Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ДОПУСТИТИ ДО ЗАХИСТУ:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
_____ Іван ПАРХОМЕНКО
«__» _____ 2025 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

галузь знань *12 Інформаційні технології*
(шифр і назва галузі знань)

спеціальність *125 Кібербезпека та захист інформації*
(код і назва спеціальності)

освітній ступень *магістр*

освітньо-наукова програма *Кібербезпека*
(назва освітньої програми)

на тему: «Методи та засоби забезпечення безпеки серверних застосунків»

Виконавець: студент ІІ курсу, групи КБм-22

_____ **Олександр МАЛИХІН**
(підпис) (Ім'я, ПРІЗВИЩЕ)

	Ім'я, ПРІЗВИЩЕ	**Підпис**
Науковий керівник	Сергій ТОЛЮПА	
Нормоконтроль	Юрій БАБЕНКО	

Київ 2025

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ЗАТВЕРДЖЕНО:
В.о. завідувача кафедри
кібербезпеки
та захисту інформації
_____ Іван ПАРХОМЕНКО
«25» жовтня 2024 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *125 Кібербезпека та захист інформації*
(код і назва спеціальності)

освітній ступень *магістр*

Здобувача(ки) КБм-22 Малихіна Олександра Олександровича
(група) (прізвище ім'я по-батькові)

Тема кваліфікаційної роботи Методи та засоби забезпечення безпеки серверних застосунків

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Рішення засідання кафедри кібербезпеки та захисту інформації факультету інформаційних технологій протокол № 4 від 24.10.2024 р.

2. МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень	Процес захисту інформації в серверних застосунках.
Предмет досліджень	Методи та засоби захисту серверних застосунків.
Мета	Забезпечення цілісності і конфіденційності інформації за рахунок розробки системи безпеки серверного застосунку.
Вихідні дані для проведення роботи	Концепції захисту серверних застосунків, види їх архітектур та процес розробки серверних застосунків.

3. ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна Удосконалення методу захисту інформації за рахунок поєднання методів шифрування даних та технології блокчейн.

Практична цінність Впровадження удосконаленої системи забезпечення цілісності і конфіденційності інформації значно підвищить рівень захисту серверних застосунків.

4. ЕТАПИ ВИКОНАННЯ РОБОТИ

Найменування етапів робіт	Строки виконання робіт (початок-кінець)
Уточнення постановки задачі	25.10.2024 – 29.12.2024
Аналіз літературних джерел	30.12.2024 – 12.02.2025
Обґрунтування вибору рішення	13.02.2025 – 23.02.2025
Розгляд концепції серверного застосунку	24.02.2025 – 03.03.2025
Дослідження проблем забезпечення інформаційної безпеки в серверних застосунках	04.03.2025 – 10.03.2025
Аналіз досліджень щодо вразливостей та загроз у серверних застосунках	11.03.2025 – 20.03.2025
Дослідження існуючих методів та засобів захисту серверних застосунків	21.03.2025 – 14.04.2025
Розробка удосконаленої системи захисту серверних застосунків, яка буде забезпечувати цілісність і конфіденційність інформації	15.04.2025 – 05.05.2025
Оформлення пояснювальної записки згідно методичних рекомендацій	06.05.2025 – 15.05.2025
Подача пакету документів на розгляд ЕК	15.05.2025 – 19.05.2025

Завдання видав _____ Сергій ТОЛЮПА
(підпис) (Ім'я, ПРІЗВИЩЕ)

Завдання прийняв до виконання _____ Олександр Малихін
(підпис) (Ім'я, ПРІЗВИЩЕ)

Дата видачі завдання: 25.10.2024 р.
Термін подання кваліфікаційної роботи до ЕК 19.05.2025 р.

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Методи та засоби забезпечення безпеки серверних застосунків»: 99 сторінок, 9 рисунків та 62 літературних джерела.

Об’єкт дослідження – процес захисту інформації в серверних застосунках.

Мета роботи – забезпечення цілісності і конфіденційності інформації за рахунок розробки системи безпеки серверного застосунку.

Методи дослідження – методи симетричного та асиметричного шифрування даних, принципи децентралізованих ідентифікаторів, використання смарт-контрактів в блокчейні.

У роботі розглянуто архітектуру сучасних серверних застосунків та основні вектори атак, які загрожують їхній безпеці. Проведено аналіз існуючих механізмів захисту, включно з контролем доступу, запобіганням ін’єкційним атакам і XSS. Розроблено удосконалену систему безпеки, яка поєднує клієнтське шифрування зберіганих даних з можливістю їх розміщення у смарт-контрактах, що гарантує їхню цілісність та захист від несанкціонованого доступу.

Наукова новизна: удосконалено метод захисту інформації в серверних застосунках шляхом інтеграції шифрування даних із зберіганням у блокчейні, що дозволяє зберігати зашифровані дані та перевіряти їх достовірність.

Актуальність теми: У сучасних умовах стрімкого розвитку цифрових технологій та зростання кількості кіберзагроз питання захисту серверних застосунків набуває особливої актуальності. Серверні застосунки є ключовим елементом IT-інфраструктур, саме на них зосереджено обробку, зберігання та передавання критично важливої інформації. Будь-яке порушення їхньої безпеки може призвести до витоку персональних даних та порушення конфіденційності користувачів.

Ключові слова: серверний застосунок, дворівнева архітектура, трирівнева архітектура, контроль доступу, DID, смарт-контракт, шифрування.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

API – Application Programming Interface
URL – Uniform Resource Locator
XSS – Cross-Site Scripting
AES – Advanced Encryption Standard
SQL – Structured Query Language
CBC – Cipher Block Chaining
IP – Internet Protocol
ACL – Access Control List
RBAC – Role-Based access-control
ABAC – Attribute-Based access-control
EVM – Ethereum Virtual Machine
HTTP – Hypertext Transfer Protocol
DAC – Discretionary Access Control
DID – Decentralized Identifiers
JSON – JavaScript Object Notation
JWT – JSON Web Tokens
DOS – Denial-of-service
ОС - Операційна система
СУБД - Система управління базами даних
ПЗ - Програмне забезпечення

ЗМІСТ

ВСТУП

У сучасному цифровому середовищі постійно зростає важливість захисту серверних застосунків. Зі швидким розвитком технологій та збільшенням кількості кіберзагроз, збереження конфіденційності, цілісності та доступності інформації стає критично важливим для будь-якої організації, що працює з критичними даними. Серверні додатки все частіше стають об'єктом атак, оскільки містять великі обсяги особистих, фінансових та конфіденційних даних, які можуть бути цінними для кібезлочинців.

Вразливості в архітектурі або логіці обробки запитів, помилки в реалізації контролю доступу, а також ін’єкційні атаки або атаки типу міжсайтовий скриптинг загрожують не тільки окремим додаткам, але й репутації компаній в цілому. У таких умовах впровадження ефективних і сучасних методів захисту стає нагальною потребою. Традиційні підходи вже не завжди здатні протистояти складним і комбінованим атакам, тому зростає інтерес до нових концепцій, серед яких особливе місце займає поєднання криптографічних методів з технологією блокчейн.

Саме тому дана кваліфікаційна робота присвячена розробці вдосконаленої системи захисту серверних застосунків, що забезпечує високий рівень цілісності та конфіденційності даних. Запропоноване рішення базується на поєднанні шифрування інформації та її зберігання у вигляді зашифрованих метаданих у смарт-контрактах з використанням децентралізованих ідентифікаторів, що дозволяє здійснювати гнучкий, верифікований та децентралізований контроль доступу.

Метою даної роботи є забезпечення цілісності та конфіденційності інформації шляхом розробки системи безпеки для серверного застосунку.

Для досягнення цієї мети необхідно виконати наступні завдання:

- проаналізувати архітектурні особливості сучасних серверних застосунків та їх вразливі місця;

- дослідити проблематику захисту інформації в сучасних серверних застосунках;
- дослідити методи шифрування, особливості смарт-контрактів;
- дослідити існуючі методи захисту інформації при побудові серверного застосунку;
- реалізувати систему, що поєднує методи шифрування і блокчейн технології в єдину модель безпеки;

Об'єктом дослідження є процес захисту інформації в серверних додатках.

Предмет дослідження – методи та засоби захисту серверних застосунків, зокрема комбіноване використання технологій шифрування та блокчейну для забезпечення цілісності та конфіденційності даних.

Методи дослідження, використані в кваліфікаційній роботі:

- аналіз літератури та нормативних документів з інформаційної безпеки;
- аналіз існуючих архітектур серверних застосунків та їх вразливих місць;
- порівняння традиційних та інноваційних підходів до інформаційної безпеки;

РОЗДІЛ 1
ЗАГАЛЬНІ ВІДОМОСТІ ПРО СУЧАСНІ СЕРВЕРНІ ЗАСТОСУНКИ

1.1. Загальне поняття серверного застосунку

У сучасному цифровому середовищі серверні додатки відіграють життєво важливу роль, забезпечуючи операційну логіку, зберігання даних, безпечний зв'язок і масштабованість більшості інформаційних систем. Під серверними додатками зазвичай розуміють програмне забезпечення, яке працює на стороні сервера комп'ютерної системи, обробляє запити клієнтів і надає їм відповідні послуги. Зазвичай такі програми не взаємодіють безпосередньо з кінцевими користувачами, а виступають посередником між кінцевими користувачами та ресурсами інформаційної системи [1].

Серверні застосунки реалізують модель клієнт-сервер, де сервер надає ресурси або послуги, а клієнт отримує доступ до сервера через відповідні запити. Серверні застосунки зазвичай є багатопоточними, можуть обробляти велику кількість одночасних з'єднань і підтримують такі комунікаційні протоколи, як HTTP або HTTPS, TCP/IP тощо.

Основними функціями серверного застосунку є:

- Обробка бізнес-логіки: сервер виконує основні обчислення, перевірки, аналіз даних і реалізує системні правила ізольовано від клієнта.
- Доступ до бази даних: виконання запитів, збереження, оновлення та отримання інформації з бази даних. У більшості випадків сервер відповідає за цілісність, узгодженість і захист даних.
- Безпечна обробка запитів: аутентифікація користувачів, надання або заборона доступу до певних ресурсів, шифрування даних, що передаються.
- Масштабованість і резервування: адаптація до змін навантаження, резервне копіювання, кластеризація та реплікація для підвищення доступності.

- Моніторинг та аудит: реєстрація подій, помилок, спроб доступу та інших критичних подій для забезпечення тестування та обслуговування системи.

Типи серверних застосунків відрізняються за архітектурою, призначенням та взаємодією з іншими компонентами. Основними категоріями яких є:

- Веб-сервери – призначені для обробки HTTP-запитів від веб-клієнтів (браузерів), надання доступу до веб-сторінок, REST API та ресурсів, до прикладу, Apache HTTP Server, Nginx, LiteSpeed.
- Сервери додатків – реалізують складні обчислення, бізнес-логіку, роботу з чергами повідомлень, мікросервіси. До них відносяться Node.js, Spring Boot, ASP.NET Core.
- Сервери баз даних – зберігають і обробляють структуровані SQL запити або неструктуровані NoSQL дані, до прикладу: MySQL, PostgreSQL, MongoDB, Cassandra.
- Файлові сервери - забезпечують централізоване зберігання та доступ до файлів, часто використовуються у внутрішніх корпоративних мережах. Приклади: Samba, NFS.
- Поштові сервери – забезпечують відправлення, доставку та зберігання електронної пошти, до прикладу, Microsoft Exchange Server або Postfix.
- Проксі-сервери та балансувальники навантаження – проміжні сервери, які виконують кешування, фільтрацію та маршрутизацію трафіку, до прикладу, HAProxy, Squid, Varnish.
- API-сервери – надають набір функцій через відкриті інтерфейси для інших систем, до прикладу, інтеграція з платіжними системами або зовнішніми сервісами.

Кожен з перерахованих вище типів серверів має власну архітектуру, протоколи та рівні безпеки, які необхідно враховувати при створенні безпечних серверних додатків [2].

Для реалізації сучасних серверних додатків використовуються такі основні технологічні мови для написання серверних додатків:

Мови програмування, які найбільш популярні серед розробників для написання серверних застосунків:

- JavaScript – популярна для розробки високопродуктивних серверів, керованих подіями.
- Python – широко використовується для веб-серверів, аналітики та автоматизації.
- Java – стабільна платформа для великих корпоративних застосунків.
- C# – широко використовується в бізнес-середовищі, інтегрована в інфраструктуру Windows.
- Go, Rust – сучасні мови, орієнтовані на продуктивність, безпеку та конкурентоспроможність.

Фреймворки та платформи, які розробники частіше всього використовують для розробки серверних застосунків:

- Express.js – легкий фреймворк для Node.js.
- Django та Flask – фреймворки для Python.
- Spring Boot – фреймворк для архітектури Java з високою конфігурацією.
- ASP.NET Core – потужна платформа для серверних .NET додатків.

Також розробники часто використовують такі хмарні платформи для полегшення взаємодії адміністратора з серверним застосунком [3]:

Amazon Web Services (далі буду використосувати позначення AWS) - це найбільша у світі платформа хмарних обчислень, яка пропонує понад 200 сервісів для обчислень, зберігання, баз даних, аналітики, машинного навчання, безпеки тощо.

Google Cloud Platform – це хмарна платформа від Google, орієнтована на високопродуктивні обчислення, аналітику та обробку даних. GCP активно використовується стартапами, фінтех-компаніями, розробниками.

Microsoft Azure – хмарна платформа від Microsoft, популярна серед корпоративного сегменту завдяки глибокій інтеграції з Windows Server, Active Directory та Microsoft 365.

Також для контейнеризації серверних застосунків використовують такі платформи та системи, наприклад Docker, який є платформою для контейнеризації, яка дозволяє створювати, розгортати та запускати додатки в ізольованих середовищах, які називаються контейнерами. Кожен контейнер має власне середовище виконання, тобто свої бібліотеки, залежності, конфігурації, що у свою чергу забезпечує стабільність і передбачуваність роботи додатка незалежно від ОС або середовища. Або Kubernetes, схожа система оркестрації контейнерів, яка керує великою кількістю контейнеризованих застосунків у кластерному середовищі. Автоматичне масштабування Kubernetes збільшує або зменшує кількість екземплярів додатка залежно від навантаження. Також якщо контейнер виходить з ладу, Kubernetes автоматично перезапускає його або створює новий.

Основні вимоги, які застосовуються під час написання серверних застосунків, сучасні серверні застосунки повинні відповідати таким основним принципам та вимогам [4]:

- Масштабованість буває двох типів, горизонтальна, тобто додавання серверів або вертикальна це додавання ресурсів, масштабованість системи при збільшенні навантаження.
- Висока доступність, оскільки застосунок повинен бути доступним 24/7 з мінімальним часом простою, щоб забезпечити безперебійну роботу навіть у разі часткових відключень.
- А також безпека, частіше за все розробники виконують впровадження багаторівневих заходів захисту, яка може включати в себе:

1. Аутентифікацію, яка визначає, хто має право отримати доступ до API, гарантує, що лише авторизовані користувачі або системи можуть звертатися до ресурсів сервера.
2. Шифрування даних TLS або SSL, є важливим аспектом забезпечення конфіденційності та цілісності переданих даних;
3. Захист від атак, оскільки серверні застосунки все більше і більше стають основною мішенню для численних видів атак, тому не треба нехтувати гарними видами захисту від різноманітних атак;

4. Аудит дій користувачів дозволяє відслідковувати, хто та як використовує API, що є критично важливим для забезпечення безпеки та відповідності політикам організації. Аудит допомагає виявляти аномальні або підозрілі дії, а також забезпечує можливість відновлення або перевірки дій у разі інцидентів безпеки.

- Продуктивність, у сучасних серверних застосунках, потрібна високошвидкісна обробка запитів за допомогою оптимізованих алгоритмів, кешування та асинхронного програмування.
- Підтримка моніторингу, тобто забезпечення наглядниї метриків, журналів та повідомлення про помилки збираються за допомогою таких інструментів, як Prometheus, Grafana.
- Модульність та підтримка архітектури мікросервісів, тобто розподіл великих додатків на окремі сервіси дозволяє гнучко керувати ресурсами та масштабувати функціональність.
- Розробники стикаються з багатьма викликами при розробці серверної частини системи, найтиповіші виклики при створенні серверних застосунків включають в себе:
- Роботу з системами з високим навантаження, тобто розробнику треба досягнути, щоб система була дуже гарно оптимізована для тисяч одночасних підключень.
- Забезпечення безпеки, застосування методів які повинні включати в себе регулярні оновлення компонентів серверних застосунків, а також виявлення вразливостей та аудит безпеки.
- Сумісність з різними клієнтськими платформами, тобто щоб серверних застосунок міг “правильно” працювати у веб середовищі, у мобільному застосунку та застосунку на ПК.
- Тестування та налагодження, тобто розробнику треба реалізувати гарну перевірку продуктивності, функціональності та безпеки за допомогою модульних, інтеграційних та навантажувальних тестів.

Серверні застосунки є серцем будь-якої сучасної інформаційної системи. Їх правильне проектування, впровадження та підтримка визначають не тільки продуктивність і зручність використання системи для кінцевого користувача, але й рівень її захисту від кіберзагроз. У контексті глобальної цифровізації наявність безпечної, масштабованої та надійної серверної архітектури є обов'язковою для будь-якої організації, яка працює з даними або взаємодіє з користувачами.

1.2. Структура побудови серверних застосунків

1.2.1. Клієнт-серверна модель

Архітектура кліент-сервер є однію з найпоширеніших моделей для організацій взаємодій у комп'ютерних мережах та розподілених інформаційних системах. Вона визначає можливості між кліентами, які ініціюють запити та серверами які їх обробляють. Цей погляд дозволяє розробляти гнучкі, маштабовані користувацькі застосунки що ефективно обслуговують багато користувачів одночасно [5].

У такому випадку кліент, ПЗ або часом пристрій, що дійсно виступає як посередник між користувачем та сервером. Основне завдання полягає у створенні запитів, наприклад на отримання даних або збереження нової інформації і передачі його на сервер для обробки і виконання. Найпоширенішими прикладами кліентських програм можуть бути:

- Веб-браузери, найпопулярніші це Google Chrome, Firefox, які надсилають запити на веб-сервер;
- Мобільні додатки, які підключаються до віддаленого сервера для отримання інформації по запиту;
- Настільні додатки, які отримують доступ до бази даних або сервера авторизації.

- Клієнт зазвичай обробляє лише частину обчислювальної потужності, тобто забезпечує інтерфейс користувача, надсилає запити, передає дані на сервер і отримує результати обробки від самого сервера.
- Сервер, у цьому контексті виступає потужною комп'ютерною системою або спеціалізованим програмним забезпеченням, яке обробляє вхідні запити клієнтів, зберігає дані, виконує бізнес-логіку та генерує відповіді. Сервер може бути фізичною машиною, віртуальним екземпляром або контейнером у хмарному середовищі, наприклад використовуючи AWS.

Сервер зазвичай розробляється, щоб він мав можливість підтримувати кілька одночасних підключень і реалізувати зі своєї сторони:

- Обробку запитів користувачів, тобто сервер постійно очікує на вхідні запити від клієнтів, у додатках це будуть HTTP-запити. Отримавши запит, він розпізнає його тип, тобто визначає що це, наприклад отримання, додавання, оновлення або видалення даних, після цього виконує необхідну логіку та формує відповідь. У більшості випадків використовується модель запит-відповідь, де клієнт ініціює взаємодію, а сервер відповідає на кожен запит окремо;
- Також підтримує доступ до бази даних оскільки обробка запитів часто вимагає взаємодії з системою управління базами даних (далі буду використовувати позначення СУБД). Сервер виконує SQL-запити, отримує або змінює дані відповідно до запиту користувача. Зазвичай між додатком і базою даних існує рівень абстракції, який забезпечує безпечний доступ;
- Забезпечує аутентифікацію та авторизацію, оскільки для захисту інформації серверу треба реалізовувати механізми аутентифікації користувачів за допомогою логіна та пароля, або ж токенів, біометричних даних або визначення рівня доступу користувача до ресурсів і операцій;
- Логіку додатків, оскільки уся бізнес-логіка додатка реалізована на сервері, тобто уся обробка транзакцій, перевірка правил, обчислення, обробка подій, ведення журналів. Це дозволяє централізовано контролювати поведінку системи, забезпечуючи цілісність даних і єдину обробку.

- Шифрування та захист інформації у сховищі, наприклад, шифрування паролів або передача трафіку за допомогою зашифрованих протоколів HTTPS.

Комунікація між клієнтом і сервером зазвичай базується на принципі запит-відповідь, як я вже казав раніше, клієнт відправляє HTTP запит, а сервер у свою чергу обробляє цей запит, виконує відповідні дії,наприклад, зчитує дані з бази даних, і повертає клієнту HTML-документ у вигляді відповіді.

Цей обмін відбувається асинхронно, тобто клієнти можуть надсилати запити незалежно один від одного, а сервер може обробляти кілька запитів одночасно за допомогою черг запитів або потокової обробки [6].

У порівнянні з сервером, на клієнті будуть реалізовуватися такі функції:

- Візуалізація та взаємодія з користувачем, тобто клієнтська сторона відповідає за відображення користувальницького інтерфейсу та забезпечення зручної взаємодії з користувачем. Вона формує елементи інтерфейсу, такі як форми, кнопки, повідомлення, списки, інформаційні панелі тощо. На цьому рівні також реалізовані компоненти UX, тобто перевірка введених даних, спливаючі підказки, анімації;

- Створення запитів і передача даних виконується коли користувач виконує дію, наприклад, натискає кнопку, заповнює форму, клієнтська сторона формує запит, зазвичай HTTP, і відправляє його на сервер. Запит може містити параметри, токени аутентифікації, дані форми тощо;

- Обробка відповідей сервера, отже після отримання відповіді від сервера клієнт обробляє її, наприклад, відображає дані, повідомляє про помилки, оновлює інтерфейс або виконує локальні обчислення. Клієнт також може кешувати отриману інформацію для підвищення продуктивності.

Також не слід забувати про те, які додаткові функції можуть розгортатися на сервері, якщо цього захоче розробник:

- Обробка помилок і ведення журналу, якщо під час виконання запитів виникають помилки, сервер генерує відповідні коди помилок і повідомлення, які повертаються клієнту. Крім того, він веде журнали подій, в яких фіксуються

ключові дії, помилки та запити, оскільки це важливо для аудиту та усунення несправностей;

- Масштабування, розробники для забезпечення високої доступності та продуктивності масштабують серверну частину, як вертикально,збільшуючи ресурси одного сервера, так і горизонтально, тобтто додаючи нові сервери. Для рівномірного розподілу вхідних запитів між декількома вузлами використовуються балансувальники навантаження.

Найрозповсюдженіші типи клієнт-серверних моделей:

Слабкий клієнт та сильний сервер: клієнт виконує тільки основні функції інтерфейсу, тоді як вся логіка, обробка та управління даними зосереджені на сервері. Ця модель є типовою для веб-браузерів [7].

Сильний клієнт та сильний сервер: клієнт бере на себе частину обчислень, наприклад, кешування, попередня обробка, тим самим зменшуючи навантаження на сервер. Ця модель популярна в настільних системах і складних клієнтських додатках,наприклад, бухгалтерському програмному забезпеченні.

Для прикладу можна визначити такі приклади додатків з архітектурою клієнт-сервер:

- Електронна пошта, тобто, поштові клієнти,такі як Outlook або Gmail, використовують протоколи SMTP для надсилання електронних листів. Усі дії координуються через поштові сервери.
- Онлайн-ігри, тобто там клієнтська частина гри надсилає дії гравця на ігровий сервер, який синхронізує ці зміни з усіма учасниками гри в режимі реального часу, забезпечуючи єдину ігрову сесію.
- Обмін файлами, тобто, користувач може завантажувати файли на сервер для зберігання або завантажувати їх із сервера на свій пристрій.

Переваги, які я можу виділити для моделі клієнт-сервер:

- Централізоване управління, тобто всі ключові ресурси, логіка та дані зберігаються та обробляються на сервері, що дозволяє адмініструвати систему з одного центру. Це спрощує управління доступом, моніторинг активності та контроль безпеки.

- Простота оновлення, оскільки логічна обробка та зберігання даних відбуваються на стороні сервера, оновлювати потрібно лише сервер, і всі клієнти автоматично отримають доступ до нових функцій або виправлень. Це зменшує витрати на підтримку великої кількості клієнтських пристроїв;
- Розподіл обов'язків, тобто завдяки чіткому розподілу функцій між клієнтом і сервером спрощується розробка системи: клієнт відповідає за інтерфейс, а сервер - за логіку та обробку даних. Це дозволяє різним командам працювати незалежно, що прискорює розробку та полегшує тестування і обслуговування.

Недоліки, які я можу виділити:

- Сервери є критичними вузлами, оскільки сервер є центральним елементом системи, тому в разі його виходу з ладу, збій, атака, перевантаження, весь сервіс може стати недоступним для клієнтів. Це вимагає впровадження механізмів відмовостійкості, резервного копіювання та автоматичного відновлення.
- Клієнти залежать від підключення до мережі, оскільки для взаємодії з сервером клієнт повинен мати стабільне мережеве з'єднання, у цьому разі втрати з'єднання або поганого доступу до мережі Інтернет, доступ до функцій додатка може бути обмеженим або повністю недоступним.
- Канали передачі даних потребують захисту, оскільки передача даних між клієнтом і сервером може бути вразливою до атак типу перехоплення, модифікації, підміни. Тому необхідно використовувати безпечні протоколи, наприклад, HTTPS, TLS, шифрування, аутентифікацію та інші механізми для забезпечення конфіденційності та цілісності інформації.

Архітектура клієнт-сервер широко використовується в різних типах додатків, включаючи веб-браузери, системи електронної пошти, служби обміну файлами, що робить її ключовою концепцією в сучасних мережах та обчислювальних системах. Цей вид архітектури майже завжди дозволяє ефективно розподіляти навантаження, централізовано керувати ресурсами, а також спрощує обслуговування та оновлення програмного забезпечення. Завдяки цим властивостям, ця модель залишається основою для побудови сучасних мережевих та обчислювальних систем.

1.2.2. Базові структурні компоненти серверних застосунків

Всі ключові ресурси, логіка та дані зберігаються та обробляються на сервері, що дозволяє адмініструвати систему з одного центру. Це спрощує управління доступом, моніторинг активності та контроль безпеки [8].

Оскільки логічна обробка та зберігання даних відбуваються на стороні сервера, оновлювати потрібно лише сервер, і всі клієнти автоматично отримають доступ до нових функцій або виправлень. Це зменшує витрати на підтримку великої кількості клієнтських пристроїв.

Чітке розділення функцій між клієнтом і сервером спрощує розробку системи: клієнт відповідає за інтерфейс, а сервер - за логіку та обробку даних. Це дозволяє різним командам працювати незалежно, що прискорює розробку та полегшує тестування і обслуговування.

Веб-додатки зазвичай складаються з декількох ключових компонентів, які взаємодіють між собою для забезпечення стабільної роботи, зручного інтерфейсу та безпечної обробки даних. Архітектура веб-додатків включає як клієнтську, так і серверну частини, кожна з яких виконує певні функції. Нижче наведено основні компоненти, які найчастіше використовуються при створенні сучасних веб-додатків.

Клієнтські компоненти:

- Користувацький інтерфейс, тобто візуальна частина додатка, з якою взаємодіє кінцевий користувач. Він включає елементи управління, такі як кнопки, поля введення, списки, що розкриваються, панелі навігації, таблиці та повідомлення. Основна мета корист. інтерфейсу - зробити взаємодію інтуїтивно зрозумілою та приємною. При розробці користувацького інтерфейсу особлива увага приділяється адаптивності підтримка різних розмірів екрану, зручності використання, або як його ще називають – юзер експіріенсу та доступності.
- Рівень презентації, оскільки цей рівень відповідає за візуалізацію інтерфейсу та форматування вмісту. Для його реалізації використовуються такі

технології: HTML - відповідає за структуру сторінки, визначає логічну структуру документа, дозволяє створювати інтерактивні форми з полями введення, створювати гіперпосилання для навігації між сторінками та надає можливість вбудовувати мультимедіа (відео, аудіо, зображення). CSS - дозволяє стилізувати сторінку, встановлювати кольори, шрифти, розміщення елементів, анімацію та переходи між сторінками, а також створювати сторінки, які будуть коректно відображатися на різних пристроях. JavaScript - мова програмування, що забезпечує динамічну поведінку веб-сторінок. Вона дозволяє реалізувати логіку на стороні клієнта, тобто включає в себе такі основні функції: забезпечує інтерактивність: перевірка форм, динамічне завантаження контенту, анімація, реакція на події.

Компоненти серверної частини:

- Логіка додатка, тобто ядро веб-додатка, яке управляє бізнес-правилами, обробкою запитів, перевірками, розрахунками та взаємодією з іншими компонентами системи. Саме тут виконуються ключові функції додатка, такі як обробка замовлень, реєстрація користувачів або розрахунок вартості доставки.
- Веб-сервер та API, тобто у цьому випадку веб-сервер приймає HTTP(S) - запити від клієнтів і перенаправляє їх до відповідних обробників. Результати обробки повертаються клієнту у вигляді відповідей. Наш API визначає, як клієнт і сервер взаємодіють між собою. Вони використовуються, оскільки дозволяють розробникам реалізувати обмін даними у структурованому вигляді.
- Формати обміну даними: існує два найпоширеніші формати обміну даними: JSON (JavaScript Object Notation) – зручний формат, який широко використовується завдяки своїй компактності та сумісності з JavaScript. Другий, це XML, який раніше був популярним для структурованого обміну, але є більшим і складнішим. У сучасних серверних застосунках переважає JSON, оскільки він забезпечує легкість, швидкість в обробці та краще підходить для передачі вкладених структур.
- Перевірка та верифікація даних. Перевірка даних завжди повинна здійснюватися на сервері, навіть якщо вона була виконана на клієнті. Це, в свою чергу, захистить сервер від різних атак.

- Для відстеження помилок і запису їх у файли журналів слід виконувати моніторинг і ведення журналів, щоб ми могли виявляти аномалії в роботі системи, тобто використовувати такі інструменти, як Prometheus, Grafana,ELK Stack або APM, які дозволяють розробникам і адміністраторам своєчасно реагувати на збої та оптимізувати систему. Також необхідно організувати процес збору аналітики продуктивності, щоб розробник міг детально відстежувати, як працює створений веб-додаток.

Всі вищезазначені компоненти взаємодіють між собою, щоб забезпечити стабільну роботу веб-додатку, обробку запитів користувачів, управління інформацією та доставку динамічного контенту.

Також слід зазначити, що набір використовуваних компонентів і технологій може відрізнятися залежно від специфіки проекту, його обсягу та функціональних або технічних вимог.

1.2.3. Дворівнева модель архітектури серверного застосунку

Дворівнева архітектура є класичною формою реалізації клієнт-серверної моделі. Вона характеризується простотою реалізації, але має обмеження при роботі з великими системами або високими навантаженнями. У цій архітектурі всі елементи системи умовно поділяються на два рівні: клієнт і сервер [9].

Рівень клієнта відповідає за взаємодію з кінцевим користувачем і формування інтерфейсу. Цей рівень зазвичай реалізується у вигляді настільної програми або веб-інтерфейсу. Всі дії користувача, такі як введення даних у поля, натискання кнопок, або щось схоже обробляються на стороні клієнта, включаючи базову перевірку даних, до прикладу, перевірка формату адреси електронної пошти або відсутність порожніх полів.

Отримання та відображення відповідей сервера, тобто після обробки запиту сервер повертає результат клієнту, до прикладу, повідомлення про успішне збереження або дані для таблиці, який клієнтська програма відображає в інтерфейсі.

Локальне зберігання даних, якщо використовується, у цьому випадку клієнт може мати локальну базу даних або кеш для тимчасового зберігання інформації, що зменшує навантаження на сервер і прискорює доступ до часто використовуваних даних.

Серверний рівень у дворівневій архітектурі реалізує основну логіку та програмний код, обробляє запити клієнтів та забезпечує взаємодію з базами даних та інформацією, що в них зберігається. В цій архітектурі сервер також:

Сервер отримує запити від клієнтів, розпізнає, яку дію потрібно виконати, до прикладу, отримати список продуктів, зберегти нове замовлення, і викликає відповідні обробники.

Сервер реалізує логіку, яка визначає, як повинна працювати система. Наприклад, перевірка, чи може користувач розмістити замовлення, чи є в наявності достатня кількість товару, або розрахунок ціни зі знижкою [10].

Після виконання дій і запитів сервер повертає відповідь клієнту у форматі, в якому був отриманий запит, і відправляє її назад клієнту.

Комунікація між клієнтом і сервером у дворівневій архітектурі проходить наступним чином, тобто у дворівневій архітектурі комунікація між клієнтським і серверним рівнями є прямою, тобто немає проміжного рівня, до прикладу, як рівня додатків у трирівневій архітектурі. Це означає, що клієнтський додаток безпосередньо комунікує з сервером, який обробляє запити та взаємодіє з базою даних.

Робота з базою даних у дворівневій архітектурі:

- У дворівневій моделі сервер безпосередньо отримує доступ до бази даних для читання, оновлення або збереження інформації. Вся відповідальність за підключення, відправку запитів і отримання результатів лежить на серверній стороні системи.
- Сервер зазвичай оснащений спеціальними модулями або бібліотеками, що забезпечують взаємодію з базами даних. Комунікація здійснюється за допомогою специфічних протоколів, характерних для конкретних систем управління базами даних, а для виконання операцій з даними використовується

мова запитів, зокрема SQL, яка є стандартною мовою для роботи з реляційними базами даних.

- Розгортання та масштабування дворівневої архітектури. Для запуску дворівневої архітектури зазвичай потрібно розмістити серверний модуль на окремому фізичному або віртуальному сервері, який виконує всі серверні функції системи. Процес масштабування в дворівневій моделі є досить складним, оскільки один і той самий сервер обробляє як логіку додатка, так і взаємодію з базою даних, що може спричинити затримки в роботі. Для підтримки зростаючого навантаження часто доводиться модернізувати апаратне забезпечення сервера або додавати нові сервери для рівномірного розподілу обробки запитів. Слід зазначити, що із зростанням кількості користувачів і ускладненням даних система може зазнавати зниження продуктивності. Це пов'язано з тим, що всі запити обробляються централізовано одним або обмеженою кількістю серверів, ресурси яких можуть швидко вичерпатися під час обробки даних [11].

Переваги дворівневої архітектури:

- Простота реалізації, що означає, що створити та впровадити таку архітектуру набагато простіше, ніж більш складні моделі. Вона також простіша для розуміння з точки зору структури та принципів роботи.
- Прямий контроль інтерфейсу, що означає, що сторона клієнта має повний контроль над зовнішнім виглядом та поведінкою інтерфейсу, що дозволяє легко адаптувати зовнішній вигляд програми до потреб користувача.
- Мінімізація складності: оскільки між клієнтом і сервером немає додаткових проміжних ланок, таких як серверні додатки або посередники, загальна структура системи спрощується, що також знижує пов'язані з цим витрати.

Недоліки дворівневої архітектури:

- Складність масштабування та зниження продуктивності: у цьому типі архітектури із зростанням кількості користувачів архітектура швидко вичерпує свої ресурси. Це призводить до зниження продуктивності, оскільки все навантаження зосереджується на обмеженій кількості серверів.

- Обмежена гнучкість: оскільки клієнтська та серверна частини в цьому типі архітектури дуже тісно пов'язані, змінити або додати нові функції до системи складніше. Також буде складно адаптувати систему до нових технологічних вимог.
- Проблеми безпеки: у цьому типі архітектури пряме з'єднання між клієнтом і сервером збільшує ризик несанкціонованого доступу до даних або виконання небажаних дій. Вразливості в коді клієнта можуть безпосередньо вплинути на безпеку всієї системи.

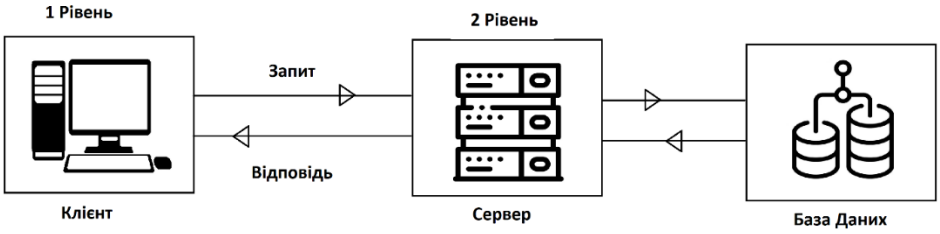

Рисунок 1.1 – Дворівнева модель архітектури.

На ранніх етапах розвитку веб-технологій багато веб-додатків будувалися на дворівневій архітектурі. У цій моделі клієнтська частина — зазвичай веб-браузер або користувальницький інтерфейс, він безпосередньо спілкується з веб-сервером. Веб-сервер, у свою чергу, відповідає не тільки за обробку вхідних запитів, але й за виконання всієї бізнес-логіки та доступ до бази даних. Усі ці функції об'єднані в одному шарі серверної частини системи.

Такий підхід передбачає тісний зв'язок між клієнтськими та серверними компонентами, що означає, що стабільність і швидкість роботи клієнта повністю залежать від відповіді сервера. У разі перевантаження або збою сервера інтерфейс клієнта також перестає працювати належним чином, що негативно позначається на користувацькому досвіді.

Однак з часом стало зрозуміло, що дворівнева архітектура не здатна ефективно обробляти великі обсяги трафіку та складні обчислювальні процеси. Тому для побудови більш потужних і гнучких інформаційних систем розробники почали використовувати багаторівневі архітектури, такі як трирівнева або n-рівнева. Такі архітектурні моделі дозволяють розподілити обов'язки між різними

рівнями, презентація, логіка додатків, доступ до даних, що значно покращує масштабованість, продуктивність і спрощує обслуговування та оновлення системи.

1.2.4. Трирівнева модель архітектури серверного застосунку

Трирівнева архітектура, яка іноді називається багаторівневою, є сучасним і гнучким способом організації програмної системи, що дозволяє розділити логіку системи на окремі, ізольовані рівні. Це сприяє масштабованості, простоті обслуговування системи, та забезпечує підвищення безпеки серверних застосунків [12].

Клієнтський рівень у трирівневій архітектурі:

- Цей рівень у розглянутій нами трирівневій архітектурі відповідає за взаємодію між користувачем і системою. Його основним завданням, як і в попередній архітектурі, є надання інтерфейсу, за допомогою якого користувач може працювати з додатком.
- Рівень клієнта обробляє клієнтську частину візуального інтерфейсу для відображення даних, введення даних користувачем, перевірки значень введених даних (наприклад, чи заповнені всі обов'язкові поля) та обробки подій, таких як натискання кнопок, перехід між сторінками тощо.
- Для створення привабливих і динамічних інтерфейсів зазвичай використовуються такі технології, як HTML для структури сторінок, CSS для стилізації та JavaScript для інтерактивності.

Рівень додатків у трирівневій архітектурі [13]:

- Отримання запитів від клієнта, коли користувач взаємодіє з інтерфейсом, наприклад, натискає кнопку, заповнює поля перед натисканням кнопки, вводить свої дані, введені дані надсилаються на рівень додатків. Цей рівень приймає запити, аналізує їхній вміст і визначає подальші дії.
- Виконання бізнес-логіки, тобто під час обробки покупки цей рівень може розрахувати загальну вартість з урахуванням знижок, перевірити наявність товарів на складі або підтвердити дійсність банківських реквізитів, тобто

переглянути весь інтерфейс, в якому буде відображатися інформація. Цей рівень також реалізує правила, за якими працює вся система.

- Взаємодія з рівнем даних, тобто після обробки логіки, рівень додатків звертається до рівня даних, наприклад, для отримання інформації про продукт, оновлення кількості одиниць на складі або збереження нового замовлення.
- Для клієнта генерується відповідь. Коли всі операції завершені, система генерує відповідь, наприклад, підтвердження замовлення на придбання ліцензійного програмного забезпечення, і надсилає її на рівень клієнта для відображення користувачеві.
- Цей рівень також часто включає додаткові компоненти, такі як сервери авторизації, контролери доступу, служби для виклику сторонніх API, наприклад API для платіжних систем або служби для отримання даних про географічне розташування клієнтів.

Рівень даних у трирівневій архітектурі:

- Цей рівень відповідає за надійне зберігання інформації, забезпечення її цілісності, безпечний доступ та виконання запитів на читання, зміну або видалення даних.
- Обробка запитів на дані включає такі операції, як створення нових записів, наприклад, створення нового користувача, читання, тобто отримання списку даних, оброблених системою, оновлення, тобто процес зміни даних, або видалення даних.
- Рівень даних підтримує послідовності дій, які повинні бути виконані повністю або не виконуватися взагалі, наприклад, при передачі змін даних. Якщо відбувається помилка, система скасовує зміни.
- Саме тут реалізуються обмеження цілісності, такі як унікальність адрес електронної пошти, а також шифрування, аутентифікація користувачів та контроль доступу до окремих таблиць або полів, які повинні бути заповнені користувачем.
- На рівні даних розробники зазвичай використовують системи управління реляційними базами даних, тобто програмні пакети, що дозволяють зберігати, організовувати та керувати інформацією у вигляді взаємопов'язаних

таблиць. Прикладами таких систем є MySQL, Oracle Database або Microsoft SQL Server. Вони забезпечують високий рівень структуризації всієї системи, також підтримують мову SQL для формулювання запитів до даних і дозволяють виконувати різні види складних операцій, такі як фільтрування, сортування, агрегація та об'єднання таблиць.

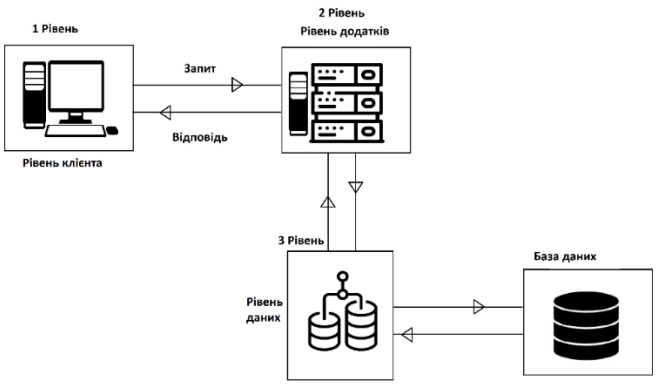

Рисунок 1.2 – Трирівнева модель архітектури.

Переваги трирівневої архітектури [14]:

- Однією з ключових переваг трирівневої архітектури є можливість масштабування кожного рівня, клієнтського, рівня додатків і даних, незалежно один від одного. Це означає, що, наприклад, якщо кількість користувачів і навантаження на логіку додатка збільшується, ви можете просто додати більше серверів додатків без необхідності змінювати клієнтський інтерфейс або базу даних. Такий підхід забезпечує ефективне горизонтальне масштабування і забезпечує стабільну роботу всієї системи навіть при стиканні з високими навантаженнями.
- У трирівневій архітектурі кожен рівень відповідає за окрему частину функціональності: клієнтський рівень відповідає за взаємодію з користувачем, рівень додатків відповідає за реалізацію бізнес-логіки, а рівень даних відповідає за зберігання та обробку інформації. Таке розділення дозволяє розробникам працювати над різними частинами додатка паралельно, зменшуючи складність кожного окремого компонента.

У свою чергу це також полегшує обслуговування, оновлення та налагодження, оскільки зміни на одному рівні зазвичай не вимагають змін на інших рівнях.

- Завдяки логічному та фізичному розділенню системи на рівні, в трирівневій архітектурі можна реалізувати багаторівневі заходи забезпечення інформаційної безпеки.

Наприклад, рівень додатків може виступати так званим посередником між користувачем або клієнтом, та базою даних, яка в свою чергу запобігає прямому доступу до конфіденційної інформації. Це дозволяє використовувати аутентифікацію, шифрування, контроль доступу та інші механізми безпеки на кожному етапі обробки даних, що значно підвищує загальний рівень безпеки системи [15].

Трирівнева архітектура, виступає перевіреним і надійним підходом до побудови програмних систем, особливо тих, що передбачають активну взаємодію користувачів з системою, або ж велику кількість операцій з різними видами даних або вимагають високої продуктивності.

1.2.5. Порівняння дворівневої та трирівневої архітектурної моделі

На мій погляд, для кращого розуміння особливостей сучасних програмних систем доцільно провести детальний порівняльний аналіз двох найпоширеніших архітектур серверних додатків, тобто дворівневої та трирівневої. Це дозволяє побачити не тільки їхні переваги та недоліки з точки зору продуктивності та безпеки, але й те, як вони відповідають різним типам вимог. Нижче я представлю свій аналіз на основі ключових порівняльних характеристик [16].

Порівняння структури цих двох архітектур:

Дворівнева архітектура складається з двох основних компонентів: клієнтської частини та серверної частини. Клієнт надсилає запити безпосередньо на сервер, який обробляє всю бізнес-логіку, взаємодіє з базою даних і повертає результат. Вся

логіка, включаючи перевірку, обчислення, обробку даних і відповідь на запити, централізована на сервері.

Трирівнева архітектура додає до цього ланцюга ще один рівень - рівень додатків. У цій моделі клієнт надсилає запит до сервера додатків, який відповідає за логіку обробки, а потім отримує доступ до баз даних для отримання або зберігання інформації. Відповідь формується на рівні додатків і надсилається назад клієнту. Така структура дозволяє розподілити обов'язки між різними службами, покращуючи керованість і масштабованість системи.

На мій погляд, трирівнева модель більш природно відображає реальні бізнес-процеси, де між інтерфейсом користувача та даними існує логічна “прослойка”, яка контролює правила, перевірки та обчислення.

Порівняння масштабованості цих двох архітектур:

Дворівнева архітектура, як я вже описував раніше в цьому розділі, має обмеження з точки зору масштабованості. Всі клієнти отримують доступ до одного сервера, і з збільшенням кількості користувачів це може призвести до перевантаження інформації та сервера. Масштабування тут можна досягти лише шляхом впровадження більш потужного сервера, але, як ми знаємо, будь-який сервер має свої технічні обмеження, і розширення лише технічних можливостей збільшує фінансові витрати на це рішення.

Трирівнева архітектура, в свою чергу, підтримує масштабування кожного рівня окремо. До прикладу, ви можете додати додаткові логічні сервери без зміни сховища даних або інтерфейсу клієнта. Це дозволяє більш ефективно обробляти велику кількість одночасних запитів без ризику коллапсу інформаційної системи або перевантаження сервера.

На мій особистий погляд, у сучасному веб-середовищі, де навантаження часто є динамічними і непередбачуваними, можливість незалежного масштабування частин системи є вирішальною перевагою трирівневої архітектури.

Порівняння гнучкості та модульності цих двох архітектур:

У дворівневій архітектурі компоненти є дуже взаємозалежними. Зміни в логіці сервера можуть вимагати змін на стороні клієнта і навпаки. Це дуже сильно ускладнює оновлення, тестування та обслуговування системи.

Трирівнева архітектура забезпечує високу модульність. Розробники серверних застосунків можуть створювати окремі сервіси, які відповідають тільки за свою функцію, що значно спрощує оновлення та обслуговування окремих частин загальної системи. Наприклад, зміна формули розрахунку ціни не вимагатиме змін у клієнті або базі даних.

Порівняння складності реалізації кожної з двох архітектур:

Дворівнева архітектура проста в реалізації, що робить її привабливою для невеликих проектів або стартапів. Менша кількість рівнів означає менше коду і менше налаштувань, а тобто і меньші витрати на розробку.

Трирівнева архітектура додає складності розробці, оскільки вимагає налаштування взаємодії між рівнями, розробки API, управління авторизацією та балансування навантаження, яке буде надходити на різні рівні системи. Однак складність такої розробки буде виправдана при тривалому використанні системи або при великій кількості користувачів [17].

На мій особистий погляд, поріг входу в трирівневу архітектуру, безумовно, вищий, але розробка забезпечить компанію/стартап/підприємство дуже надійною, стабільною системою, яка буде проста в обслуговуванні та дозволить розмежовувати процеси оновлення різних рівнів системи.

Порівняння продуктивності двох архітектур:

У дворівневій архітектурі все навантаження з боку клієнта припадає на один сервер, що з часом може створити іформаційну вразливість, тобто сервер може просто перевантажитися. Це можна помітити, коли система одночасно обробляє запити від багатьох користувачів і виконує важкі процеси або обробляє складні функції.

Трирівнева архітектура, в свою чергу, дозволяє ефективно розподіляти обробку, тобто логіка системи виконується окремо, база даних не перевантажується

операціями, а клієнт отримує відповідь швидше. Звичайно, це позитивно впливає на швидкість і стабільність системи в цілому.

На мій особистий погляд, продуктивність не тільки має на увазі швидкість, але й стабільність під навантаженням, і тут трирівнева архітектура також демонструє свої очевидні переваги.

Порівняння безпеки в цих двох архітектурах:

Як я вже згадував раніше, дворівнева модель змушує клієнта взаємодіяти безпосередньо з сервером, що збільшує кількість потенційних точок вторгнення в інформаційну систему.

У трирівневій архітектурі можна реалізувати захист на декількох рівнях. Наприклад, на рівні додатків можна виконувати аутентифікацію, реєструвати дії та шифрувати дані до того, як запит досягне бази даних. Це дозволяє краще контролювати доступ до критичної інформації або інформації з обмеженом доступом, конфіденційної інформації, робочої таємниці тощо, та забезпечує надійну безпеку серверного додатка.

Підсумовуючи це порівняння, я б сказав, що трирівнева архітектура клієнт-сервер забезпечує вищий рівень масштабованості, гнучкості, модульності та продуктивності в порівнянні з дворівневою архітектурою. Хоча дворівневий підхід може бути цілком прийнятним для простих систем або невеликих стартапів, які не можуть дозволити собі розробку більш складних систем, а також для них притаманна невелика кількість користувачів, та їх системі притаманні мінімальні функціональні вимоги, тому в даному випадку розробка більш складного серверного застосунку не має сенсу. Однак, виходячи з порівняння, трирівнева архітектура є кращим вибором для складних додатків, які вимагають розподілу обов'язків, легкої масштабованості та підвищеної адаптивності до сучасних вимог.

Висновки за розділом 1

У першому розділі наведено детальний аналіз видів архітектури сучасних серверних додатків та підходів до їх розробки. Серверні додатки відіграють

ключову роль у сучасній інформаційній інфраструктурі, починаючи від обробки запитів користувачів і управління базами даних до реалізації складних бізнес процесів, та авжеж забезпечення інформаційної безпеки системи. Їх ефективне проектування безпосередньо впливає на продуктивність, масштабованість та зручність обслуговування програмних систем.

Було досліджено основні принципи двохрівневої архітектури клієнт-сервер, яка є основою для більш складних структур. Хоча підхід двохрівневої архітектури є простим, він більше підходить для зручності користування невеликим проектам, вале треба розуміти, що він виявляється обмеженим при роботі з великою кількістю користувачів або роботи зі складними запитами.

Особливу увагу було приділено порівнянню двох основних архітектур клієнт-серверної моделі, дворівневої та трирівневої. Дворівнева архітектура характеризується простотою реалізації, що робить її привабливою для стартапів. Однак пряма взаємодія клієнтської сторони з логікою сервера та базами даних обмежує масштабованість і можливість внесення змін до існуючих систем.

На відміну від цього, трирівнева архітектура демонструє значно вищу гнучкість, модульність та реалізацію розподілених оновлень системи. Її структура включає в себе, чіткий поділ на рівень презентації, рівень додатків та рівень доступу до даних, що у свою чергу дозволяє розробляти кожен компонент незалежно, масштабувати за потреби та підтримувати без втручання в інші частини системи. Такий підхід не тільки підвищує ефективність розробки самого серверного застосунку, але й значно спрощує взаємодію з ним, а також спрощує впровадження нових функцій або змін до існуючих.

У реальному середовищі, трирівнева архітектура краще справляється з високими навантаженнями, оскільки кожен рівень можна оптимізувати для обробки інформації та даних, або для захисту від інформаційних атак окремо. Ця архітектура також створює умови для більш надійного впровадження заходів інформаційної безпеки, оскільки дозволяє встановлювати контроль доступу та фільтри безпеки на кожному рівні серверного застосунку. Завдяки цим перевагам трирівнева модель є

на сьогодні найпоширенішим рішенням для розробки складних, розподілених або корпоративних серверних застосунків.

Результати цього розділу підтверджують, що глибоке розуміння особливостей архітектури серверних застосунків є основою для прийняття вибору виду архітектури, під час розробки програмного забезпечення. Але основною важливою вимогою буде врахування, а чи варто витрачати фінансові ресурси підприємства на розробку більш складних видів клієнт-серверної архітектури, а також враховувати вимоги до масштабованості, підтримки бізнес-логіки, безпеки та подальшого розвитку системи. Якщо ці вимоги треба врахувати, вибір в більшості випадків очевидний і приводить нас до вибору трирівневої архітектури у розробці серверних застосунків.

РОЗДІЛ 2
ДОСЛІДЖЕННЯ ВРАЗЛИВОСТЕЙ СЕРВЕРНИХ ЗАСТОСУНКІВ ТА ЇХ ЗАХИСТ

2.1. Потенційні вектори атак і слабкі місця серверних застосуснків

Серверні додатки, як невід'ємна частина сучасної IT-інфраструктури, постійно піддаються ризику атак зловмисних осіб. Порушення основних властивостей інформаційної безпеки, таких як конфіденційність, цілісність і доступність, найчастіше здійснюються шляхом використання вразливостей в програмному забезпеченні. Вразливості можуть виникати як на рівні коду, а також на рівні архітектури або конфігурації інформаційної системи.

Велика кількість таких вразливостей пов'язана з тим, що серверні додатки не можуть правильно обробляти непередбачені або ненадійні вхідні дані від клієнтів. Найчастіше це відбувається, до прикладу, коли очікується обробка певного типу вхідної інформації, але через недостатній контроль даних зловмисник вводить спеціально сформоване значення або запит, яке порушує нормальне виконання програми або навіть цілої системи. Крім того, із зростанням складності програмного забезпечення практично неможливо повністю уникнути таких проблем. Складні взаємодії між компонентами, численні залежності від зовнішніх бібліотек та швидкі темпи розробки серверних додатків призводять до недостатнього тестування, що робить сучасні серверні застосунки вразливими до широкого спектру атак.

До основних факторів, що сприяють появі та успішному використанню вразливостей у серверних застосунках, належать:

У трирівневій архітектурі існує черезмірна залежність від компонентів і бібліотек. Багато сучасних серверних додатків використовують відкриті бібліотеки, фреймворки або готові програмні рішення, які можуть містити відомі вразливості, але розробники нехтують захистом від цих відомих вразливостей. До прикладу,

використання застарілої версії фреймворку або пакету, в якому вже виявлено критичну помилку безпеки, робить весь додаток вразливим до відомих ризиків інформаційної безпеки. Дуже часто розробники не мають часу для своєчасного оновлення таких компонентів, або не мають достатньої автоматизації для відстеження оновлень, що створює сприятливе середовище для успішних атак зловмисників.

Недостатні знання в галузі інформаційної безпеки серед розробників серверних застосунків приводить до того, що вразливості часто виникають через людські помилки, до прикладу, неправильне застосування методів перевірки даних, отриманих сервером, неправильне управління сесіями, незахищене виконання аутентифікації або недостатнє шифрування даних. Якщо розробники не мають достатньої підготовки в галузі інформаційної безпеки або не дотримуються рекомендацій у процесі розробки з усуненням усіх відомих вразливостей, ризик виникнення вразливостей значно зростає. Це особливо важливо у великих командах, де немає чіткої політики внутрішнього контролю якості розробки програмного забезпечення або створення комплексних систем.

Слід також зазначити, що більшість розробників не знають про нові вразливості, що виникають. У багатьох випадках розробники або групи розробників не мають часу або ресурсів для своєчасного моніторингу нових типів атак або вразливостей у програмному забезпеченні. Наприклад, без системи виявлення загроз та проведення регулярних аудитів інформаційної безпеки в компанії загрози можуть залишатися непоміченими, до моменту коли атака вже буде успішною. Крім того, більшість команд розробників серверних застосунків не мають ефективних каналів комунікації між відділами розробників та фахівцями з кібербезпеки.

WASC (Web Application Security Consortium Threat Classification) [18], це всесвітньо визнана система класифікації вразливостей, пов'язаних із безпекою серверних застосунків. Вона була розроблена міжнародною некомерційною організацією, яка об'єднує провідних фахівців із кібербезпеки та розробки з усього світу. Основна мета цієї класифікації це створити загальне розуміння типів

вразливостей та сприяти їх кращому виявленню, аналізу та усуненню в серверних застосунках.

Відмінною рисою моделі WASC є її орієнтація на весь життєвий цикл серверної програми. Класифікація враховує джерело вразливості на кожному етапі створення, розробки та впровадження системи:

- Вразливості проектування виникають на етапі проектування системи. Зазвичай вони є результатом помилок у логіці проектування архітектури, відсутності принципів безпеки при плануванні взаємодії компонентів або неправильного розуміння потенційних загроз.
- Вразливості реалізації з'являються під час розробки програмного коду. Цей тип помилок виникає через недосконаліть кодування, використання небезпечних системних компонентів або відсутність перевірки вхідних даних. Саме на цьому етапі з'являються класичні приклади ін'єкцій, міжсайтового скриптингу та помилок активного управління сеансами.
- Вразливості конфігурації формуються під час розгортання додатків, коли система налаштовується для роботи в реальному середовищі. Неналежне відкриття портів, неправильне управління дозволами або зберігання облікових даних у вигляді відкритому виді – це приклади проблем, які можуть стати критичними загрозами через людські помилки на етапі конфігурації застосунку.

Окрім класифікації за фазами життєвого циклу, WASC також класифікує вразливості за типом атак. Серед цих класифікацій, виділяють такі основні категорії:

- Атаки на аутентифікацію [19], оскільки вони спрямовані на порушення механізмів аутентифікації користувачів системи. Зокрема, це можуть бути атаки методом грубої сили для розсекречення паролів, використання слабких політик аутентифікації або обхід системи через недостатньо захищені процедури відновлення доступу. Особливо небезпечними є випадки порушення безпеки облікових записів адміністраторів, які відкривають доступ до критично важливих функцій всієї системи, і можуть привести до інформаційного колапсу.

- Атаки на авторизацію. Цей тип атак спрямований на обхід або використання механізмів контролю доступу. Зловмисники можуть використовувати передбачувані або незахищені ідентифікатори сеансів користувачів, здійснювати атаки шляхом підстановки параметрів або використовувати недоліки в контролі доступу. В результаті порушується ізоляція користувачів і виникає ризик витоку даних або несанкціонованих дій.
- Атаки на стороні клієнта. Ці види загроз включають спроби вплинути на браузери користувачів або ввести шкідливий вміст. Прикладами є атаки міжсайтового скриптингу, підміна HTML-вмісту, використання неправильних заголовків HTTP або маніпулювання фрагментацією HTTP-запитів. Мета цієї атаки полягає у тому, щоб виконати код на стороні клієнта, викрасти сесії або паролі або ввести користувача в оману.
- Атаки на серверну інфраструктуру. У цьому випадку зловмисники прагнуть отримати контроль над сервером або повністю порушити його роботу. Це можуть бути атаки переповнення буфера, SQL, LDAP, XML-ін'єкції або виконання довільного коду в серверному середовищі, коли був отриманий доступ до консолі адміністратора. Крім того, вразливості операційних систем, можуть теж бути використані для виконання команд з правами адміністратора.
- Інші загрози, пов'язані з логікою або архітектурою додатків. До них відносяться менш очевидні, але не менш небезпечні вразливості, такі як, передбачуваність імен файлів або каталогів, витік метаінформації, до прикладу, шляхи до критично важливих ресурсів, витік конфіденційних запитів пошуковими системами та несанкціонований доступ через обхід механізмів контролю.

Загалом, класифікація загроз WASC є цінним інструментом для структурованого аналізу ризиків безпеки серверних застосунків. Вона дозволяє не тільки виявити конкретні вразливості, але й системно підійти до їх запобігання, враховуючи різні аспекти життєвого циклу програмного продукту, тобто від проектування до фактичного використання в реальному інформаційному середовищі.

Існує також загальновизнаний перелік найкритичніших вразливостей у серверних застосунках OWASP Top 10 [20]. Він складений організацією OWASP (Open Worldwide Application Security Project), що працює над поліпшенням інформаційної безпеки. Цей перелік слугує довідником для розробників та фахівців з інформаційної безпеки.

Далі я хочу привести наглядний перелік загроз, виходячи з огляду OWASP Top 10 (2019 року):

Порушення контролю доступу, тобто ця вразливість серверного застосунку виникає, коли обмеження доступу до функцій або даних реалізовано неправильно. Це дозволяє зловмисникам змінювати дані інших користувачів, отримувати несанкціонований доступ до інших облікових записів, змінювати права доступу або отримувати доступ до обмежених ресурсів.

Атаки типу ін’єкцій, коли зловмисник вставляє шкідливі дані, до прикладу, SQL-запити, у поля введення програми, що дозволяє йому виконувати небажані команди або отримувати доступ до конфіденційних даних користувачів або навіть адміністраторів системи.

Міжсайтовий скриптинг, успішне використання цієї вразливості дозволяє виконувати шкідливий код у браузері жертви. Найчастіше використовується для викрадення сеансів, заміни вмісту веб-сторінок або перенаправлення користувачів на шкідливі підроблені сайти.

Порушення аутентифікації, цей тип атаки включає в себе помилки в реалізації механізмів входу, зберігання сеансів або зміни пароля. Це дозволяє зловмисникам захоплювати облікові записи, отримувати доступ до сеансів користувачів і навіть адміністраторів.

Витік конфіденційних даних, цей вид загрози може реалізуватися, коли програма не захищає дані належним чином, до прикладу, не шифрує передачу або зберігання, це відкриває можливість викрадення особистої інформації, фінансових даних тощо.

Вразливість XML [21], вона виникає коли додаток неправильно перевіряє чи очищає вводяться користувачем дані перед їх аналізом у форматі XML, а також у

старих або неправильно налаштованих XML-процесорах можуть оброблятися шкідливі зовнішні ресурси, що призводить до витоку даних, віддаленого виконання коду або DoS-атак.

Неправильна конфігурація безпеки, полягає у недостатній увазі стандартним налаштуванням, надто докладними повідомленнями про помилки, відкритим хмарним сховищем або застарілим програмним забезпеченням. Важливо регулярно перевіряти конфігурації та застосовувати оновлення.

Небезпечна десеріалізація полягає у тому, що зловмисники можуть використовувати механізми десеріалізації для виконання віддаленого коду, зловмисник може замінювати вміст, підвищувати собі привілеї або отримати доступ до здійснення інших атак.

Використання компонентів із відомими вразливостями, наприклад таких компонентів, як бібліотеки, фреймворки, які часто використовують ті самі привілеї, що й сама програма. Якщо вони мають вразливості, це ставить під загрозу всю систему. Дуже важливо оновлювати всі компоненти системи та перевіряти їх.

Недостатнє логування та моніторинг системи, без належного моніторингу атаки можуть залишатися невиявленими протягом тривалого часу. Це ускладнює реагування на інциденти інформаційної безпеки та дозволяє зловмисникам діяти без перешкод. Дослідження показують, що порушення часто виявляються не внутрішніми системами, а сторонніми організаціями в процесі проведення аудиту інформаційних систем.

Списки з класифікаціями вразливостей, такі як WASC та OWASP Top 10, є критично важливими інструментами для розуміння та попередження загроз інформаційній безпеці. Та повинні бути ключовими інструментами, коли розробник планує створення серверного застосунку.

2.2. Порушення механізмів контролю доступу

Контроль доступу є критично важливим компонентом системи безпеки серверних застосунків. Його завдання полягає в обмеженні дій користувачів

відповідно до їхніх прав доступу. Для цього система повинна забезпечити два ключові етапи: спочатку аутентифікацію, тобто ідентифікацію особи користувача, а потім авторизацію, тобто визначення дозволених дій після успішного входу в систему. Авторизація відповідає за те, щоб користувачі мали доступ тільки до тих ресурсів і функцій, які вони фактично мають право використовувати. Неправильна реалізація контролю доступу або його відсутність часто призводить до критичних вразливостей системи.

Важливо розуміти, що авторизація, це не одноразова перевірка. Вона повинна бути вбудована в логіку кожної операції, яка вимагає обмеження. Якщо система не перевіряє права доступу до інформаційної системи кожного разу, коли користувач намагається виконати дію або отримати доступ до ресурсу, існує ризик, що користувач, або зловмисник який отримав несанкціонований доступ до акаунту користувача системи, зможе діяти поза межами своїх дозволів.

Порушення контролю доступу можна класифікувати за типом контролю, який було реалізовано неправильно. Найпоширенішими є два типи порушення контролів доступу, таких як вертикальні та горизонтальні.

Вертикальний контроль доступу передбачає обмеження прав користувачів на основі їх ієрархічного положення або ролі в системі. Це означає, що користувачі з різними ролями мають доступ до різних функцій або рівнів даних. До прикладу, у системі управління персоналом звичайний співробітник може бачити тільки свої дані, менеджер може бачити дані своєї команди, а адміністратор має повний доступ до всієї системи, тобто може редагувати складові системи, або привілеї усіх користувачів. Суть вертикального контролю полягає у визначенні чітких рівнів дозволів та призначенні обмеженого набору дій для кожного рівня. До прикладу, побудова системи може мати наступний вигляд:

- Адміністратор може мати право змінювати або видаляти облікові записи, або конфігурацію системи;
- Менеджер системи може переглядати статистику команди та звітність, але не може змінювати налаштування системи;

- Користувач може користуватися лише тими правами, які йому потрібні, без можливості впливати на інші облікові записи чи процеси діяльності системи в цілому.

Тепер можемо роздивитись основні принципи, що лежать в основі вертикального контролю доступу:

- Розділення рівнів авторизації, тобто кожен рівень користувачів повинен мати чітко визначені привілеї;
- Дозволи на дії та об'єкти, маю на увазі, що доступ до ресурсів системи повинен бути визначений відповідно до ролі користувача;
- Найкраща практика інформаційної безпеки у цьому випадку, це принцип мінімальних привілеїв, коли кожному користувачеві повинні бути надані мінімальні дозволи, необхідні для виконання його завдань пов’язаних з інформаційною системою;

Горизонтальний контроль доступу у свою ж чергу регулює доступ до ресурсів одного типу, які належать різним користувачам, але мають однаковий рівень прав. Його мета полягає у тому, щоб запобігти роботі одного користувача з даними іншого, навіть якщо вони мають однакові загальні дозволи. Наприклад, усі користувачі в системі можуть редагувати власний профіль, але ніхто не повинен мати можливість редагувати профіль іншої особи.

Цей тип контролю особливо актуальний в онлайн-сервісах, таких як банківські, медичні, освітні установи тощо, де кожен користувач повинен мати доступ тільки до своїх транзакцій, медичних записів або навчальних матеріалів.

Типовим прикладом порушення горизонтального контролю доступу є ситуація, коли користувач змінює ідентифікатор ресурсу в URL-адресі або запиті API і отримує доступ до інформації іншого користувача. Це може статися, якщо система не перевіряє, чи дійсно запитувані дані належать цьому конкретному авторизованому користувачу.

Такі порушення часто трапляються, коли логіка реалізована спрощеним чином на стороні клієнта, без реалізацій повторних перевірок на рівні сервера, а

також коли права доступу реалізовані лише для формальності або умовно, тобто ніде не регламентовані (хоча б мінімально).

Загалом порушення контролю доступу становлять значну загрозу для будь-якого серверного застосунку. Вони не тільки дозволяють зловмисникам отримати доступ до конфіденційної інформації або змінити критичні налаштування, але й можуть повністю скомпрометувати систему інформаційної безпеки в цілому. Розробники повинні постійно бути обізнаними про цей вид загроз та впроваджувати надійні та доцільно продумані механізми авторизації.

2.3. Протидія вразливості типу «Порушення контролю доступу»

2.3.1. Система керування доступом на основі ролей

Одним з найефективніших методів запобігання порушенням контролю доступу є впровадження моделі контролю доступу на основі ролей [22] (далі, буду використовувати позначення RBAC). Це один з найпоширеніших і перевірених підходів у всьому світі, для забезпечення контролю до авторизації в сучасних інформаційних системах, включаючи серверні застосунки.

Суть RBAC полягає в тому, що доступ користувачів до системних ресурсів, буде визначатися не їхніми індивідуальними ідентифікаторами, а заздалегідь визначеними ролями, які відповідають їхнім посадовим обов'язкам або функціям в організації. Замість того, щоб призначати дозволи окремим користувачам, дозволи пов'язуються з ролями, а користувачам призначаються відповідні ролі. Такий підхід забезпечує централізоване, контрольоване та структуроване управління правами доступу.

Модель RBAC є, по суті, еволюцією дискреційного контролю доступу (DAC), але вона краще відображає структуру реального бізнесу, оскільки ролі відображають посадові обов'язки і відповідають функціональним потребам організації [23].

У такій рольовій моделі можна виділити кілька ключових елементів:

- Визначення ролей, оскільки ролі створюються відповідно до функціональних потреб організації, кожна роль, яку створює адміністратор, буде відображати конкретну посаду або набір обов'язків. Наприклад, типовими ролями можуть бути: адміністратор, керівник відділу, менеджер системи, користувач. Роль, у цьому сенсі, це визначення набору дозволів, необхідних для виконання певної дії.
- Призначення дозволів ролям. Кожна роль пов'язана зі списком прав, які визначають дії та функціональні обов’язки, що можуть бути виконані, до прикладу, перегляд, створення, редагування, видалення або виконання операцій над даними. До прикладу, роль адміністратора може мати повний доступ до всього вмісту бази даних, тоді як роль користувача може мати лише доступ для читання тільки певної інформації.
- Призначення ролей користувачам, тобто користувачі отримують доступ до ресурсів, коли їм призначаються відповідні ролі, це дозволяє централізовано керувати правами доступу, замість того, щоб вручну змінювати дозволи для кожного користувача, адміністратор просто додає або видаляє ролі, пов'язані з конкретними обов'язками користувача.
- Політика доступу. В рамках RBAC визначаються політики, які формалізують відносини між ролями, дозволами та ресурсами. До прикладу, політика може передбачати, що доступ до звітів мають лише ролі деякі ролі, людей, у посадові обов’язки яких входить передача або просмотр звітів.
- Ієрархія ролей, тобто у просунутих реалізаціях RBAC використовується ієрархічна структура ролей, де ролі нижчого рівня можуть успадковувати дозволи від ролей вищого рівня, я маю на увазі, що деякі дозволи у підрозділі будуть повторюватися у керівника підрозділом та звичайного користувача, оскільки в їх функціональні обов’язки будуть входити взаємодії з одними й тими ж системами, це авжеж, спрощує адміністрування, зменшує повторення конфігурації та забезпечує більшу гнучкість.

- Принцип розподілу обов'язків, розмежовування можна налаштувати так, щоб жоден користувач не мав одночасного доступу до функцій створення та затвердження фінансових операцій або модифікування регулятивних документів.
- Гнучкість, у деяких рідкісних випадках RBAC можна доповнити умовами, за яких активуються певні ролі. Наприклад, тимчасова роль може бути призначена тільки на час робочої зміни або тільки для доступу з корпоративної мережі.

Модель рольового контролю доступу має ряд таких переваг:

- Підвищена безпека, оскільки RBAC дозволяє чітко обмежити дії, які може виконувати користувач, що значно зменшує ризик випадкового або зловмисного доступу до критично важливих систем. Користувачі мають доступ тільки до тих ресурсів, які необхідні їм для виконання своїх службових обов'язків. Наприклад, бухгалтер не може змінювати налаштування сервера, а технічний адміністратор не може бачити конфіденційну інформацію про клієнтів. Це зменшує збитки від атаки в разі злому облікового запису, оскільки зловмисник матиме обмежений набір дозволів.
- Масштабованість, оскільки RBAC є ефективним рішенням як для малих, так і для великих організацій, оскільки легко адаптується до змін у структурі або кількості персоналу. Коли структура компанії змінюється, до прикладу, додаються відділи, посади, проекти, немає необхідності вручну змінювати доступ кожного користувача, треба просто оновити або створити відповідну роль. Нову роль можна призначити одразу великій кількості користувачів.
- Легкість при підготовці до аудиту інформаційних систем, оскільки RBAC полегшує аудит інформаційної безпеки завдяки прозорій структурі дозволів. Всі дії користувачів можна чітко відстежити до їх ролей. Журнали доступу будуть логічно зрозумілі, тобто можна буде легко визначити, хто мав доступ до певного ресурсу і на якій підставі. Це спрощує перевірку відповідності стандартам безпеки (ISO 27001, PCI DSS) і створює чіткі звіти для регулюючих органів.
- Використовуючи RBAC, можна дотримуватись принципу мінімальних привілеїв, який є одним з найкращих світових практик, це гарантує, що кожен

користувач має мінімальні привілеї, необхідні для виконання своєї роботи. Це зменшує потенційні наслідки зловживання доступом або помилок користувачів.

Таким чином, RBAC є ефективним засобом організації контролю доступу в інформаційних системах, що дозволяє підвищити загальний рівень безпеки серверних застосунків.

2.3.2. Контроль доступу на основі атрибутів

Контроль доступу на основі атрибутів [24] (далі ABAC), це потужна та гнучка модель контролю, яка дозволяє приймати рішення про доступ не просто на основі ролей або імен користувачів, а на основі комбінації атрибутів. Такий підхід дозволяє враховувати широкий спектр параметрів, що описують як користувача, так і ресурс, до якого здійснюється доступ, а також поточні умови, за яких надходить запит.

Основними компонентами, які входять до контролю доступу ABAC:

- Атрибути, тобто загальні описові характеристики, які використовуються як вхідні дані для оцінки запиту на доступ. Вони поділяються на кілька категорій, серед яких можна виділити, атрибути користувача, тобто властивості, що описують суб'єкта, який ініціює запит. (до прикладу його роль або посада, відділ або команда, до якої належить користувач, а також його рівень доступу, поточний статус у якому знаходиться співробітник, активний, звільнений або у відпустці).
- Також є атрибути ресурсу, тобто ті атрибути, які описують об'єкт, до якого запитується доступ. (до прикладу, тип або формат ресурсу, класифікація даних, тобто конфіденційна, секретна, внутрішня або публічна, а також зазначення власника ресурсу).
- Також треба зазначити атрибути середовища, тобто фактори, пов'язані з часом або місцем запиту, до прикладу, час доби а також день тижня, географічне розміщення користувача та сегмент мережі, тип пристрою, канал зв’язку, а також рівень ризику.

- Не можна забувати про політики, тобто набори правил, сформульованих у вигляді умов, які визначають, за яких обставин доступ дозволяється або забороняється. Ці правила частіше за все базуються на логічних виразах, які порівнюють значення атрибутів, зазвичай виражаються за допомогою спеціальних мов, які дозволяють описувати політики у форматі, зручному для обробки машинами, а також підтримують ієрархії та успадкування, що спрощує масштабування у великих системах. Вони дозволяють централізовано керувати доступом без необхідності редагувати кожен об'єкт окремо.
- Реалізація оцінки запиту, тобто процесу, під час якого система аналізує всі атрибути та порівнює їх з існуючими політиками, щоб прийняти рішення про доступ, потім після отримання запиту система автоматично збирає всі відповідні атрибути користувача, ресурсу та середовища, і визначає, які політики можуть бути застосовані до цього випадку, після цього атрибути порівнюються з правилами політики і вже після цього приймається рішення дозволити, заборонити або переслати на перевірку (якщо треба вручну перевірити). Всі ці дії робляться динамічно, в режимі реального часу, що дозволяє враховувати поточні обставини.
- Реалізація гнучкість і точність, оскільки ABAC дозволяє створювати дуже детальні сценарії доступу, він може бути дозволений тільки за певних комбінацій умов (до прикладу, доступ до ресурсу можливий тільки з корпоративної мережі), а також можуть враховуватися тимчасові фактори (до прикладу, тільки під час активної дії сертифікату користувача). Також можна створювати політики, які автоматично адаптуються до змін в середовищі або політиках без втручання адміністратора.
- Також ABAC, підтримує адаптацію до змін умов доступу без необхідності ручного виправлення дозволів, тобто зміна будь-якого атрибуту, наприклад, переведення користувача в іншу країну або закінчення робочого дня - може автоматично призвести до перегляду усіх прав доступу, це у свою чергу дозволяє організації впроваджувати адаптивні політики безпеки, які реагують на нові ризики або події в режимі реального часу. Такий підхід зменшує ризик

несанкціонованого доступу, коли зміни статусу користувача ще не були враховані вручну.

- Для должної реалізації та ефективної роботи ABAC необхідні надійні джерела, з яких система може отримувати значення атрибутів, наприклад з каталогів користувачів, таких як LDAP або Active Directory, або з системи управління ідентифікацією, яка забезпечує централізоване зберігання та синхронізацію атрибутів, або з зовнішніх служб, таких як служби геолокації або аналізу пристроїв.

Рисунок 2.1 – Керування доступом на основі атрибутів.

Переваги моделі контролю доступу на основі атрибутів [25]:

- Максимальна гнучкість, оскільки ця модель дозволяє створювати політики, що враховують десятки параметрів одночасно, що робить її ідеальною для складних і динамічних систем.
- Підтримка складних сценаріїв, оскільки, доступ може залежати від графіка роботи або наявності сертифіката.
- Масштабованість системи доступу, тому що завдяки централізованим політикам можна легко додавати нових користувачів або правила без перегляду всієї системи дозволів.
- Також система може приймати рішення на основі поточних значень атрибутів, забезпечуючи актуальність та відповідність без ручного втручання.

Контроль доступу на основі атрибутів особливо ефективний у сучасних серверних застосунках, також використовується у хмарних платформах та середовищах з високими вимогами до безпеки. Його здатність враховувати величезну кількість умов дозволяє не тільки захищати конфіденційні дані, але й автоматично реагувати на мінливі ризики інформаційних систем.

2.3.3. Дискреційне керування доступом

Дискреційний контроль доступу [26] (далі я буду використовувати позначення DAC), він є одним з найстаріших і найпоширеніших підходів до контролю доступу в інформаційних системах. Основна ідея цієї моделі полягає в тому, що рішення про те, хто має доступ до певного ресурсу, приймаються власником або адміністратором цього ресурсу на власний розсуд.

У керуванні доступом DAC повноваження надавати дозволи на доступ делегуються окремим користувачам, які володіють ресурсами, а не централізованому адміністратору або системі політик. Такий новий підхід забезпечує велику гнучкість, але водночас може створювати ризики безпеки через ненавмисні помилки користувачів або зловмисне використання привілеїв.

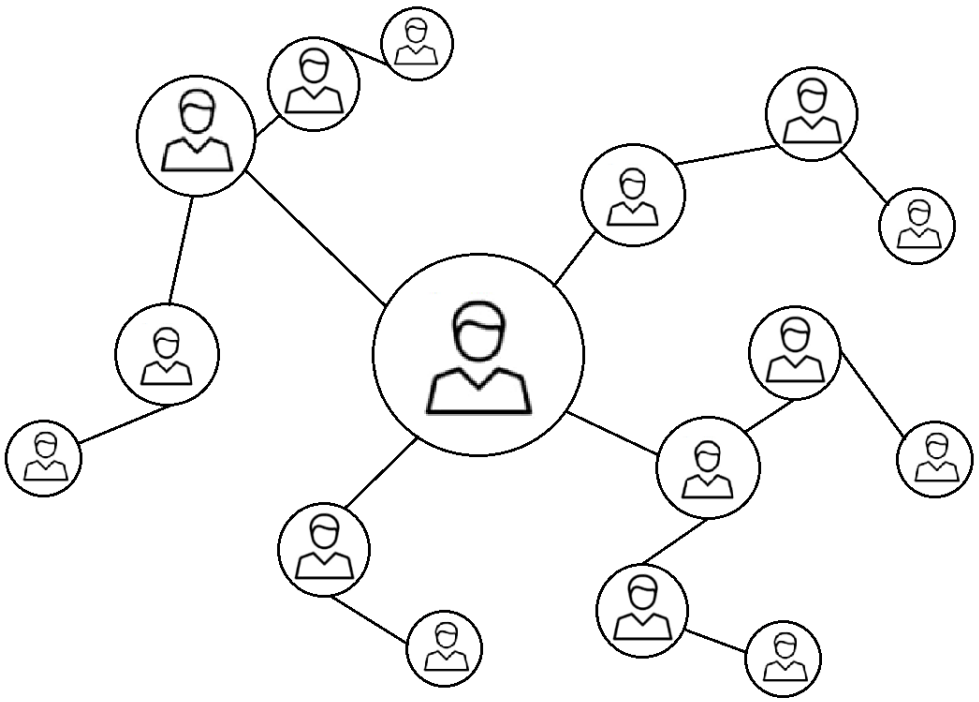

Рисунок 2.2 – Дискреційне керування доступом.

Основні особливості та принципи моделі DAC [27]:

- Контроль на основі власності об'єктів, оскільки в цій моделі кожен об'єкт у системі, наприклад файл, каталог, запис бази даних, буде мати свого власника, частіше за все це буде користувач, який його створив. Цей власник автоматично отримує повні права на ресурс, включаючи права щоб надавати або обмежувати доступ іншим користувачам. Це дозволяє побудувати модель контролю доступу, орієнтовану на суб'єктів, що значно спрощує управління в середовищах з невеликою кількістю користувачів або ресурсів. До прикладу, користувач може створити документ і може дозволити одному колезі переглядати його, іншому редагувати, а решті взаглі мати повний доступ.
- Списки контролю доступу, тобто кожен об'єкт у системі DAC має пов'язаний із ним список контролю доступу, тобто структурований набір записів, що визначає, які користувачі або групи мають які дозволи для цього об'єкта. Кожен запис у цьому списку зазвичай містить такі дані, як ідентифікатор користувача або групи, а також тип наданого доступу або заборону доступу. ACL, дуже крутий інструмент для налаштування доступу, але управління ними у великих системах може бути складним і вимагати спеціальних інструментів моніторингу та аудиту.
- Успадкування дозволів, тобто системи, що реалізують DAC, часто передбачають успадкування прав доступу, дозволяючи об'єктам автоматично успадковувати дозволи від своїх начальників або інших об’єктів. Наприклад, якщо користувач створює новий файл у певній папці, новий файл може за замовчуванням успадкувати дозволи цієї папки. Це спрощує налаштування дозволів, забезпечує узгодженість політик доступу в межах одного каталогу або області системи та мінімізує ручну роботу адміністратора або користувача. Однак автоматичне успадкування іноді може призвести до небажаного доступу, якщо власник не перевірив дозволи на об'єкт після його створення, тому с цим треба бути обережним.

- Об'єктно-орієнтована авторизація, тобто це означає, що система приймає рішення про доступ на основі того, хто робить запит і які права він має на вибраний ресурс. Таким чином, авторизація базується не на загальних правилах або ролях, а на індивідуальних налаштуваннях. Перевага цього підходу полягає в тому, що адміністратор або власник ресурсу може швидко призначити права конкретним користувачам. Але у великих організаціях така деталізація може стати дуже-дуже неефективною і складною в управлінні, оскільки прав в великих компаніях дуже багато.

Я можу виділити такі обмеження та проблеми моделі DAC:

- Відсутність централізованого контролю, оскільки кожен користувач самостійно визначає права доступу до своїх об'єктів, що ускладнює впровадження єдиної політики безпеки.
- Схильність до людських помилок, оскільки власник може випадково надати доступ не тій людині або не скасувати дозвіл після звільнення співробітника, що може в майбутньому призвести до отримання потенційного доступу зловмисником.
- Недостатнє дотримання нормативних вимог, наприклад, у середовищах з високими вимогами до дотримання нормативних вимог банки, державні органи, дискреційний контроль доступу зазвичай замінюють більш контрольованими моделями.

Модель DAC [28] наприклад використовується в таких середовищах, наприклад файлові сервери, оскільки ресурси розподіляються між користувачами, які мають автономію в управлінні доступом, а також гарним прикладом будуть соціальні мережі, оскільки користувачі вирішують, хто може бачити їхній контент, наприклад, усі, друзі або тільки я.

Дискреційний контроль доступу, це проста і зрозуміла модель, яка добре підходить для середовищ з низьким рівнем складності, невеликою кількістю користувачів і відносно низькими вимогами до інформаційної безпеки. Однак у сучасних корпоративних і хмарних інфраструктурах, де необхідні централізація та

аудит, DAC дуже сильно програє іншим моделям контролю доступу, які можна краще реалізувати.

2.3.4. Системи для забезпечення керування доступом

Одним із поширених та ефективних механізмів реалізації контролю доступу у серверних застосунках є JSON Web Token (JWT). Це легкий та автономний формат представлення інформації у вигляді токенів, що дозволяє безпечно передавати дані між сторонами, зазвичай між клієнтом і сервером. JWT [29] широко використовується для цілей аутентифікації та авторизації, оскільки дозволяє надійно підтверджувати ідентичність користувача та його права доступу до ресурсів системи.

Структура JWT отак виглядає, та він складається з трьох частин:

- Заголовок, завжди містить інформацію про тип токена та алгоритм, який буде використовуватися для підпису, до прикладу алгоритм RSA.
- Корисне навантаження, яке включає в себе так звані твердження, які представляють собою набір даних, таких як ідентифікатор користувача, ролі, дозволи, час створення, термін дії тощо.
- Підпис, який створюється шляхом хешування закодованих заголовка та навантаження разом із секретним ключем. Підпис дозволяє перевірити аутентичність токена і виявити спроби його модифікації.

JWT не лише компактний, а й не потребує збереження на сервері, оскільки вся необхідна інформація міститься всередині самого токена. А використання JWT у системах контролю доступу, використовується у таких випадках [30]:

- Аутентифікація користувача, тобто на етапі входу в систему користувач передає свої облікові дані серверу, після чого сервер формує токен JWT. У цей токен вшивається інформація про користувача, до прикладу, його унікальний ідентифікатор, яка потім використовується для підтвердження особи в усіх подальших запитах. Токен підписується, що унеможливлює його фальсифікацію без знання секретного ключа.

- Авторизація доступу, оскільки для перевірки прав доступу користувача до конкретних ресурсів чи функцій. У корисному навантаженні можуть бути вказані ролі або дозволи користувача. Коли клієнт надсилає запит до захищеного ресурсу, сервер перевіряє ці твердження і вирішує, дозволити чи відхилити запит.
- Передача і повторне використання токена, оскільки після того як токен виданий, клієнт зберігає його, зазвичай у локальному сховищі, і додає до кожного наступного HTTP-запиту, це у свою чергу дозволяє серверу перевіряти ідентичність користувача без повторної аутентифікації при кожному запиті.
- Перевірка підпису та терміну дії, оскільки сервер при отриманні токена перевіряє його підпис, використовуючи або секретний ключ, або відкритий ключ, якщо використовується алгоритм із відкритим ключем. Також перевіряється термін дії токена, якщо час, вказаний у полі experation, вийшов, токен вважається недійсним і запит відхиляється.
- Реалізація контролю доступу на основі ролей, оскільки JWT легко інтегрується з RBAC. У токені можна зафіксувати, до якої ролі належить користувач, а сервер, отримавши запит, приймає рішення на основі вказаної ролі. Гарним прикладом буде, що якщо в користувача роль admin або local admin, то вони можуть мати доступ до певних заздалегідь визначених адміністративних фукнцій або доступ до ресурсів з обмеженим доступом.
- Тонке налаштування прав доступу, тобто можна окрім загальних ролей, до JWT можна додавати більш специфічні твердження, що у свою чергу дозволяє будувати складну і детальну систему контролю доступу, що враховує контекст.
- Закінчення дії та відкликання токенів, оскільки JWT має механізм автоматичного завершення дії через параметр expiration. Це знижує ризик використання компрометованого токена. У разі необхідності примусового відкликання токена, до прикладу, при зміні пароля або видаленні користувача, можна реалізувати список відкликаних токенів або скоротити термін їхньої дії до мінімального.

Переваги використання JWT у моделях контролів доступу:

1. Масштабованість, він гарно підходить для розподілених систем і мікросервісів, де централізоване збереження сесій є складним.

2. Безсерверний підхід, немає потреби зберігати стан сесій на сервері, а підтвердження відбувається через токен.

3. Безпека, оскільки підпис токена запобігає його компроментації зловмисником.

4. Портативність, оскільки токен містить всю необхідну інформацію, та не потребує постійного звернення до бази даних.

Використання JWT дозволяє реалізувати надійну та гнучку систему керування доступом у серверних застосунках. Технологія підтримує як прості механізми аутентифікації, так і складні багаторівневі моделі авторизації, з урахуванням ролей, дозволів, термінів дії, а також специфічних вимог.

2.4. Атаки ін'єкційного типу

Ін'єкційні атаки, це один із типів кібератак, під час яких зловмисник вставляє шкідливий код у поля введення або інші джерела даних, які потім обробляються сервером або базою даних. Одним із найнебезпечніших і найпоширеніших типів таких атак є SQL-ін'єкція. Коли програма динамічно генерує SQL-запити на основі введених користувачем даних без належної перевірки або фільтрації цих даних. Це дозволяє зловмиснику вставляти фрагменти SQL-коду, які будуть інтерпретовані СУБД як частина запиту. Це відкриває можливість виконання довільних дій над базою даних, від перегляду конфіденційної інформації до її повного знищення [31].

Успішна ін'єкційна атака, проведена зловмисником може призвести до таких дій:

- Несанкціоноване читання даних з бази даних, включаючи облікові записи, особисту інформацію, паролі тощо;
- Зміна або видалення інформації, що може призвести до втрати важливих даних або зміни поведінки програми;

- Додавання нових даних до бази даних, наприклад створення фальшивих користувачів;
- Виконання додаткових команд на сервері, що в найгіршому випадку може призвести до компрометації всієї інфраструктури сервера;
- Обхід аутентифікації, коли зловмисник отримує доступ до облікових записів, не маючи доступу до облікових записів.

SQL-ін'єкції часто використовуються в автоматизованих атаках, оскільки вразливості цього типу досить легко виявити за допомогою сканерів або навіть вручну. Небезпека також полягає в тому, що інфекційні фрагменти системи можуть залишатися невиявленими на протязі тривалого часу, особливо в разі більш прихованих варіантів атак.

Існує кілька основних типів SQL-ін'єкцій, які відрізняються способом взаємодії з системами керування базами даних та отриманням зворотного зв'язку. Найбільш поширені типи ін’єкційних атак включають:

1. Класичні SQL-ін'єкції;
2. Сліпі ін'єкції;
3. Ін'єкції на основі помилок, які використовують системні повідомлення про помилки для вилучення внутрішньої інформації про структуру бази даних.

Вразливість до SQL-ін'єкцій часто є прямим результатом ігнорування практик безпечного програмування, зокрема, відсутності перевірки вхідних даних, використання динамічної побудови SQL та неправильного використання бібліотек баз даних. Ось чому надзвичайно важливо, щоб розробники, тестувальники та фахівці з кібербезпеки чітко розуміли природу таких атак і вживали відповідних заходів захисту [32].

Своєчасне та регулярне тестування безпеки серверних застосунків, включаючи аудит інформаційних систем та тестування на проникненнядозволяє виявити та усунути потенційні вразливості, перш ніж ними зможуть скористатися зловмисники.

2.4.1. Базовий вид SQL ін'єкції

Класична SQL-ін’єкція є одним із найстаріших і найвідоміших методів атак на серверні застосунки, які використовують реляційні бази даних, такі SQL-ін’єкції зазвичай трапляються в тих випадках, коли серверний застосунок динамічно формує SQL-запити шляхом простого з’єднання тексту, який містить вхідні параметри користувача. У такій ситуації відсутня перевірка або екранування введених даних, і шкідливий код може вільно потрапити в запит [33].

Метод проведення класичної SQL-ін’єкції зазвичай виглядає так:

1. Зловмисник ідентифікує вразливу форму або URL у веб-додатку, де дані користувача безпосередньо використовуються для створення SQL-запитів, до прикладу, форма входу або пошуку.

2. Потім зловмисник вводить спеціально підготовлений текст, який порушує початкову структуру запиту й додає шкідливий SQL-код, це у свою чергу змінює логіку умови перевірки так, щоб вона завжди повертала істину, що дозволяє обійти перевірку облікових даних.

3. У результаті попереднього кроку сервер виконує змінений SQL-запит, який вже не відповідає початковим вимогам розробника. Це може дозволити зловмиснику отримати доступ до даних, які йому не призначені, або навіть впливати на саму структуру бази даних.

Класичні SQL-ін’єкції небезпечні тим, що їх реалізація часто не потребує спеціальних технічних знань або складних інструментів. Багато з таких атак можна здійснити вручну, просто експериментуючи з введенням даних.

Загалом, класичні SQL-ін’єкції є основою для розуміння більш складних типів ін’єкційних атак. Їх важливо вивчити детально, оскільки вони залишаються актуальними навіть у сучасних системах, якщо не дотримуватись базових принципів безпечного програмування. У наступних розділах буде розглянуто інші варіанти SQL-ін’єкцій, які не дають прямих відповідей від системи, але все одно дозволяють отримати несанкціонований доступ до даних.

2.4.2. SQL ін'єкція без прямої відповіді сервера

Сліпа SQL-ін'єкція, один із типів ін'єкційної атаки, при якій сервер не надає прямих відповідей, які могли б підтвердити або спростувати успішність SQL-запиту. Однак навіть без відкритого доступу до результатів зловмисник може отримати конфіденційні дані, використовуючи непрямі ознаки, такі як зміни в поведінці програми або затримки відповідей. Цей тип ін'єкції поширений в системах, де обробка помилок прихована або ретельно фільтрується [34].

На відміну від класичного SQL-ін'єкції, де результат запиту може бути негайно відображений на сторінці, при сліпій ін'єкції зловмисник повинен витягувати інформацію поетапно, покладаючись лише на ознаки поведінки програм, та процес атаки зазвичай включає такі етапи:

- Першим кроком є визначення форми або URL-адреси, яка потенційно вразлива до SQL-ін'єкції. Це можна зробити вручну, тестуючи різні запити, або автоматично за допомогою спеціальних інструментів сканування.
- Далі зловмисник вводить спеціально сформовані SQL-оператори в поля введення або параметри запиту. Вони призначені для тестування поведінки програми у відповідь на певні логічні умови.
- Далі, якщо сервер не надає прямої інформації, зловмисник аналізує зміни в структурі сторінки, повідомленнях або часі, необхідному серверу для відповіді.
- Далі він аналізує запит, і якщо умова в SQL-запиті є істинною, програма поводиться як очікується, до прикладу, завантажується сторінка профілю, а якщо вона є хибною, структура або вміст сторінки змінюється. Це дозволяє поступово вгадати структуру бази даних.
- Також є й інший спосіб отримати інформацію, з використанням функцій, які затримують виконання запиту, до прикладу, sleep або waitfor delay). Якщо після виконання певного запиту відповідь сервера значно сповільнюється, це може означати, що умова в SQL-виразі була істинною.

- Потім після підтвердження вразливості зловмисник приступає до поступового вилучення інформації, далі можна визначити довжину імені таблиці, а потім по черзі перевірити кожен символ.

Хоча сліпі SQL-ін'єкції не дають миттєвих результатів, вони можуть бути надзвичайно ефективними, особливо проти додатків, які намагаються приховати свої помилки або не відображають результати запитів. Цей тип атак вимагає більше часу і зусиль, але в руках.

2.4.3. SQL-ін'єкція з використанням повідомлень про помилки СУБД

SQL-ін'єкція на основі помилок, також ще один метод ін’єкційної атаки, при якому зловмисник навмисно викликає помилки в запитах до бази даних, щоб отримати додаткову інформацію про її структуру або вміст. Така атака можлива лише в тому випадку, якщо система або серверний застосунок не приховує технічні деталі помилок, а відображає повідомлення безпосередньо користувачеві або зберігає їх у доступних для перегляду користувачем журналах [35].

Цей тип ін'єкції вважається одним з найпростіших у реалізації, якщо додаток не має належних механізмів обробки помилок, ось як зазвичай відбувається така атака:

- Спочатку зловмисник аналізує серверний застосунок, шукаючи місця, де можна вставити SQL-код, до прикладу, у форми авторизації, поля пошуку, фільтри, параметри URL. Можна використовувати як ручні тести, так і автоматизовані сканери вразливостей.
- Далі спеціально створений SQL-запит вставляється в уразливе поле з метою викликати помилку. Наприклад, зловмисник може ввести неправильну команду або спробувати звернутися до неіснуючого елемента, який входить до бази даних.
- У випадку якщо система не фільтрує помилки, вона повертає зловмиснику текст повідомлення про помилку, згенерований системою управління

базами даних. Це повідомлення може містити технічну інформацію, таку як імена таблиць, імена стовпців, типи даних, структури SQL і навіть фрагменти запитів.

- Після цього зібрані помилки аналізуються з метою виявлення структури бази даних, логіки взаємозв'язків між таблицями та наявності конфіденційних об'єктів. На основі цієї інформації зловмисник може розробити більш точні запити для подальшого вилучення даних.
- Далі, вже маючи знання про внутрішні деталі СУБД, зловмисник продовжує атаку, він може атакувати читання вмісту таблиць, зміну даних або підвищення прав доступу.

На відміну від сліпих SQL-ін'єкцій, де інформацію доводиться витягувати поступово, атаки на основі повідомлень про помилки можуть надати значний обсяг інформації одночасно, особливо якщо СУБД налаштована незахищено, тобто якщо вона не обмежує виведення помилок до клієнтської частини.

2.5. Методи протидії ін'єкційним атакам

Як я згадував у минулих розділах, SQL-ін'єкція є одним з найпоширеніших і найнебезпечніших видів атак на веб-додатки. Суть цієї атаки полягає в маніпулюванні SQL-запитами шляхом введення в них шкідливого коду. Наслідки успішної атаки можуть бути серйозними, від крадіжки конфіденційних даних до повного контролю над базою даних. Саме тому захист від SQL-ін'єкцій повинен бути реалізований на всіх рівнях обробки даних.

Для забезпечення ефективного захисту необхідний комплексний підхід, що включає як захист на рівні коду, так і організаційні та інфраструктурні заходи. Нижче я розглянув основні методи протидії SQL-ін'єкціям, які використовують розробники серверних застосунків:

- Одним з основних способів захисту від SQL-ін'єкцій є забезпечення контролю над усіма даними, що вводяться користувачем, тобто реалізація процесу перевірки відповідності значень, введених користувачем, оскільки треба зробити так, щоб поле, яке повинно приймати тільки числові значення, не дозволяло

введення букв або спеціальних символів. Наступним кроком треба реалізувати очищення даних, тобто процес очищення вхідних даних від небезпечних символів або конструкцій, які можуть бути інтерпретовані як частина коду баз даних. Одним із найкращих рішень є використання заздалегідь передбачених SQL-запитів за допомогою шаблонів із ячейками для значень, а самі значення передаються окремо. В результаті, навіть якщо вхідні дані містять SQL-код, він буде оброблятися як звичайні дані, а не як виконуваний код.

- Іншим важливим заходом безпеки є контроль прав облікового запису бази даних, через який працює веб-додаток. Застосування принципу мінімальних прав означає, що кожному компоненту надається лише мінімальний набір прав, необхідний для його функціонування. Наприклад, сторінка, яка лише відображає інформацію, не повинна мати права видаляти або змінювати дані. Такий підхід зменшує потенційні наслідки навіть у разі успішної атаки.
- Також можна реалізувати такі методи, як кодування, тобто, перетворення спеціальних символів на їх безпечні еквіваленти. Це допомагає гарантувати, що будь-які вхідні дані будуть інтерпретуватися як звичайний текст, а не як частина SQL-запиту, тобто зловмисник не зможе занести шкідливий запит.
- Також можна реалізувати використання збережених процедур, що у свою чергу дозволяє перемістити частину логіки взаємодії з базою даних на сам сервер. Таким чином, замість прямого формування SQL-запитів у коді, програма викликає заздалегідь визначені процедури з параметрами, що знову ж таки в свою чергу забезпечує додатковий рівень ізоляції між логікою програми та базою даних, зменшуючи ймовірність маніпулювання кодом.
- Іншим ефективним методом контролю вхідних даних є використання білих списків, тобто списку дозволених значень або символів, які може вводити користувач.
- Реалізація фільтрування дозволяє виявляти та видаляти небезпечні фрагменти вхідних даних, які можуть вказувати на спробу SQL-атаки, розробник може реалізувати, як ручне, так і автоматичне фільтрування, за допомогою спеціальних бібліотек або служб.

Також треба розуміти, що не тільки такі “великі” пласти захисту можуть принести захист від атак, не варто забувати і про більш “незначні” кроки захисту, серед них я б виділив:

- Постійне навчання розробників серверних частин сучасним типам загроз та методам їх запобігання.
- Використання перевірених фреймворків, що мають вбудовані механізми захисту від SQL-ін'єкцій.
- Регулярний аудит коду, який використовується з метою виявлення потенційних вразливостей.
- Також треба стовідсотків дотримуватися стабільного оновлення компонентів та системи, оскільки застарілі фреймворки, сервери або СУБД можуть містити відомі вразливості, які легко експлуатувати. Тому дуже важливо регулярно оновлювати програмне забезпечення, включаючи сторонні бібліотеки.
- Також можна застосувати більш комплексне рішення у вигляді брандмауеру веб-додатків, який буде діяти як фільтр між користувачем і сервером, аналізуючи весь вхідний трафік. Його можна налаштувати для виявлення конкретних ознак SQL-ін'єкцій, хоча WAF не замінює внутрішні механізми безпеки, він значно підвищує загальний рівень безпеки.
- Треба реалізовувати постійне тестування безпеки, як внутрішнє так і зовнішнє, оскільки постійне тестування безпеки, як автоматизоване за допомогою сканерів вразливостей, так і ручне тестування на проникнення, дуже сильно впливає на загальний ступінь безпеки. Проведення симуляцій атак дозволяє виявити вразливості до того, як ними зможе скористатися реальний зловмисник. Сканери допомагають виявити відомі шаблони SQLi, а компанії, які будуть проводити пентест, можуть використовувати нестандартні техніки та тестувати складні сценарії.

У підсумку цього, можна сказати, що ефективний захист від SQL-ін'єкцій вимагає поєднання перевірки вхідних даних, правильної обробки запитів, обмеження доступу та постійного моніторингу, жоден з цих підходів не є достатнім

сам по собі, але у поєднанні один з одним, вони створюють надійну систему захисту для застосунку.

2.6. Атака типу міжсайтовий скриптинг

Міжсайтовий скриптинг, це один з найпоширеніших і найнебезпечніших типів вразливостей серверних застосунків. Суть цієї атаки полягає у введенні шкідливого скрипту, зазвичай зловмисники пишуть на JavaScript-і, який потім виконується в браузері іншого користувача. Це відбувається через недостатню перевірку або очищення даних, наданих користувачем, а також через неправильну обробку цих даних у відповіді сервера [36].

Атаки міжсайтового скриптингу зазвичай націлені на елементи веб-інтерфейсу, де можна вводити дані, наприклад поля пошуку, і якщо введені дані не перевіряються належним чином і відображаються на сторінці як частина HTML-коду, зловмисник може вставити шкідливий скрипт, який буде виконаний у браузерах інших відвідувачів [37].

Залежно від способу ініціації шкідливого коду зловмисником, існує кілька основних типів атаки:

- Постійний міжсайтовий скриптинг, тобто атака, під час якої шкідливий скрипт зберігається на сервері і відображається іншим користувачам щоразу, коли вони завантажують уражену сторінку.
- Відображений міжсайтовий скриптинг, коли атака передбачає передачу шкідливого коду в запиті і повернення його в необробленому вигляді у відповіді веб-додатку. У цьому випадку скрипт не зберігається на сервері, а виконується відразу в браузері жертви, коли вона відкриває шкідливе посилання.

Типовий процес XSS-атаки може виглядати так, зловмисник вводить або передає спеціально створений код жертві, потім коли веб-додаток вставляє цей код в HTML без фільтрації або кодування, він виконується в браузері як звичайний скрипт. Це дозволяє зловмиснику взяти під контроль сеанс користувача, викрасти конфіденційні дані або змінити зовнішній вигляд сторінки.

Найбільш очевидні потенційні наслідки XSS-атак включають:

- Викрадення сеансу, тобто зловмисник може отримати доступ до файлів cookie або токенів авторизації та використовувати їх для входу в систему без авторизації від імені користувача.
- Також зміна вмісту сторінки, коли зловмисник може вносити зміни в веб-інтерфейс, наприклад замінювати текст або зображення, тим самим підриваючи довіру до ресурсу.
- Найнебезпечніші наслідки я б сказав, це крадіжка особистих даних, тобто шкідливий скрипт може перехоплювати та надсилати інформацію, введену користувачем, до прикладу, логіни, паролі, реквізити платіжних карток тощо, на сторонні сервери.

Успішна XSS-атака не тільки ставить під загрозу безпеку окремих користувачів, але й може мати серйозні наслідки для репутації організації, яка є власником веб-додатку. Саме тому виявлення та запобігання таким вразливостям є пріоритетом для розробників та фахівців з кібербезпеки.

2.7. Методи протидії міжсайтовому скриптингу

Захист від міжсайтового скриптингу, як уже було зазначено в попередньому розділі, XSS-атаки становлять серйозну загрозу для безпеки веб-додатків, дозволяючи зловмисникам виконувати шкідливі скрипти у браузерах користувачів. Далі я роздивлюсь декілька методів, які використовують розробники серверних застосунків, для того щоб забезпечити захист від таких типів атак [38].

Перший і найважливіший крок у запобіганні XSS-атакам, це перевірка всієї інформації, яку користувачі вводять у систему, розробники зазвичай реалізовують валідацію введення, тобто перевірку, чи дані відповідають очікуваним форматам, типам, до прикладу, рядок, число, електронна адреса і допустимим довжинам.

Реалізація політика безпеки контенту, розробники використовують сучасну технологію захисту, яка дозволяє вказувати, які джерела контенту дозволено використовувати на сайті. З допомогою CSP можна заборонити виконання

вбудованих скриптів або завантаження JavaScript з неперевірених джерел. Правильно налаштована політика може значно знизити ризик успішної XSS-атаки, навіть якщо в коді є інші вразливості.

Також розробники уділяють велику увагу встановленням атрибута HTTPOnly для файлів кукі, оскільки сесійні файли cookie часто є мішенню для XSS-атак, адже за їх допомогою зловмисник може отримати доступ до облікового запису жертви. Щоб захистити ці дані, слід використовувати прапорець HttpOnly і усе, але розробники часто нехтують цим.

Крім цього рекомендується робити фільтрацію на вході та виході, тобто реалізовувати механізми фільтрації на етапах як отримання даних, так і виведення їх у інтерфейс користувача. Фільтри повинні аналізувати дані на наявність небезпечних шаблонів та видаляти або блокувати їх. Рекомендовано використовувати як білі списки, так і чорні списки, тобто блокувати заздалегідь відомі шкідливі шаблони, щоб охопити ширший спектр загроз.

Також є більш регулярний метод, у якому розробники використовують фреймворки і бібліотеки, орієнтовані на безпеку, оскільки багато сучасних веб-фреймворків мають вбудовані засоби захисту від XSS. Наприклад, фреймворки типу Django або ASP.NET автоматично кодують вихідні дані в шаблонах, що унеможливлює виконання скриптів.

Завжди треба дотримуватись принципів безпечної розробки, оскільки не варто забувати, що безпека повинна бути інтегрованою частиною процесу розробки програмного забезпечення, тобто увесь штат розробників має бути ознайомлений з практиками безпечного кодування,необхідністю перевіряти усі вхідні дані, уникати використання функцій, які можуть виявитися потенційно небезпечними.

Дуже важливим кроком є регулярне тестування безпеки, для того щоб переконатися у відсутності вразливостей XSS, необхідно систематично перевіряти серверний застосунок на наявність таких загроз.

Хоча відповідальність за безпеку в першу чергу лежить на розробниках, користувачі також мають розуміти базові принципи безпечної поведінки в мережы. Варто інформувати їх про ризики, пов’язані з переходом за сумнівними

посиланнями або введенням особистих даних на ненадійних сайтах. Крім того, слід рекомендувати користувачам тримати свої браузери й ОС оновленими, оскільки в них регулярно виправляють відомі вразливості. Це дуже важливо, оскільки більшість успішних атак зловмисників відбуваються через людський фактор.

Застосування усіх наведених методів у комплексі формує багаторівневу систему захисту, яка дозволяє істотно знизити ймовірність успішної XSS-атаки. Регулярний моніторинг, оновлення механізмів безпеки та постійне вдосконалення підходів до розробки допоможуть підтримувати належний рівень захисту серверних застосунків у сучасному середовищі кіберзагроз.

Висновки за розділом 2

У цьому розділі наведено комплексний аналіз основних загроз безпеці сучасних серверних застосунків та описано ефективні методи їх захисту. Особливу увагу було приділено основним типам атак, які становлять найбільшу небезпеку в реальних веб-системах, а саме атака порушення контролю доступу, ін'єкційні атаки та міжсайтовий скриптинг.

Порушення контролю доступу розглядалися в контексті різних моделей управління правами користувачів, включаючи ролі, атрибути та дискреційні механізми. Це дозволило краще зрозуміти підходи до забезпечення гнучкої та надійної системи авторизації.

Ін'єкційні атаки були детально класифіковані за типами, від класичних SQL-ін'єкцій до сліпих. У відповідь було запропоновано комплекс захисних заходів, включаючи параметризовані запити, фільтрацію та перевірку вхідних даних.

Також було розглянуто одну з найпоширеніших вразливостей XSS, яка полягає в ін'єкції шкідливого JavaScript-коду, а також було розглянуто, як використання таких інструментів, як політика безпеки контенту, фільтрація даних та обмеження доступу до cookie-файлів, може значно зменшити ризик атак.

Підсумовуючи, можна зробити висновок, що забезпечення безпеки серверних додатків вимагає багаторівневого підходу. Виявлення та усунення вразливостей на

етапі розробки, впровадження відповідних механізмів контролю, а також постійне тестування та оновлення системи безпеки є критично важливими елементами ефективного захисту. Виходячи з цього, у результаті виконання цього розділу було підтверджено необхідність побудови багаторівневого підходу до захисту серверного застосунку, який буде забезпечувати надійний захист даних на всіх рівнях архітектури застосунку, це у свою чергу забезпечить надійний захист від несанкціонованої модифікації даних, навіть під час часткової компроментації системи, оскільки використовуючи комплесні засоби захисту, ми не розраховуємо на однин конкретний засіб захисту, а розраховуємо на систему в цілому.

РОЗДІЛ 3
ЗАБЕЗПЕЧЕННЯ КОНФІДЕНЦІЙНОСТІ ТА ЦІЛІСНОСТІ ДАНИХ У СЕРВЕРНИХ ЗАСТОСУНКАХ

3.1. Проблематика забезпечення цілісності та конфіденційності даних у серверних застосунках

Питання цілісності та конфіденційності даних у серверних додатках є одними з ключових питань спеціалістів з інформаційної безпеки, особливо з огляду на сучасні тенденції у сфері цифрових технологій, хмарних обчислень та глобальної цифровізації. Оскільки сучасні серверні додатки, які обробляють великі обсяги запитів і конфіденційну інформацію користувачів, стикаються з дедалі складнішими проблемами, пов'язаними з ризиками витоку, фальсифікації або несанкціонованого доступу до даних. У цьому контексті забезпечення цілісності та конфіденційності інформації є не просто вимогою, а необхідною умовою надійного функціонування будь-якої інформаційної системи, яка входить до складу повноцінного серверного застосунку.

Цілісність даних у серверному середовищі означає, що будь-яка інформація, яка зберігається, передається або обробляється, не повинна бути змінена неавторизованими особами або випадково пошкоджена під час використання. Порушення цілісності дуже часто залишаються не поміченими відразу, що робить проблему ще більш критичною, оскільки, пошкоджені або замінені дані можуть продовжувати використовуватися в системі, що призводить до неправильних результатів, порушення багатьох бізнес-процесів або навіть втрати довіри користувачів і партнерів.

Конфіденційність, у свою чергу, означає, що доступ до конфіденційної інформації повинен бути наданий тільки уповноваженим особам або службам. Порушення цього принципу призводить до витоку особистих, фінансових, комерційних або інших конфіденційних даних, що може мати як фінансові, так і

репутаційні наслідки для організації. В останні роки, з ростом кількості кіберінцидентів і зловмисних атак, питання захисту приватної інформації стало предметом пильної уваги усіх регуляторних органів, зокрема в таких нормативних документах, як GDPR, HIPAA, ISO/IEC 27001[39], PCI DSS [40].

Основними причинами порушення цілісності та конфіденційності в серверних додатках є:

Недостатній контроль доступу, частіше за все, це неправильне впровадження політик аутентифікації та авторизації відкриває зловмисникам можливості для доступу до даних або їх зміни без дозволу.

Вразливості в програмному коді, тобто, неправильна обробка вхідних даних, відсутність перевірки типів даних, використання застарілих бібліотек або залежностей може призвести до таких атак, як SQL-ін'єкція або міжсайтовий скриптинг, які безпосередньо загрожують цілісності та конфіденційності даних.

Неаутентифіковані джерела даних, тобто, без механізмів перевірки походження або цілісності вхідної інформації серверні додатки можуть приймати та обробляти підроблені або скомпрометовані дані.

Ненадійні канали передачі, оскільки передача даних у вигляді звичайного тексту або з використанням застарілих протоколів, або протоколів без шифрування, створює ризик перехоплення інформації під час її маршрутизації між клієнтом і сервером.

Окремою проблемою є те, що в багаторівневих розподілених архітектурах, типових для сучасних веб та серверних додатків, інформація постійно передається між різними вузлами, інтерфейсами та сторонніми службами. Це ускладнює забезпечення її цілісності та унеможливлює централізоване управління без спеціальних інструментів. У таких умовах необхідно впроваджувати системи, які не тільки зберігають дані, але й фіксують їх стан та зміни, а також контролюють доступ до них протягом усього процесу обробки. Розробники проектуючи сучасні заходи безпеки, що будуть використовуватися для збереження цілісності та конфіденційності, часто включають такі пункти:

Криптографічні алгоритми шифрування, що забезпечують конфіденційність шляхом перетворення даних у форму, яка не може бути прочитана сторонніми особами.

Хеш-функції [41] та цифрові підписи, що дозволяють перевіряти аутентичність та цілісність інформації без необхідності зберігати її в оригінальній формі.

Механізми багатофакторної аутентифікації [42] та контролю доступу, що зменшують ризик несанкціонованого доступу до критичних компонентів системи.

Реєстрація подій та аудит активності користувачів, що забезпечують моніторинг змін у режимі реального часу, швидке реагування на інциденти та юридичну відповідальність за порушення.

Таким чином, питання забезпечення цілісності та конфіденційності даних у серверних додатках не є абстрактним або суто технічним. Воно має прямий вплив на стабільність цифрових систем, довіру користувачів та дотримання законодавчих вимог. На даному етапі розвитку IT-інфраструктури існує нагальна потреба у впровадженні інноваційних підходів до захисту даних, серед яких особливу увагу привертають технології блокчейн, децентралізовані ідентифікатори та протоколи конфіденційного зв'язку нового покоління. Вони стають основою для побудови безпечних серверних додатків, стійких до сучасних кіберзагроз.

3.2. Основи криптографічного захисту даних: симетричні та асиметричні методи.

Захист даних у серверних додатках неможливий без впровадження криптографічних методів, що забезпечують конфіденційність та цілісність інформації на всіх етапах її обробки, починаючи від введення на стороні клієнта до зберігання або обробки на сервері. Існує два основних типи криптографічних підходів, є симетричне та асиметричне шифрування [43], кожен з яких відіграє свою роль в архітектурі безпеки та має свої переваги та обмеження.

Симетричне шифрування [44] передбачає використання одного і того ж ключа для шифрування та дешифрування інформації. Цей підхід є надзвичайно ефективним з точки зору продуктивності, що робить його придатним для швидкої обробки великих обсягів даних. Одним з найнадійніших і найпоширеніших алгоритмів симетричного шифрування є AES [45] (Advanced Encryption Standard), особливо в режимі 256-CBC [46] (Cipher Block Chaining). Цей реалізує блокове шифрування з довжиною блоку 128 бітів за допомогою 256-бітного ключа, у режимі CBC кожен блок відкритого тексту перед шифруванням піддається операціям з попередньо зашифрованим блоком, що забезпечує більш високу криптографічну міцність і запобігає утворенню повторюваних шаблонів у вихідному тексті навіть при однакових вхідних даних.

Цей алгоритм шифрування, особливо важливий в контексті шифрування клієнта, тобто виконання криптографічних операцій перед передачею даних на сервер. Це гарантує, що навіть у разі перехоплення трафіку або компрометації сервера, сторонні особи не зможуть отримати доступ до змісту переданої інформації без ключа [47].

Механізм шифрування клієнта за допомогою працює наступним чином, тобто на стороні клієнта під час підготовки запиту до серверної програми дані, такі як метаінформація або особиста інформація користувача, перетворюються в зашифрований формат. Для цього генерується випадковий 256-бітний ключ і 128-бітний вектор ініціалізації. Вхідний текст розділяється на 128-бітні блоки, кожен з яких по черзі обробляється за принципом CBC: перший блок шифрується за допомогою четвертого, другий, за допомогою попередньо зашифрованого блоку і так далі. В результаті отримується набір зашифрованих блоків, які об'єднуються в одне повідомлення, придатне для безпечної передачі.

Однією з переваг симетричного шифрування є його швидкість і відносно низьке навантаження на систему, що особливо важливо в додатках з високим навантаженням, де обробка даних повинна бути не тільки безпечною, але й швидкою. Однак головним викликом є проблема розподілу ключів, як клієнт і

сервер повинні безпечно домовитися про спільний ключ, не піддаючи його ризику перехоплення.

Для вирішення цієї проблеми використовується асиметричне шифрування [48], яке базується на парі пов'язаних ключів: відкритому ключі, тобто такий який можна вільно розповсюджувати і закритому ключі,який зберігається в таємниці. У типовому сценарії захисту даних клієнта клієнт може зашифрувати симетричний ключ AES за допомогою відкритого ключа сервера, після чого сервер може розшифрувати його за допомогою свого закритого ключа. Цей підхід поєднує переваги обох методів: швидкість симетричного шифрування та безпеку обміну ключами завдяки асиметричному протоколу.

Найпоширенішим алгоритмом асиметричного шифрування є RSA, який дозволяє безпечно передавати ключ AES, згенерований на стороні клієнта. Наприклад, під час першого встановлення з'єднання клієнт шифрує симетричний ключ за допомогою відкритого ключа RSA сервера і надсилає його в запиті разом із зашифрованими даними. Таким чином, після дешифрування сервер отримує доступ до ключа і може обробляти дані без ризику витоку інформації.

Ця модель, яка поєднує AES-256-CBC для базового шифрування даних і RSA для безпечного обміну ключами, є основою багатьох протоколів конфіденційного зв'язку, включаючи HTTPS, а також деякі реалізації взаємодії клієнт-сервер у середовищах криптовалют і Web3 [49,50]. Її надійність перевірена десятиліттями практичного використання і відповідає сучасним вимогам криптографічної безпеки.

Підсумовуючи всі ці пункти, можна сказати, що поєднання симетричних і асиметричних методів шифрування забезпечує гнучкий, ефективний і безпечний механізм захисту даних у серверних застосунків. Цей підхід є основою системи, яку я буду розробляти в цій роботі, яка шифрує метадані на стороні клієнта перед їх передачею до смарт-контракту, що унеможливлює доступ до них навіть у випадку, якщо потенційний зловмисник має повний контроль над серверною частиною.

3.3. Принципи роботи смарт-контрактів у блокчейн середовищах

Смарт-контракти [51,52] є одним з ключових інструментів сучасних блокчейн-систем, що дозволяють реалізувати автоматизовану логіку взаємодії між користувачами без необхідності залучення довірених посередників. Робота смарт-контрактів базується на ідеї детермінованого виконання умов контракту, вбудованих у вигляді програмного коду, що зберігається в блокчейні та виконується децентралізовано. Це відкриває нові можливості для захисту даних, особливо в тих випадках, коли важливо не тільки зберігати інформацію, але й гарантувати її незмінність, прозорість, цілісність та доступність у майбутньому.

Смарт-контракт, по суті є програмою, яка розгортається в розподіленому реєстрі та виконується віртуальною машиною, такою як Ethereum Virtual Machine [53] (EVM). Його код написаний мовою програмування, придатною для компіляції в байт-код, який підтримується блокчейном. Найпоширенішою мовою програмування для смарт-контрактів є Solidity, яка спеціально розроблена для Ethereum. Після створення контракт зберігається в блокчейні як незмінний фрагмент даних, і кожна взаємодія з ним, наприклад записувати у ньому інформацію, чи проводити будь-які транзакції, але він вимагає транзакції типу підтверження, яка реєструється мережею, для виконання запиту.

Важливо розуміти, що смарт-контракт не має доступу до зовнішнього світу в звичайному розумінні. Він функціонує тільки з даними, які були передані йому під час виклику функцій або які вже зберігаються в його стані. Для взаємодії із зовнішніми джерелами використовуються спеціалізовані сервіси, які надають зовнішню інформацію смарт-контрактам, однак у контексті забезпечення конфіденційності та цілісності, як у цій роботі, такі сервіси не будуть використовуватися, я буду покладатися виключно на заздалегідь підготовлені дані, які передаються від клієнта в зашифрованому вигляді.

Однією з найважливіших властивостей смарт-контрактів є незмінність. Після публікації контракту в блокчейні його код не може бути змінений. Це забезпечує надзвичайно високий рівень довіри до його функціональності, оскільки жодна

сторона не може модифікувати контракт або змінити правила. Водночас змінні, такі як ключі, хеші, зашифровані метадані можуть зберігатися в смарт-контракті та оновлюватися за допомогою викликів функцій контракту відповідно до вбудованої логіки.

Смарт-контракт виконується всіма учасниками мережі, що забезпечує детермінованість і перевірюваність результатів. Якщо одна і та ж функція викликається з однаковими параметрами, всі вузли блокчейну повинні отримати однаковий результат. Це забезпечує відтворюваність і прозорість логіки, вбудованої в код.

Серед характерних властивостей смарт-контрактів, що мають безпосереднє відношення до захисту даних, можна виділити наступні пункти:

- Прозорість потрібна для того, щоб операції у контракті були відкриті для перегляду всіма учасниками мережі.
- Цілісність, оскільки жоден учасник не може змінити дані або логіку контракту без відповідної транзакції.
- Автоматизація, оскільки контракт може самостійно виконувати встановлені правила без необхідності втручання третіх осіб у цей захищенний процес.
- Незворотність, оскільки усі дії, які виконуються контрактом, є остаточними, тобто неможливо скасувати транзакцію або відновити дані чи повернути помилково відправлені кошти.

З огляду на вищезазначене, смарт-контракти ідеально підходять для завдань, пов'язаних із безпечним зберіганням зашифрованих даних. У рамках цієї роботи вони використовуються для інтеграції зашифрованих метаданих, які клієнт шифрує на своєму боці і передає до функції контракту. У відповідь смарт-контракт записує ці дані у своє сховище, зберігаючи їх як незмінні об'єкти. Якщо потрібна перевірка або аудит, будь-яка сторона може отримати доступ до запису, не маючи можливості змінити його зміст.

Таким чином, у цьому випадку смарт-контракт виступає незалежним, відкритим і незмінним посередником, гарантуючи, що дані не були

сфальсифіковані, змінені або знищені. У поєднанні з криптографічними методами та децентралізованою інфраструктурою блокчейну це дозволяє створити архітектуру, стійку до більшості відомих векторів атак на цілісність і конфіденційність у серверних застосунках.

3.4. Технологія децентралізованих ідентифікаторів (DID) та її значення для безпеки.

У сучасних цифрових системах, особливо тих, що працюють у децентралізованому середовищі, ідентифікація користувачів є одним з ключових аспектів безпеки. Традиційні моделі ідентифікації базуються на централізованих реєстрах, серверах авторизації та надійних третіх сторонах, що створює низку ризиків, від компрометації бази даних до витоку персональних даних, а також вводить додаткові точки відмови. У відповідь на ці виклики було розроблено концепцію децентралізованих ідентифікаторів [54,55] (DID), тобто новий підхід до цифрової ідентифікації, який дозволяє користувачам управляти своїми обліковими записами без залежності від будь-якого центрального органу.

DID, це унікальний глобальний ідентифікатор, який не потребує централізованого реєстру або постачальника для функціонування. Він зберігається в децентралізованому реєстрі, в нашому випадку на блокчейні, і пов'язаний з суб'єктом, тобто особою, організацією, пристроєм або навіть додатком. DID вказує на відповідний документ DID, структурований документ, що містить метаінформацію про власника: відкриті криптографічні ключі, методи аутентифікації, точки доступу до послуг.

Ключова відмінність DID від традиційних ідентифікаторів полягає у відмові від централізованого управління. Якщо, наприклад, адреса електронної пошти або ім'я користувача надається організацією і може бути скасована або змінена адміністративно, DID створюється безпосередньо користувачем або додатком і записується в блокчейні, що робить його незалежним, стійким до змін і довговічним [56].

З точки зору безпеки, використання DID має кілька стратегічних переваг, серед яких я б виділив:

- Контроль над ідентифікацією, оскільки користувачі самі володіють своїми DID, управляють пов'язаними з ними ключами та можуть змінювати методи аутентифікації без залучення третіх осіб. Це означає, що жодна централізована служба не може змінити, заблокувати або зламати ідентифікатор.
- Захист персональних даних, оскільки DID дозволяє використовувати модель мінімального розкриття інформації, що означає, що користувачі можуть вибірково ділитися лише тією інформацією, яка необхідна в конкретному контексті, не розкриваючи всю структуру свого DID-документа. Це особливо корисно в разі конфіденційних транзакцій.

У системах, де конфіденційність та цілісність мають критичне значення, таких як серверні додатки, що обробляють приватні або конфіденційні дані, DID може слугувати основою довіри, навколо якої будуються інші елементи безпеки. Використання DID у поєднанні з криптографічним шифруванням та смарт-контрактами дозволяє створити безпечну, стійку до атак систему, в якій ідентичність перевіряється без розкриття персональних даних, і доступ до інформації має тільки власник.

У типовому випадку, коли користувач хоче зашифрувати та зберегти метадані в смарт-контракті, спочатку створюється DID і пов'язується з відкритим ключем. Відповідно, тільки власник приватного ключа може розшифрувати або перевірити дані. Таким чином, DID діє як безпечний токен доступу, який не вимагає перевірки третьою стороною і не має вразливостей, типових для класичних методів авторизації.

Отже, DID відкриває можливість переходу до моделі самодостатньої ідентичності, яка стає надзвичайно важливою з швидким розвитком децентралізованих серверних застосунків. Його впровадження в серверні системи не тільки захищає інформацію від несанкціонованого доступу, але й забезпечує її цілісність та аутентичність, що є головним завданням у контексті моєї роботи.

3.5. Переваги використання блокчейну для забезпечення цілісності даних у серверних застосунках

Забезпечення цілісності даних є одним із основних проблем безпеки сучасних інформаційних систем, особливо в серверних застосунках, що обробляють великі обсяги інформації з обмеженим доступом. Дуже часто порушення цілісності, включаючи навмисну або випадкову модифікацію даних, може мати серйозні наслідки, від втрати надійності інформації до компрометації всієї системи. Через це існує потреба в технологічному рішенні, яке гарантувало б незмінність даних навіть у потенційно ненадійному середовищі. Одним з найперспективніших підходів є використання блокчейну, тобто розподіленої, незмінної бази даних, яка зберігає інформацію у вигляді ланцюжка пов'язаних між собою блоків.[57,58]

Унікальна особливість блокчейну полягає в тому, що кожен новий блок містить хеш попереднього, утворюючи криптографічно захищений ланцюг. Це означає, що будь-яка спроба змінити навіть один байт інформації в одному з блоків автоматично порушить хеш-ланцюг і буде миттєво виявлена. Такий підхід не тільки перевіряє дані, але й захищає їх від фальсифікації на рівні усієї інфраструктури.

Я б привів такі переваги використання блокчейну в контексті забезпечення цілісності даних:

Незмінність інформації, оскільи після того, як блок увійшов до ланцюжка і був підтверджений мережею, його вміст не може бути змінений або видалений. Це створює надійну систему протидії будь яким змінам, яку неможливо підробити.

Розподіленість і відсутність єдиної точки відмови, оскільки дані в блокчейні зберігаються на великій кількості вузлів, тому навіть у разі втрати або пошкодження окремих серверів цілісність і доступність записів не порушується.

Криптографічна надійність, оскільки кожна транзакція або запис підписується криптографічними ключами, а цілісність блоків гарантується хеш-функціями. Це в свою чергу усуває необхідність у централізованому органі, що забезпечує довіру.

Можливість аудиту та відстеження, це дуже полегшує завдання аудиторів, оскільки всі зміни є відкритими та незмінними, це в свою чергу дозволяє легко відстежити, хто, коли та яку інформацію створив або відредагував, що є надзвичайно важливим для безпеки та дотримання нормативних вимог.

Автоматичне забезпечення дотримання логіки доступу за допомогою смарт-контрактів, якщо реалізувати систему у поєднанні зі смарт-контрактами блокчейн може автоматично перевіряти умови зберігання, читання або оновлення даних, усуваючи людський фактор.

У контексті моєї роботи блокчейн служить довгостроковим, незмінним і криптографічно захищеним сховищем метаданих про зашифровану інформацію. Шифрування відбувається на стороні клієнта, після чого хеш або інші допоміжні атрибути записуються в блокчейн за допомогою смарт-контракту. Це гарантує, що зашифровані дані не були змінені або замінені після публікації, і що користувач завжди може перевірити аутентичність інформації. У такій архітектурі користувач має повний контроль над дешифруванням, але авжеж лише за допомогою приватного ключа можна буде перевірити аутентичність та отримати доступ до вмісту. Це створює високий рівень довіри до цілісності даних, навіть коли сама система працює в умовах Інтернету.

Коротко кажучи, блокчейн, це не просто засіб для фінансових або інших видів транзакцій, а повноцінна технологія, здатна забезпечити стабільну та прозору інфраструктуру для захисту даних.

3.6. Порівняльний аналіз існуючих підходів до захисту метаданих у серверних застосунках

У процесі розробки та впровадження блокчейн-орієнтованих архітектур захисту вкрай важливим є проведення моделювання загроз. Це дозволяє заздалегідь виявити потенційні вразливості, оцінити ризики для конфіденційності, цілісності й доступності даних, а також сформувати та реалізувати дієві стратегії захисту. Особливість блокчейн-систем полягає у тому, що вони мають високий рівень

прозорості та незмінності, проте це не робить їх автоматично захищеними від усіх типів загроз. Моделювання загроз у цьому випадку повинне враховувати, загальні принципи кібербезпеки, авжеж так само як і специфіку технологій розподіленого реєстру.

Одним з основних підходів до моделювання загроз є використання методології STRIDE [59]. Розроблена експертами Microsoft, ця модель є однією з найвідоміших і найпоширеніших платформ для моделювання загроз інформаційним системам безпеки. Її основною метою є виявлення потенційних векторів атак шляхом класифікації загроз за категоріями, що охоплюють найпоширеніші порушення безпеки. STRIDE, що походить від шести ключових категорій: Spoofing, тобто підміна ідентичності, Tampering, тобто,фальсифікація даних, Repudiation, тобто,заперечення дій, Information Disclosure, тобто, розкриття інформації, Denial of Service, тобто відмова в обслуговуванні, Elevation of Privilege, тобто підвищення привілеїв. У класичних архітектурах клієнт-сервер кожна з цих загроз має добре досліджені сценарії реалізації. Однак у контексті систем, орієнтованих на блокчейн, ці категорії набувають нових, специфічних форм через децентралізований характер, особливості зберігання даних, незворотність транзакцій та публічний характер блокчейну.

Спуфінг в екосистемі блокчейну проявляється у вигляді підміни ідентичності користувача, що, зокрема, може бути реалізовано шляхом компрометації приватного ключа. Оскільки учасники мережі ідентифікуються не за допомогою традиційних логінів і паролів, а шляхом перевірки цифрового підпису транзакції, втрата або крадіжка приватного ключа еквівалентна повній втраті контролю над обліковим записом, тобто ішними словами це означає можливість несанкціонованого переказу активів, зміни логіки смарт-контракту (якщо це був обліковий запис володаря контракту). Загроза посилюється відсутністю централізованого механізму відновлення доступу: ніхто не може скасувати або змінити ключ.

Фальсифікація даних у традиційному розумінні майже неможлива в блокчейні завдяки його незмінності та криптографічним гарантіям цілісності

блоків. Однак ця загроза може бути реалізована шляхом маніпулювання вхідними даними смарт-контракту або формування транзакцій із зловмисним вмістом, які, хоча і є законними за своєю структурою, викликають небажану поведінку системи, наприклад скрита функція на переказ усіх коштів, чи скрита функція на підписання якоїсь дії від вашого імені.

Відмова від авторства дій технічно складна в блокчейні, оскільки кожна транзакція підписується приватним ключем. Однак за умови недостатньої прозорості логіки смарт-контракту або при використанні проміжних вузлів, наприклад передача інформації через посереднє API, може бути складно довести намір автора транзакції або відокремити дії користувача від автоматизованого процесу.

Розкриття інформації є особливо чутливим питанням для публічних блокчейнів, де всі транзакції та дані смарт-контрактів відкриті для будь-кого. Навіть якщо дані формально зашифровані або представлені у хешованій формі, зловмисник може проаналізувати метадані, які знаходяться у відкритому доступі, визначити частоту взаємодій між адресами, обсяги переказів або шаблони транзакцій, що в свою чергу дозволяє зробити висновки про ділову діяльність, відносини між сторонами і навіть встановити реальні особи за допомогою порівняння з відкритими джерелами.

Відмова в обслуговуванні [60] (DoS) в децентралізованих системах реалізується шляхом навмисного перевантаження мережі транзакціями, створення атаки на окремі вузли інфраструктури. Смарт-контракти з надто складною або неоптимізованою логікою також можуть піддаватися DoS-атакам. Наприклад, один користувач може регулярно надсилати обчислювально дорогі запити до контракту, обмежуючи доступність ресурсу для інших.

Підвищення привілеїв, в блокчейні зазвичай пов'язане чисто з помилками в системі управління ролями або правами доступу в смарт-контрактах, тобто якщо автор контракту неправильно реалізував перевірку прав доступу, зловмисник може отримати адміністративний контроль над функціями, які не повинні бути доступними, такими як видалення або зміна критичних параметрів, передача

токенів. Реалізація таких видів вразливостей найчастіше виникає через людський фактор.

Таким чином, методологія STRIDE дозволяє не тільки систематизувати потенційні загрози в архітектурах блокчейну, але й адаптувати існуючі стратегії безпеки серверних застосунків до особливостей взаємодії з блокчейн середовищем.

У результаті моделювання загроз формується карта ризиків, яка дозволяє пріоритезувати заходи захисту, оцінити економічну доцільність їх впровадження та виявити критичні точки архітектури. Важливо підкреслити, що моделювання загроз не є одноразовою процедурою, воно повинно постійно проходити перерозгляд відповідно до змін у функціональності системи, появи нових векторів атак або зміни в контексті використання технології. Успішне моделювання є запорукою побудови надійної, стійкої до зловмисних дій архітектури захисту, яка забезпечує довіру до обробки, зберігання та перевірки даних.

Висновки за розділом 3

У третьому розділі було розглянуто теоретичні та практичні аспекти забезпечення конфіденційності та цілісності даних у сучасних серверних додатках з урахуванням як класичних методів криптографії, так і інноваційних рішень на основі технології блокчейн. Було зрозуміло, що забезпечення цілісності та захисту даних у розподілених та відкритих середовищах вимагає не тільки звичних протоколів шифрування, а й більш гнучких підходів, здатних протистояти сучасним кіберзагрозам.

Особлива увага приділена симетричним і асиметричним методам шифрування, які є основою криптографічного захисту. Механізм шифрування даних на стороні клієнта за допомогою алгоритму AES-256-CBC розглянуто як ефективне рішення для збереження конфіденційності під час передачі та зберігання даних. Цей підхід гарантує, що навіть у разі перехоплення інформації вона залишається захищеною.

Особливий акцент було зроблено на технологіях блокчейн та смарт-контрактів, які відкривають нові можливості для створення децентралізованих архітектур із вбудованими механізмами безпеки. Запровадження децентралізованих ідентифікаторів (DID) описано як перспективний напрямок для підвищення довіри між учасниками цифрових взаємодій та зменшення залежності від централізованих систем аутентифікації.

Загалом, цей розділ демонструє, що поєднання класичних криптографічних інструментів з новітніми технологіями дозволяє побудувати ефективну систему захисту даних у серверних застосунках, яка здатна не тільки протистояти сучасним загрозам, але й адаптуватися до зростаючих вимог безпеки, прозорості та автономності цифрових систем.

РОЗДІЛ 4
РЕАЛІЗАЦІЯ УДОСКОНАЛЕНОЇ СИСТЕМИ ЗАБЕЗПЕЧЕННЯ ЦІЛІСНОСТІ І КОНФІДЕНЦІЙНОСТІ У СЕРВЕРНИХ ЗАСТОСУНКАХ

4.1. Мета та завдання практичної частини

Практична частина моєї кваліфікаційної роботи спрямована на розробку удосконаленої системи, яка забезпечує безпечне зберігання конфіденційних даних у розподіленому середовищі з використанням технологій блокчейну, зокрема мережі Ethereum, смарт-контрактів та децентралізованих ідентифікаторів. Основна ідея полягає в інтеграції методів криптографічного захисту з можливостями платформи блокчейну для створення моделі, яка дозволяє зберігати зашифровану інформацію таким чином, щоб забезпечити як її цілісність, так і конфіденційність, з відкритим доступом до даних. Очікується, що результатом стане рівень захисту, при якому зберігання даних у публічному реєстрі не створює ризику несанкціонованого доступу або компрометації, а дані можна перевірити без їх розкриття.

Це буде досягатися шляхом розробки архітектури, в якій конфіденційна інформація не зберігається у вигляді звичайного тексту в блокчейні, а шифрується безпосередньо на стороні клієнта перед записом. Потім ці зашифровані дані інтегруються в смарт-контракт, який забезпечує їх децентралізоване зберігання.

Для досягнення мети, я поставив низку конкретних завдань щодо створення комплексної системи забезпечення безпеки серверного застосунку:

1. Розробка смарт-контракту, відповідального за створення, реєстрацію, зберігання та перевірку децентралізованих ідентифікаторів, які слугують основою для унікальної ідентифікації користувачів у системі без залежності від централізованих реєстрів.

2. Впровадження механізму шифрування даних на стороні клієнта з використанням алгоритму AES-256 у режимі CBC, що забезпечує високий рівень

криптографічної стійкості та запобігає розкриттю вмісту навіть у разі доступу до шифрованого тексту.

3. Інтеграція зашифрованих даних у смарт-контракт, включаючи передачу цих даних до мережі блокчейну Ethereum через відповідний інтерфейс.

4. Забезпечення конфіденційності та контролю доступу шляхом створення приватних ключів, які гарантують, що тільки аутентифіковані користувачі з відповідними ключами можуть розшифрувати та отримати доступ до вмісту зашифрованих даних.

5. Тестування функціональності удосконаленої системи, включаючи перевірку основних модулів, логіки шифрування/дешифрування, зберігання даних у блокчейні та аналіз ефективності запропонованого підходу з точки зору безпеки, продуктивності та сумісності з сучасними серверними застосунками.

Загалом, виконання цих завдань продемонструє доцільність і практичність створення системи безпеки нового покоління, в якій надійність буде гарантуватися не тільки безпечним середовищем виконання, але й на рівні самої логіки обробки та зберігання інформації за допомогою децентралізованих підходів.

4.2. Вибір інструментів та технологій для досягнення мети

Розробка системи для безпечного зберігання зашифрованих метаданих у блокчейн-середовищі вимагала ретельного підбору технологічного стеку, який би забезпечував не тільки функціональність, але й високу надійність, криптографічну безпеку та простоту інтеграції. Основною вимогою було використання перевірених, сучасних інструментів, які активно використовуються у розробці децентралізованих додатків.

В першу чергу, в якості блокчейн-платформи було обрано Ethereum та мережу Sepolia, оскільки вона є найбільш зрілою, поширеною та гнучкою для створення смарт-контрактів та децентралізованих сервісів. Ethereum підтримує стандартизовану інфраструктуру для розгортання контрактів, має активну

екосистему розробників та забезпечує високий рівень сумісності з інструментами для тестування, налагодження та інтеграції.

Для написання смарт-контрактів використовувалася мова Solidity, яка є основною мовою розробки в екосистемі Ethereum. Solidity дозволяє описувати логіку взаємодії, верифікації та зберігання даних в блокчейні, а також має широку підтримку бібліотек і фреймворків.

Ключовим компонентом для забезпечення конфіденційності є симетричне шифрування з використанням алгоритму AES-256 у режимі Cipher Block Chaining. Цей алгоритм був обраний завдяки своїй криптографічній міцності, високій продуктивності при реалізації на стороні клієнта та здатності ефективно обробляти навіть великі обсяги даних. Шифрування відбувається перед відправкою даних у блокчейн, що гарантує, що жодна третя сторона, включаючи вузли мережі Ethereum, не матиме доступу до відкритого вмісту. Для реалізації цього шифрування використовувалася вбудована криптобібліотека в середовищі Node.js, яка дозволяє легко реалізувати як процес шифрування, так і дешифрування. Її функціональність забезпечує підтримку сучасних криптографічних стандартів та інтеграцію з іншими частинами додатка без необхідності використання сторонніх залежностей. Бібліотека ethers.js використовується для взаємодії з мережею Ethereum, надаючи зручний API для підключення до смарт-контрактів, виконання транзакцій, зчитування даних з блокчейну, підписання повідомлень та управління ключами. Вона має високу продуктивність, активну підтримку та добре інтегрується з іншими інструментами, що робить її ідеальним вибором для клієнтської частини додатка.

Платформа Hardhat відіграла значну роль у процесі розробки та тестування. Вона використовувалася мною для написання модульних тестів, налагодження коду та імітації взаємодії з блокчейном. Hardhat дозволяє створювати тестові середовища з емульованими вузлами Ethereum, інтегрується з ethers.js та надає розширені можливості для автоматизації процесу тестування контрактів.

Таким чином, я обрав такий стек технологій: Ethereum, Solidity, AES-256-CBC, Node.js (crypto), ethers.js, та Hardhat для тестування і створення, що дозволить

мені створити безпечну, гнучку та ефективну систему для зберігання та перевірки зашифрованих даних у смарт-контрактах, яка може інтегруватись у серверні застосунки. Кожен інструмент доповнює інші, забезпечуючи комплексну архітектуру, що відповідає цілям безпеки, конфіденційності та цілісності.

4.3. Реалізація частини смарт-контракту

Для реалізації смарт-контракту було використано мову програмування Solidity, яка є основним стандартом розробки для платформи блокчейну Ethereum. Solidity забезпечує високий рівень сумісності з Ethereum Virtual Machine, підтримує контрактно-орієнтовану модель програмування та дозволяє безпечно взаємодіяти з децентралізованими структурами даних.

На відміну від традиційних відкритих сховищ інформації, смарт-контракт у цій системі не містить розшифрованих метаданих. Замість цього, для збереження конфіденційності система зберігає вже зашифровані дані, які користувач попередньо обробляє за допомогою симетричного шифрування на стороні клієнта. Таке рішення зменшує ризик витоку конфіденційної інформації навіть у разі повного доступу до контракту третіх осіб, оскільки смарт-контракт не виконує розшифрування, він тільки зберігає зашифровані записи.

Основна логіка контракту реалізована за допомогою двох ключових функцій [61]:

Перша функція буде createDID, функція, яка дозволяє користувачеві створити новий запис DID. До неї передаються такі параметри:

- ідентифікатор користувача;
- зашифровані метадані;
- відкритий ключ, який згодом може бути використаний іншими сторонами для перевірки аутентичності інформації або, в майбутньому, для реалізації асиметричного шифрування;

Функція зберігає цю інформацію у внутрішньому сховищі контракту, де кожен запис пов'язаний з унікальним DID. Таким чином, кожен користувач може

мати тільки один DID у контракті, що гарантує унікальність записів і запобігає дублюванню.

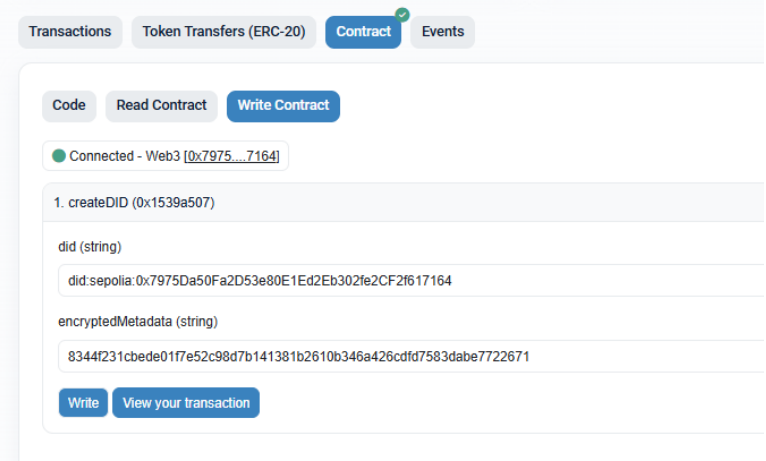

Рисунок 4.1 – Функція createDID.

Друга функція яку я буду використовувати називається getDID, така функція, яка дозволяє отримати дані про конкретний DID. Коли викликається ця функція, користувач вказує ідентифікатор, і контракт повертає відповідні зашифровані метадані разом із раніше збереженим відкритим ключем. Ця інформація може бути використана на стороні клієнта для перевірки цілісності або аутентичності або, в майбутньому, для дешифрування даних.

Оскільки контракт не розкриває внутрішній вміст зашифрованих метаданих, конфіденційність забезпечується навіть при публічному доступі до блокчейну. Це дозволяє дотримуватися принципів захисту даних, таких як мінімізація доступу, шифрування за замовчуванням та можливість перевірки походження інформації.

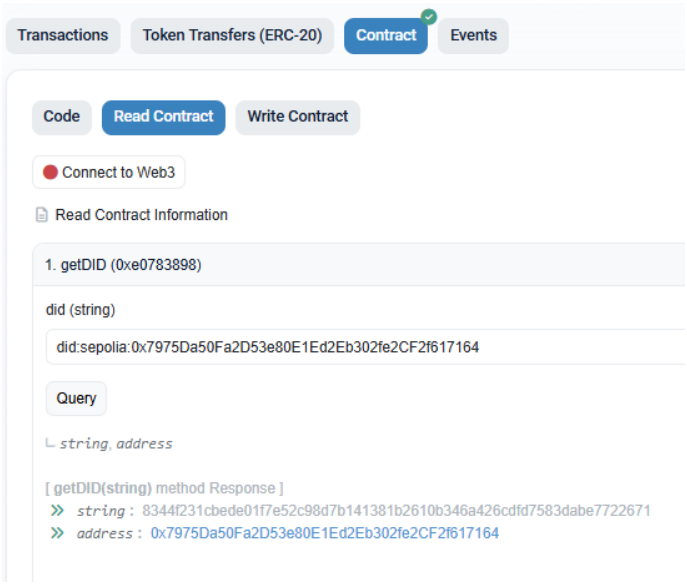

Рисунок 4.2 – Функція getDID та механізм отримання даних.

Реалізація частини смарт-контракту була протестована в локальному середовищі за допомогою інструменту Hardhat і довела свою функціональність під час збереження та читання записів DID.

4.4. Реалізація частини шифрування/дешифрування

У рамках практичної частини розробки системи важливим стало впровадження механізму шифрування та дешифрування даних на стороні клієнта. Це дозволило зберігати конфіденційну інформацію в смарт-контракті у формі, недоступній для сторонніх осіб, навіть при публічній прозорості мережі блокчейн. Шифрування реалізовано за допомогою алгоритму AES-256-CBC [62], одного з найнадійніших алгоритмів симетричного шифрування, що широко використовується в сучасних системах захисту даних.

Для реалізації процесу шифрування та дешифрування було створено два окремі скрипти JavaScript з використанням модуля crypto, вбудованого в платформу Node.js:

Скрипт encrypt.js, цей скрипт відповідає за шифрування публічних метаданих перед їх передачею в блокчейн. Основні етапи реалізації такі:

- Генерація випадкового симетричного ключа довжиною 256 бітів (32 байти), який буде використовуватися для шифрування.
- Генерація вектора ініціалізації, який є обов'язковим і забезпечує унікальність результату навіть при багаторазовому шифруванні одних і тих самих даних.

```
user@Zephyrus:~/did-secure-store/scripts$ node encrypt.js
Зашифровано і збережено у data.json
```

Рисунок 4.3 – Реалізація ecrypt.js

Далі відбувається збереження результату у файлі data.json, який містить: зашифровану інформацію у вигляді шістнадцяткового рядка, а також закритий ключ;

Рисунок 4.4 – Вигляд файлу data.json

Скрипт decrypt.js, який виконує зворотну операцію, тобто розшифровує дані, якщо користувач має доступ до відповідного симетричного ключа. Логіка скрипта включає:

- Зчитування даних з файлу data.json, зашифровану інформацію та ключ.
- Розшифрування за допомогою AES-256-CBC з використанням вказаного ключа.

```
user@Zephyrus:~/did-secure-store/scripts$ node decrypt.js
Дешифровано: Malykhin_SecurityMethods
```

Рисунок 4.5 – Вигляд виконання файлу decrypt.js

Файл decrypt забезбечує виведення розшифрованої інформації на консоль або дає можливість її подальшого використання в іншій частині серверного застосунку.

4.5. Переваги удосконаленої системи забезпечення цілісності і конфіденційності.

Розроблена вдосконалена система базується на концепції шифрування на стороні клієнта, що є надзвичайно важливою умовою для створення безпечних і децентралізованих серверних застосунків нового покоління. У цьому підході всі операції, пов'язані з обробкою конфіденційної інформації, включаючи шифрування та дешифрування, виконуються виключно на стороні користувача до того, як дані потрапляють у публічну мережу блокчейну. Це означає, що смарт-контракт діє лише як безпечне сховище для зашифрованих метаданих, без доступу до їх відкритого вмісту. Така архітектура дозволяє кінцевому користувачу зберігати повний контроль над конфіденційною інформацією.

Оскільки шифрування відбувається до надсилання даних до блокчейну, сам блокчейн, як і будь-який сторонній учасник мережі, не має доступу до незашифрованої інформації. Це значно зменшує ризики, пов'язані з витоком або несанкціонованим доступом до особистих або бізнес-даних. Такий підхід дозволяє системі залишатися продуктивною, що є важливою особливістю для інтеграції в реальні серверні застосунки.

Ще однією сильною стороною підходу є відкритість і прозорість технічної реалізації. Для шифрування використовуються стандартні інструменти середовища Node.js, зокрема бібліотека шифрування, а формат зберігання та передачі даних реалізований на блокчейні. Це забезпечує легкість інтеграції з іншими системами забезпечення інформаційної безпеки, а також надає можливість проведення зовнішнього аудиту, також надає прозорість усіх дій користувачів та адміністраторів, та гнучкість для подальшого масштабування системи та модифікації рішення в цілому. Ця відкрита архітектура створює сприятливе середовище для розвитку та адаптації проєкту відповідно до мінливих вимог безпеки.

Висновки до розділу 4

В результаті цього розділу було створено вдосконалену системи забезпечення цілісності та конфіденційності даних у серверних застосунках з використанням сучасних технологій блокчейну, зокрема платформи Ethereum. Розробка підтвердила доцільність використання смарт-контрактів як безпечного та прозорого сховища для зашифрованих метаданих, що дозволяє зберігати важливу інформацію без її розголошення у відкритому доступі.

Відмінною особливістю запропонованої архітектури є реалізація шифрування на стороні клієнта, що запобігає передачі відкритих даних до блокчейну та надає повний контроль над інформацією в руки користувача. Такий підхід забезпечує високий рівень конфіденційності, дозволяє уникнути централізованих точок контролю та зменшує ймовірність витоку даних. Смарт-контракт, у свою чергу, виступає децентралізованим контейнером для зберігання зашифрованих записів, що дозволяє перевірити аутентичність інформації без розкриття її змісту.

Розроблена система довела можливість поєднання криптографічних методів і децентралізованих технологій для створення безпечного середовища для обробки конфіденційних даних у серверних застосунках. Успішна реалізація цього підходу відкриває перспективи для подальшого розвитку інтегрованих рішень у сфері інформаційної безпеки.

ВИСНОВКИ

У ході кваліфікаційної роботи було всебічно досліджено сучасні серверні застосунки, особливості їх архітектури, типові вразливості та методи захисту, що дозволило сформувати цілісне уявлення про ефективні підходи до захисту даних у серверних додатках.

У першому розділі розглянуто загальну структуру серверних застосунків, особливості клієнт-серверної моделі, різні типи архітектур, такі як дворівнева та трирівнева. Проведено порівняльний аналіз цих архітектур з акцентом на їх стійкість до типових загроз. Встановлено, що зростання складності архітектури безпосередньо впливає на необхідність застосування гнучких і багаторівневих систем захисту.

Другий розділ присвячений аналізу вразливостей серверних застосунків. Особливу увагу я приділив таким поширеним загрозам, як ін'єкційні атаки типу SQL-ін'єкцій, міжсайтовий скриптинг та порушення контролю доступу. Для кожного типу загроз представлено сучасні засоби протидії, включаючи системи рольового контролю доступу та контролю доступу за атрибутами, а також спеціалізовані засоби безпеки. Проаналізовано ефективність підходів до запобігання використанню вразливостей за допомогою перевірки вхідних даних, фільтрації запитів та належної конфігурації прав доступу.

У третьому розділі розглядаються питання забезпечення конфіденційності та цілісності даних. Особлиу увагу приділив методам криптографічного захисту, як симетричним, так і асиметричним, а також сучасним підходам захисту на основі технології блокчейн. Детально проаналізував використання технологій смарт-контрактів у блокчейні для зберігання зашифрованих даних та застосування децентралізованих ідентифікаторів, що дозволяє гарантувати аутентичність походження даних та їх незмінність. Окремо розглядаються переваги інтеграції блокчейну в процес забезпечення безпеки, зокрема усунення централізованих точок відмови та підвищення прозорості доступу до інформації.

У четвертому розділі реалізовано вдосконалену систему безпеки, що поєднує локальне шифрування даних клієнта та зберігання зашифрованих даних у смарт-контрактах на блокчейні. Такий підхід забезпечує високий рівень цілісності та конфіденційності інформації, а також контроль доступу без залучення централізованих служб. Запропонована модель базується на сучасних концепціях, що робить систему менш вразливою до атак, спрямованих на інфраструктуру серверних застосунків.

В результаті виконаної роботи:

- проаналізовано проблеми безпеки серверних додатків в контексті сучасних загроз;
- проаналізовано типові архітектурні моделі та потенційні вектори атак, а також ефективні підходи до захисту від найбільш критичних типів атак;
- реалізовано вдосконалену систему, що інтегрує шифрування, смарт-контракти для забезпечення високого рівня безпеки серверного застосунку;
- доведено, що поєднання методів криптографічного захисту з децентралізованими технологіями може бути ефективним доповненням до традиційних підходів.

Таким чином, результати дослідження мають практичну цінність і можуть бути використані при створенні безпечних серверних застосунків, особливо в критичних сферах, де важлива цілісність і конфіденційність даних, таких як охорона здоров'я, фінанси, державне управління тощо.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Kumar S. A Review on Client-Server based applications and research opportunity //International Journal of Recent Scientific Research. – 2019. – Т. 10. – №. 7. – С. 33857-3386.

2. Inmon W. H. Developing client/server applications. – QED Information Sciences, Inc., 1991.

3. Sinz E. J., Knobloch B., Mantel S. Web-Application-Server //Wirtschaftsinformatik. – 2000. – Т. 42. – С. 550-552.

4. Radhakrishnan R., John L. K. A performance study of modern Web server applications //Euro-Par'99 Parallel Processing: 5th International Euro-Par Conference Toulouse, France, August 31–September 3, 1999 Proceedings 5. – Springer Berlin Heidelberg, 1999. – С. 239-247.

5. Duchessi P., Chengalur-Smith I. S. Client/server benefits, problems, best practices //Communications of the ACM. – 1998. – Т. 41. – №. 5. – С. 87-94.

6. Lile E. A. Client/Server architecture: A brief overview //Journal of Systems Management. – 1993. – Т. 44. – №. 12. – С. 26.

7. Blażewicz J., Dell'Olmo P., Drozdowski M. Scheduling of client-server applications //International Transactions in Operational Research. – 1999. – Т. 6. – №. 4. – С. 345-363.

8. Kistijantoro A. I. et al. Enhancing an application server to support available components //IEEE Transactions on Software Engineering. – 2008. – Т. 34. – №. 4. – С. 531-545.

9. Gallaugher J. M., Ramanathan S. C. Choosing a client/server architecture: a comparison of two-and three-tier systems //Information Systems Management. – 1996. – Т. 13. – №. 2. – С. 7-13.

10. Steele R. et al. A two-tier architecture for automated mobile applications //International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. – IEEE, 2004. – Т. 2. – С. 281-285.

11. Nyabuto M. G. M., Mony V., Mbugua S. Architectural review of client-server models //International journal of scientific research and engineering trends. – 2024. – Т. 10. – №. 1. – С. 139-143.

12. Aarsten A., Brugali D., Menga G. Patterns for three-tier client/server applications //Proceedings of Pattern Languages of Programs (PLoP'96). – 1996. – Т. 4. – №. 6.

13. Fernandez E. B. et al. The secure three-tier architecture pattern //2008 International Conference on Complex, Intelligent and Software Intensive Systems. – IEEE, 2008. – С. 555-560.

14. Tie J., Jin J., Wang X. Study on application model of three-tiered architecture //2011 Second International Conference on Mechanic Automation and Control Engineering. – IEEE, 2011. – С. 7715-7718.

15. Aarsten A., Brugali D., Menga G. Patterns for three-tier client/server applications //Proceedings of Pattern Languages of Programs (PLoP'96). – 1996. – Т. 4. – №. 6.

16. Dorofeev D., Shestakov S. 2-tier vs. 3-tier architectures for data processing software //Proceedings of the 3rd International Conference on Applications in Information Technology. – 2018. – С. 63-68.

17. Housel T., Meaney S. The Three Tier Model: Step by Step. – 2002.

18. Auger R., Barnett R. Web Application Security Consortium: Threat Classification Version 1.0 //Web Application Security Consortium (www. webappsec. org). – 2004.

19. Wang X. et al. Attacks and defenses in user authentication systems: A survey //Journal of Network and Computer Applications. – 2021. – Т. 188. – С. 103080.

20. Sedek K. A. et al. Developing a Secure Web Application Using OWASP Guidelines //Comput. Inf. Sci. – 2009. – Т. 2. – №. 4. – С. 137-143.

21. Muscat I. Web vulnerabilities: identifying patterns and remedies //Network Security. – 2016. – Т. 2016. – №. 2. – С. 5-10.

22. Bertino E. RBAC models—concepts and trends //Computers & Security. – 2003. – Т. 22. – №. 6. – С. 511-514.

23. Ferraiolo D. et al. Role-based access control (RBAC): Features and motivations //Proceedings of 11th annual computer security application conference. – 1995. – С. 241-48.

24. Yuan E., Tong J. Attributed based access control (ABAC) for web services //IEEE International Conference on Web Services (ICWS'05). – IEEE, 2005.

25. Morisset C., Willemse T. A. C., Zannone N. Efficient extended ABAC evaluation //Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies. – 2018. – С. 149-160.

26. Sandhu R., Munawer Q. How to do discretionary access control using roles //Proceedings of the third ACM workshop on Role-based access control. – 1998. – С. 47-54.

27. Moffett J., Sloman M., Twidle K. Specifying discretionary access control policy for distributed systems //Computer Communications. – 1990. – Т. 13. – №. 9. – С. 571-580.

28. Born E., Stiegler H. Discretionary access control by means of usage conditions // Computers & Security. – 1994. – Т. 13. – №. 5. – С. 437-450.

29. Ahmed S., Mahmood Q. An authentication based scheme for applications using JSON web token //2019 22nd international multitopic conference (INMIC). – IEEE, 2019. – С. 1-6.

30. Mahindraka P. Insights of JSON Web Token //International International Journal of Recent Technology and Engineering (IJRTE) ISSN. – 2020. – С. 2277-3878.

31. Halfond W. G. J. et al. A Classification of SQL Injection Attacks and Countermeasures //ISSSE. – 2006.

32. Mukherjee S. et al. SQL Injection: A sample review //2015 6th International Conference on Computing, Communication and Networking Technologies (ICCCNT). – IEEE, 2015. – С. 1-7.

33. Yunus M. A. M. et al. Review of SQL injection: problems and prevention //JOIV: International Journal on Informatics Visualization. – 2018. – Т. 2. – №. 3-2. – С. 215-219.

34. Sharma K., Bhatt S. SQL injection attacks-a systematic review //International journal of information and computer security. – 2019. – Т. 11. – №. 4-5. – С. 493-509.

35. Aliero M. S. et al. Review on SQL injection protection methods and tools //Jurnal Teknologi (Sciences & Engineering). – 2015. – Т. 77. – №. 13.

36. Gupta S., Gupta B. B. Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art //International Journal of System Assurance Engineering and Management. – 2017. – Т. 8. – С. 512-530.

37. Grossman J. XSS attacks: cross site scripting exploits and defense. – Syngress, 2007.

38. Cui Y., Cui J., Hu J. A survey on XSS attack detection and prevention in web applications //Proceedings of the 2020 12th International Conference on Machine Learning and Computing. – 2020. – С. 443-449.

39. Everett C. Is ISO 27001 worth it? //Computer Fraud & Security. – 2011. – Т. 2011. – №. 1. – С. 5-7.

40. Morse E. A., Raval V. PCI DSS: Payment card industry data security standards in context //Computer Law & Security Review. – 2008. – Т. 24. – №. 6. – С. 540-554.

41. Carter J. L., Wegman M. N. Universal classes of hash functions //Proceedings of the ninth annual ACM symposium on Theory of computing. – 1977. – С. 106-112.

42. Ometov A. et al. Multi-factor authentication: A survey //Cryptography. – 2018. – Т. 2. – №. 1. – С. 1.

43. Simmons G. J. Symmetric and asymmetric encryption //ACM Computing Surveys (CSUR). – 1979. – Т. 11. – №. 4. – С. 305-330.

44. Alenezi M. N., Alabdulrazzaq H., Mohammad N. Q. Symmetric encryption algorithms: Review and evaluation study //International Journal of Communication Networks and Information Security. – 2020. – Т. 12. – №. 2. – С. 256-272.

45. Selent D. Advanced encryption standard //Rivier Academic Journal. – 2010. – Т. 6. – №. 2. – С. 1-14.

46. Bellare M., Kilian J., Rogaway P. The security of cipher block chaining //Annual International Cryptology Conference. – Berlin, Heidelberg : Springer Berlin Heidelberg, 1994. – С. 341-358.

47. De Canniere C., Biryukov A., Preneel B. An introduction to block cipher cryptanalysis //Proceedings of the IEEE. – 2006. – Т. 94. – №. 2. – С. 346-356.

48. Bellare M., Rogaway P. Optimal asymmetric encryption //Advances in Cryptology—EUROCRYPT'94: Workshop on the Theory and Application of Cryptographic Techniques Perugia, Italy, May 9–12, 1994 Proceedings 13. – Springer Berlin Heidelberg, 1995. – С. 92-111.

49. Liu W., Cao B., Peng M. Web3 technologies: Challenges and opportunities //IEEE Network. – 2023. – Т. 38. – №. 3. – С. 187-193.

50. Wan S. et al. Web3: The next internet revolution //IEEE Internet of Things Journal. – 2024. – Т. 11. – №. 21. – С. 34811-34825.

51. Zheng Z. et al. An overview on smart contracts: Challenges, advances and platforms //Future Generation Computer Systems. – 2020. – Т. 105. – С. 475-491.

52. Kolvart M., Poola M., Rull A. Smart contracts //The Future of Law and etechnologies. – 2016. – С. 133-147.

53. Ma F. et al. Security reinforcement for Ethereum virtual machine //Information Processing & Management. – 2021. – Т. 58. – №. 4. – С. 102565.

54. Mahalle P. N., Shinde G., Shafi P. M. Rethinking decentralised identifiers and verifiable credentials for the Internet of Things //Internet of things, smart computing and technology: A roadmap ahead. – 2020. – С. 361-374.

55. Rodríguez-Doncel V. Web Technologies for Decentralised Identity //Governance and Control of Data and Digital Economy in the European Single Market. – 2025. – С. 111.

56. Giannopoulou A. Allocating control in decentralised identity management //European Review of Digital Administration & Law. – 2021. – Т. 2. – №. 2. – С. 73-88.

57. Di Pierro M. What is the blockchain? //Computing in Science & Engineering. – 2017. – Т. 19. – №. 5. – С. 92-95.

58. Zheng Z. et al. Blockchain challenges and opportunities: A survey //International journal of web and grid services. – 2018. – Т. 14. – №. 4. – С. 352-375.

59. Potter B. Microsoft SDL threat modelling tool //Network Security. – 2009. – Т. 2009. – №. 1. – С. 15-18.

60. Gu Q., Liu P. Denial of service attacks //Handbook of Computer Networks: Distributed Networks, Network Planning, Control, Management, and New Trends and Applications. – 2007. – Т. 3. – С. 454-468.

61. Reed D. et al. Decentralized identifiers (dids) v1. 0 //Draft Community Group Report. – 2020.

62. Biryukov A., Khovratovich D. Related-key cryptanalysis of the full AES-192 and AES-256 //Advances in Cryptology–ASIACRYPT 2009: 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings 15. – Springer Berlin Heidelberg, 2009. – С. 1-18.

ДОДАТОК А

СПИСОК ОПУБЛІКОВАНИХ ПРАЦЬ ЗА ТЕМОЮ РОБОТИ

Тези наукових доповідей:

1. Ivan Sofiyenko, Oleksandr Malykhin, Lavryk Oleksandr. FEATURES OF WEB APPLICATION SECURITY. Міжнародна науково-практична конференція «Інформаційні технології та впровадження» (Information technology and implementations) (IT&I-2024). С. 122-124.

2. Malykhin Oleksandr, Serhii Toliupa. Security strategies for protecting server applications. VIII Міжнародна науково-практична конференція «Проблеми кібербезпеки інформаційно-комунікаційних систем» (PCSICS).

ДОДАТОК Б

ФАЙЛИ З ПРОГРАМНИМ КОДОМ

Файл DIDStorage.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

contract DIDStorage {
    struct DIDData {
        string encryptedMetadata;
        address owner;
    }

    mapping(string => DIDData) private dids;

    function createDID(string memory did, string memory encryptedMetadata) public {
        require(dids[did].owner == address(0), "DID already exists");
        dids[did] = DIDData(encryptedMetadata, msg.sender);
    }

    function getDID(string memory did) public view returns (string memory, address) {
        DIDData memory data = dids[did];
        require(data.owner != address(0), "DID not found");
        return (data.encryptedMetadata, data.owner);
    }
}
```

Файл deploy.js

```
const hre = require("hardhat");

async function main() {
```

```
  const DIDStorage = await hre.ethers.getContractFactory("DIDStorage");
  const contract = await DIDStorage.deploy();
  await contract.deployed();
  console.log(Contract deployed to: ${contract.address});
}

main().catch((error) => {
  console.error(error);
  process.exitCode = 1;
});
```

Файл encrypt.js

```
const plaintext = 'Malykhin_SecurityMethods';

// Генеруємо ключ (256 біт) і IV (128 біт)
const key = crypto.randomBytes(32);
const iv = crypto.randomBytes(16);

// Створення шифратора
const cipher = crypto.createCipheriv('aes-256-cbc', key, iv);

let encrypted = cipher.update(plaintext, 'utf8', 'hex');
encrypted += cipher.final('hex');

// Збереження даних у файл
const output = {
  encryptedData: encrypted,
  key: key.toString('hex'),
  iv: iv.toString('hex')
};

fs.writeFileSync('data.json', JSON.stringify(output, null, 2));
```

```
console.log('Зашифровано і збережено у data.json');
```

Файл decrypt.js

```
const crypto = require('crypto');
const fs = require('fs');

// Зчитування зашифрованих даних
const rawData = fs.readFileSync('data.json', 'utf8');
const { encryptedData, key, iv } = JSON.parse(rawData);

// Дешифрування
const decipher = crypto.createDecipheriv(
  'aes-256-cbc',
  Buffer.from(key, 'hex'),
  Buffer.from(iv, 'hex')
);

let decrypted = decipher.update(encryptedData, 'hex', 'utf8');
decrypted += decipher.final('utf8');

console.log('Дешифровано:', decrypted);
```