завершення будь-якої резервної копії; [15]

Як приклад, розглянемо створення повної резервної копії для бази даних. Для цього, в MsSql необхідно підключитись до цієї бази даних, лівою кнопкою мишки натиснути на базу даних, для якої будемо робити резервне копіювання, відкрити контекстне меню, в цьому меню перейти в розділ «Tasks» і там обрати «Back Up…». Натиснувши на «Back Up…», відкриється вікно, у якому ми зможемо виконати ряд налаштувань резервного копіювання, що буде створено. Детальніше процес відкриття модального вікна, для створення резервного копіювання бази даних, зображено на рисунку 1.3.

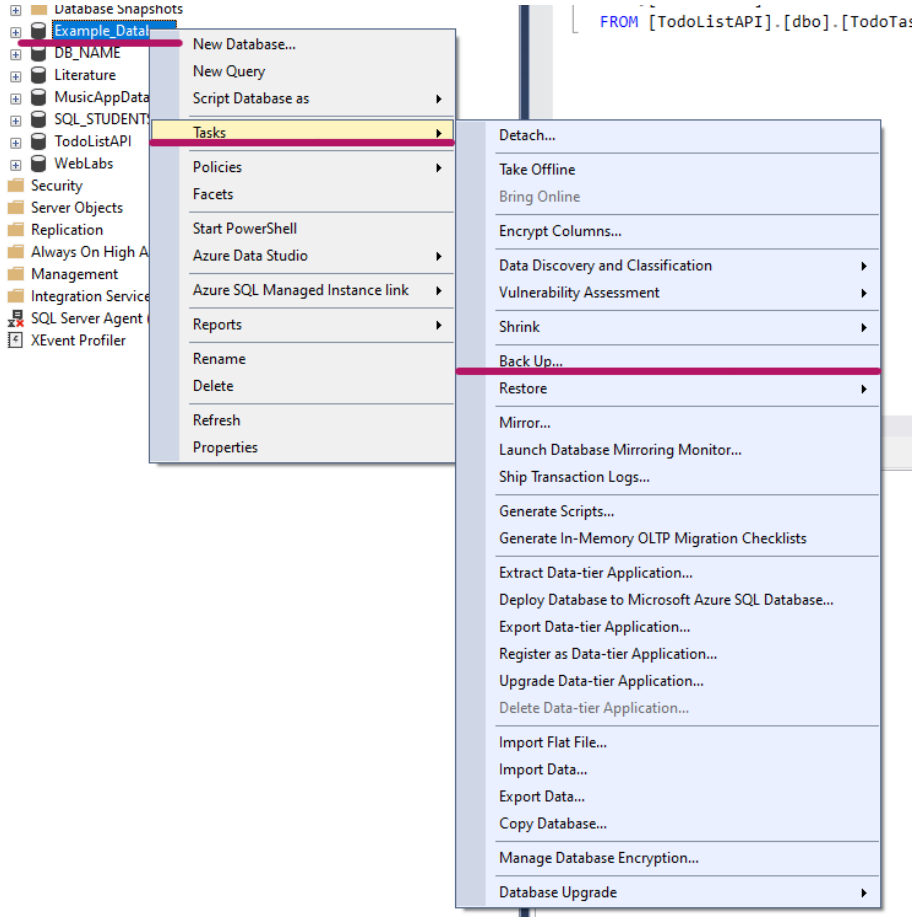


Рисунок 1.3 – Контекстне меню

Ми можемо обрати тип резервного копіювання (Full, Differential, Transaction Log), обрати чи це буде copy-only, чи звичайне копіювання, обрати чи ми будемо копіювати базу, чи окремі файли чи файлові групи. Після цього, ми можемо

обрати шлях, по якому буде створений файл резервної копії. За замовчуванням це диск, шлях по якому встановлено ms sql, але також можна обрати варіант «url». В такому разі, необхідно буде ввести посилання на Azure storage. [16] Оскільки ми працюємо без Azure, та робимо перше резервне копіювання, то оберемо будемо виконувати повне резервне копіювання бази даних, без опції copy-only, зберігаючи на диск. Окрім цих налаштувань, ми можемо вказати, як нова резервна копія буде взаємодіяти з вже існуючими, тобто чи новий бекап буде доданий до кінця існуючого набору, чи новий бекап перезапише старі. В розділі «Reliability» ми маємо змогу перевірити правильність бекапу після його створення, перевірити контрольну суму перед записом в медіа файли, та опцію продовжити створення резервної копії після помилки. Також ми маємо можливість стиснути файл резервної копії. Вікно з наявними налаштуваннями резервного копіювання можемо спостерігати на рисунку 1.4.

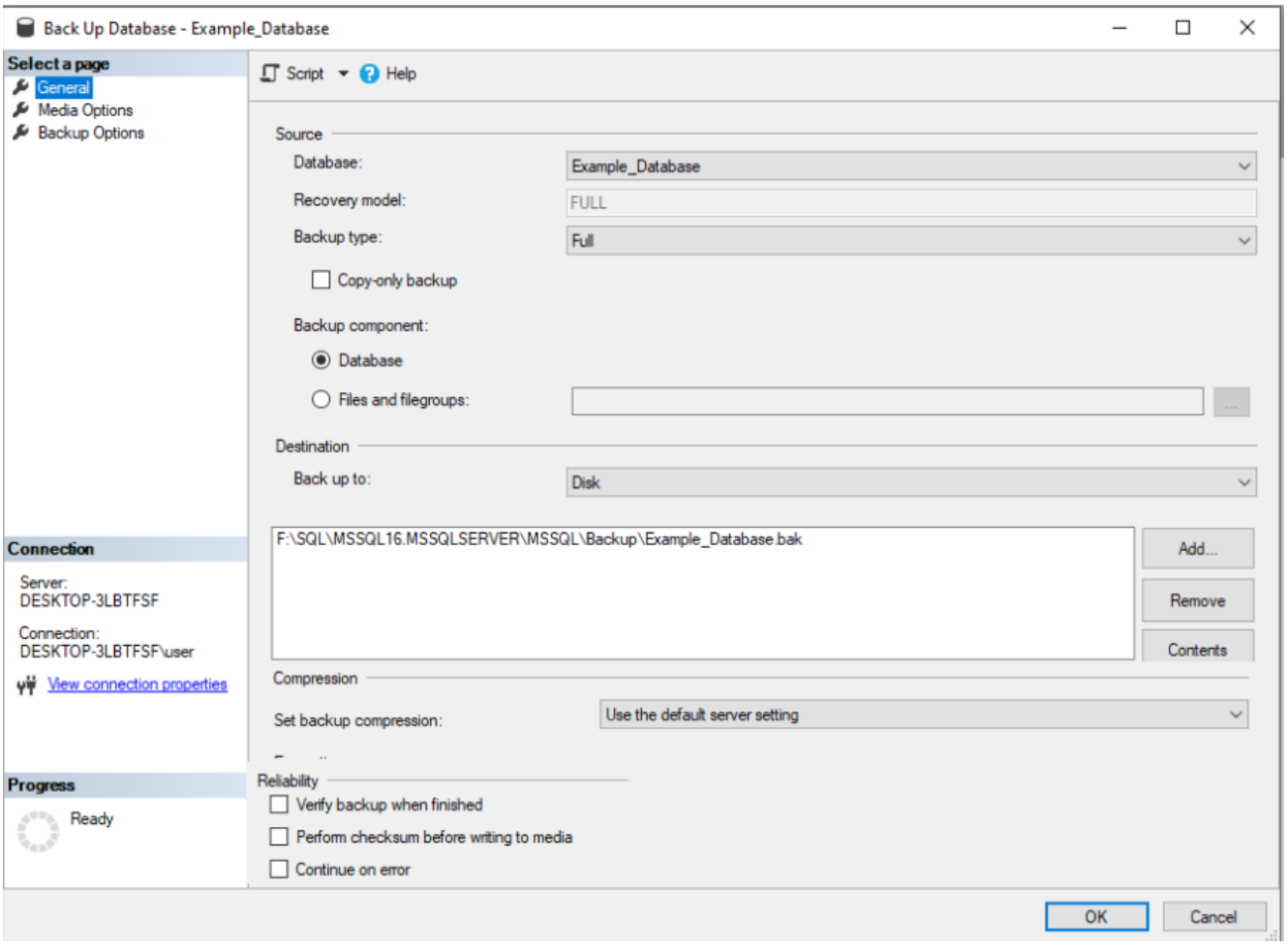


Рисунок 1.4 – Вікно налаштувань резервного копіювання

Аналогічно було створено часткове резервне копіювання бази даних. В результаті, ми маємо 2 файли резервного копіювання, використовуючи які, ми можемо відновити базу даних. Оскільки ми створювали повне резервне копіювання, то ми маємо можливість відкотити базу до будь-якого періоду в часі в рамках наявних резервних копій. Процес відновлення є майже ідентичним процесу створення резервного копіювання. Для відновлення нам необхідно вибрати яку базу будемо відновлювати, на основі яких файлів резервного копіювання, та на який період в часі. Детальніше цей процес можемо спостерігати на рисунку 1.5.

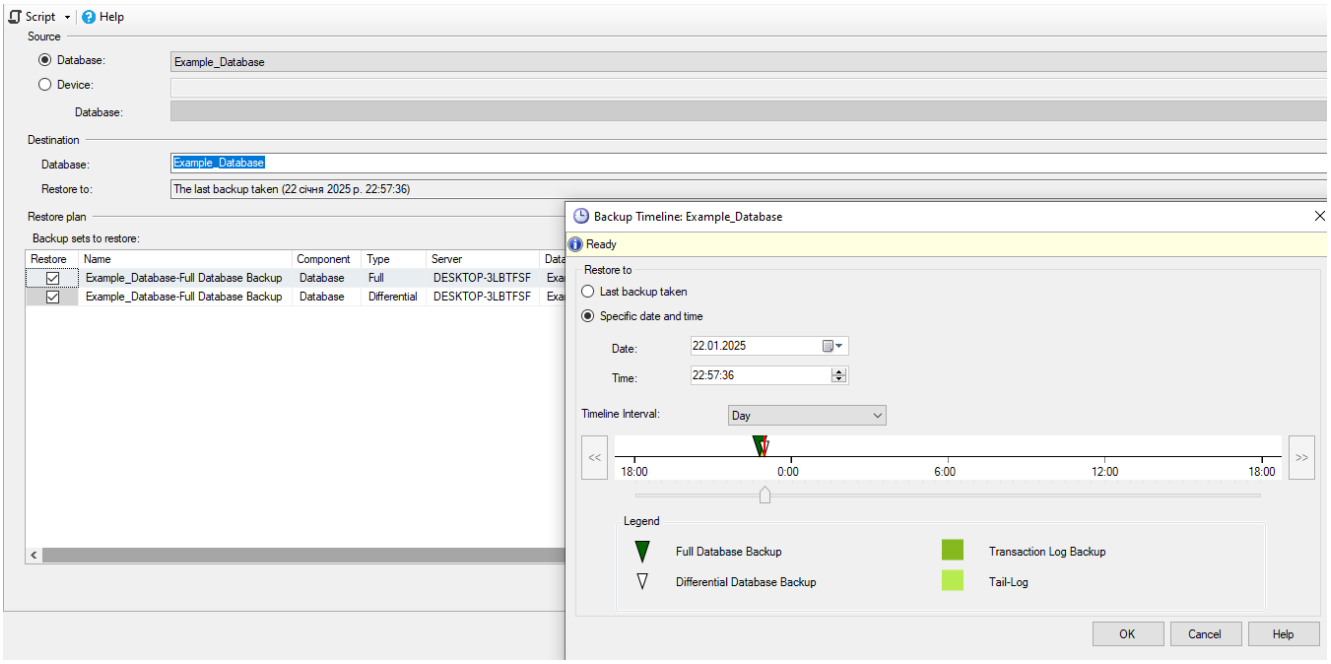


Рисунок 1.5 – Вікно відновлення бази даних

Доступність даних є не менш важливим аспектом. Доступність означає, що база даних має безперебійно працювати в будь-який момент часу, щоб забезпечити безперебійний доступ до інформації, та щоб отримання даних не займало надмірної кількості часу.

Доволі простим прикладом порушення доступності до даних є DoS та DDoS атаки. DoS (Denial of Service) атака - атака на комп’ютерні мережі, метою якої є порушення діяльності певного сервісу, шляхом надсилання на сервер значної кількості запитів. Принцип роботи доволі простий – скориставшись інсуючими

інстурментами для DoS атак, чи написавши власний скрипт, ми надсилаємо на сервер безліч запитів, які сервер намагається опрацювати. Чим складніший запит сервер має опрацювати – тим більше ресурсів сервер буде витрачати на опрацювання запиту і відповідно, тим меньше запитів потрібно буде аби спричинити неполадки в роботі сервера. Прикладом утиліт, які можуть виконувати DoS атаки є LOIC, Hping3, Slowloris. DdoS атака відрізнається від DoS атаки тим, що DoS атака здійснюється з одного пристрою, а DdoS атака – з декількох пристроїв [17].

Врахувати всі варіанти, які можуть спричинити до збоїв в роботі неможливо, але все ж можна знизити ризики, захистившись від найбільш ймовірних загроз. Розглянемо варіанти захисту від вище описаних DDoS атак. Найпростішим варіантом є імплементація «Rate Limiter». Це дозволяє нам встановити обмеження на кількість запитів за певний період часу. Як приклад, розглянемо Rate Limiter для API, написаного використовуючи ASP NET [18]. Налаштувати Rate Limiter дуже просто, оскільки ASP NET надає готове рішення, в якому нам необхідно лише налаштувати опції та стратегію. На вибір є наступні стратегії:

- Fixed window;
- sliding window;
- token bucket;
- concurrency limiting;
- custom/chained;

Fixed window – кількість запитів обмежена фіксованим лімітом для певного відрізку в часу. Наприклад, 10 запитів кожну хвилину. Якщо користувач зробить 10 запитів в 00:01, то зробити наступний запит користувач зможе рівно в 00:02. Немає значення, як користувач розподілить свої запити, тобто чи це буде 10 запитів за одну секунду, чи 10 запитів рівно розподілених на одну хвилину;

Sliding window – ліміт запитів динамічно оновлюється, залежно від того, коли запити було здійснено. Такий підхід є більш гнучним, порівняно з фіксованим;

Token bucket – цей підхід дозволяє виконувати запити, поки «відро токенів» не буде вичерпано. Якщо відро порожнє, то запити буде відкладено до моменту, поки токени не відновляться. Ми моєемо налаштувати інтервал відновлення токенів, та скільки токенів буде відновлено за один інтервал. Наприклад, можна налаштувати відновлення 1 токена в секунду, або відразу 10 токенів за 10 секунд, залежно від стратегії обмеження трафіку;

Concurrency limiting – обмежує кількість одночасних запитів до ресурсу. Це може бути корисним у випадках, якщо відбуваються складні запити до бази даних, або до стороніх API. Це допомагає уникнути перевантаження ресурсів, які мають обмежену пропускну здатність;

Custom/chained – дозволяє налаштувати свою стратегію, використовуючи та поєднуючи існуючі. Варто використовувати, якщо жодна з попердньо описаних стратегій не підходить, та необхідна більш складна логіка. Більшість ситуацій покриваються вже існуючими стратегіями, тому використання користувацької стратегії не є рекомендованим, бо невдало поєднавши існуючі стратегії, можна отримати непередбачувані результати.

Розглянувши наявні стратегії для налаштування Rate Limiting, розглянемо як він створюється. Для цього, розглянемо стратегію Fixed Window, в якій запити будуть відслідковуватись за IP адресою, з якої запит був надісланий. Для цього використаємо наявну імплементацію Rate Limiter-а від ASP NET. Використовуючи IserviceCollection, ми маємо готові рішення вище описаних Rate Limiter-ів. Нам необхідно лише додати Rate Limiter до сервісів, дати йому ім’я (policy), та встановити ключ, за яким будемо відслідковувати запити. На щастя, ми також маємо доступ до інформації про запит через HttpContext, звідки ми і будемо отримувати інформацію про айпі адресу, з якої було здійснено запит, а також інформацію про те, на який ендпоінт було здійснено запит. Після налаштувань, нам необхідно лише зареєструвати Rate Limiter та встановити його на необхідний ендпоінт, чи повністю на весь контролер повністю, що є еквівалетом встановлення Rate Limiter на всі ендпоінти даного контролера. Розглянути описані вище інструкції можемо на лістингу 1.1.

```
public static IServiceCollection WithRateLimiting(this IServiceCollection
services)
{
    services.AddRateLimiter(option =>
    {
        option.RejectionStatusCode = StatusCodes.Status429TooManyRequests;

        option.AddPolicy("fixed", httpContext =>
            RateLimitPartition.GetFixedWindowLimiter(
                partitionKey:
httpContext.Connection.RemoteIpAddress?.ToString(),
                factory: _ => new FixedWindowRateLimiterOptions()
                {
                    PermitLimit = 10,
                    Window = TimeSpan.FromSeconds(10)
                }
            ));
    });

    return services;
}
//Реєструємо Rate Limiter
builder.Services.WithRateLimiting();
app.UseRateLimiter();

//Встановлюємо Rate Limiter
app.MapGet("/{query}", async (string query) =>
{
…
}).RequireRateLimiting("fixed");
```

Лістинг 1.1 – Fixed Rate Limiter

Окрім обмеження кількості запитів, ми можемо встановити обробник подій, на випадок, якщо вичерпано кількість запитів. Обробник подій є дуже потужним розширенням до rate limiter-а, оскільки це дає нам можливість виконувати будь-які дії у випадку, коли вичерпано кількість запитів. Як приклад, будемо логувати айпі адресу та ендпоінт, до якого була спроба доступу. Для цього розширимо Rate Limiter, додавши до нього обробник подій при вичерпані спроб. Сервіс логування ми можемо отримати з контейнеру сервісів. Можемо використовувати сервіс логування за замовчуванням, який надає asp net, але в даному прикладі використовується розширений сервіс логувань. Розглянемо доповнення для Rate Limiter, яке дозволяє обробляти подію перевищення кількості запитів шляхом логування айпі адреси, з якої був здійснений запит, а також ендпоінта, на лістингу 1.2.

```
services.AddRateLimiter(option =>
{
…
option.OnRejected = (context, token) =>
{
    var logger =
context.HttpContext.RequestServices.GetRequiredService<ILoggerService>();
    var ipAddress = context.HttpContext.Connection.RemoteIpAddress?.ToString();
    var path = context.HttpContext.Request.Path;

    logger.Log(message: $"REQUEST LIMIT", LogLevel.Warning, context: new {
ipAddress, path });

    return ValueTask.CompletedTask;
};
…
}
```

Лістинг 1.2 – Логування при вичерпанні запитів

В результаті, якщо буде виконано більше 10 спроб за 10 секунд, буде повернено код 429, а айпі адресу, з якої було здійснено запит, та ендпоінт, до якого була спроба доступу, буде записано в журнал логувань.

Також можна використати підхід балансування навантаження. Суть полягає у тому, що замість використання одного сервера, на який йде все навантаження, використовується декілька серверів, мій якими розподіляється весь трафік. З розвитком хмарних технологій, все частіше використовуються віртуальні сервери для розподілення трафіку. Значним недоліком є те, що ціна за кожен такий сервер є доволі високою, а використання декількох серверів значно ускладнює процес розробки.

Аби точно виявити DDoS атаку, варто звернути увагу на використання IDS/IPS/IDPS систем. IDS (Intrusion Detection System) – інструмент, що використовується для пасивного моніторингу трафіку комп'ютерній мережі, шляхом копіювання трафіку в мережі та його подальшого аналізу. Окрім захоплення трафіку, IDS також сканує захоплені пакети на наявність вірусів, використовуючи сигнатурний аналіз. IDS самостійно не вживає жодних заходів, тому для обробки загроз, виявлених ним, необхідне людське втручання, або наявність IPS. IPS (Intrusion Prevention System) – інструмент, що на основі проаналізованих пакетів може блокувати пакети трафіку. IDPS поєднують в собі функціонал IDS та IPS. Відомими інструментами є Snort(IDS) та Suricata(IDPS). Обидва інструменти чудово справляються з виявленням DDoS атак. [19]

Більш примітивним прикладом порушення доступності інформації є природні явища, які можуть спичинити до фізичного знищення серверів, чи до збоїв електропостачання. Підхід балансування навантаження, що був описаний вище, в поєднанні з регулярним створенням резервних копій частково вирішує цю проблему. Наявність декількох серверів, що знаходяться на значній відстані один від одного, значно забезпечує роботу хоча б одного з них, а наявність резервних копій забезпечує можливість відновлення даних. Також можна використовувати хмарні сервіси, як частину гібридної інфраструктури, може мінімізувати ризики пов’язані з фізичними загрозами. Не меньш важливим пунктом є наявність джерел безперебійного живлення, на випадок відключення електроенергії та наявність автоматизованих систем моніторингу роботи серверів.

1.3 Засоби та інструменти захисту конфіденційної інформації в СКБД MS SQL від sql-атак

Неправильна обробка користувацьких введень може призвести до вразливостей SQL-ін'єкцій, що дозволяє зловмисникам маніпулювати запитами, отримувати несанкціонований доступ до даних або компрометувати систему, що ставить під загрозу конфіденційність, цілісніть та доступність інформації, що може призвести до втрати конфіденційних даних та до грошових втрат. Наприклад, хакери можуть обійти автентифікацію, отримати конфіденційну інформацію або пошкодити дані, вводячи несанкціоновані команди SQL у запити. Здавалося б, що з розвитком технологій та появою ORM, які автоматично створюють безпечні запити, шляхом їх параметризації, загроза SQL-ін’єкцій мала б зникнути, але насправді це не так. Значна кількість додатків та веб-сайтів використовують застріле програмне забезпечення чи старі версії фреймворків, що збільшує шанси на успішне виконання SQL-ін’єкцій. Згідно з останньою статистикою від OWASP, зображеною на рисунку 1.6, станом на 2021 рік, SQL-ін’єкції досі входять в топ 10 загроз, а точніше – займають 3 місце серед усіх загроз. Це менше, порівняно з 2017 роком, коли SQL-ін’єкції були на першому

місці, але сам факт, що ця загроза досі входить в топ 10 вказує на те, що нехтувати захистом від SQL-ін’єкцій не варто [20].

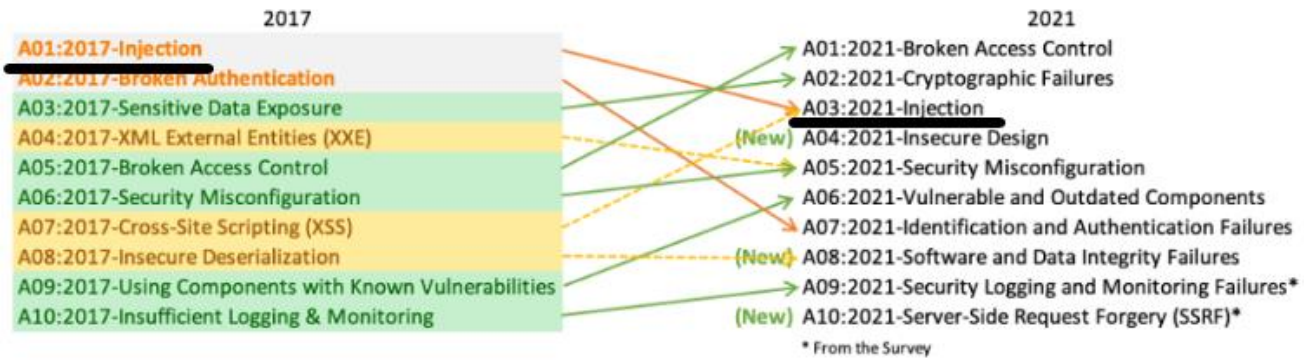


Рисунок 1.6 – Статистика OWASP TOP 10 за 2021 рік

Розглянемо основні види SQL- ін’єкцій:

– Error based;
– union based;
– blind boolean;
– blind time based;

Error based SQL Injections – найпростіший тип ін’єкцій. Щоб виявити загрозу error-based ін’єкції, достатньо ввести символ «’» в поле вводу, яке вразливе до ін’єкції. В результаті, ми можемо отримати повідомлення з помилкою, яке містить інформацію про структуру бази даних або про запит. Це корисно для зловмисника, бо можна отримати інформацію про наявність певних таблиць чи стовпців;

Union based SQL Injections – в основі цих ін’єкцій лежить оператор UNION, який дозволяє об’єднувати результати кількох запитів. Варто зауважити, що для успішного використання оператора UNION, обидва запити мають повертати однакову кількість стовпів. Щоб взнати кількість стовпців, можна скористатись оператором ORDER BY, оскільки він дозволяє формат запису ORDER BY N, де N – порядковий номер стовпця по якому буде відбуватись сортування. Варто перебирати N, поки не отримаємо помилку, і таким чином взнаємо кількість

стовпців. Тепер можемо виконати команду «?id=-1‘ UNION SELECT 1,version(),database() --+», яка поверне нам версію та тип бази даних;

Blind boolean SQL Injections – ін’єкція, в якій ми не бачимо результатів, але можемо перевірити наявність вразливості використовуючи логічні вирази. Наприклад, додавши до запита «' AND 2=2;--» або «' AND 1=2;--». Визначивши, що вразливість наявна, зловмисник може почати перебір логічних значень, які розкриють критичну інформацію про базу;

Blind time based SQL Injections – схожі на blind boolean, оскільки чіткого повідомлення з помилкою ми не отримуємо. Для перевірки на цей тип вразливості використовується SLEEP N або DELAY FOR N. Вразливість наявна, якщо запит буде виконуватись N часу. [21]

Існують також і інші види ін’єкцій, які глибше конкретизують спосіб виконання ін’єкції, але в цілому вони всі зводяться до типів описаних вище. Уникнути SQL-ін’єкцій доволі просто. Більшість SQL-ін’єкцій трапляються внаслідок відсутності параметризації. Конкатенуючи вхідні параметри в запит, ми відкриваємо можливість SQL-ін’єкцій. Використовуючи параметризацію, SQL-ін’єкція неможлива, оскільки параметри екрануються і сприймаються як текст, а не команда, тобто навіть у випадку, коли зловмисник введе шкідливий запит в поле вводу, завдяки параметризації запитів, шкідливі дії вчинити не вийде. Використовуючи ORM (Object Relational Mapping) ми можемо ще сильніше захиститись від SQL-ін’єкцій, оскільки нам навіть не треба задумуватись про параметризаці. ORM самостійно складають SQL запити, використовуючи параметризацію та інші заходи безпеки. Незважаючи на це, найкращим вирішенням цієї проблеми є валідація будь-яких даних, що користувач передає нам. Як приклад, розглянемо запит для авторизації користувача за логіном та паролем. У випадку наявності вразливості, зловмисник може увійти не знаючи пароль, або ж виконати власний шкідливий запит. Приклад вразливого та виправленого коду для авторизації користувача можемо розглянути на лістингу 1.3.

```
public class DapperVulnerableExample
{
    public void ExecuteVulnerableDapperQuery(string username, string password)
    {
        string query = "SELECT * FROM Users WHERE Username = '" + username + "'
AND Password = '" + password + "'";

        using (SqlConnection connection = new SqlConnection("connectionString"))
        {
            var users = connection.Query<User>(query).ToList();
        }
    }
}

public void ExecuteFixedDapperQuery(string username, string password)
{
    string query = "SELECT * FROM Users WHERE Username = @Username AND Password =
@Password";

    using (SqlConnection connection = new SqlConnection("connectionString"))
    {
        var users = connection.Query<User>(query, new { Username = username,
Password = password }).ToList();
    }
}
}
```

Лістинг 1.3 – Приклад параметризації

Приклад з попереднього лістингу є особливо актуальним, коли взаємодія з базою даних відбувається через ADO NET, або Dapper, та меньш актуальним, коли використовуються ORM, які автоматично генерують запити, наприклад Entity Framework (за вийнятком, коли запити пишуться вручну), оскільки ORM сам згенерує параметризований запит. Приклад згенерованого параметризованого запиту продемонстровано на лістингу 1.4. [22]

```
SELECT p."Guid", p."CreatedAt", p."CreatedBy", p."ModifiedAt", p."Source",
p."ThumbnailId", p."ThumbnailSource", p."Title", u."UserGuid" IS NOT NULL AND
u."PlaylistGuid" IS NOT NULL AND u."IsFavorite", (
                SELECT count(*)::int
                FROM "PlaylistSong" AS p0
                WHERE p."Guid" = p0."PlaylistGuid")
            FROM "Playlist" AS p
            LEFT JOIN "UserFavoritePlaylist" AS u ON p."Guid" =
u."PlaylistGuid" AND @__request_UserGuid_0 = u."UserGuid"
            WHERE p."CreatedBy" = @__request_UserGuid_0
```

Лістинг 1.4 – Приклад згенерованого ORM параметризованого запиту

1.4 Постановка задачі

В цьому розділі було розкрито тему MS SQL, наявних механізмів захисту в MS SQL, зокрема таких, як захищені анклави, розмежування прав користувачів, шифрування окремих колонок чи таблиць. Також було розглянуто види резервних копіювань, для чого вони використовуються, та як їх поєднувати для підтримки балансу часу копіювання/відновлення, та захисту інформації.

Також було розглянуто методи захисту від DoS та DdoS атак, зокрема шляхом використання різних типів Rate Limiter-ів, які дозволяють обмежити кількість запитів, а також встановити власну логіку обробки події перевищення кількості дозволених запитів, наприклад логування дій та айпі адреси, що ініціювала запит, та шляхом використання IDPS систем.

Найважливіше, було розкрито тему SQL ін’єкцій, їх типів, найпростіших шляхів їх запобігання, з прикладами реалізації. Розглянуто різницю між написанням запитів вручну, та використання ORM.

На основі розглянутої інформації, ми можемо перейти до постановки задачі. Задача полягає у розробці програмного забезпечення, що зможе виявляти потенційно вразливі SQL запити, та обробляти їх. Програмне забезпечення має забезпечити логування факту запиту шляхом, що унеможливлює знищення запису логування. Окрім цього, необхідно забезпечити простоту інтеграції нашого ПЗ у клієнтські програми, а також надати можливість розширення логіки обробки перехоплених запитів. Окрім основного ПЗ передбачається розробки допоміжних утиліт. Розроблена система захисту має легко розширюватись, легко інтегруватись, та не мати жодних вимог щодо розгортання зі сторони клієнта. Система захисту має виявляти всі види SQL ін’єкцій, а також передбачати можливість перехоплення запитів, залежно від описаної клієнтом додаткової логіки.

2 ПРОЄКТУВАННЯ СИСТЕМИ ЗАХИСТУ ІНФОРМАЦІЇ В БАЗАХ ДАНИХ SQL ВІД SQL-АТАК

2.1 Кібератаки на конфіденційність інформації в базах даних та її захист

Оскільки бази даних зберігають та обробляють конфіденційну іннформацію, вони завжди були основною цілю зловмисників. Технології для розробки застосунків постійно змінюються та вдосконалюються, але проблема конфіденційності завжди залишається незміною. Хоча ця загроза є доволі старою, вона й досі залишається актуальною. Ця загроза була описана ще у 1998 році у журналі «Phrack». У цій статі продемонстровано та описано, як відсутність параметризації призводить до обходу перевірки прав користувача, отримання доступу до інформації, до якої доступу не мало б бути. [23] Як приклад новіших інцидентів порушення конфіденційності інформації, що могли б мати критичні наслідки для власників застосунку, а також їх клієнтів, можна розглянути інцидент, що стався з адміністрацією транспортної безпеки Сполучених Штатів Америки у 2024 році. Вони використовували сторонє програмне забезпечення, під назвою FlyCASS (Fly Cockpit Access Security System), яке спрощувало керування кабіною керування літака. В цьому програмному забезпеченні, було вразливе поле для sql-ін’єкції. Прямо на головній сторінці, було вразливе поле для входу користувача, за допомогою якого вдалось здійснити ін’єкцію. Це дозволило повністю обійти механізм аутентифікації, що дозволяє переглядати та змінювати інформацію сторонім користувачам. [24] Цю загрозу було виявлено, та ідентифіковано як критичну, з рейтингом загрози 9.3 [25]

Оскільки база даних не може самостійно повністю забезпечити функціонування застосунку, вектор загроз значно розширюється. Тепер окрім загроз для самої бази даних, ми також розглядаємо загрози, які можуть виникнути під час розробки застосунку, загрози, які пов’язані із конфігурацією систем. Це важливо, оскільки навіть у випадку, коли застосунок використовує параметризовані запити, доступ до конфіденційних даних може бути отриманий внаслідок помилки у коді застосунку. Цей сценарій стає все більш популярним і це пов’язано з розвитком штучного інтелекту. Значна кількість застосунків є

згенерованими за допомогою штучного інтелекту, який часто нехтує загальною архітектурою застосунка, та доволі часто генерує код, який вразливий до SQL-ін’єкцій та XSS атак. Окрім цього, є окремі випадки, коли чутлива інформація по типу API-ключів, connectionString до бази даних прописані прямо в коді, а не в конфігураційних файлах, чи окремих сховищах. [26]

Ще одним вектором для атак, є застрілі бібліотеки, що використовуються у програмному забезпеченні, яке взаємодіє з базою даних. Навіть якщо у застосунку немає вище описаних вразливостей, але є застарілі бібліотеки, існує загроза порушення конфіденційності інформації. Як прилад, якщо застосунок використовує бібліотеку, з вразливістю command injection, то зловмисник зможе отримати доступ до консолі хостового пристрою. Маючи доступ до консолі, зловмисник з легкістю зможе отримати доступ до бази даних. [27] Вразливості з використанням застарілих бібліотек є доволі поширеними, оскільки існує безліч legacy-застосунків, які перебувають в режимі підтримки, та в яких використання найновіших бібліотек є неможливим.

Вразливості конфіденційності, які пов’язані з неправильною конфігурацією є доволі рідкісними, але натомість їх найлегше використати у власних цілях, оскільки у конфігураційних файлах майже завжди міститься конфіденційна інформація. Переважно використовуються декілька конфігураційних файлів, один для розробки, та інший для готового застосунку. Конфігураційний файл для розробки часто містить чутливу інформацію у відкритому вигляді, і доступ до цього файлу має бути лише у довірених осіб, яким необхідний доступ до додатку при розброці. Конфігураційний файл готового затосунку має використовувати плейсхолдери і не містити жодної чутливої інформації. Якщо конфігураційний файл для розробки буде оприлюднено, то можуть бути втрачені такі дані:

- connection string до бази даних;
- apі-ключі;
- логін та пароль адміна за замовчуванням;
- адреси внутрішніх сервісів;
- деталі про провайдера автентифікації.

Якщо буде отримано connection string – то буде отримано і доступ до бази даних. Втрата арі-ключів завжди призводить до значних фінансових збитків, оскільки хтось інший зможе робити запити за рахунок власника цього ключа. Інформація про адреси внутрішніх сервісів може допомогти зловмиснику в подальшій експлуатації системи. Цього всього можна уникнути, дотримуючись загальних правил роботи з конфігураційними та секретними файлами. Наприклад, при роботі з github, всі конфігураційні та секретні файли мають бути додані в .gitignore.

Найкращим способом забезпечити секретність конфігураційних, .env файлів є використання систем децентралізованого управління секретами, таких як Azure Key Vault, AWS Secret Manager Vault, чи Koyeb Secrets. Перелічені системи мають доволі широкий спектр функцій, зокрема автоматичну ротацію паролів бази даних, зберігання даних у зашифрованому вигляді, використання у CI/CD pipelines. [28] Точніша характеристика зображена у таблиці 2.1.

Таблиця 2.1 – Порівняння Azure Key Vault, AWS Secrets Manager, Koyeb Secrets

Характеристика	Azure Key Vault	AWS Secrets Manager	Koyeb Secrets
Шифрування	Azure-managed (з використанням Azure KMS)	AWS KMS (за замовчуванням)	Шифрується автоматично на рівні платформи
Автоматична ротація	Частково (через Azure Automation або API)	Вбудована підтримка для DB та Lambda	Немає автоматичної ротації
Інтеграція з іншими сервісами	Azure App Service, Azure Functions тощо	Lambda, ECS, RDS, EC2 тощо	Лише з Koyeb Workloads
Керування доступом	Role-Based Access Control (RBAC), Azure AD	IAM Policies	Просте обмеження доступу до сервісів Koyeb
Ціна	Залежить від операцій	Вища, враховуючи автоматичну ротацію	Безкоштовна (у межах плану)
Використання у CI/CD	Підтримується (через GitHub Actions, DevOps)	Підтримується (через CodePipeline, GitHub)	Через вбудоване керування секретами

Всі перелічені вище системи мають функцію захищеного збереження секретів. В контексті захисту конфіденційної інформації, нас переважно цікавить наявність можливості зашифрованого зберігання даних. Автоматична ротація не є критичною, оскільки це робиться доволі швидко вручну.

На основі зібраної вище інформації, будемо використовувати Koyeb для безпечного збереження секретів. Koyeb підтримує додавання секретів вручну, вказуючи назву ключа та значення, або імпорт. Імпорт є кращим варіантом, оскільки додаючи секрету вручну легко допустити помилку, особливо коли секрет є не однорядковим, оскільки вони обробляються іншим способом. Для імпорту нам потрібні всі змінні з конфігураційного файлу у форматі, який підтримує Docker. [29] Аби повністю уникнути людського фактору при імпорті секретів з конфігураційного файлу в Koyeb, можемо автоматизувати цей процес, використавши невелику консольну програму, яка буде зчитувати конфігураційний файл, та експортувати відформатовані секрети у файл чи консоль. Суть алгоритму полягає у переборі значень файлу, та форматуванні отриманих значень для експорту в Koyeb чи Docker. Перебір розпочинається з вершини файлу, та рекурсивно продовжується для кожного вкладеного в файл об’єкту, доки не буде отримано лише значення. Отримані значення представлені у форматі «Ключ: значення», де ключ – повний шлях до значення, відносно початку файла. Тобто ключ буде містити у собі назви всіх дочірніх об’єктів, що дозволяє відслідкувати кому належить це значення. Для більшої гнучкості, реалізовано чорний список, який дозволяє виключити з перебору певні значення, або ж цілі секції чи підсекції, реалізовано різні типи форматування, та опції виводу, що зберігають зберегти результат як файл, або ж просто вивести на екран. Розглянемо описаний алгоритм, що рекурсивно витягує секрети з конфігураційного файлу на рисунку 2.1. Повний код зображено у додатку Б.

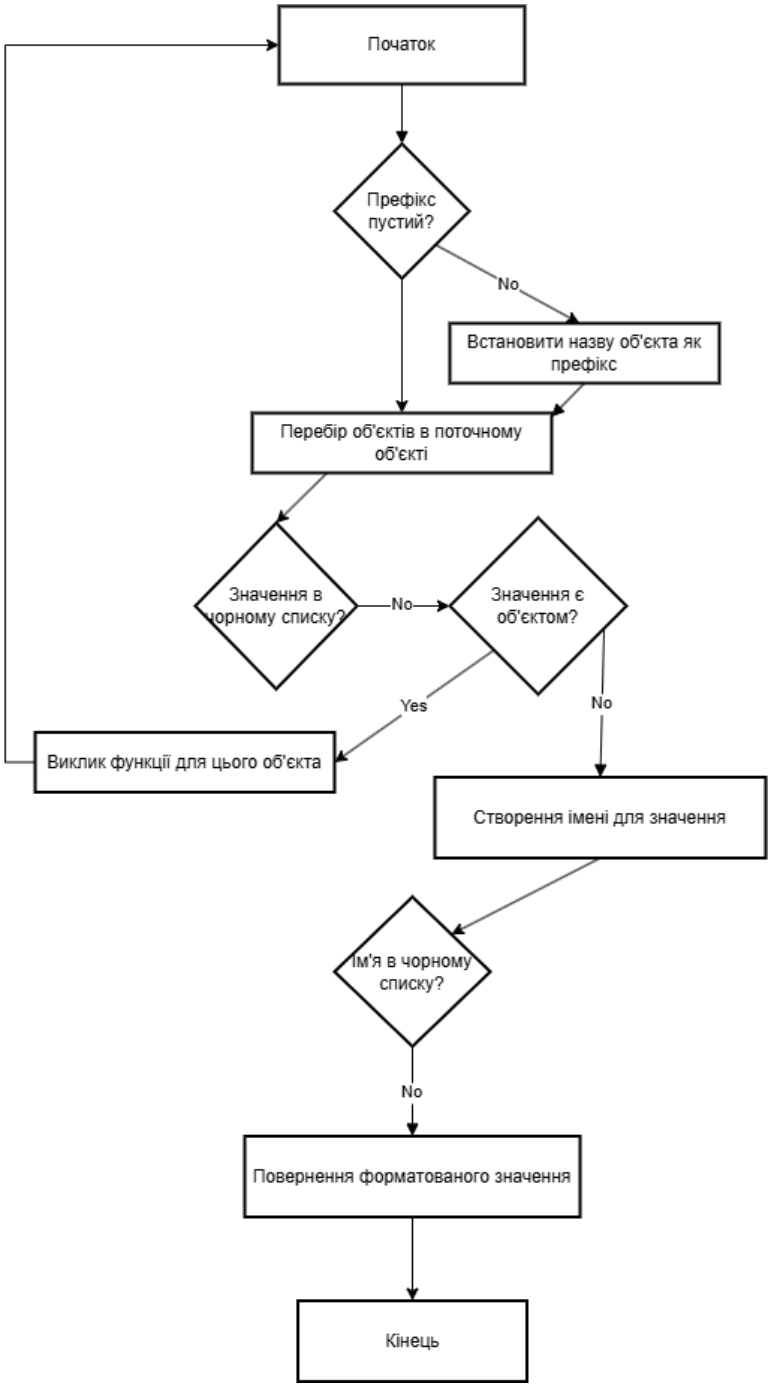


Рисунок 2.1 – Алгоритм експорту з конфігураційного файлу

Завантаживши отриманий файл у Koyeb, ми зможемо використовувати секрети конфігураційного файлу, не ризикуючи випадково порушити їх конфіденційність, а також отримуємо підвищену захищеність, оскільки тепер вони зберігаються в зашифрованому вигляді і в нас повністю відпадає потреба зберігати секрети локально, а також знижується ризик пов’язаний із людським фактором.

2.2 Проєктування плейсхолдерів як основний засіб захисту даних від SQL-ін’єкцій

Коли ми працюємо з SQL запитами, параметризація є обов’язковою. Варто керуватись правилом - якщо якась частина запиту є динамічною і приходить від користувача, то варто використовувати параметризацію. Як приклад, візьмемо базовий запит для пошуку користувачів за ролями, мінімальним віком, флагом «активний», та пошуковим словом, та будемо сортувати їх за стовпцями.

Всі ці параметри приходять до нас від клієнта, тому всі параметри мають бути параметризованими, включаючи orderByColumn і orderDirection. Якщо для фільтрів, таких як роль, мінімальний вік, пошукове слово вистачить параметризації, то для сортування цього буде замало. Припустимо, що ми не хочемо, щоб при пошуку користувачів була можливість визначити хто був зареєстрований в системі раніше, а хто пізніше. Результат вибірки повертає всі поля, крім CreatedAt. Наш запит параметризований, але зловмисник всеодно може прописати «CreatedAt» як значення параметру orderByColumn і таким чином дізнатись хто і в якій послідовності був зареєстрований в системі. Ще один сценарій - зловмисник здійснити перебір існуючих колонок і дізнатись схему таблиці. Ні параметризований запит, ні використання ORM не рятує від перебору колонок. Рішення цієї проблеми є дуже простим, потрібно ніколи не довіряти тому, що вводить користувач, та проводити валідацію параметрів перед тим як формувати запит. В цьому випадку нам необхідно додати перевірку на правильність колонок сортування. Це можна зробити як збереженою процедурою, так і через будь-який ORM. Використовуючи збережену процедуру в Ms Sql, нам достатньо додати значення за замовчуванням для параметрів, а також перевірити їх на входження в перелік дозволених значень. Така ж ситуація при використанні ORM по типу Dapper чи ADO NET. Використовуючи ORM по типу Entity Framework, де ми можемо не писати запис вручну, і динамічна фільтрація є вже імплементованою, ця проблема залишається. По черзі розглянемо рішення для кожного з підходів. Почнемо з збереженої процедури в MsSql, яка збережена на лістингу 2.1.

```
CREATE PROCEDURE dbo.GetFilteredUsers
    @Role NVARCHAR(50) = NULL,
    @MinAge INT = NULL,
    @IsActive BIT = NULL,
    @SearchTerm NVARCHAR(100) = NULL,
    @SortColumn NVARCHAR(50) = 'Username',
    @SortDirection NVARCHAR(4) = 'ASC'
AS
BEGIN
    SET NOCOUNT ON;
    IF @SortColumn NOT IN ('Id', 'Username', 'Email', 'Age', 'Role', 'IsActive')
        SET @SortColumn = 'Username';
    IF UPPER(@SortDirection) NOT IN ('ASC', 'DESC')
        SET @SortDirection = 'ASC';
    SELECT Id, Username, Email, Age, Role, IsActive
    INTO #FilteredUsers
    FROM Users
    WHERE (@Role IS NULL OR Role = @Role)
 AND (@MinAge IS NULL OR Age >= @MinAge)
 AND (@IsActive IS NULL OR IsActive = @IsActive)
    DECLARE @Sql NVARCHAR(MAX) = '
        SELECT * FROM #FilteredUsers
        ORDER BY ' + QUOTENAME(@SortColumn) + ' ' + @SortDirection;
    EXEC sp_executesql @Sql;
    DROP TABLE #FilteredUsers;
END
```

Лістинг 2.1 – Безпечна збережена процедура

Оскільки в нас є перевірка на правильність параметрів сортування і значення за замовчуванням, ми можемо викликати цю процедуру з будь-якими параметрами, при цьому не ризикуючи розкрити небажану інформацію, а також впевнені в тому, що процедура просто не впаде з помилкою. Також відпадає вразливість перебору колонок. Якщо зловмисник спробує здійснити перебір колонок і введе вище згаданий «CreatedAt», то перевірка на дозволить йому відсортувати за часом створення – натомість буде відсортовано за колонкою за замовчуванням, в нашому випадку – за колонкою Username. [30]

Щоб досягти такого ж результату, використовуючи ORM, нам необхідно провести таку ж перевірку на входження фільтру в список дозволених значень. Реалізація вийде значно меншою за розміром, але ідентичною за логікою роботи. Список дозволених колонок можна прописувати окремо для кожного виклику, або ж розширити наявну логіку шляхом використання допоміжної функції, яка буде

на основі таблиці повертати список дозволених колонок. Приклад зображено на лістингу 2.2.

```
var allowedColumns = new HashSet<string> { "Id", "Username", "Email", "Age",
"Role", "IsActive" };
var allowedDirections = new HashSet<string> { "ASC", "DESC" };

var sortColumn = allowedColumns.Contains(filter.SortColumn) ? filter.SortColumn
: "Username";
var sortDirection = allowedDirections.Contains(filter.SortDirection?.ToUpper())
? filter.SortDirection.ToUpper() : "ASC";
```

Лістинг 2.2 – Перевірка колонок для ORM

Деякі ORM мають можливість автоматично генерувати безпечні, параметризовані процедури, шляхом їх зчитування з бази даних. Тобто спочатку нам необхідно реалізувати бізнес логіку в Ms Sql збереженою процедурою, а потім запустивши спеціальну утиліту, як от наприклад EDMX, ми маємо можливість автоматично створити таблиці, безпечні методи для виклику збережених процедур, які використовують параметризацію.

Якщо ми працюємо з ORM і маємо DTO (Data Transfer Object) для результату запита, ми можемо спростити процес перевірки входження колонки в об’єкт ще більше. Це найкраще працює у поєднанні з Code First Approach, оскільки сутності в коді і в базі будуть ідентичними. Як приклад, розглянемо допоміжний метод для ORM, який буде розширювати згенерований запит, та додавати до нього валідні налаштування сортування. Перевірка відбувається шляхом перевірки входження колонки, яку надав користувач, у список колонок, які були витягнуті з переліку колонок DTO, чи іншого наданого класу. Напрямок сортування буде встановлюватись флагом, та оброблятись допоміжним методом, через що зникає потреба створювати білий список для значень напрямку сортування. Розглянемо приклад перевірки на правильність колонки, та допоміжний хелпер метод для спрощення механізму сортування для Entity Framework на лістингу 2.3.

```
public bool IsValidSortColumn<T>(string columnName)
{
    return typeof(T).GetProperties()
        .Any(p => string.Equals(p.Name, columnName,
StringComparison.OrdinalIgnoreCase));
}

public IQueryable<T> ApplyOrdering<T>(IQueryable<T> query, string orderBy, bool
isDescending) where T : class
{
    if(!IsValidSortColumn<T>(orderBy))
    {
        orderBy = typeof(T).GetProperties().First().ToString()!;
    }

    return string.IsNullOrEmpty(orderBy)
        ? query
        : isDescending
            ? query.OrderByDescending(entity => EF.Property<object>(entity,
orderBy))
            : query.OrderBy(entity => EF.Property<object>(entity, orderBy));
}

//Приклад використання
_uow.ApplyOrdering(query, request.OrderBy, request.IsDescending)
```

Лістинг 2.3 – Методи перевірки правильності колонок для EF

На всіх вище зображених прикладах суть одна – якщо інформація в запит приходить від користувача, то спочатку необхідно здійснити валідацію, а потім параметризацію. В жодному разі не можна відразу підставляти значення, що прийшли від користувача, навіть якщо вони параметризовані. Якщо не проводити цю перевірку, то ми частково відкриваємось вже до зовсім іншого типу атак – XSS атак. Цей тип атак не загрожує безпосередньо базі даних, але може призвести до порушення конфіденційності даних на рівні клієнтського інтерфейсу. Якщо інформація з бази даних відображається без належного екранування на клієнтській частині, то зловмисник може впроваджувати шкідливі скрипти в інтерфейс, внаслідок чого він зможе виконати будь-який шкідливий java script код у власних цілях, і як приклад, викрасти кукі жертви, що дає зловмиснику можливість до Session Hijacking. Зловмисник зможе виконати набажані дії від лиця жертви, та отримати інформацію, до якої він не повинен мати доступ. Залежно від того, наскільки застосунок вразливий до XSS, зловмисник також може поширювати віруси-черв’яки чи модифікувати вміст сторінки, шляхом відображення контенту з власних джерел, чи переадресації на неї.

2.3 Інтеграція блокчейн-технології для підвищення ефективності захисту даних від sql-ін'єкцій

Як додатковий шар захисту, ми можемо використати технологію блокчейн. Блокчейн – це покращений метод баз даних, що дозволяє бізнес мережам організовувати відкритий обмін інформацією. Структура блокчейна нагадує двозв’язний список, оскільки кожен блок, окрім бізнес інформації, містить хеш попереднього блока та хеш наступного. Оскільки кожен блок є пов’язаним з іншими, та утворює хронологічну послідовність, видалення певного блока є неможливим без згоди мережі, оскільки це порушить послідовність всього блокчейна. Іншими словами, блокчейн – це децентралізована розподілена база даних, дані якої зберігаються на декількох пристроях, та де транзакції перевіряються на основі механізму консенсусу. Консенсус означає дотримання набору правил мережі. Транзакція буде створена тільки за умови більшості користувачів.

В блокчейні використовується криптографія. Кожен блок має свій хеш, що не дозволяє скомпоментувати навіть один байт інформації, оскільки хеш бути зовсім іншим. Кожен учасник мережі має відкритий і закритий ключі шифрування, що дозволяє перевірити, що транзакцію ініціював саме той користувач, який має на це необхідні права.

Основна перевага блокчейна в тому, що жодна з сторін не може скомпроментувати інформацію транзакції, завдяки властивості розподіленості блокчейнів. Це означає, що кожен учасник мережі має копію всієї історії транзакцій. Наприклад, якщо було укладено договір, про надання послуг між замовником та виконавцем, замовник не зможе збрехати, що він заплатив, точно так само, як виконавець не зможе збрехати, що не отримав кошти, оскільки інформація про транзакцію буде записана у блок, а цей блок буде пов’язано із іншими блоками в мережі. [31] Візуально принцип роботи блекчейна зображено на рисунку 2.2.

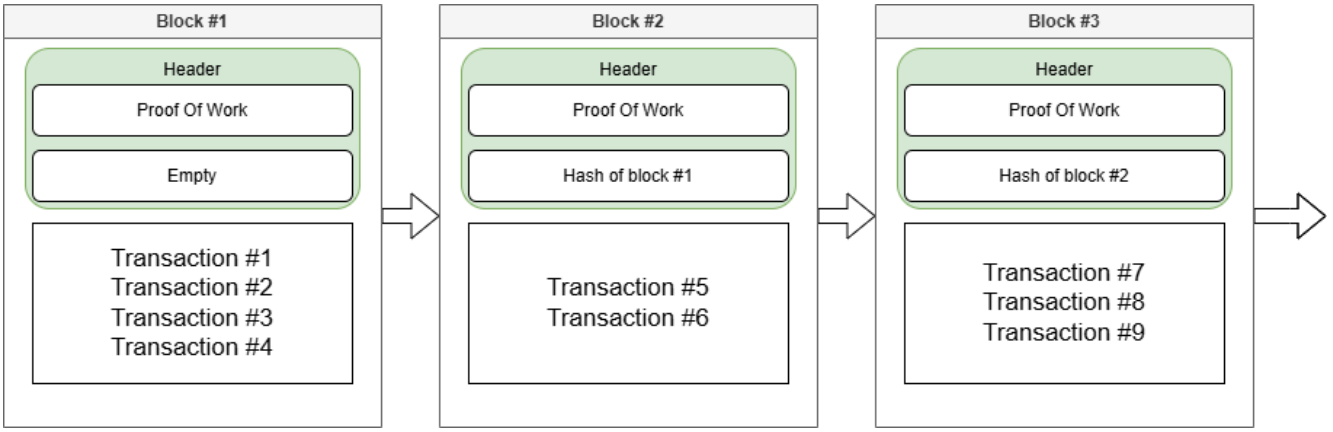


Рисунок 2.2 – Візуалізація блоків у блокчейні

Область застосування блокчейну є доволі широкою. Найчастіше блокчейн використовується у фінансовій сфері, точніше – у криптовалюті, для забезпечення прозорості транзакцій та унеможливлення компроментації даних транзакції. В сфері логістики блокчейн використовується для відслідковування надходження товарів, щоб не було ситуації, коли одна з сторін намагається привласнити кошти чи товар шляхом обману. Окрім цього, блокчейн може використовуватись у сфері музики, фільмів для затвердження авторських прав. У сфері освіти, блокчейн може використовуватись для затвердження факту отримання диплому, чи здачі екзамену. З наведених прикладів можемо дійти висновку, що блокчейн використовується для затвердження факту виконання певної дії, гарантує достовірність даних, оскільки жодна з сторін не може скомпроментувати дані. Доступ до даних мають лише авторазовані учасники мережі. Завдяки тому факту, що всі блоки у блокчейні пов’язані між собою, можна легко відслідкувати повний шлях, що пройшла транзакція.

Окрім зберігання інформації про транзакції, блокчейн дає змогу створювати смарт-контракти. Смарт контракти дозволяють автоматизовувати бізнес процеси за допомогою описаного алгоритму, який виконується в спеціальному середовищі, без необхідності втручання третьої сторони. Як приклад, за допомогою смарт контрактів можна автоматизувати процес оплати замовлень між постачальником та замовником. Одним із можливих сценаріїв автомазиції – налаштувати смарт контракт таким чином, щоб постачальник отримував кошти лише в момент реєстрації факту прибуття товару замовнику.

Існує чотири типи блокчейнів:

– Публічний;

– приватний;

– гібридний;

– consortium (співтовариство).

Публічний блокчейн не обмежує дій користувачів. Користувач, який є частиною публічного блокчейну має доступ до поточних та минулих записів, має доступ до перевірки транзакцій, виконання proof-of-work. Саме публічний блокчейн використовується у криптовалютах, таких як Bitcoin, Ethereum, Litecoin.

Приватний блокчейн обмежує дії користувачів, що працює лише в закритій мережі. Зазвичай цей тип блокчейну використовується у закритих організаціях, де лише деякі обрані члени є учасниками мережі. Компанія-власник блокчейну повністю керує безпекою, авторизацією дозволами блокчейну.

Співтовариський блокчейн є напівдецентралізованим блокчейном, у якому декілька організацій керує мережею блокчейн. В співтовариському блокчейні декілька організацій може виступати у ролі вузла.

Гібридний блокчейн поєднує у собі властивості публічного і приватного блокчейнів. В такому блокчейні може існувати і публічна мережа, доступ до якої мають всі члени, так і приватна мережа, в якій доступ мають лише учасники з певними правами. [32]

Всі типи блокчейнів використовують алгоритм консенсусу, щоб зробити компроментацію даних неможливою. Цей алгоритм дає змогу комп’ютерам, які я членами блокчейн мережі, перевірити факт достовірності інформації. Це необхідно, оскільки серед членів мережі можуть бути потенційні зловмисники, тобто створювати транзакції чи блоки без попередньої перевірки не можна. Кожен член в мережі має повну копію історії транзакцій. Ціль у тому, щоб всі історії транзакцій збігались у всіх членів мережі. Це дозволяє перевірити факт підробки певних транзакцій зловмисником, оскільки у всіх інших учасників мережі історія транзакцій залишиться, навіть у випадку, якщо зловмисник певним чином зможе модифікувати дані транзакції у себе. В алгоритмі консенсусу використовуються

алгоритми, які перевіряють певний факт, на ооснові якого учасник потім може (або не може) створити новий блок. Є доволі багато алгоритмів алгоритмів, що використовуються у алгоритмі консенсусу, серед яких найпоширенішими є:

- Proof of work (PoW);
- proof of stake (PoS);
- proof of authority (PoA);
- proof of elapsed time (PoET);
- proof of burn (PoB);
- proof of capacity (PoC).

Розглянемо кожен із наведених алгоритмів детальніше. Proof of Work є одним із найстаріших алгоритмів, але при цьому і одним із найбільш надійних. Суть полягає у тому, що перший комп'ютер, який зможе розв'язати хеш-задачу, зможе створити блок. Для розв'язання задачі беруться початкові дані, до них додається nonse (число) і на основі цих даних вираховується хеш, частіше всього SHA256. Наступним кроком є перевірка отриманого хешу на відповідініть правилам PoW. Найвидшив варіантом є перевірка, чи отриманий хеш починається з певної кількості нулів. Кількість нулів залежить від рівня складності алгоритму. Більш безпечним та поширеним, але при цьому більш затратним є метод перевірки отриманого хешу на на входження в діапазон очікуваного значення. Грубо кажучи, відбувається брут-форс, поки не отримаємо правильне значення. Це потребує доволі значного часу і значного навантаження на пристрій-майнер. За умови, що правильний хеш знайдено створюється новий блок у блокчейні, який окрім бізнес інформації містить інформацію про значення nonce а також отриманий хеш. Таким чином кожен може перевірити, що хеш справді було знайдено шляхом перебору. Ці дані фактично і є доказом роботи. Швидкість роботи цього алгоритму повністю залежить від потужності пристрою, який буде виконувати операцію перебору хешів. [33]

Proof of Stake, на відміну від Proof of Work, не змушує вирішувати хеш задачі, щоб створити новий блок. Тут ймовірність створення нового блоку повністю залежить від кількості монет, які на даний момент зберігаються у вузлі у

замороженому вигляді, тобто власники монет забов’язуються зберегти цю кількість монет, без права їх використання. Це зроблено з метою того, щоб забезпечити захист від зловмисників, оскільки у разі виявлення зловмисних намірів, заморожені монети, які виступають у ролі застави, будуть зписані з зловмисників, а учасники, які дотримувались правил, будуть винагороджені. Основними факторами відбору на право створити блок виступають кількість монет та час їх зберігання (чим більше часу, тим краще). Цей алгоритм не потребує значних потужностей комп’ютера, але має значний мінус – новим учасникам буде набагато важче отримати право на створення блоку, оскільки чим багатший учасник, тим більше шансів у нього, що призводить до нерівної конкуренції за блок. [34]

Proof of Authority полягає у тому, що валідатори ризикують не монетами, а своєю репутацією. Тут захист відбувається на основі довірених вузлів валідаторів, які вже є затвердженими і виступають у ролі модераторів. Цей алгоритм є набагто швидшим у порівнянні з попередніми і дозволяє виконувати набагто більше транзакцій в секунду. Стати довіреним валідатором доволі важко, оскільки треба внести інформацію про свою спрвжню особу, інколи внести грошову заставу. [35]

Proof of Elapsed Time це алгоритм, розроблений компанією Intel, що використовується у приватних блокчейн мережах. Оскільки це приватна мережа, всі її учасники мають підтвердити свою особу, внаслідок чого, можна мінімізувати затрати на валідацію довіреності та надійності осіб. В цьому алгоритмі кожен учасник має рівні шанси на право створення нового блоку. Суть полягає у тому, що кожен учасник отримує випадково згенерований час очікування, протягом якого вони мають очікувати, або виконувати будь-яку іншу дію, узгоджену правилами. Учасник, що завершить виконувати ці дії першим, буде мати право на шанс створення нового блоку. Важливо зауважити, що всі операції мають виконувати в Trusted Execution Environment, щоб учасники не могли підмінити час очікування. Для підтвердежння, кожен учасник разом із випадковим часом, отримує сертифікат, що підтверджує його достовірність. Цей процес зображено на рисунку 2.3.

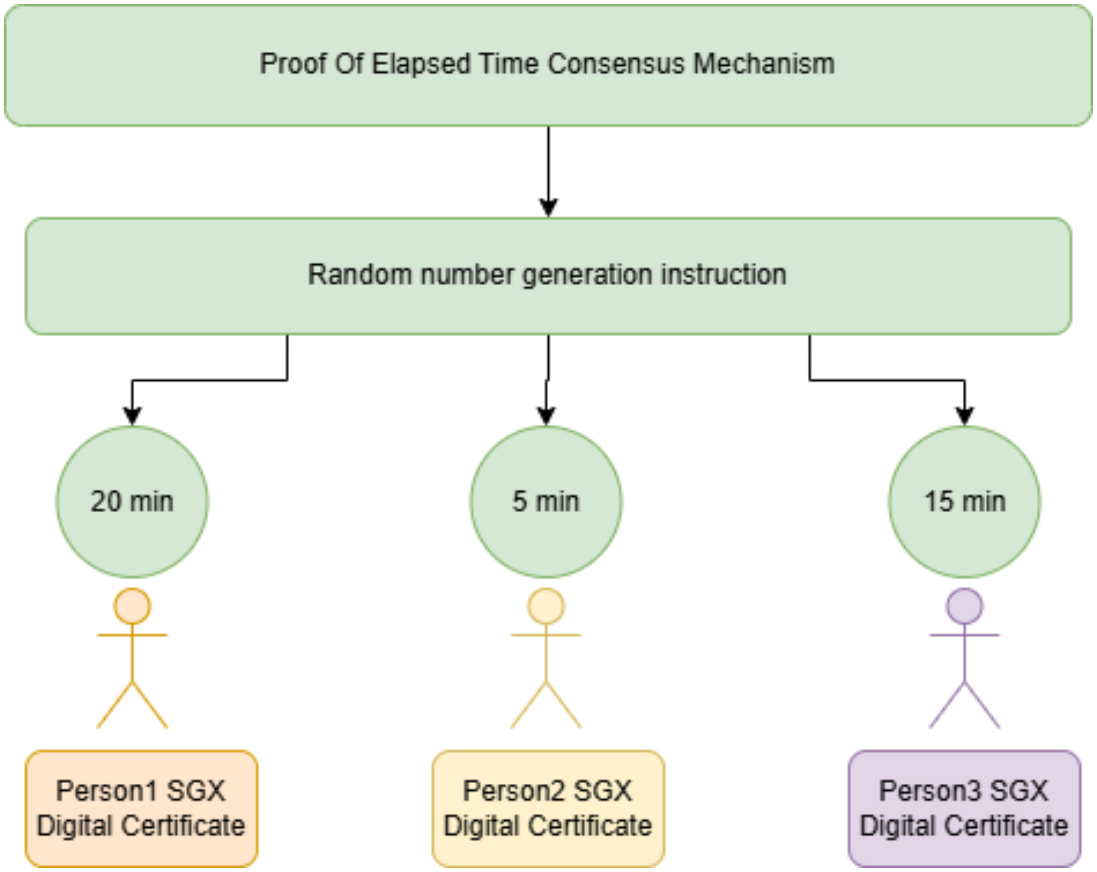


Рисунок 2.3 – Вхідні данні учасників

Цей алгоритм не є поширеним, та вважається експерементальним, але при цьому він є достатньо швидким у виконанні і незатратним по ресурсам, оскільки важких операцій підрахунку не відбувається. [36]

Proof of burn відрізняється від попередніх алгоритмів, оскільки для того, щоб отримати право на створення блоку, учасники мають «спалити» частину своїх монет. Учасники переводять монети на адресу, яка називається «eater», в результаті чого вони назавжди втрачають доступ до переведених монет, без можливості їх повернути, та отримують право на створення блоку, пропорційно до кількості втрачених монет. Цей алгоритм є дуже швидким у виконанні, але як і у PoS, чим багатший учасник – тим більше шансів у нього на створення блоку. Також, через спалювання монет, ціна на відповідну монету є нестабільною та змінюється частіше ніж інші. [37]

Proof of Capacity, як і Proof of Work, вимагає від учасників виконання процесу майнінгу. PoC вимагає від учасників значного місця для зберігання на носіях, так як вся суть алгоритму полягає у тому, що учасники використовують

вільне місце для майнінгу та зберігання результатів. Цей алгоритм є значно швидшим за PoW, приблизно у 2.5 рази, але все ж загальний час залежить тільки від потужності пристрою, на якому відбуваються криптографічні обчислення. Тут використовується криптографічна хеш функція під назвою Shabal, результати виконання якої і зберігаються на носіях учасників. Цей результат можна порівняти з nonce, що використовується у PoW – чим більше згенеровано, тим більший шанс на генерацію блока. Після генерації блоків (та вичерпання місця на носіях) відбувається процес генерації «дедлайнів». Дедлайн це період часу, що має пройти від моменту генерації попереднього блоку, перш ніж майнер зможе додати новий. Цей процес відбувається для кожного згенерованого nonce на диску, і з цієї вибірки обирається найкоротший дедлайн. Якщо щоден інший учасник не додав новий блок протягом цього дедлайну, то учасник, що згенерував відповідний дедлайн зможе додати новий блок. Цей алгоритм є відносно швидким, але вимагає від учасника значних обсягів для зберігання даних на власному фізичному диску. [38] Розглянемо порівняння характеристик алгоритмів у таблиці 2.2.

Таблиця 2.2 – Порівняння різних алгоритмів консенсусу

Критерій	Proof of Work (PoW)	Proof of Stake (PoS)	Proof of Authority (PoA)	Proof of Elapsed Time (PoET)	Proof of Burn (PoB)	Proof of Capacity (PoC)
1	2	3	4	5	6	7
Принцип дії	Обчислення складних задач	За кількістю монет	Авторитетні вузли затверджують блоки	Випадкове очікування часу	Спалювання моент	Майнери зберігають дані на диску
Споживання енергії	Дуже високе	Низьке	Дуже низьке	Низьке	Середнє	Помірне
Обладнання	GPU/ASIC	Гаманці	Сервер з надійною ідентифікацією	Процесор з TEE	Монети	Жорсткий диск великого об’єму
Умова	За витрачені обчислення	За кількість монет	Фіксована або за довіру	За час	За кількість монет	За мінімальний «дедлайн»

Кінець таблиці 2.2

1	2	3	4	5	6	7
Безпека	Висока	Висока	Від надійності вузлів	Від довіри до АЗ	Надійна	Висока
Централізація	Відносно децентралізована	Середній рівень централізації	Висока (контроль кількома вузлами)	Середня	Середня	Низька
Швидкість блоків	Низька (висока затримка)	Висока	Дуже висока	Висока	Висока	Висока
Стійкість до атак	Висока	Висока	Залежить від довіри до вузлів	Залежить від довіри до Intel SGX	Висок	Середня, залежить від фізичного носія
Приклад використання	Bitcoin, Ethereum (до 2.0)	Ethereum 2.0, Cardano	VeChain, POA.Network	Hyperledger Sawtooth	Slimcoin, Counterparty	Burstcoin, Chia

Невід'ємною частиною блокчейнів є смарт контракти. Смарт контракти допомагають автоматизувати бізнес процеси, шляхом виконання певного алгоритму, який був закладений у цей контракт, за умови виконання певної дії. Як приклад, можна автомазивувати оплату за надання послуг, залежно від факту прибуття товару, чи інших необхідних умов. Це дозволяє уникнути потреби третьої довіреної сторони, оскільки контракт працює суто за алгоритмом, і жодна сторона на це вплинути не може. Що не меньш важливо, на відміну від звичайних програм, де можна змінити код, і про це ніхто не дізнається, з смарт контрактами так не вийде. Смарт контракт не можна редагувати, лише видалити та замінити на новий, а це не може залишитись не поміченим іншими учасниками, що забезпечує повну прозорість процесу, та високу довіру. Після деплою, смарт контракт зберігається на блокчейні і його код може переглянути кожен бачаючий учасник мережі. Для написання смарт контрактів використовують різні мови програмування, залежно від платформи серед яких:

- Solidity;
- vyper;
- rust;

– move;

– go;

– java script;

– java;

– c#;

З перелічених, найпопулярнішою мовою є Solidity, який використовується у Ethereum, Binance Smart Coin. Ця мова програмування дозволяє доволі легко створювати смарт контракти, має зрозумілий синтаксис, та широкий набір вбудованих інструментів для прискорення розробки смарт контрактів. [39]

Тепер постає питання, яким чином технологія блокчейн може допомогти нам підвищити захист конфіденційних даних бази даних від sql-ін'єкцій. Насправді, способів безліч, серед яких варто виокремити два. Перший спосіб – прибрати всю потенційно критичну конфіденційну інформацію з бази даних, та перенести її на приватний блокчейн. При такому підході, навіть у разі порушення конфіденційності бази даних, критичні дані, по типу інформації про адреси, кредитні карти користувачів, конфіденційну бізнес логіку, номера телефонів користувачів, інформацію про транзакцій і тому подібне, не буде розкрито, оскільки вона децентралізовано зберігається на блокчейні. Хоча це забезпечує високий захист інформації, та високу стійкість до компоментації, цей підхід також створює безліч проблем. Користувач ніколи не зможе повністю видалити інформацію про себе, оскільки видалення неможливо у блокчейні. Можна буде створити окрему транзакцію, та оновити дані, але старі дані назавжди залишаться у історії транзакцій. Також це може бути доволі затратно у реалізації, оскільки за кожен запис треба сплачувати «gas fee».

Другий запропонований спосіб полягає у використанні технології блокчейн не для зберігання конфіденційних даних, а для ведення журналу підозрілих запитів. Цей підхід не створює додаткових проблем, оскільки всі дані залишаються у базі даних. Суть полягає у тому, що кожен раз, коли ми виявляємо підозрілий запит до бази даних, ми будемо створювати нову транзакцію у блокчейн. Як приклад, можна вносити інформацію про запит у

параметризованому вигляді, інформацію про передані параметри, які можуть бути потенційно шкідливими в контексті sql-ін’єкцій, IP адресу, з якої цей запит прийшов, дата та час, коли була спроба. Тепер вибір стоїть лише у тому, чи будемо ми логувати абсолютно всі запити, чи лише шкідливі. Якщо будемо логувати всі, то отримаємо детальне логування всіх подій, що дасть нам не тільки підвищений захист, але й можливість проведення аналізу, для подальших покращень функціонування системи. При цьому підході обов’язково треба налаштувати кешування, наприклад використовуючи Redis, щоб зайвий раз не навантужвати систему і не створювати зайві транзакції. Інший підхід полягає у тому, щоб записувати у блокчейн лише підозрілі та потенційно шкідливі запити, а всі інші «здорові» запити записувати у таблицю в базі даних, чи логувати в інше місце, таке як файл, чи спеціальні системи, призначенні для цього. Як на мене, другий підхід є набагато кращим, оскільки ми будемо значно меньше навантажувати систему, створювати значно менше транзакцій, а одже і зменьшимо потенційні витрати на реалізацію. Окрім цього, ми будемо мати доступ до всіх шкідливих запитів, і нам не доведеться придумувати механізми фільтрування. І на останок, оскільки ми записуємо лише потенційно шкідливі записи, це значно спростить інтеграцію з смарт контрактами, якщо у них виникне потреба. Варто зауважити, що хоча ми й не записуємо абсолютно всі запити, сам по собі запит може бути доволі великим. Оскільки ціна gas fee залежить від кількості байтів, то варто також опрацювати і цей момент. Як приклад, встановити максимальну довжину запиту, та додати можливість налаштування логування. Як приклад, можна додати конфігураційне значення, яке буде змінювати вигляд, у якому запит записується. Можна його або зашифрувати, та повертати ключ для розшифрування разом з зашифрованим запитом, або ж замість запита записувати назву ендпоінта, та назву метода, в якому була спроба виконати SQL-ін’єкцію. Це дасть можливість розробникам програмного забезпечення власноруч перевірити цей метод на правильність, оскільки тепер вони точно знатимуть, що там є вразливість до ін’єкцій.

2.4 Висновки

У цьому розділі було проведено проєктування системи захисту інформації в базах даних SQL від sql-атак. Розглянули загально відомі випадки, коли внаслідок sql-ін’єкцій, компанії ризикували втратити конфіденційну інформацію, з чи це було пов’язано, та як вони це вирішили. Було описано, як неправильна конфігурація системи, та неправильне зберігання секретів конфігураційних файлів програми може призвести до повної втрати конфіденційних даних, та як цього можна уникнути, використовуючи стороні сервіси Key Vault, Secret Manager, чи без використання стороніх сервісів, розроблено допоміжну утиліту для безпечного експорту секретів конфігураційних файлів.

Найважливішою розкритою темою, є використання параметризації, як основного методу боротьби з SQL-ін’єкціями, наведено приклади реалізації за допомогою збережених процедур, а також використовуючи різні типи ORM, сформовано загальні рекомендації, щодо роботи з запитами бази даних, а особливо з параметрами, що потрапляє в запит шляхом вводу даних користувачем.

Як додатковий шар захисту, запропоновано використання технологію blockchain у поєднанні з смарт контрактами, для ведення незмінного журналу потенційно шкідливих запитів у базу даних. Також було оглянуто типи блокчейн мереж, їх плюси та мінуси, алгоритми роботи та область їх застосування.

Окрім всього вище перечисленого, згадувалась тема XSS загроз, чим вони дотичні до SQL-загроз, та як за допомогою лише одного підходу, зменшити ризик виникнення і SQL-ін’єкцій, і XSS.

На основі інформації, та підходів, що описані у цьому розділі, ми можемо перейти до розробки програмного забезпечення для захисту від sql-ін’єкцій.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ВИЯВЛЕННЯ SQL-ІН'ЄКЦІЙ У ДОДАТКАХ

3.1 Розробка варіантів використання програмного забезпечення системи захисту даних від sql-атак

Для проектування варіантів використання, нам необхідно визначити акторів. Оскільки наша система захисту складається з окремих модулів, які налаштовуються в момент запуску застосунку, і після цього не потребують активного людського втручання, то акторами є лише адміністратор веб-застосунку, який відповідальний за моніторинг справності застосунку, та розробник, який буде інтегрувати розроблені нами модулі, та проводити їх первинне налаштування, яке частково здатне змінювати поведінку модулів, або розширює їх функціонал. Після проведення налаштувань, модулі працюють повністю самі, без подальшої потреби людського втручання.

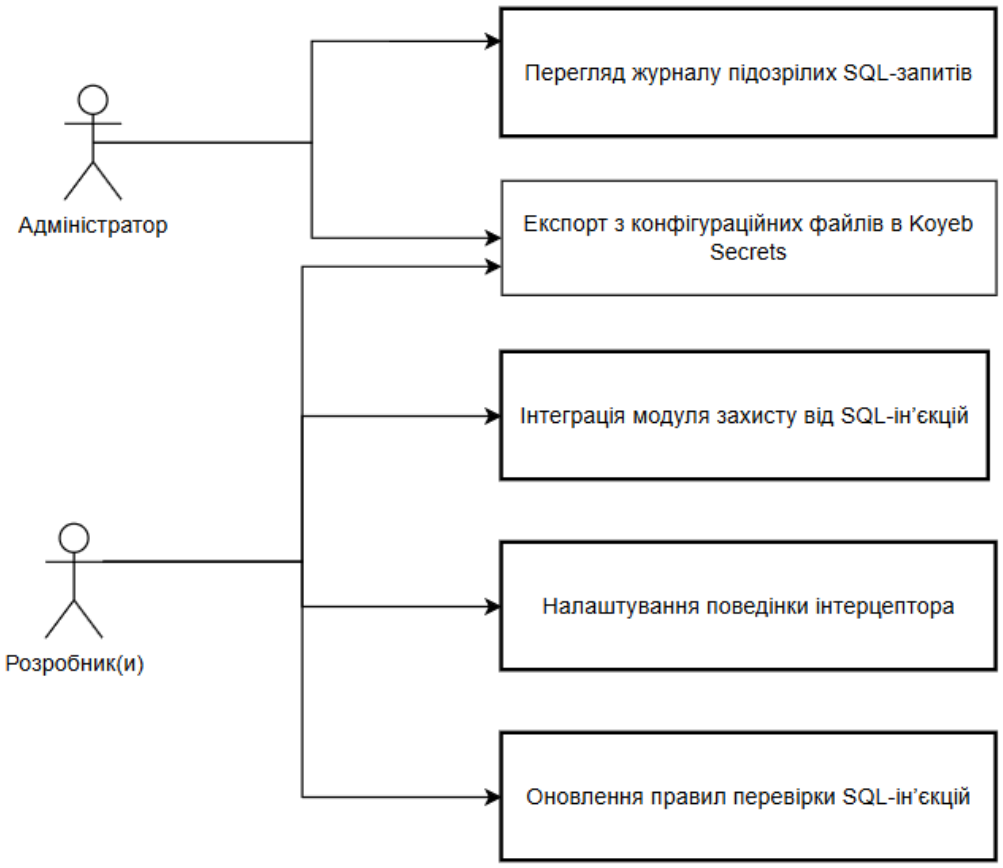


Рисунок 3.1 – Структурна схема варіантів використання програмного забезпечення акторами

Детальний опис кожного з варіантів, зображених на рисунку 3.1, наведено у таблицях 3.1 – 3.4.

Таблиця 3.1 – Варіант використання «Перегляд журналу підозрілих SQL-запитів»

Параметр	Опис
Найменування	Перегляд журналу підозрілих SQL-запитів
Первинний актор	Адміністратор
Інші актори	Всі члени мережі блокчейн
Опис	Отримання доступу до збережених записів потенційно небезпечних запитів
Попередні умови	Система функціонує, журнал ведеться
Вихідні умови	Актор переглянув потрібну інформацію

Таблиця 3.2 – Варіант використання «Інтеграція модуля захисту від SQL-ін’єкцій»

Параметр	Опис
Найменування	Інтеграція модуля захисту від SQL-ін’єкцій
Первинний актор	Розробник
Інші актори	Немає
Опис	Додавання інтерцептора до коду застосунку та ініціалізація налаштувань
Попередні умови	Розробник має доступ до коду та документації
Вихідні умови	Модуль інтегровано, система готова до виявлення підозрілих запитів

Таблиця 3.3 – Варіант використання «Налаштування інтерцептора»

Параметр	Опис
Найменування	Налаштування поведінки інтерцептора
Первинний актор	Розробник
Інші актори	Немає
Опис	Конфігурація параметрів, що впливають на правила визначення небезпечних запитів
Попередні умови	Модуль інтегровано, є доступ до конфігурацій
Вихідні умови	Налаштування застосовано, поведінка модуля змінена

Таблиця 3.4 – Варіант використання «Експорт з конфігураційних файлів в Koyeb Secrets»

Параметр	Опис
Найменування	Експорт з конфігураційних файлів в Koyeb Secrets
Первинний актор	Розробник або Адміністратор
Інші актори	Немає
Опис	Експорт секретів з конфігураційних файлів в безпечне сховище Koyeb Secrets
Попередні умови	Наявність конфігураційного файлу з секретами, як от наприклад стрічка підключення до бази даних, чи ключі стороніх API
Вихідні умови	Секрети надійно зберігаються у Koyeb Secrets, що забезпечує їх захист від несанкціонованого доступу

Взагалі, варіанти використання програмного забезпечення не обмежуються тільки на описаних вище. Завдяки модульності нашої системи захисту, доволі легко розширити наявну логіку, додавши перевірку на нові вразливості, патерни, покращити наявну логіку перевірок. Наприклад, у випадку з нашою системою, клієнт, що буде її використовувати, може розширити наявну логіку використовуючи спеціально залишені для цього методи, які можна перевизначити, або ж може розширити чи змінити логіку просто створивши відповідний інтерцептор, який наслідує інтерцептор реалізований у нашій системі захисту. Якщо ж сталось так, що клієнтський додаток потребує кардинальних змін, які не можуть бути реалізовані шляхом розширення наявного функціоналу через спеціально відведені для цього параметри, та методи, що можна перевизначити, то клієнт має можливість отримати вихідний код системи захисту, оскільки ця система є open source.

3.2 Розробка логічної та фізичної структури програмного забезпечення системи захисту даних від sql-атак

Розглянемо логічну структуру нашої системи захисту даних. Логічна структура передбачає у собі опис та відображення високорівневого дизайну

системи, з особливою увагою до функціонально-логічних модулів. До таких модулів можна віднести будь-які об’єкти програмного забезпечення, які беруть участь у виконані бізнес процесів системи. Оскільки наша система захисту складається з окремих модулів, які самі по собі не створюють жодних таблиць у базі даних Ms SQL, а лише доповнюють функціонал клієнтського коду, в якості об’єктів логічної структури будуть розглянуті наші окремі модулі, їх взаємодія між собою, а також між модулями, клієнтським застосунком, та мережею блокчейн. Структура система дозволяє додавати до клієнтського застосунку нові модулі, замінювати їх, чи розширювати, при цьому не ризикуючи зламати наявну логіку клієнтського застосунку. За потреби, у клієнта є можливість повністю перевизначити логіку розпізнання потенційно вразливих запитів, змінити провайдера секретів з Koyeb Secrets, на будь-якого іншого, оскільки утиліта експорту підтримує ряд найпоширеніших форматів для імпорту секретів конфігураційних файлів.

Для кращого розуміння інтеграції системи захисту у існуючий клієнтський застосунок, візьмемо застосунок, що призначений для завантаження та керуванням музичними треками з платформи Youtube та Youtube Music. Цей застосунок зберігає інформацію про користувачі, виконавців, службову інформацію для Youtube, що можна віднести до конфіденційних даних, які потребують захисту. Окрім цього, клієнтський застосунок використовує доволі багато стороніх сервісів, втрата ключів до яких призведе до критичних наслідків, серед яких:

- Firebase blob storage;
- supabase;
- hangfire;
- youtube api;
- yt-dll;
- youtube explode;
- firebase authentication.

Спочатку розглянемо принцип дії самої системи, не вдаючись у інфраструктуру клієнтського застосунку. Клієнтський застосунок поки позначимо умовно пустим. Загальну логічну структуру інтеграції системи захисту з клієнтським застосунком продемонстровано на рисунку 3.2 у вигляді UML діаграми.

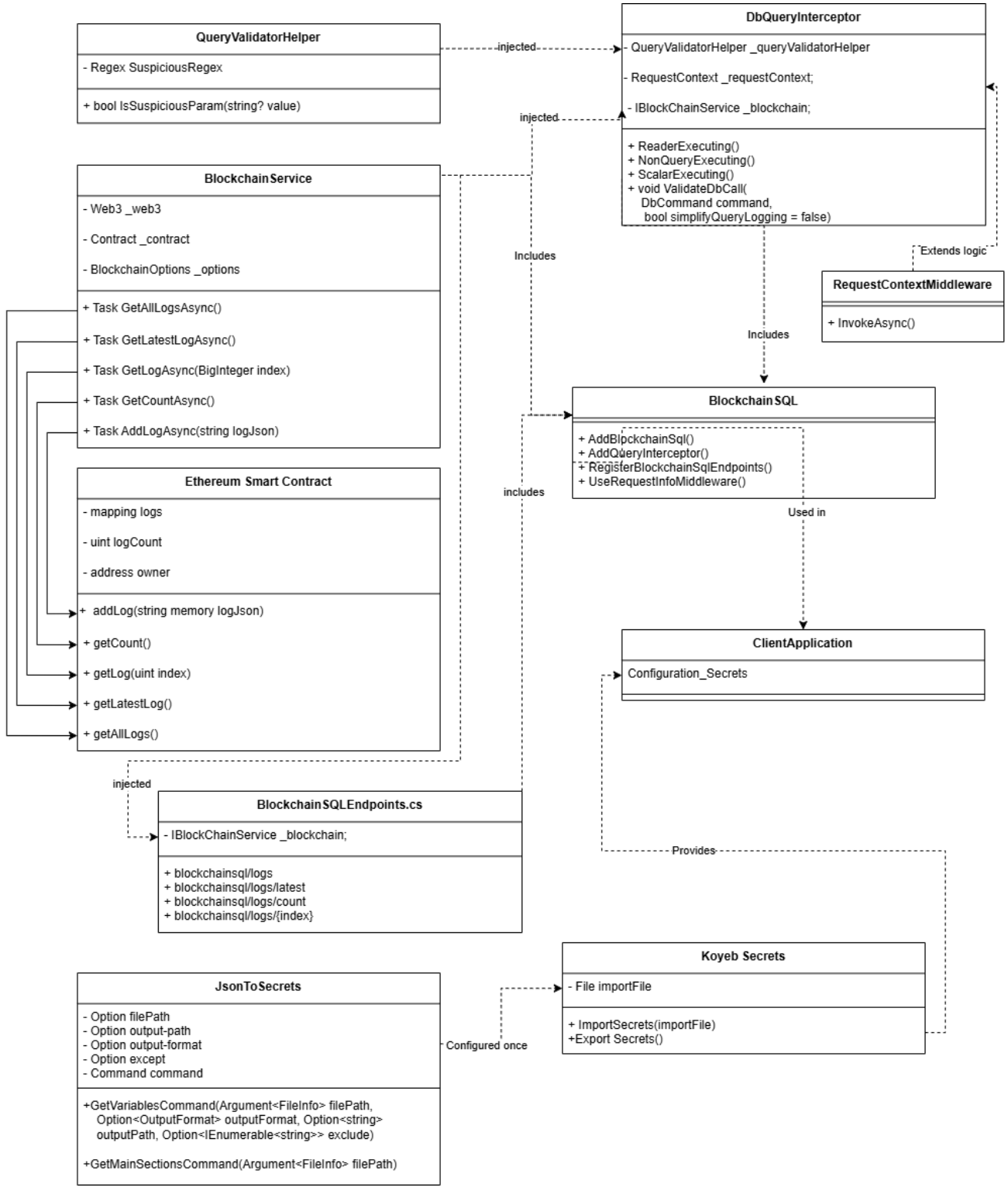


Рисунок 3.2 – Логічна структура програмного засобу

Лише по логічній діаграмі доволі важко зрозуміти призначення кожного з модулів, тому детальніше розглянемо призначення кожного з модулів та класів у таблиці 3.5.

Таблиця 3.5 – Список модулів програмного забезпечення та їх загальна характеристика

Модуль	Загальна характеристика призначення
JsonToSecrets	Призначений для конвертації конфігураційних файлів, що містять секрети, у файл готовий до імпорт в Koyeb Secrets, чи будь-який інший secrets manager
Koyeb Secrets	Secrets manager, який зберігає всі секрети з конфігураційних файлів, та в подальшому постачає їх клієнту, забезпечуючи високу безпеку
BlockchainSQL	Головний nu get пакет, що містить у собі модулі BlockchainSQLEndpoints, Blockchain Service, QueryValidatorHelper, DbQueryInterceptor, RequestContextMiddleware
BlockchainSQLEndpoints	Модуль, що дозволяє клієнтському додатку отримати доступ до готових кінцевих точок BlockchainSQL
Blockchain Service	Сервіс взаємодії з смарт контрактами
Ethereum Smart Contract	Смарт контракт у мережі Ethereum, який відповідає за керування логами
QueryValidatorHelper	Допоміжний клас, що містить логіку перевірки параметрів SQL запиту на вміст потенційних вразливостей
DbQueryInterceptor	Перехоплювач запитів, який залежно від результату перевірки запиту на входження вразливостей передає інформацію керування блокчейн сервісу, який в свою чергу логує інформацію про запит, використовуючи ethereum smart contract
RequestContextMiddleware	Допомагає збирати додаткову інформацію для логування запитів, зокрема таку, як айпі адреса, з якої надіслано запит, кінцева точка, з якої запит було ініційовано
Client Application	Довільний клієнтський додаток, в який буде інтегровано BlockchainSQL

Список ключів методів для основних модулів та класів, описаних у таблиці 3.5 наведено у таблицях 3.6 – 3.11.

Таблиця 3.6 – Список основних методів JsonToSecrets

Сигнатура метода	Опис
string ReadAllText(this FileInfo file)	Метод розширення, призначений для зчитування всього тексту з файлу
(bool IsValid, string ErrorMessage) IsValidJson(this FileInfo file)	Метод розширення, призначений для перевірки, чи файл у форматі Json
IEnumerable<JsonVariable> Extract(JsonProperty property, IEnumerable<string>? exclude = null)	Рекурсивно витягує та форматує значення з Json файлу. Надає змогу виключити певні ключі, чи повністю секції за допомогою опції exclude.
Command GetMainSectionsCommand(Argument <FileInfo> filePath)	Команда, яка виводить на екран користувача список секцій конфігураційного файла
Command GetVariablesCommand(Argument<FileInfo> filePath, Option<OutputFormat> outputFormat, Option<string> outputPath, Option<IEnumerable<string>> exclude)	Команда, що запускає процес експорту секретів з конфігураційного файлу на консоль користувача, або у файл

Таблиця 3.7 – Список основних методів BlockchainSQL

Сигнатура метода	Опис
AddBlockchainSql(this IServiceCollection services, IConfiguration configuration)	Здійснює реєстрацію необхідних сервісів для взаємодії з blockchain.
AddQueryInterceptor(this IServiceCollection services, IConfiguration configuration)	Здійснює реєстрацію необхідних сервісів та компонентів для перехоплення запитів SQL
UseRequestInfoMiddleware(this IApplicationBuilder app)	Здійснює реєстрацію проміжного програмного забезпечення, яке дозволяє логувати більш точну інформацію про запит
RegisterBlockchainSqlEndpoints(this IEndpointRouteBuilder app)	Здійснює реєстарцію набору кінцевих точок, які клієнтський додаток може використовувати для перегляду логів
virtual bool IsSuspiciousParam(string? value)	Метод з QueryValidatorHelper, який визначає, чи є параметр потенційно небезпечним

Таблиця 3.8 – Список основних методів BlockchainSQLEndpoints

Кінцева точка	Опис
blockchainsql/logs	Повертає список всіх записів логування
blockchainsql/logs/latest	Повертає найновіший запис логування
blockchainsql/logs/count	Повертає загальну кількість записів логування
blockchainsql/logs/{index}	Повертає запис логування під певнив індексом

Таблиця 3.9 – Список основних методів Blockchain Service

Сигнатура метода	Опис
string GetLatestLogAsync();	Викликає функцію смарт контракту, для отримання останього запису з логів
string GetLogAsync(BigInteger index)	Викликає функцію смарт контракту, для отримання логу за індексом
BigInteger GetCountAsync()	Викликає функцію смарт контракту, для отримання кількості лог записів
string AddLogAsync(string logJson)	Викликає функцію смарт контракту, для додавання нового запису в блок чейн
List<string> GetAllLogsAsync()	Викликає функцію смарт контракту, для додавання отримання всього списку логів

Таблиця 3.10 – Список основних методів Ethereum Smart Contract

Сигнатура метода	Опис
addLog(string memory logJson)	Додає новий лог у мережу блок чейн
getLog(uint index) returns (string memory)	Повертає лог за заданим індексом
getLatestLog() public view returns (string memory)	Повертає останній лог
getCount()returns (uint)	Повертає загальну кількість логів
getAllLogs() returns (string[] memory)	Повертає всі лог записи.
event LogAdded(string log, uint indexed index, address indexed sender)	Подія, яка генерується при додавані логу. Дозволяє легше відстежувати джерела змін.

Таблиця 3.11 – Список основних методів DbQueryInterceptor

Сигнатура метода	Опис
ReaderExecuting()	Перехоплення запитів читання
NonQueryExecuting()	Перехоплення запитів запису
ScalarExecuting()	Перехоплення скалярних запитів
ValidateDbCall(DbCommand command, bool simplifyQueryLogging = false)	Основний метод перевірки, що виконується при будь-якому виді перехоплення. Має спрощений режим, який зменшує використання gas

Тепер розглянемо фізичну структуру програмного забезпечення. Фізична структура програмного забезпечення передбачає відображення структури файлів програмного забезпечення, та як вони взаємодіють між собою. Оскільки під час опису логічної структури програмного забезпенння ми частково описали взаємодію між компонентами, в логічній структурі відобразимо лише раніше не описані частини. Під час розгляду фізичної структури програмного забезпечення буде відображено приклад справжнього клієнтського застосунку, в який інтегровано систему захисту від sql-ін'єкцій шляхом установки nuget пакета. Приклад зображено на рисунку 3.3.

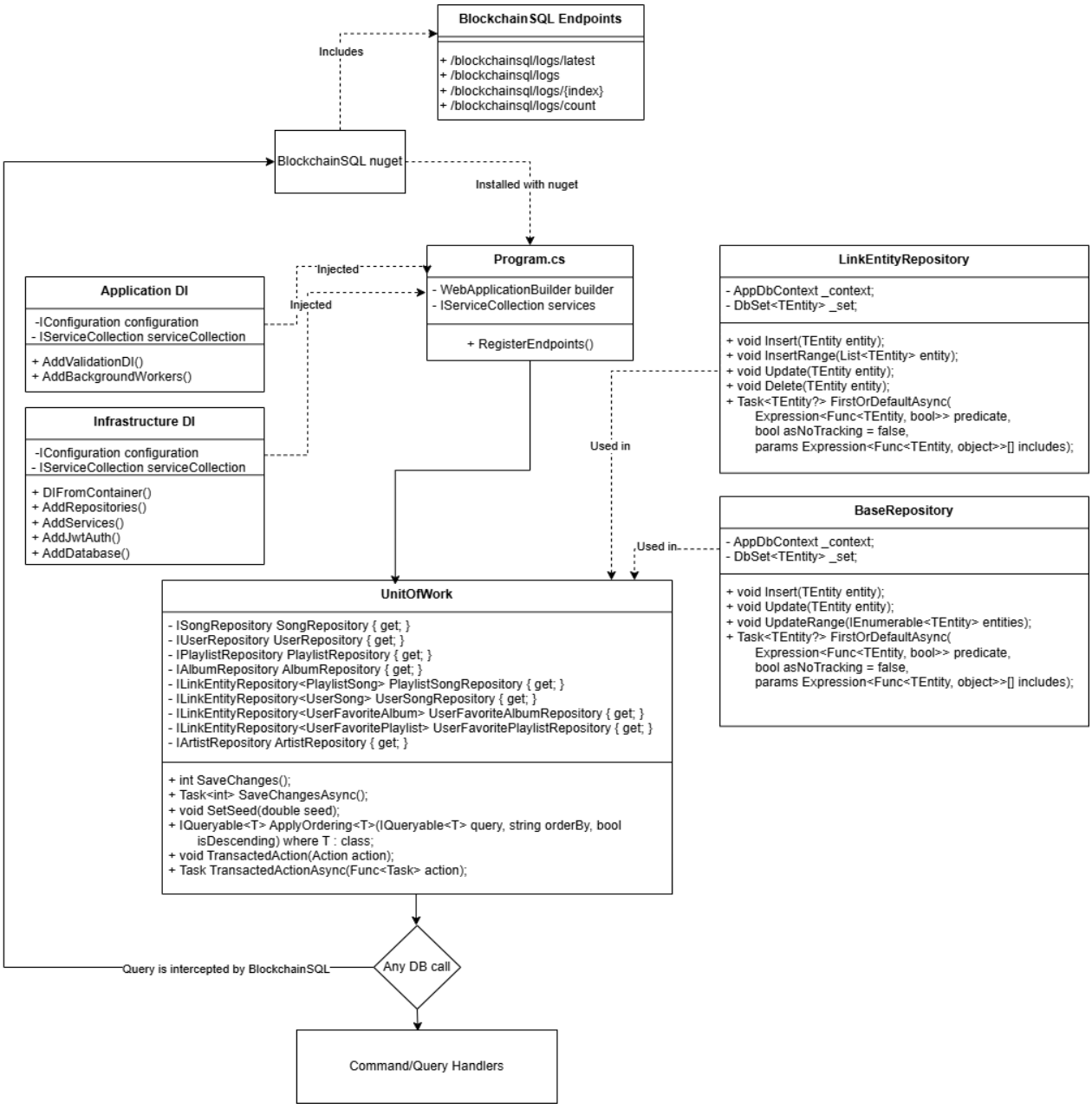


Рисунок 3.3 – Фізична структура з прикладом клієнта

З рисунка 3.3, ми бачимо, що наша система захисту, яка реалізована у вигляді nuget пакету, з легкістю інтегрується у будь-який клієнтський застосунок. Незалежно від інфраструктури рішень, що використовуються у клієнта, достатньо лише інсталювати nuget та використати допоміжні методи для реєстрації нашої системи у клієнтському додатку. Як приклад, клієнтський додаток, що зображений на рисунку 3.3, має окремі репозиторії для кожної сутності, кожен з яких міститься у єдиному unit of work. Без жодних змін в кодову базу клієнта, та без жодних змін у вже існуючій логіці додатка клієнта, вдалось інтегрувати систему захисту від sql-атак.

3.3 Розгортання та тестування програмного забезпечення системи виявлення SQL ін’єкцій у додатках

Основною умовою для розгортання системи захисту від sql-ін’єкцій є наявність всіх ключових компонентів для підключення до мережі блокчейн, а саме:

- Rpc url;
- contract address;
- abi;
- private key;
- account address;

Rpc url – це кінцева точка, яка дозволяє програмі взаємодіяти з мережею блокчейн. Через цю кінцеву точку ми можемо взаємодіяти з смарт контрактами, транзакціями, та в цілому із мережею блокчейн. Залежно від вузла, до якого ми підключились за допомогою rcp url, ми можемо помітити зміни у швидкодії інтеракцій з смарт контрактами, та загалом це впливає на надійність. Можна використовувати вже існуючі публічні вузли, або ж створити та налаштувати свій власний приватний вузол, все залежить від того, для яких потреб ми будемо його використовувати. [40]

Contract address – це унікальна адреса, за якою розгорнуто смарт контракт у мережі блокчейн. Оскільки в мережі може бути безліч смарт контрактів, цю адресу необхідно вказувати, аби звернутись до конкретного смарт контракту у мережі.

Abi (Application Binary Interface) - опис інтерфейсу смарт контракту у форматі, що робить можливою взаємодію смарт контракту і стороніх застосунків. Abi генерується автоматично при компіляції коду смарт контракту, та містить у собі повну інформацію про всі можливі функції контракту, про їх параметри та типи даних. [41]

Private key це приватний ключ, що дозволяє підписувати контракти від імені користувача-власника. Account address, в свою чергу, є публічною адресоб, що дозволяє ідентифікувати користувача. Ця адреса буде використовуватись при записуванні даних до смарт контракту.

За умови, що у клієнта, який буде використовувати нашу систему захисту, є вище описані дані, подальших трудностей з запуском системи виникнути не має. Клієнту залишилось лише інсталювати nuget пакет BlockchainSQL, інсталювати залежності, що випливають з цього пакету, та здійснити конфігурацію. Оскільки нашу систему захисту немає сенсу розгортати саму по собі, логічно буде розглянути приклад розгортання застосунку, що використовує нашу систему захисту від sql-ін’єкцій. Можемо розробити покрокову інструкцію розгортання клієнтського додатку, який використовує систему захисту BlockchainSQL:

– Інсталювання nuget BlockchainSQL;

– експорт секретів файлів конфігурацій;

– налаштування системи захисту під потреби клієнтського застосунку;

– формування dockerfile;

– розгортання клієнтського застосунку;

– перевірка функціонування системи BlockchainSQL шляхом спроби проведення sql-ін’єкцій.

Перший крок є найпростішим. Достатньо виконати наступну команду, перебуваючи у директорії проєкту «dotnet add package BlockchainSQL». Це завантажить останню версію пакета, а також вставовить необхідні залежності.

Наступний крок також є доволі простим. Необхідно перенести секрети конфігураційних файлів на безпечну платформу, яку обере клієнт. У нашому випадку це Koyeb Secrets. Використовуючи утиліту «JsonToSecrets» виконуємо наступні дії: спочатку робимо перелік всіх конфігураційних розділів, потім робимо експорт необхідних. За потреби, можемо скористуватись командою –help, щоб отримати підказки щодо процесу експорту. Цей процес зображено на рисунку 3.4.

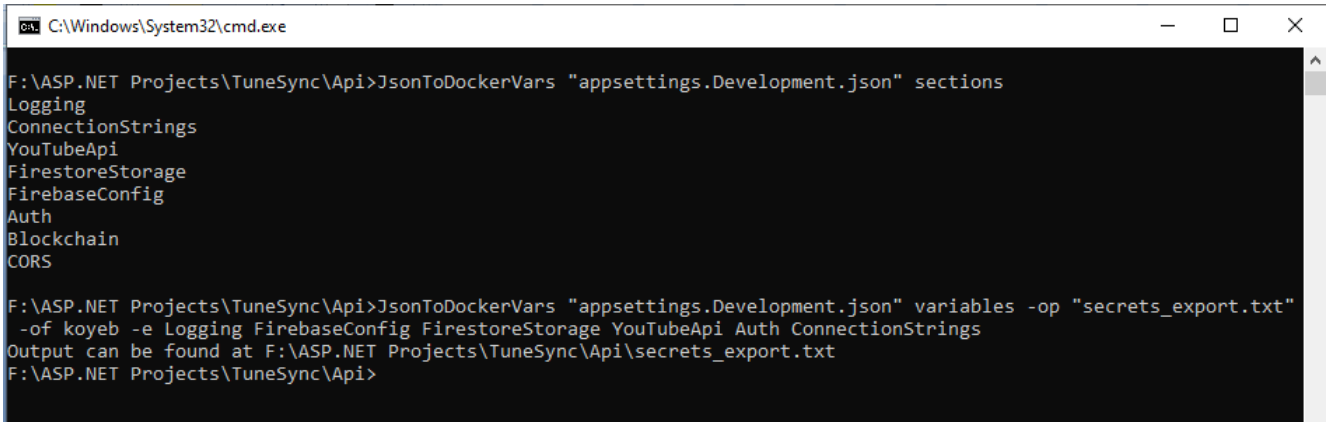


Рисунок 3.4 – Експорт секретів з конфігураційного файлу у форматі Koyeb

Як бачимо, файл було створено, нам залишилось лише завантажити його у розділі Secrets на платформі Koyeb. Після цього ми зможемо видалити конфігураційний файл з конфіденційними даними, та зтягувати їх з Koyeb Secrets. Також утиліта дає нам можливість експортувати дані у форматі сумісному з docker, що є корисним для тестування системи перед її остаточним розгортаням. Для цього нам достатньо замінити параметр output format на docker_file, або docker_string. Обидва варіанти працюють однаково, різниця лише в сумісності з секретами, які містять спеціальні символи, зокрема «\n» та «\t». Якщо такі символи є, то рекомендується використовувати параметр docker_string, в іншому ж випадку можна використати docker_file, та отримати команду у більш читабельному вигляді.

З інстальованим пакетом, та з налаштованими конфігураційними файлами, ми можемо перейти безпосередньо до налаштування системи захисту під потреби клієнта. Розпочинаємо налаштування з виклику методів ініціалізації. Вони зареєструють необхідні сервіси в контейнер dependency injection. Якщо у зв'язку з певними обставинами клієнт не хоче використовувати наш перехоплювач, але зацікавлений у використанні блокчейн сервісу, достатньо просто не реєструвати перехоплювач та пов'язані із ним сервіси. Це не забов'язує клієнта до використання не потрібного йому функціоналу, та робить систему більш гнучкою. Приклад реєстрації сервісів та перехоплювача зображено на лістингу 3.1.

```
serviceCollection
    .AddBlockchainSql(configuration)
    .AddQueryInterceptor(configuration);

serviceCollection.AddDbContext<AppDbContext>((serviceProvider, options) =>
{
    var interceptor =
serviceProvider.GetRequiredService<DbQueryInterceptor>();

    options.UseNpgsql(connectionString)
        .AddInterceptors(interceptor);
});
```

Лістинг 3.1 – Реєстрація блокчейн сервісу та перехоплювача запитів

Впринципі, цієї конфігурації достатньо для мінімального функціонування системи. Ми зможемо логувати запити, але не будемо знати хто їх ініціював, з якої кінцевої точки. Для того, щоб логувати цю інформацію, нам необхідно додати всього один рядок. Нам необхідно зареєструвати проміжне програмне забезпечення (middleware), який імплементує BlockchainSQL, а також за потреби, можемо додати набір кінцевих точок для перегляду останього запиту, запиту за індексом, всіх запитів, а також кількості запитів. Реєстрацію обох варіантів проілюстровано на лістингу 3.2.

```
app.UseRequestInfoMiddleware();
app.RegisterBlockchainSqlEndpoints();
```

Лістинг 3.2 – Реєстрація проміжного програмного забезпечення та набору готових кінцевих точок

Система захисту повністю налаштована та готова до використання. Якщо клієнт має необхідність, він може змінити чи розширити наявну логіку перевірки параметрів на вразливість. Для цього, він може перевизначити метод «IsSuspiciousParam(string? value)» у класі QueryValidatorHelper. Як приклад, можна доповнити наявну, та перевіряти запити, які містять певні значення з чорного списку, створеного клієнтом, чи зробити перевірку на інші типи вразливостей. Наприклад, перевірку на вміст параметрів, які при збережені в базу даних та одальшому виведені на інтерфейс можуть спичинити вразливість до XSS атак. Окрім цього, клієнт може перевизначити абсолютно кожен метод перехоплювача запитів, та як приклад, додати до нього додаткові умови перевірок, чи добавити можливість логування вразливостей у файл, чи консоль, а не тільки у блокчейн.

Розібравшись з налаштуванням системи, можемо перейти до формування dockerfile. Dockerfile є надзвичайно корисним, оскільки він допомагає уникнути проблеми, коли застосунок працює на одному пристрої, але викидає помилки на іншому пристрої. Він допомагає нам досягти однакової роботи додатку шляхом контейнеризації. Це означає, що застосунок, разом із всіма його залежностями буде упаковано в контейнер, що може бути запущений на будь-якій системі, де є підтримка docker, чи іншого відповідного контейнерного рушія. Також це дозволяє налаштувувати автоматизовані процеси CI/CD, що значно спрощують процес тестування, оновлення, та розгортання застосунку, але доцільність цього залежить від масштабу та потреб застосунку. Для створення справного dockerfile, нам необхідно покроково прописати у ньому всі інструкції, необхідні для створення додатку. Немає сенсу вдаватись у повні деталі створення docker файлу для конкретно нашого застосунку, оскільки там не буде жодних спеціальних інструкцій для забезпечення роботи BlockchainSQL. Натомість, розглянемо мінімальну інструкцію для створення dockerfile для asp net застосунку:

- Обираємо базовий образ;
- копіюємо залежні файли;
- встановлюємо залежності (по типу nuget);

– налаштування середовища (налаштування портів, запуск скриптів, налаштування змінних середовища);

– команда для запуску контейнера;

– побудова docker-image (docker build -t image_name -f Dockerfile ..);

Тепер, маючи Dockerfile та docker image, ми можемо створити контейнер, викликавши «docker run --name container_name image_name:latest». За потреби, в цю команду передаються такі параметри, як налаштування портів, а також змінні. Повний Dockerfile для клієнтського додатку буде наведено у додатках. Як приклад, у випадку, якщо ми працюємо локально, ми можемо передати всі секрети конфігураційних файлів прямо у команді створення контейнера, і це буде працювати. [42] Для генерації стрічки з конфігураційними секретами для docker ми можемо використати JsonToSecrets з форматом виводу docker_file або docker_string. Тепер у нас є декілька варінтів подальшого розгортання додатку. Якщо доступ до нього необхідний лише з локальної мережі, то достатньо буде виконати операцію переадресації портів, додати відповідні налаштування портів при запуску контейнера та запустити його. [43] Якщо все зробили правильно, то застосунок буде доступний іншим користувачам у мережі за IP адресою хостової машини, та вказаним під час налаштувань портом. Якщо потрібно, щоб додаток був доступний абсолютно всім, з будь-якого пристрою в мережі інтернет, то треба обрати хостинг-провайдера. Як приклад, будемо використовувати безкоштовний хостинг від Koyeb. Koyeb є дуже простим у використанні, та робить значну частину роботи за нас, що робить його чудовим вибором для малих застосунків, та для людей, які не є добре ознайомлені з процесом хостингу. Як приклад, koyeb дає нам можливість створити деплой з репозиторію github. Якщо у репозиторії міститься dockerfile, то при деплої буде використаний саме він, тому рекомендується створювати його власноруч, оскільки в проектах з складною інфраструктурою він може бути автоматично згенерований неправильно. Додавши посилання на репозиторій з dockerfile, залишилось лише підтягнути секрети з Koyeb Secrets, та натиснути кнопку «deploy». [44] В нашому випадку було використано безкоштовний варіант хостингу. Безкоштовний варіант

передбачає у собі вибір з двох серверів: у Франкфурті, та у Вашингтоні. Було обрано Франкфурт, оскільки цей сервер має значно меньшу затримку. В безкоштовному варіанті, ми отримуємо доволі обмежені потужності хостового пристрою:

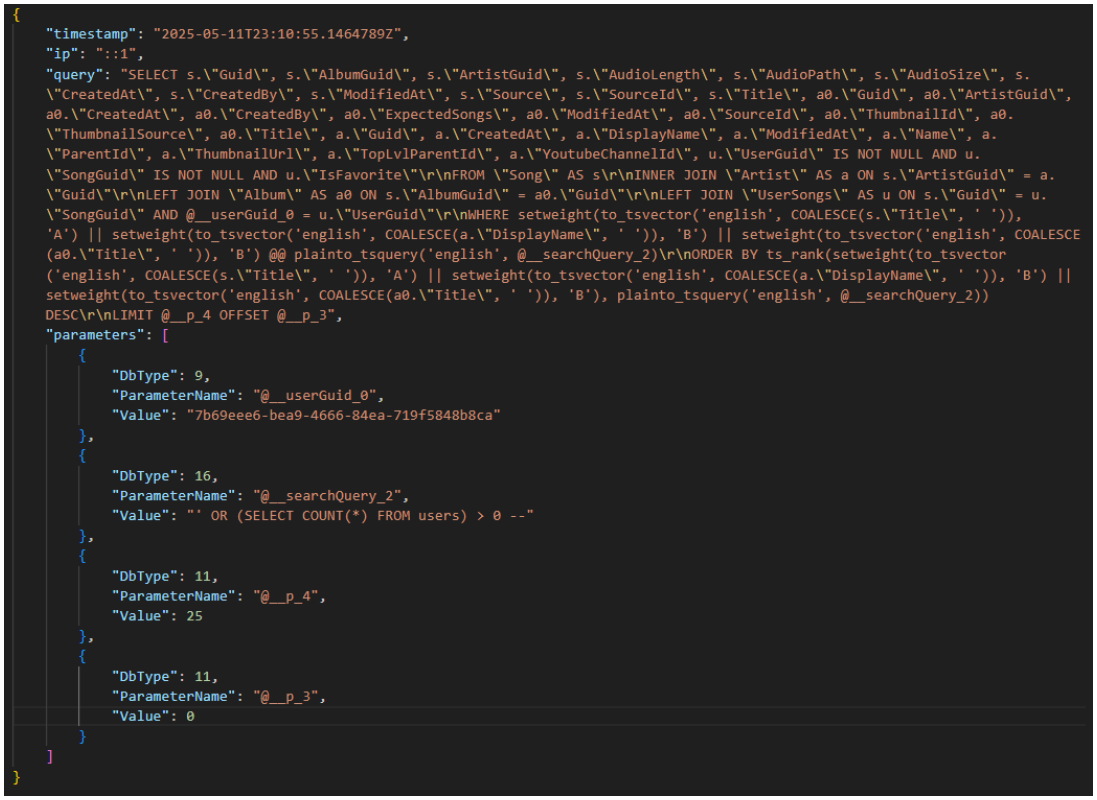
- 0.1 vCPU;
- 512mb RAM;
- 512mb DISK.

Може здатися, що цього замало, але насправді, цих ресурсів повністю вистачить, щоб забезпечити роботу невеличкого застосунку, яким щоденно користуються десятки, а інколи й сотні користувачів, за умови, що база даних буде хоститись окремо (наприклад на Supabase), 512 мегабайт RAM вистачить для організації механізму кешування, та запису тимчасових файлів. За потреби, завжди можна придбати доступ до потужнішої хостової машини.

Тепер, коли наш застосунок працює як локально, так і на хостингу, необхідно удостовіритись у тому, що наша система захисту від sql-ін'єкцій справді працює. Для перевірки, використаємо інструмент під назвою Ganache. Ganache дозволяє нам розгорнути локальну мережу приватного блокчейну, що дозволяє нам виконувати смарт контракти, та при цьому не створювати спам-записів у справжньому блокчейні, а що головніше – це повністю безкоштовно, оскільки ми працюємо у «пісочниці». Ganache повністю імітує справжнє Ethereum блокчейн середовище, що дозволяє миттєво створювати нові блоки та транзакції. Окрім цього, Ganache є продуктом з відкритим вихідним кодом, що підвищує фактор довіри до нього, та дозволяє створити свою модифіковану версію Ganache, завантаживши репозиторій, та модифікувавши його під свої потреби. [45]

Для використання Ganache з метою тестування системи захисту від sql-ін'єкцій, нам достатньо завантажити Ganache, створити тестову локальну блокчейн мережу, та повторити конфігураційні кроки, описані вище. Після створення контейнера, з відповідними налаштуваннями, та його запуску, ми нарешті можемо перейти до тестування системи захисту на sql-ін'єкції. Перевірка системи буде проводитись на всі види sql ін'єкцій, які були описані у попередніх

розділах. Тестування будемо проводити використовуючи різні кінцеві точки, для більшого покриття. Було проведено тестування всіх кінцевих точок, в яких є можливість пошуку, додавання, редагування даних, а також на сторінці авторизації. Приклади результату тестування можемо бачити на рисунках 3.5 – 3.8.



```
{
    "timestamp": "2025-05-11T23:10:55.1464789Z",
    "ip": "::1",
    "query": "SELECT s.\"Guid\", s.\"AlbumGuid\", s.\"ArtistGuid\", s.\"AudioLength\", s.\"AudioPath\", s.\"AudioSize\", s.
    \"CreatedAt\", s.\"CreatedBy\", s.\"ModifiedAt\", s.\"Source\", s.\"SourceId\", s.\"Title\", a0.\"Guid\", a0.\"ArtistGuid\",
    a0.\"CreatedAt\", a0.\"CreatedBy\", a0.\"ExpectedSongs\", a0.\"ModifiedAt\", a0.\"SourceId\", a0.\"ThumbnailId\", a0.
    \"ThumbnailSource\", a0.\"Title\", a.\"Guid\", a.\"CreatedAt\", a.\"DisplayName\", a.\"ModifiedAt\", a.\"Name\", a.
    \"ParentId\", a.\"ThumbnailUrl\", a.\"TopLvlParentId\", a.\"YoutubeChannelId\", u.\"UserGuid\" IS NOT NULL AND u.
    \"SongGuid\" IS NOT NULL AND u.\"IsFavorite\"\r\nFROM \"Song\" AS s\r\nINNER JOIN \"Artist\" AS a ON s.\"ArtistGuid\" = a.
    \"Guid\"\r\nLEFT JOIN \"Album\" AS a0 ON s.\"AlbumGuid\" = a0.\"Guid\"\r\nLEFT JOIN \"UserSongs\" AS u ON s.\"Guid\" = u.
    \"SongGuid\" AND @__userGuid_0 = u.\"UserGuid\"\r\nWHERE setweight(to_tsvector('english', COALESCE(s.\"Title\", ' ')),
    'A') || setweight(to_tsvector('english', COALESCE(a.\"DisplayName\", ' ')), 'B') || setweight(to_tsvector('english', COALESCE
    (a0.\"Title\", ' ')), 'B') @@ plainto_tsquery('english', @__searchQuery_2)\r\nORDER BY ts_rank(setweight(to_tsvector
    ('english', COALESCE(s.\"Title\", ' ')), 'A') || setweight(to_tsvector('english', COALESCE(a.\"DisplayName\", ' ')), 'B') ||
    setweight(to_tsvector('english', COALESCE(a0.\"Title\", ' ')), 'B'), plainto_tsquery('english', @__searchQuery_2))
    DESC\r\nLIMIT @__p_4 OFFSET @__p_3",
    "parameters": [
        {
            "DbType": 9,
            "ParameterName": "@__userGuid_0",
            "Value": "7b69eee6-bea9-4666-84ea-719f5848b8ca"
        },
        {
            "DbType": 16,
            "ParameterName": "@__searchQuery_2",
            "Value": "' OR (SELECT COUNT(*) FROM users) > 0 --"
        },
        {
            "DbType": 11,
            "ParameterName": "@__p_4",
            "Value": 25
        },
        {
            "DbType": 11,
            "ParameterName": "@__p_3",
            "Value": 0
        }
    ]
}
```


Рисунок 3.5 – Перехоплений запит з sql-ін’єкцією

Запит, в якому була спроба інк’єкції з шкідливим параметром «' OR (SELECT COUNT(*) FROM users) > 0 --» був перехоплений, та записаний на блокчейн. Бачимо створений блок на рисунку 3.6.

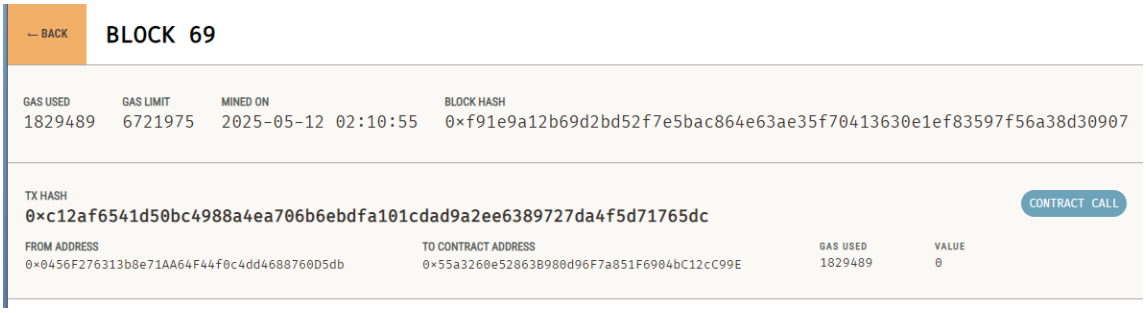


Рисунок 3.6 – Створений внаслідок перехоплення блок

Спробуємо проексплуатувати кінцеву точку, яка записує дані у базу даних. Було обрано ендпоінт, який створює нову групу треків, та дає можливість користувачу ввести назву для цього списку. Спробуємо використати time based sql-injection. Результат продемонстровано на рисунку 3.7.

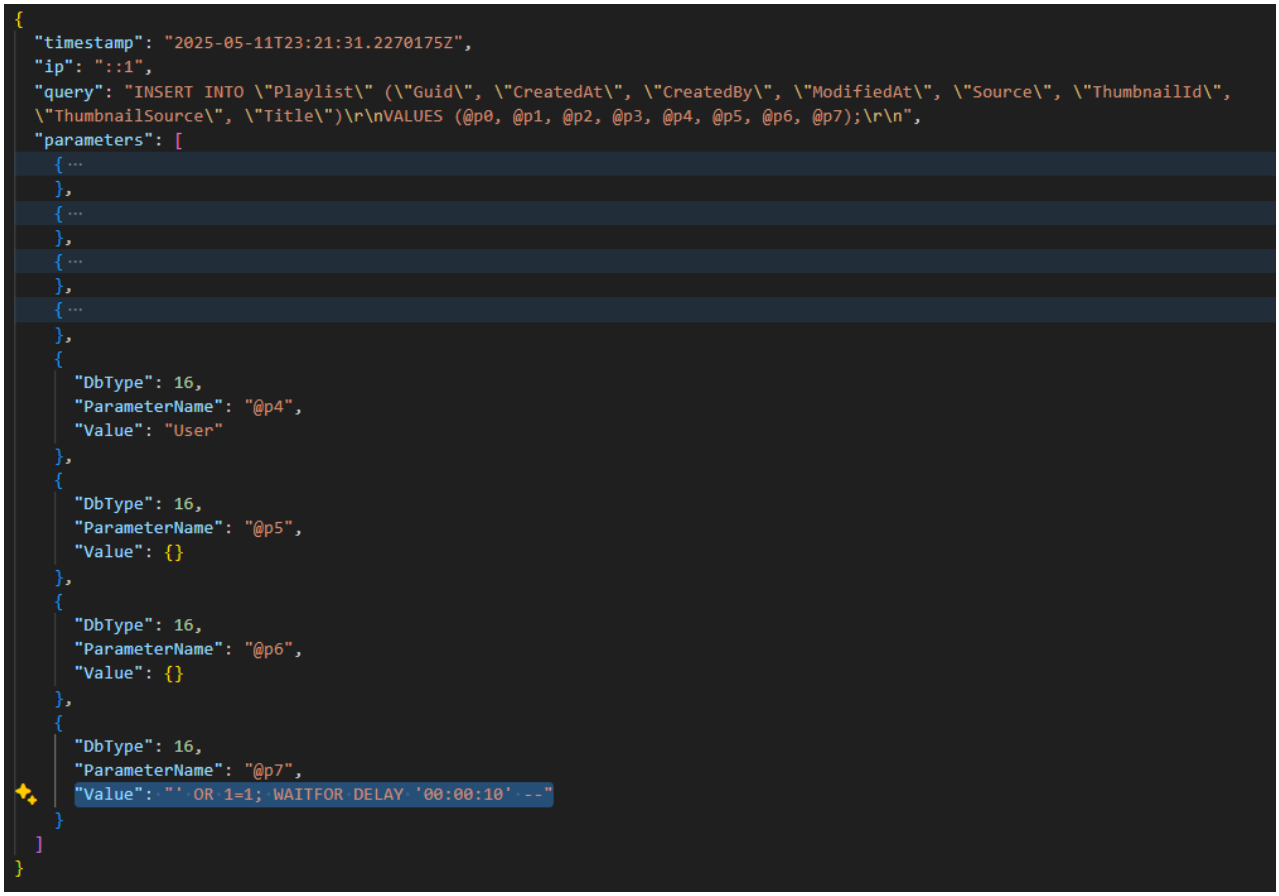


Рисунок 3.7 – Перехоплений запит з time based sql-injecton

Запит також було успішно виявлено то внесено в мережу блокчейн. Бачимо новий блок на рисунку 3.8.

Рисунок 3.8 – Створений блок внасладіок blind injection

3.4 Висновки

В цьому розділі було проведено розробку програмного забезпечення для виявлення sql-ін’єкцій в додатках.

Зформовано варіанти використання програмного забезпечення, логічне та фізичне представлення системи, описано основні компоненти та модулі, їх загальне призначення, принцип роботи, та як ці компоненти взаємодіять один з одним, чи з зовнішніми системами.

На основі зібраних даних та описаних представленях системи, було прийнято рішення імплементувати систему захисту у вигляді nuget пакету BlockchainSQL, щоб забезпечити простоту інтеграції з клієнтськими застосунками. Цей пакет включає у себе основний функціонал – перехоплення потенційно шкідливих запитів, з метою їх подальшого логування, а також, додатковий функціонал у вигляді простоти розширення логіки розпізнання зі сторони клієнта, готових кінцевих точок, що надають інформацію про перехоплені запити.

BlockchainSQL використовує смарт-контракт, що написаний на мові програмування Solidity, та призначена для роботи з Ethereum Blockchain.

Для збільшення безпеки даних конфігураційних файлів було розроблено утиліту JsonToSecrets, яка дозволяє експортувати необхідні секції, чи окремі значення конфігураційних файлів у файл, чи на консоль у різних форматах, зокрема форматі, що сумісний з Koyeb Secrets, два формати для Docker, а також експорт у інший json, з метою фільтрування секцій.

Для демонстрації працездатності системи, було розгорнуто клієнтський додаток, у який інтегровано BlockchainSQL. Додаток розгортався локально, та на платформі Koyeb, в обох випадках за допомогою Docker.

Проведено тестування на всі види sql-ін’єкцій на всіх можливих кінцевих точках клієнтського додатку. Всі потенційно вразливі запити було виявлено, перехоплено, та записано у мережу блокчейн.

Система захисту від sql-ін’єкцій працює, успішно виявляє потенційно вразливі захисти, та загалом готова до використання у реальних додатках.

ВИСНОВКИ

Завдання дипломної роботи було виконано, здійснено розробку програмного забезпечення для виявлення sql-ін’єкцій у додатках. Зокрема, здійснено розробку nuget пакета BlockchainSQL, смарт-контракту на Solidity, що працює з мережею Ethereum, а також допоміжну утиліту JsonToSecrets, щоб мінімізувати ризики втрати даних внаслідок людського фактору.

Окрім розробки програмного забезпечення, було проведено аналіз вже існуючих засобів захисту від sql-ін’єкцій, серед яких основним є використання параметризованих запитів, валідація вхідних даних, та використання методу «whitelist». Сформовано загальні рекомендації щодо роботи з базами даних, та оглянуто вбудовані механізми захисту конфіденційної інформації в MsSql.

Проведено загальний аналіз загроз конфіденційній інформації в MsSql, сформовано рекомендації щодо їх вирішення, серед яких можна виокремити використання сервісів Secret Manager, як от Koyeb Secrets, та захист від XSS атак, який частково покривається за допомогою BlockchainSQL, оскільки є точка розширення, в яку доволі легко це інтегрувати.

Були побудовані логічна та фізична моделі структури програмного забезпечення, сформовано основні варіанти використання. Розглянуто варіанти інтеграції з клієнтськими застосунками.

Програмне забезпечення було успішно протестовано на всі види ін’єкцій, на інтеграцію з смарт контрактами та мережею блокчейн. Усі шкідливі запити було виявлено та внесено до мережі блокчейн, яку потім було успішно переглянуто, щоб виявити деталі спроби вчинення sql-ін’єкції.

СПИСОК ДЖЕРЕЛ

1. Transparent data encryption 2024. С. 1. URL: https://learn.microsoft.com/en-us/sql/relational-databases/security/encryption/transparent-data-encryption?view=sql-server-ver17 (дата звернення: 09.05.2025)

2. Always Encrypted 2025. С. 1. URL: https://learn.microsoft.com/en-us/sql/relational-databases/security/encryption/always-encrypted-database-engine?view=sql-server-ver17 (дата звернення: 09.05.2025)

3. Always Encrypted with secure enclaves 2025. С. 1. URL: https://learn.microsoft.com/en-us/sql/relational-databases/security/encryption/always-encrypted-enclaves?view=sql-server-ver17 (дата звернення: 09.05.2025)

4. What is SQL Server? 2025. С. 1. URL: https://learn.microsoft.com/en-us/sql/sql-server/what-is-sql-server?view=sql-server-ver17 (дата звернення: 09.05.2025)

5. Require Strong Passwords 2025. С. 1. URL: https://www.cisa.gov/secure-our-world/require-strong-passwords (дата звернення: 09.05.2025)

6. Що таке двохфакторна автентифікація? 2025. С. 1. URL: https://www.microsoft.com/uk-ua/security/business/security-101/what-is-two-factor-authentication-2fa (дата звернення: 09.05.2025)

7. Best Practices for Dev, QA, and Production Environments: A Comprehensive Guide 2023. С. 1. URL: https://www.bunnyshell.com/blog/best-practices-for-dev-qa-and-production-environments/ (дата звернення: 09.05.2025)

8. Copy only backups 2024. С. 1. URL: https://learn.microsoft.com/uk-ua/sql/relational-databases/backup-restore/copy-only-backups-sql-server?view=sql-server-ver17 (дата звернення: 09.05.2025)

9. Full database backups 2024. С. 1. URL: https://learn.microsoft.com/en-us/sql/relational-databases/backup-restore/full-database-backups-sql-server?view=sql-server-ver17 (дата звернення: 09.05.2025)

10. Differential backups 2024. С. 1. URL: https://learn.microsoft.com/en-us/sql/relational-databases/backup-restore/differential-backups-sql-server?view=sql-server-ver17 (дата звернення: 09.05.2025)

11. Transaction log backups 2024. С. 1. URL: https://learn.microsoft.com/en-us/sql/relational-databases/backup-restore/transaction-log-backups-sql-server?view=sql-server-ver17 (дата звернення: 09.05.2025)

12. Tail-log backups 2024. С. 1. URL: https://learn.microsoft.com/en-us/sql/relational-databases/backup-restore/tail-log-backups-sql-server?view=sql-server-ver17 (дата звернення: 09.05.2025)

13. Full File Backups 2024. С. 1. URL: https://learn.microsoft.com/en-us/sql/relational-databases/backup-restore/full-file-backups-sql-server?view=sql-server-ver17 (дата звернення: 09.05.2025)

14. Partial Backups 2024. С. 1. URL: https://learn.microsoft.com/en-us/sql/relational-databases/backup-restore/partial-backups-sql-server?view=sql-server-ver17 (дата звернення: 09.05.2025)

15. Recovery models 2024. С. 1. URL: https://learn.microsoft.com/en-us/sql/relational-databases/backup-restore/recovery-models-sql-server?view=sql-server-ver17 (дата звернення: 09.05.2025)

16. SQL Server backup to URL for Microsoft Azure Blob Storage 2024. С. 1. URL: https://learn.microsoft.com/en-us/sql/relational-databases/backup-restore/sql-server-backup-to-url?view=sql-server-ver17 (дата звернення: 09.05.2025)

17. Що таке DDoS атака 2025. С. 1. URL: https://cip.gov.ua/ua/faqs/shotake-ddos-ataka (дата звернення: 09.05.2025)

18. Rate limiting middleware in ASP.NET Core 2025. С. 1. URL: https://learn.microsoft.com/en-us/aspnet/core/performance/rate-limit?view=aspnetcore-9.0 (дата звернення: 09.05.2025)

19. Лекція 9. Інструменти та програми для виявлення вторгнень 2025. С.1. URL:https://msn.khmnu.edu.ua/pluginfile.php/682862/mod_resource/content/1/Лекція%209.%20Інструменти%20та%20програми%20для%20виявлення%20вторгнень.pdf (дата звернення: 09.05.2025)

20. OWASP Top 10 – 2021 2021, С. 1. URL: https://owasp.org/Top10/ (дата звернення: 09.05.2025)

21. Types of SQL Injection (SQLi) 2025, С. 1. URL: https://www.geeksforgeeks.org/types-of-sql-injection-sqli/ (дата звернення: 09.05.2025)

22. SQL Injection Prevention Cheat Sheet 2024, С. 1. URL: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html (дата звернення: 09.05.2025)

23. Web Technology Vulnerabilities 1998, С. 1. URL: https://phrack.org/issues/54/8 (дата звернення: 09.05.2025)

24. Security Experts Exploit Airport Security Loophole with SQL Injection 2024, С. 1. URL: https://www.infoq.com/news/2024/09/sql-injection-airport-security/ (дата звернення: 09.05.2025)

25. CVE-2024-8395 Detail 2024, С. 1. URL: https://nvd.nist.gov/vuln/detail/cve-2024-8395 (дата звернення: 09.05.2025)

26. Finding more vulnerabilities in vibe coded apps 2025, С. 1. URL: https://www.intigriti.com/researchers/blog/hacking-tools/vibe-coding-security-vulnerabilities (дата звернення: 09.05.2025)

27. Command Injection 2024, С. 1. URL: https://owasp.org/www-community/attacks/Command_Injection (дата звернення: 09.05.2025)

28. AWS Secrets Manager vs Azure Key Vault comparison 2025, С. 1. URL: https://www.peerspot.com/products/comparisons/aws-secrets-manager_vs_azure-key-vault (дата звернення: 09.05.2025)

29. Koyeb Secrets 2025, С. 1. URL: https://app.koyeb.com/secrets (дата звернення: 09.05.2025)

30. Stored procedures (Database Engine) 2024, С. 1. URL: https://learn.microsoft.com/en-us/sql/relational-databases/stored-procedures/stored-procedures-database-engine?view=sql-server-ver16 (дата звернення: 09.05.2025)

31. What is blockchain 2025, С. 1. URL: https://aws.amazon.com/what-is/blockchain (дата звернення: 09.05.2025)

32. Features of BlockChain 2025, С. 1. URL: https://blockchain.gov.in/Home/BlockChain?blockchain=type (дата звернення: 09.05.2025)

33. Write your Own Proof-of-Work Blockchain 2019, С. 1. URL: https://www.jmeiners.com/tiny-blockchain/ (дата звернення: 09.05.2025)

34. Proof of Stake (PoS) 2024, С. 1. URL: https://academy.binance.com/uk-UA/glossary/proof-of-stake (дата звернення: 09.05.2025)

35. Proof of Authority Explained 2024, С. 1. URL: https://academy.binance.com/en/articles/proof-of-authority-explained (дата звернення: 09.05.2025)

36. What is Proof of Elapsed Time (PoET)? Unlocking the Secrets of Proof of Elapsed Time (PoET) 2023, С. 1. URL: https://medium.com/unicorn-ultra/what-is-proof-of-elapsed-time-poet-unlocking-the-secrets-of-proof-of-elapsed-time-poet-0bb7dd1b614e (дата звернення: 09.05.2025)

37. What is a Proof-of-Burn in crypto? 2025, С. 1. URL: https://www.coinbase.com/ru/learn/crypto-glossary/what-is-a-proof-of-burn-in-crypto (дата звернення: 09.05.2025)

38. Proof of Capacity 2025, С. 1. URL: https://crypto.com/glossary/proof-of-capacity (дата звернення: 09.05.2025)

39. What is a smart contract? 2025, С. 1. URL: https://www.coinbase.com/ru/learn/crypto-basics/what-is-a-smart-contract (дата звернення: 09.05.2025)

40. RPC URL 2025, С. 1. URL: https://thirdweb.com/learn/glossary/rpc-url (дата звернення: 10.05.2025)

41. What is an ABI of a Smart Contract? 2022, С. 1. URL: https://www.alchemy.com/overviews/what-is-an-abi-of-a-smart-contract-examples-and-usage (дата звернення: 10.05.2025)

42. Writing a Dockerfile 2024, С. 1. URL: https://docs.docker.com/get-started/docker-concepts/building-images/writing-a-dockerfile/ (дата звернення: 11.05.2025)

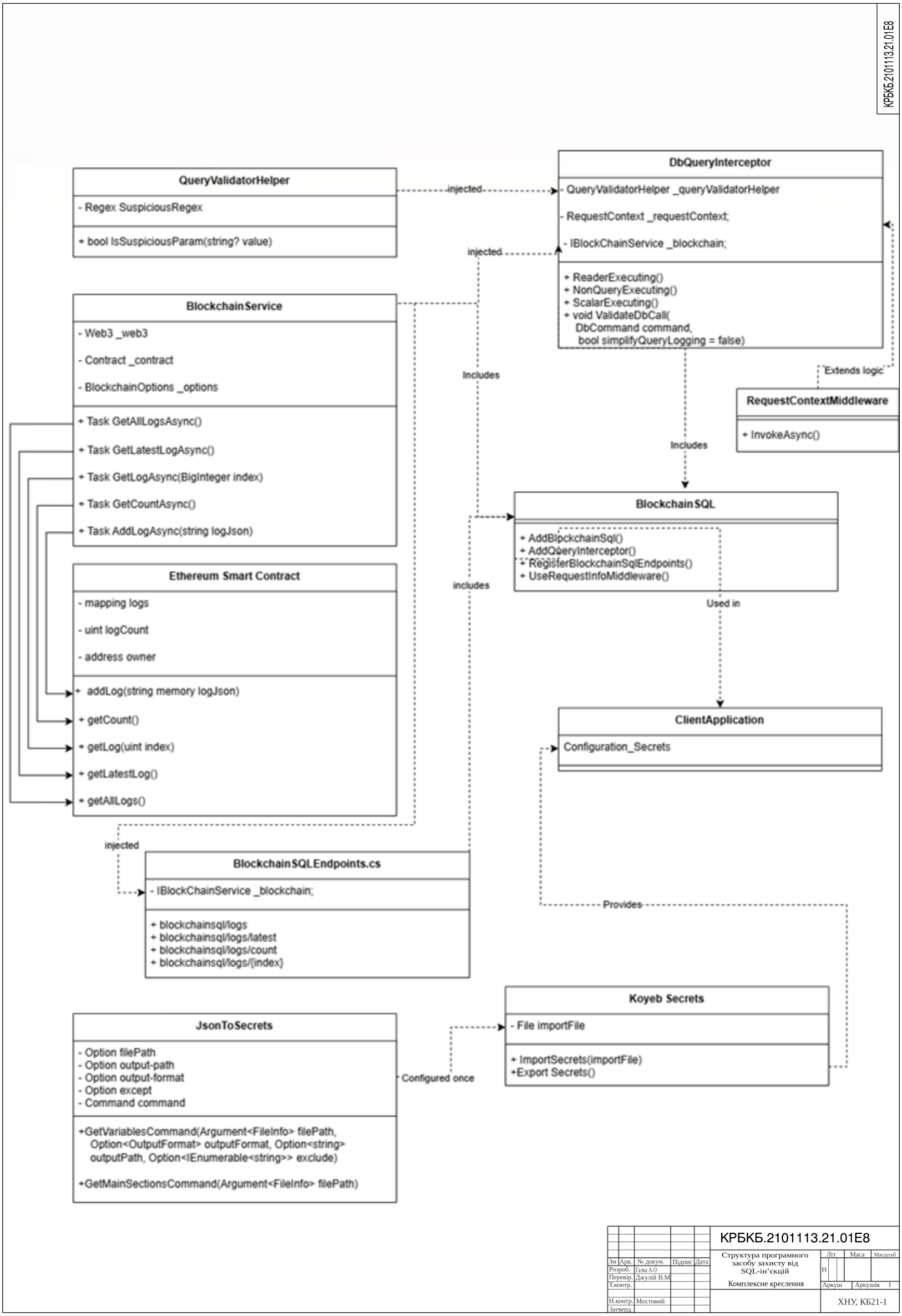
43. Port-forward access application cluster 2024, С. 1. URL: https://kubernetes.io/docs/tasks/access-application-cluster/port-forward-access-application-cluster/ (дата звернення: 11.05.2025)

44. Deploy with GitHub 2025, С. 1. URL: https://www.koyeb.com/docs/build-and-deploy/deploy-with-git (дата звернення: 12.05.2025)

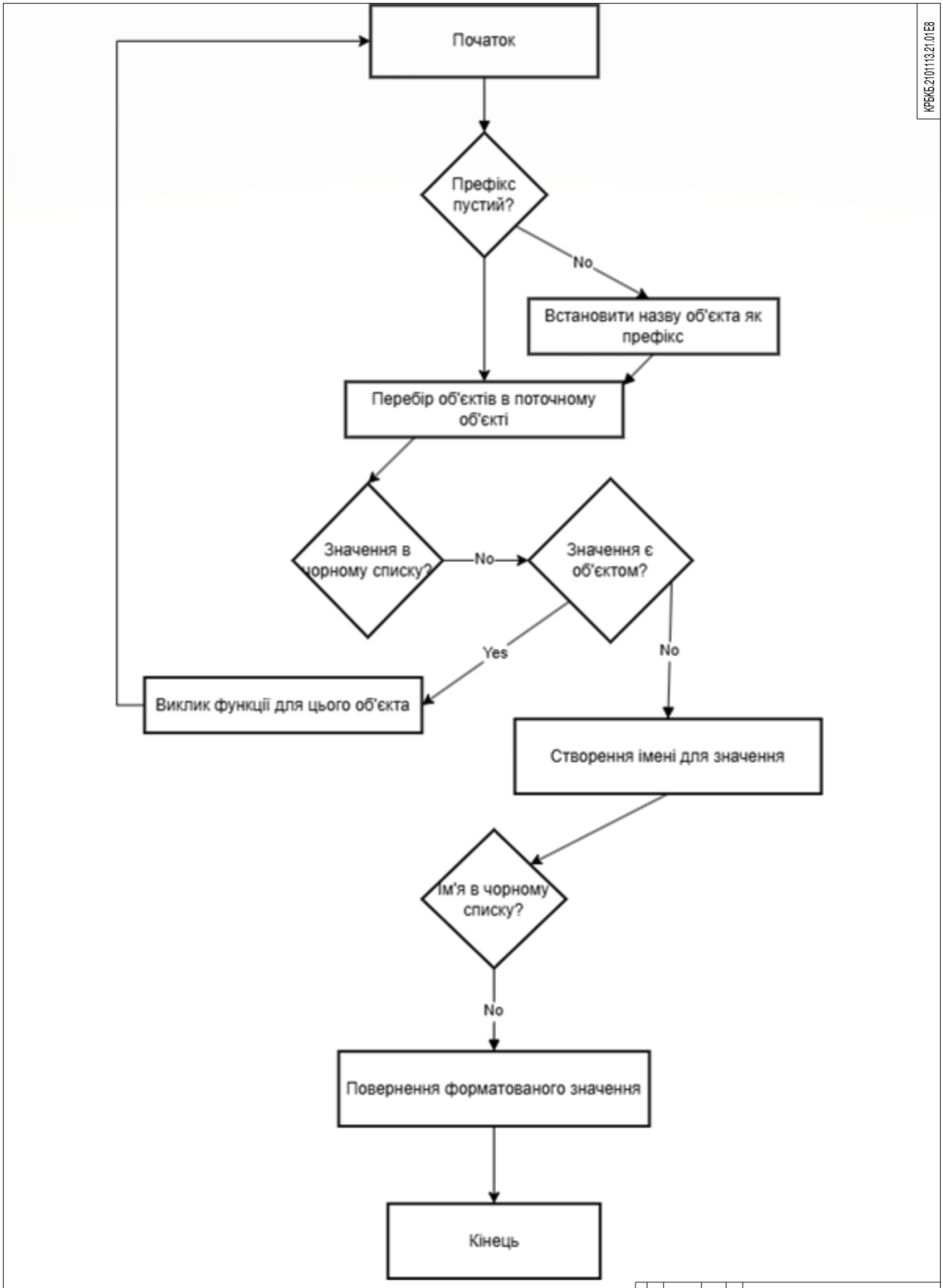
45. Ganache README 2024, С. 1. URL: https://github.com/trufflesuite/ganache-ui/blob/develop/README.md (дата звернення: 11.05.2025)

ДОДАТОК А

Копія графічної частини



					КРБКБ.2101113.21.01Е8	
Зм	Арк.	№ докум.	Підпис	Дата	Структура програмного засобу захисту від SQL-ін'єкцій	Літ. / Маса / Масштаб
Розроб.		Гоза А.О.				Н
Перевір.		Джулій В.М				
Т.контр.					Комплексне креслення	Аркуш / Аркушів 1
Н.контр.		Мостовий				ХНУ, КБ21-1
Затверд.						



КРБКБ.2101113.21.01Е8

Зм	Арк.	№ докум.	Підпис	Дата
Розроб.		Гоза А.О.		
Перевір.		Джулій В.М.		
Т.контр.				
Н.контр.		Мостовий		
Затверд.				

КРБКБ.2101113.21.01Е8

Структура програмного засобу захисту від SQL-ін'єкцій

Комплексне креслення

Літ.	Маса	Масштаб
Н		
Аркуш	Аркушів 1	

ХНУ, КБ21-1

ДОДАТОК Б

Код програмного забезпечення

Smart contract:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;
contract LogStorage {
    mapping(uint => string) private logs;
    uint private logCount;
    event LogAdded(string log, uint indexed index, address indexed sender);
    address private owner;
    constructor() {
        owner = msg.sender;
    }
    modifier onlyOwner() {
        require(msg.sender == owner, "Only the owner can add logs.");
        _;
    }
    function addLog(string memory logJson) public onlyOwner {
        logs[logCount] = logJson;
        emit LogAdded(logJson, logCount, msg.sender);
        logCount++;
    }
    function getLog(uint index) public view returns (string memory) {
        require(index < logCount, "Invalid index");
        return logs[index];
    }
    function getLatestLog() public view returns (string memory) {
        require(logCount > 0, "No logs stored");
        return logs[logCount - 1];
```

```
    }
    function getCount() public view returns (uint) {
        return logCount;
    }
    function getAllLogs() public view returns (string[] memory) {
        string[] memory allLogs = new string[](logCount);
        for (uint i = 0; i < logCount; i++) {
            allLogs[i] = logs[i];
        }
        return allLogs;
    }
}
```

BlockChainSQL.IblockChainService

```
public interface IBlockChainService
{
    Task<string> GetLatestLogAsync();
    Task<string> GetLogAsync(BigInteger index);
    Task<BigInteger> GetCountAsync();
    Task<string> AddLogAsync(string logJson);
    Task<List<string>> GetAllLogsAsync();
}
public class BlockchainOptions
{
    public string Abi { get; set; } = "";
    public string RpcUrl { get; set; } = "";
    public string PrivateKey { get; set; } = "";
    public string ContractAddress { get; set; } = "";
    public string AccountAddress { get; set; } = "";
    public bool SimplifyQueryLogging { get; set; } = false;
    public BlockchainFunctionNames Functions { get; set; } = new();
```

```
}
public class BlockchainFunctionNames
{
    public string AddLog { get; set; } = "addLog";
    public string GetLatestLog { get; set; } = "getLatestLog";
    public string GetLog { get; set; } = "getLog";
    public string GetCount { get; set; } = "getCount";
    public string GetAllLogs { get; set; } = "getAllLogs";
}
public class LogEntry
{
    public string Timestamp { get; set; }
    public string Ip { get; set; }
    public string Query { get; set; }
    public string Endpoint { get; set; }
    public List<Parameter> Parameters { get; set; }
}
public class Parameter
{
    public string DbType { get; set; }
    public string ParameterName { get; set; }
    public string Value { get; set; }
}
```

BlockChainSql.BlockChainService

```
public class BlockChainService : IBlockChainService
{
    private readonly Web3 _web3;
    private readonly Contract _contract;
    private readonly BlockchainOptions _options;
    public BlockChainService(IOptions<BlockchainOptions> options)
```

```
    {
        _options = options.Value;
        _web3 = new Web3(_options.RpcUrl);
        _contract = _web3.Eth.GetContract(_options.Abi, _options.ContractAddress);
    }
    private Web3 GetSigningWeb3() => new Web3(new
Nethereum.Web3.Accounts.Account(_options.PrivateKey), _options.RpcUrl);

    public async Task<string> GetLatestLogAsync()
    {
        var function = _contract.GetFunction(_options.Functions.GetLatestLog);
        return await function.CallAsync<string>();
    }
    public async Task<string> GetLogAsync(BigInteger index)
    {
        var function = _contract.GetFunction(_options.Functions.GetLog);
        return await function.CallAsync<string>(index);
    }
    public async Task<BigInteger> GetCountAsync()
    {
        var function = _contract.GetFunction(_options.Functions.GetCount);
        return await function.CallAsync<BigInteger>();
    }
    public async Task<string> AddLogAsync(string logJson)
    {
        var web3 = GetSigningWeb3();
        var function = web3.Eth.GetContract(_options.Abi,
_options.ContractAddress).GetFunction(_options.Functions.AddLog);
        var gas = await function.EstimateGasAsync(_options.AccountAddress, null, null,
logJson);
```

```
        var txHash = await function.SendTransactionAsync(_options.AccountAddress,
gas, null, null, logJson);
        return txHash;
    }
    public async Task<List<string>> GetAllLogsAsync()
    {
        var function = _contract.GetFunction(_options.Functions.GetAllLogs);
        var logs = await function.CallAsync<List<string>>();
        return logs;
    }
}
```

BlockChainSql.DbQueryInterceptor

```
public class DbQueryInterceptor : DbCommandInterceptor
{
    private readonly IBlockChainService _blockchain;
    private readonly QueryValidatorHelper _queryValidatorHelper;
    private readonly RequestContext _requestContext;
    private readonly bool _simplifyQueryLogging;
    public DbQueryInterceptor(
        IBlockChainService blockchain,
        QueryValidatorHelper queryValidatorHelper,
        RequestContext requestContext,
        BlockchainOptions blockchainOptions)
    {
        _blockchain = blockchain;
        _queryValidatorHelper = queryValidatorHelper;
        _requestContext = requestContext;
        _simplifyQueryLogging = blockchainOptions.SimplifyQueryLogging;
    }
    #region Async
```

```
    public override async ValueTask<InterceptionResult<DbDataReader>>
ReaderExecutingAsync(
        DbCommand command,
        CommandEventData eventData,
        InterceptionResult<DbDataReader> result,
        CancellationToken cancellationToken = default)
    {
        await TryLogIfMaliciousAsync(command);
        return await base.ReaderExecutingAsync(command, eventData, result,
cancellationToken);
    }
    public override async ValueTask<InterceptionResult<int>>
NonQueryExecutingAsync(
        DbCommand command,
        CommandEventData eventData,
        InterceptionResult<int> result,
        CancellationToken cancellationToken = default)
    {
        await TryLogIfMaliciousAsync(command);
        return await base.NonQueryExecutingAsync(command, eventData, result,
cancellationToken);
    }
    public override async ValueTask<InterceptionResult<object>>
ScalarExecutingAsync(
        DbCommand command,
        CommandEventData eventData,
        InterceptionResult<object> result,
        CancellationToken cancellationToken = default)
    {
        await TryLogIfMaliciousAsync(command);
```

```
        return await base.ScalarExecutingAsync(command, eventData, result,
cancellationToken);
    }
    #endregion
    #region Sync
    public override InterceptionResult<DbDataReader> ReaderExecuting(
        DbCommand command,
        CommandEventData eventData,
        InterceptionResult<DbDataReader> result)
    {
        TryLogIfMaliciousFireAndForget(command);
        return base.ReaderExecuting(command, eventData, result);
    }
    public override InterceptionResult<int> NonQueryExecuting(
        DbCommand command,
        CommandEventData eventData,
        InterceptionResult<int> result)
    {
        TryLogIfMaliciousFireAndForget(command);
        return base.NonQueryExecuting(command, eventData, result);
    }
    public override InterceptionResult<object> ScalarExecuting(
        DbCommand command,
        CommandEventData eventData,
        InterceptionResult<object> result)
    {
        TryLogIfMaliciousFireAndForget(command);
        return base.ScalarExecuting(command, eventData, result);
    }
    #endregion
```

```
    #region Core logic
    private bool IsSuspicious(DbCommand command)
    {
        return command.Parameters.Cast<DbParameter>().Any(p =>
_queryValidatorHelper.IsSuspiciousParam(p.Value?.ToString()));
    }
    private string BuildLogJson(DbCommand command)
    {
        var log = new LogEntry()
        {
            Timestamp = DateTime.UtcNow.ToString("o"),
            Ip = _requestContext.IpAddress ?? "unknown",
            Query = _simplifyQueryLogging ? "" : command.CommandText,
            Endpoint = _requestContext.Endpoint ?? "unknown",
            Parameters = command.Parameters.Cast<DbParameter>().Select(p=>new
Parameter()
            {
                DbType = p.DbType.ToString(),
                ParameterName = p.ParameterName,
                Value = p.Value?.ToString() ?? string.Empty
            }).ToList()
        };
        return JsonSerializer.Serialize(log);
    }
    private async Task TryLogIfMaliciousAsync(DbCommand command)
    {
        if (!IsSuspicious(command)) return;
        var json = BuildLogJson(command);
        await _blockchain.AddLogAsync(json);
    }
```

```
    private void TryLogIfMaliciousFireAndForget(DbCommand command)
    {
        if (!IsSuspicious(command)) return;

        var json = BuildLogJson(command);
        _ = Task.Run(async () =>
        {
            try
            {
                await _blockchain.AddLogAsync(json);
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Blockchain log failed: {ex.Message}");
            }
        });
    }
    #endregion
}
```

BlockChainSql.RequestContextMiddleware

```
public class RequestContextMiddleware
{
    private readonly RequestDelegate _next;

    public RequestContextMiddleware(RequestDelegate next)
    {
        _next = next;
    }
```

```
    public async Task InvokeAsync(HttpContext context, RequestContext
requestContext)
    {
        requestContext.IpAddress = context.Connection.RemoteIpAddress?.ToString();
        requestContext.Endpoint = context.Request.Path;
        await _next(context);
    }
}
public class RequestContext
{
    public string? IpAddress { get; set; }
    public string? Endpoint { get; set; }
}
```

BlockChainSQL.DependencyInjectionExtension

```
public static class DependencyInjectionExtension
{
    public static IServiceCollection AddBlockchainSql(this IServiceCollection services,
IConfiguration configuration)
    {
        services.AddScoped<RequestContext>();
       services.Configure<BlockchainOptions>(configuration.GetSection("Blockchain"));
        services.AddScoped<IBlockChainService, BlockChainService>();
        return services;
    }
    public static IServiceCollection AddQueryInterceptor(this IServiceCollection
services, IConfiguration configuration)
    {
        services.AddScoped<QueryValidatorHelper>();
        services.AddScoped<DbQueryInterceptor>();
        return services;
```

```
    }
    public static IApplicationBuilder UseRequestInfoMiddleware(this
IApplicationBuilder app)
    {
        return app.UseMiddleware<RequestContextMiddleware>();
    }
}
```

BlockChainSQL.BlockChainSqlEndpoints

```
public static class BlockchainSqlEndpoints
{
    public static IEndpointRouteBuilder RegisterBlockchainSqlEndpoints(this
IEndpointRouteBuilder app)
    {
        var group = app.MapGroup("blockchainsql/logs");
        group.MapGet("/latest", async (IBlockChainService _blockchain) =>
        {
            var dlog = await _blockchain.GetLatestLogAsync();
            var log = JsonSerializer.Deserialize<LogEntry>(dlog);
            return Results.Ok(log);
        });
        group.MapGet("", async (IBlockChainService _blockchain) =>
        {
            var dlogs = await _blockchain.GetAllLogsAsync();
            var logs = dlogs.Select(log =>
JsonSerializer.Deserialize<LogEntry>(log)).ToList();
            return Results.Ok(logs);
        });
        group.MapGet("/{index}", async (IBlockChainService _blockchain, BigInteger
index) =>
        {
```

```
            var dlog = await _blockchain.GetLogAsync(index);
            var log = JsonSerializer.Deserialize<LogEntry>(dlog);
            return Results.Ok(log);
        });

        group.MapGet("/count", async (IBlockChainService _blockchain) =>
        {
            var count = await _blockchain.GetCountAsync();
            return Results.Ok(count);
        });
        return app;
    }
}
```

JsonToSecrets.Program.cs

```
internal static class Program
{
    private static class Descriptions
    {
        public static string RootDescription = "JSON to docker/koyeb converter";
        public static string OutputPathOptionDescription = "Output path. If not specified, will write to console. WILL OVERRIDE ANY CONTENT IN PROVIDED FILE";
        public static string FilePathOptionDescription = "Path to JSON file";
        public static string OutputFormatOptionDescription = "Do not use docker_file if multi line values are present. Use docker_string instead";

        public static string ExcludeOptionDescription = "Sections/SubSections/Values to exclude. Example:" +
    "\n\tSection - will exclude whole section" +
     "\n\tSection__SubSection - will exclude only sub section" +
     "\n\tSection__SubSection__Value - will exclude only value" +
```

```
    "\nMultiple excludes are available in format -e \"first\" \"seconds\" \"third\"";
    }
    static async Task<int> Main(string[] args)
    {
        var rootCommand = new RootCommand(Descriptions.RootDescription);
        var filePathArg = new Argument<FileInfo>("filePath",
Descriptions.FilePathOptionDescription).LegalFilePathsOnly().ExistingOnly();
        var outputFormatOption = new Option<OutputFormat>(["--output-format", "-of"],
Descriptions.OutputFormatOptionDescription).FromAmong(Enum.GetNames<OutputF
ormat>());
        outputFormatOption.SetDefaultValue(OutputFormat.json);
        outputFormatOption.IsRequired = true;
        var outputPathOption = new Option<string>(["--output-path", "-op"],
Descriptions.OutputPathOptionDescription);
        var excludeOption = new Option<IEnumerable<string>>(["--except", "-e",],
Descriptions.ExcludeOptionDescription)
        {
            AllowMultipleArgumentsPerToken = true
        };
 var getVariablesCommand = VariableCommands.GetVariablesCommand(filePathArg,
outputFormatOption, outputPathOption, excludeOption);
        getVariablesCommand.AddOption(outputPathOption);
        getVariablesCommand.AddOption(outputFormatOption);
        getVariablesCommand.AddOption(excludeOption);
rootCommand.AddArgument(filePathArg);
rootCommand.AddCommand(getVariablesCommand);
rootCommand.AddCommand(SectionCommands.GetMainSectionsCommand(filePathA
rg));
        return await rootCommand.InvokeAsync(args);
    }
```

```
}
```

JsonToSecrets.Service.cs

```
public sealed record JsonVariable(string Name, string Value)
{
   public string ToJsonString() =>
@$"{this.Name.NoEscaping()}:{this.Value.NoEscaping()}";
   public string ToDockerString() => @$"-e
{this.Name.NoEscaping()}=""{this.Value.RemoveEscaping()}"" "; //Need to remove
newlines and tabs for multi line values
   public string ToDockerEnvFileLineString() =>
$@"{this.Name.NoEscaping()}=""{this.Value.RemoveEscaping()}""";
   public string ToKoyebString() => $@"{this.Name.NoEscaping()}=""{this.Value}""";
};
public static class Service
{
 public static IEnumerable<JsonVariable> Extract(JsonProperty property,
IEnumerable<string>? exclude = null)
   {
      var excludeHS = exclude != null
         ? [..exclude]
         : new HashSet<string>();
      return InternalExtractAllValues(property, excludeHS);
   }
private static IEnumerable<JsonVariable> InternalExtractAllValues(JsonProperty
property, HashSet<string> exclude, string prefix = "")
   {
      prefix = string.IsNullOrEmpty(prefix) ? "" : $"{prefix}__";
      foreach (var sub in property.Value.EnumerateObject())
      {
         var prefixedName = $"{prefix}{property.Name}";
```

```
            if (exclude.Contains(property.Name) || exclude.Contains(prefixedName))
continue;
            if (sub.Value.ValueKind == JsonValueKind.Object)
            {
                foreach (var value in InternalExtractAllValues(sub, exclude, prefixedName))
                    yield return value;
            }
            else
            {
                var name = $"{prefix}{property.Name}__{sub.Name}";
                if (exclude.Contains(name)) continue;
                yield return new JsonVariable(name, sub.Value.ToString());
            }
        }
    }
}
```

JsonToSecrets.FileInfoExtensions

```
public static class FileInfoExtensions
{
    public static string ReadAllText(this FileInfo file)
    {
        if (file is null || !file.Exists) throw new FileNotFoundException("File not found");
        using var reader = file.OpenText();
        return reader.ReadToEnd();
    }
    public static (bool IsValid, string ErrorMessage) IsValidJson(this FileInfo file)
    {
        if (file is null || !file.Exists)
            return (false, ErrorMessages.FileErrors.DoesNotExists);
        if (!file.Extension.Equals(".json", StringComparison.OrdinalIgnoreCase))
```

```
            return (false, ErrorMessages.FileErrors.InvalidExtension);
        return (true, string.Empty);
    }
}
public static class StringExtensions
{
    public static string NoEscaping(this string str) => str.Replace("\n",
@"\n").Replace("\t", @"\t");
    public static string RemoveEscaping(this string str) => str.Replace("\n",
@"").Replace("\t", @"");
}

public enum OutputFormat
{
    json,
    docker_string,
    docker_file,
    koyeb
}
public static class ErrorMessages
{
    public static class FileErrors
    {
        public const string DoesNotExists = $"File does not exist";
        public const string InvalidExtension = "The file has an invalid extension";
    }
}
```

JsonToSecrets.SectionCommand

```
public static class SectionCommands
{
   public static Command GetMainSectionsCommand(Argument<FileInfo> filePath)
   {
      var getMainSectionsCommand = new Command("sections", "Get main sections of
JSON file");
      getMainSectionsCommand.SetHandler((jsonFile) =>
      {
         var validationResult = jsonFile.IsValidJson();
         if (!validationResult.IsValid)
         {
            Console.WriteLine(validationResult.ErrorMessage);
            getMainSectionsCommand.Invoke("--help");
            return;
         }
         using var doc = JsonDocument.Parse(jsonFile.ReadAllText());
         foreach (var section in doc.RootElement.EnumerateObject())
         {
            Console.WriteLine(section.Name);
         }
      }, filePath);
      return getMainSectionsCommand;
   }
}
```

JsonToSecrets.VariableCommand

```
public static class VariableCommands
{
```

```
    public static Command GetVariablesCommand(Argument<FileInfo> filePath,
Option<OutputFormat> outputFormat, Option<string> outputPath,
Option<IEnumerable<string>> exclude)
    {
        var enumerateSectionsCommand = new Command("variables", "Get variables
from JSON file");
        enumerateSectionsCommand.SetHandler((jsonFile, format, path, except) =>
        {
            var validationResult = jsonFile.IsValidJson();
            if (!validationResult.IsValid)
            {
                Console.WriteLine(validationResult.ErrorMessage);
                enumerateSectionsCommand.Invoke("--help");
                return;
            }
            Action<string> consoleWriter = format switch
            {
                OutputFormat.docker_string => Console.Write,
                _ => Console.WriteLine
            };
            Action<string>? fileWriter = null;
            StreamWriter? streamWriter = null;
            if (!string.IsNullOrEmpty(path))
            {
                streamWriter = new StreamWriter(path, append: false);
                fileWriter = format switch
                {
                    OutputFormat.docker_string => streamWriter.Write,
                    _ => streamWriter.WriteLine
                };
```

```
        }
        using (streamWriter)
        {
            using var doc = JsonDocument.Parse(jsonFile.ReadAllText());
            foreach (var output in doc.RootElement.EnumerateObject()
                        .SelectMany(prop =>
                            EnumerateSingleProp(prop, format, except)))
            {
                (fileWriter ?? consoleWriter).Invoke(output);
            }
            if (fileWriter != null)
            {
                Console.Write($"Output can be found at {Path.GetFullPath(path)}");
            }
        }
    }, filePath, outputFormat, outputPath, exclude);
    return enumerateSectionsCommand;
}
```

Завідувачу кафедри кібербезпеки
к.т.н., доц. Кльоцу Ю.П.

Галки Артура Олексійовича
ПІБ здобувача вищої освіти

Студента ФІТ, 4 курсу, групи КБ-21-1

ЗАЯВА

З правилами чинного Положення «Про систему забезпечення академічної доброчесності у Хмельницькому національному університеті» від 31.08.2023, згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (StrikePlagiarism та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

25.05.25
дата

підпис

Протокол аналізу звіту подібності науковим керівником

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Галка Артур Олексійович

Співавтор:

Назва: Система захисту конфіденційної інформації в СКБД MS SQL від sql-атак

Науковий керівник:

Підрозділ: Кафедра кібербезпеки

Коефіцієнт подібності 1:0.9%

Коефіцієнт подібності 2:0.2%

Мікропробіли: 0

Заміна букв: 0

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2025-05-30 11:08:22.0

Після аналізу Звіту подібності констатую наступне:

☑ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

02.06.2025р.

Anti-Plagiarism (UA) v-15.281 Educational

The maximum coincidence with one document 0.0%

Dictionaries check: en_US, ru_RU, ua_UA. **Errors in the documents: 13%**

	Document		Sum coincidence on the DB	
ID: 242604 Title: Система захисту конфіденційної інформації в СКБД MS SQL від sql-атак Added in a DB: 2025-05-30 Authors: Галка Артур Олексійович Heads: Джулій В.М. Consultants: Opponents:	Symbols	Lexemes	Symbols	Lexemes
	90241	762	236 (0%)	5 (1%)

Plagiarism sources

ID	Description	Plagiarism presence in the document	
		Symbols	Lexemes

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ

КАФЕДРИ КІБЕРБЕЗПЕКИ

ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: Система захисту конфіденційної інформації в СКБД MS SQL від sql-атак.

Автор: Галка Артур Олексійович

Спеціальність: 125 – Кібербезпека

Освітня програма: Кібербезпека

Науковий керівник: Володимир Джулій, канд. техн. наук, доцент

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних). Робота приймається до захисту.	відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи (далі – зазначаються детальні та аргументовані підстави віднесення запозичень до правомірних). Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи.	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Оригінальність тексту роботи за результатами перевірки системою StrikePlagiarism складає 99,1%, оригінальність тексту роботи за результатами перевірки системою Anti-Plagiarism складає 100%.

Згідно з правилами чинного Положення «Про систему забезпечення академічної доброчесності у Хмельницькому національному університеті» від 24.09.2024, авторська робота, обсяг оригінального тексту у відсотках до загального обсягу матеріалу в якій складає 90-100%, визначається роботою з високою унікальністю тексту і допускається до захисту.

Керівник роботи _____ Володимир ДЖУЛІЙ

Гарант ОП _____ Віктор ЧЕШУН

Завідувач кафедри кібербезпеки _____ Юрій КЛЬОЦ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

освітнього ступеня «бакалавр»

Студент Галка Артур Олексійович

Тема Система захисту конфіденційної інформації в СКБД MS SQL від sql-атак

Спеціальність 125 – Кібербезпека

Обсяг кваліфікаційної роботи освітньо-кваліфікаційного рівня «бакалавр»:

кількість листів креслень 4 ; кількість сторінок записки 61 .

1. Короткий зміст роботи та прийнятих рішень У кваліфікаційній роботі була розроблена система захисту конфіденційної інформації від SQL-ін'єкцій на основі технології Blockchain. У процесі проєктування були розроблені такі компоненти: смарт-контракт для роботи з мережою Blockchain, nuget пакет BlockchainSQL, допоміжна утиліта JsonToSecrets. Крім того, надані рекомендації щодо інтеграції системи в існуючих проектах, проведено тестування системи шляхом спроби здійснення SQL-ін'єкцій.

2. Висновок про відповідність кваліфікаційної роботи завданню.У кваліфікаційній роботі було виконано поставлене завдання як у теоретичній, так і в практичній частині. Було дотримано усіх вимог щодо виконання

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У кваліфікаційній роботі була виконана низка завдань, таких як наведення загальної характеристики задачі, визначення проблематики, розглянуто предмет та методи дослідження. Виконанні задачі послугили створенню рішення системи захисту від SQL-ін'єкцій. У першому розділі проведено аналіз предметної області, розглянуто наявні методи та технології захисту в СКБД MS SQL. У другому розділі, на основі результатів аналізу та визначеного завдання, було спроєктовано систему захисту від SQL-ін'єкцій шляхом перехоплення та аналізу запитів, та подальшого їх запису у мережу Blockchain. У третьому розділі описані практичні аспекти розробки, реалізації смарт-контрактів, алгоритму перехоплення та аналізу запитів, та допоміжної утиліти для експорту конфігураційних файлів проекту. Також наведено опис глобального розгортання в мережі та локального тестування системи з виконанням дійсної криптографічної транзакції у мережі Ethereum. Система розроблена з використанням актуальних технологій та засобів.

4. Позитивні сторони роботи Розроблена система захисту від SQL-ін'єкцій є актуальною та цікавою темою, що допомагає відслідкувати та захиститись від атак. Розроблена система легко інтегрується в додатки, та має зрозумілу документацію. Тематика роботи має потенціал для подальшого розвитку та вдосконалення системи.

5. Негативні сторони роботи Система потребує підключення до мережі блокчейн, що є додатковим кроком при розгортані з боку клієнта. Не передбачено захист від спам-атак, які можуть спричинити створення значної кількості транзакцій. Наразі система може використовуватись лише з Entity Framework.

6. Оцінка графічного оформлення та пояснювальної записки роботи. Графічне оформлення кваліфікаційної роботи відповідає темі роботи та виконане з дотриманням стандартів. У цілому графічне оформлення є чітким та якісним, а пояснювальна записка відповідає нормам оформлення.

7. Відгук про роботу в цілому Кваліфікаційна робота заслуговує позитивної оцінки, оскільки весь матеріал роботи є структурованим, чітким та послідовним. Усі розділи роботи мають логічну послідовність, що сприяє зрозумінню викладеного матеріалу в рамках теми роботи. Графічний матеріал допомагає наочно продемонструвати доцільність та ефективність прийнятих рішень для досягнення мети.

8. Інші зауваження У переліку використаних джерел наявні посилання на популярні ресурси, такі як офіційна сторінка Microsoft. Такі джерела не рекомендовано використовувати при написанні кваліфікаційних робіт

9. Оцінка кваліфікаційної роботи Ураховуючи всі позитивні та негативні сторони представленої кваліфікаційної роботи, можна зробити висновок, що вона заслуговує оцінки «відмінно».

РЕЦЕНЗЕНТ (прізвище, ім'я, по батькові, посада, місце роботи)
Бойко Юлій Миколайович,
доктор технічних наук, професор кафедри ТМІТ

« 25 » травня 2025. _____ (підпис)