Вінницький національний технічний університет

Факультет менеджменту та інформаційної безпеки

Кафедра менеджменту та безпеки інформаційних систем

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

Розробка системи захисту веб-додатків із динамічним розподілом і управлінням криптографічними ключами та з можливістю шифрування у реальному часі

Виконав: студент 2(4) курсу, гр.1КІТС-216
спеціальності 125– Кібербезпека
Освітня програма – Кібербезпека
інформаційних технологій та систем
(шифр і назва напряму підготовки, спеціальності)

Лесь В.С.
(прізвище та ініціали)

Керівник: доцент каф. МБІС
Грицак А. В.
(прізвище та ініціали)
«____» _____ 2025 р.

Рецензент: к.т.н., доц., доцент каф. ОТ
Богомолов С. В.
(прізвище та ініціали)
«____» _____ 2025 р.

Допущено до захисту
Голова секції УБ кафедри МБІС
д.т.н., проф. Яремчук Ю.Є.
“____” _____2025р.

Вінниця ВНТУ – 2025

Вінницький національний технічний університет
Факультет менеджменту та інформаційної безпеки
Кафедра менеджменту та безпеки інформаційних систем

Рівень вищої освіти 1-й (бакалаврський)
Галузь знань 12 – Інформаційні технології
Спеціальність 125 – Кібербезпека
Освітньо-професійна програма - Кібербезпека інформаційних технологій та систем

ЗАТВЕРДЖУЮ
Голова секції УБ, кафедра МБІС
д.т.н., проф. Яремчук Ю.Є.
" 20 " березня 2025 р.

З А В Д А Н Н Я
на бакалаврську кваліфікаційну роботу студенту
Лесю Віталію Сергійовичу
(прізвище, ім'я, по-батькові)

1. Тема роботи розробка системи захисту веб-додатків із динамічним розподілом і управлінням криптографічним ключами та з можливістю шифрування у реальному часі.
Керівник роботи к.т.н, доц. кафедри МБІС Грицак Анатолій Васильович
(прізвище, ім'я, по-батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від " 20 " березня 2025року № 97
2. Термін подання студентом роботи 2 червня 2025 року
3. Вихідні дані до роботи Електронні джерела, наукові статті, протоколи SSL та TLS, документи NIST SP 800-57, технічна документація до MongoDB, Node.JS, OpenSSL, документи OWASP, інструменти Postman, Visual Studio Code, сервіси AWS, Azure, Google KMS.
4. Зміст текстової частини:
Для досягнення мети роботи було поставлено такі задачі; дослідити сучасні методи захисту веб-додатків і управління криптографічними ключами; оцінити наявні реалізації систем із динамічним розподілом ключів; спроєктувати та реалізувати систему захисту з шифруванням у реальному часі; протестувати працездатність і ефективність розробленого рішення.
5. Перелік ілюстративного матеріалу (з точним зазначенням обов'язкових креслень)
Блок-схема алгоритму динамічного управління ключами, схема архітектури взаємодії модулів (frontend/API/crypto/database), діаграма логування та дій користувача, схема зберігання ключів у MongoDB, скріншоти запитів до API та результатів тестування.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Розділ І	Грицак А.В. к.т.н, доц. кафедри МБІС		
Розділ ІІ	Грицак А.В. к.т.н, доц. кафедри МБІС		
Розділ ІІІ	Грицак А.В. к.т.н, доц. кафедри МБІС		

7.Дата видачі завдання 20 березня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№	Назва та зміст етапу	Термін виконання		Примітк
		початок	закінчення	
1	Аналіз предметної області	20.03.2025	23.03.2025	Виконан
2	Аналіз літературних джерел за напрямком роботи	24.03.2025	13.04.2025	Виконан
3	Розробка алгоритму роботи	14.04.2025	27.04.2025	Виконан
4	Написання розділів роботи	28.04.2025	25.05.2025	Виконан
5	Передзахист кваліфікаційної роботи	26.05.2025	27.05.2025	Виконан
6	Виправлення, уточнення, корегування, бакалаврської роботи	28.05.2025	02.06.2025	Виконан
7	Захист бакалаврської кваліфікаційної роботи	09.06.2025	10.06.2025	Виконан

Студент Лесь В.С.

(підпис) (прізвище та ініціали)

Керівник роботи Грицак А

(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська кваліфікаційна робота складається з сторінок формату А4, на яких є рисунків, список використаних джерел містить найменувань. Бакалаврська кваліфікаційна робота присвячена розробці системи захисту веб-додатків з динамічним розподілом і управлінням криптографічними ключами та шифруванням у реальному часі.

У результаті аналізу сучасних методів захисту веб-додатків і криптографічних механізмів виявлено недоліки статичних систем розподілу ключів. Запропонована система реалізує динамічну генерацію та ротацію ключів, що дозволяє зменшити ризик їх компрометації. Система включає механізми поведінкового аналізу, адаптивного шифрування та управління доступом у реальному часі. У роботі представлено структуру бази даних, алгоритми обміну ключами, описано тестування працездатності та ефективності захисту. Отримані результати підтверджують підвищення стійкості веб-додатка до сучасних кіберзагроз.

Ключові слова: захист веб-додатків, криптографічний ключ, шифрування в реальному часі, динамічний розподіл ключів, інформаційна безпека, поведінковий аналіз, адаптивна система, ротація ключів.

ABSTRACT

The bachelor's qualification thesis consists of A4 pages, containing figures, and a list of references.

The thesis is devoted to the development of a web application protection system with dynamic cryptographic key distribution and real-time encryption.

As a result of analyzing current web application security methods and cryptographic mechanisms, the limitations of static key management systems were identified. The proposed system implements dynamic key generation and rotation, reducing the risk of key compromise. It incorporates behavioral analysis, adaptive encryption, and real-time access control mechanisms. The work presents a database structure, key exchange algorithms, and a detailed description of system testing. The results demonstrate an improved level of resilience against modern cyber threats and confirm the efficiency of the proposed approach.

Keywords: web application security, cryptographic key, real-time encryption, dynamic key distribution, information security, behavioral analysis, adaptive system, key rotation.

ЗМІСТ

ВСТУП

Актуальність. У сучасному цифровому середовищі, де обробка конфіденційних даних у веб-додатках стала повсякденною практикою, забезпечення їх захисту є пріоритетом у сфері кібербезпеки. Веб-додатки нерідко виступають основною ціллю для зловмисників через вразливості, пов'язані з передачею, зберіганням та обробкою інформації. Окрім атак на логіку застосунку (SQL-ін'єкції, XSS), суттєвої загрози набувають атаки, спрямовані на компрометацію або перехоплення криптографічних ключів, що забезпечують шифрування трафіку між клієнтом і сервером.

Традиційні засоби захисту, такі як протоколи HTTPS/TLS або використання фіксованих сесійних ключів, не забезпечують належного рівня безпеки у випадках, коли ключ використовується тривалий час. Компрометація одного такого ключа відкриває зловмиснику доступ до великого обсягу переданих даних. В умовах сучасних загроз, таких як атаки повторного використання ключа (key reuse), атаки на час виконання та інциденти стороннього доступу, актуальним є підхід, що забезпечує динамічну зміну ключів у режимі реального часу та перевірку цілісності інформації.

Саме тому розробка системи захисту веб-додатків, що ґрунтується на динамічному управлінні криптографічними ключами та автоматичному реагуванні на загрози, є актуальним і важливим завданням у сфері захисту інформації.

Метою роботи є створення повнофункціональної системи захисту веб-додатків, яка реалізує динамічну генерацію, ротацію та зберігання криптографічних ключів, підтримує шифрування та дешифрування повідомлень у режимі реального часу, перевірку цілісності даних та фіксацію дій у журналі безпеки.

Задачами дослідження є:

– аналіз сучасних методів захисту веб-додатків та управління криптографічними ключами;

– вивчення технічних обмежень існуючих реалізацій та визначення

потреби у динамічному шифруванні;

– розробка архітектури системи з підтримкою автоматичної ротації ключів;

– реалізація програмного модуля з використанням Node.js, MongoDB та власного алгоритму управління ключами;

– проведення тестування, перевірка працездатності, фіксація подій та аналіз ефективності системи.

Об’єкт дослідження: процес забезпечення криптографічного захисту даних у веб-додатках із використанням механізмів динамічного управління ключами.

Предмет дослідження: структура, алгоритми та програмна реалізація системи, що забезпечує шифрування даних у режимі реального часу з підтримкою автоматичної ротації ключів та логування дій.

Практична цінність. Результатом роботи є створений програмний модуль, що може бути інтегрований у веб-додатки для забезпечення адаптивного шифрування, виявлення загроз та ведення журналу безпеки. Реалізоване рішення є автономним, не вимагає сторонніх бібліотек або сервісів, забезпечує контроль над усіма етапами криптографічного захисту та може бути використане у реальних умовах експлуатації.

1 АНАЛІЗ МЕТОДІВ ТА ЗАСОБІВ ЗАХИСТУ ВЕБ-ДОДАТКІВ

Даний розділ охоплює всебічний аналіз сучасних методів захисту веб-додатків, які базуються на криптографічних методах. Також розглядаються підходи до забезпечення безпеки, їх стоуктура та принцип роботи. Особлива увага приділена методам управління криптографічним ключам, їх динамічному розподілу. Розглядаються всі існуючі реалізації даних систем. На основі цього формуються висновки щодо недоліків існуючих рішень та обґрунтовується необхідність їх удосконалення для ефективнішого захисту веб-додатків.

1.1 Аналіз існуючих підходів до захисту веб-додатків

Захист веб-додатків є ключовим напрямом у сфері інформаційної безпеки, оскільки саме через вразливості таких систем найчастіше відбувається несанкціоноване втручання. Типовими атаками залишаються SQL-ін’єкції, XSS, CSRF, атаки через підміну елементів DOM та міжсайтове виконання коду [21].

Сучасні засоби захисту поділяються на інфраструктурні, прикладні та криптографічні. Інфраструктурний рівень охоплює міжмережеві екрани веб-застосунків (WAF), які аналізують трафік та блокують підозрілі запити. Ці системи ефективні проти відомих атак, однак не здатні виявити цільові маніпуляції з бізнес-логікою додатку [22].

Прикладний рівень реалізується, зокрема, за допомогою Content Security Policy (CSP), що обмежує виконання скриптів у браузері, та Strict-Transport-Security (HSTS), який змушує браузер використовувати лише HTTPS. Проте ці засоби зосереджені на клієнтській безпеці і не контролюють внутрішні обробники або стан даних у транспортному середовищі [23].

Криптографічний рівень забезпечується застосуванням алгоритмів шифрування (AES, RSA), хешування (SHA-256, BLAKE2), цифрових підписів, а також інфраструктур управління ключами (KMS). Наприклад, такі сервіси, як AWS KMS або Azure Key Vault, дозволяють централізовано створювати й обертати ключі шифрування. Однак використання зовнішніх сервісів є ризикованим для автономних систем або в умовах критичної інфраструктури, що

вимагає внутрішнього управління ключами без передачі їх у хмарне середовище [24].

Більшість реалізацій використовує один ключ на всю сесію користувача, що створює точку потенційного компрометування. Уразливість посилюється при тривалому життєвому циклі ключа, оскільки зловмисник, здобувши його, може отримати доступ до всієї зашифрованої інформації. Порівняльний аналіз продуктивності шифрування показує, що сучасні сервери здатні генерувати ключі AES-256 менш ніж за 1 мс, що дозволяє впровадити ротацію ключа навіть на рівні окремого запиту без втрати продуктивності [25].

Аналіз показує, що традиційні засоби мають обмежену ефективність у динамічних умовах, де необхідна реакція на інциденти без зупинки системи. Це обґрунтовує необхідність створення архітектур, що підтримують автоматизовану зміну ключів шифрування та перевірку цілісності повідомлень у режимі реального часу.

Розвиток веб-додатків став важливим етапом у цифровій трансформації бізнесу, державного управління та повсякденного життя. Проте поряд із розширенням функціональності веб-технологій зростає кількість кіберзагроз, спрямованих на порушення конфіденційності, цілісності та доступності даних, що обробляються веб-додатками.

Захист веб-додатків є складним завданням, яке потребує використання різноманітних підходів для мінімізації ризиків, пов’язаних із поширеними загрозами, такими як SQL-ін’єкції, міжсайтовий скриптинг (XSS), DDoS-атаки, несанкціонований доступ тощо.

Особливу увагу буде приділено механізмам, які забезпечують динамічне управління криптографічними ключами та шифрування даних у реальному часі, адже саме ці підходи стають дедалі актуальнішими у відповідь на нові типи атак і вимоги до підвищення безпеки у веб-додатках.

Використання веб-аплікаційних фаєрволів (WAF). Web Application Firewall – це система захисту вебдодатків від потенційно шкідливого або небезпечного трафіку з інтернету. Вона моніторить http-трафік між вебдодактом та інтернотом, а потім відфільтровує виявлені загрози. WAF працює з дотриманням набору

політик, які допомагають визначити чи шкідливий трафік [1].

Web Application Firewall убезпечує від різноманітних типів хакерських атак, наприклад:

1) SQL Injection (SQL ін'єкції). Це спосіб злому додатків, які працюють з базами даних (БД), коли зловмисник може змінювати, видаляти або отримувати конфіденційну інформацію з БД (рис. 1.1).

Рисунок 1.1 – SQL ін'єкції [4]

2) Remote Code Execution (RCE) або віддалене виконання коду. У цьому випадку хакер використовує вразливість вебдодатка, щоб запустити власний шкідливий код на сервері компанії з метою отримання незаконного доступу до нього (рис. 1.2).

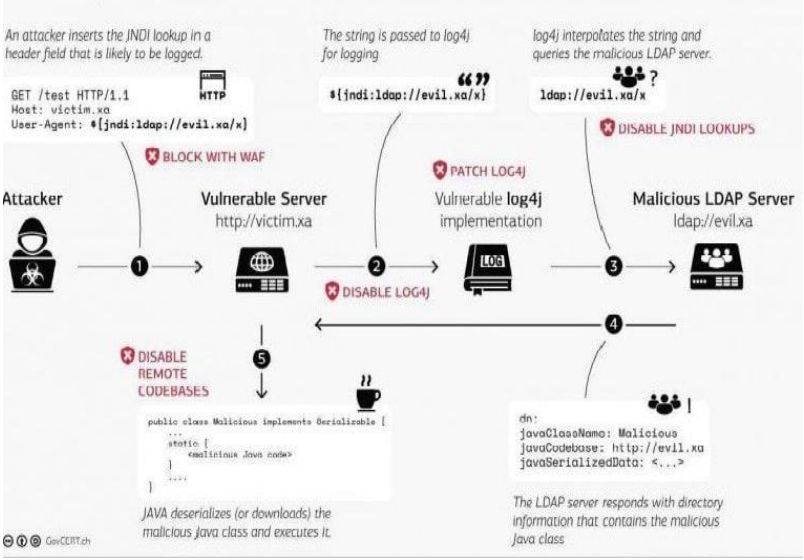


Рисунок 1.2 – Віддалене виконання коду [5]

3) Cross Site Scripting (XSS) або міжсайтовий скриптинг. Зловмисник впроваджує зловмисний скрипт у вебсторінку, який потім запускається в браузері користувача. Так можна отримати доступ до сесійних cookie, перехоплювати дані відвідувачів ресурсу або модифікувати веб сторінки (рис. 1.3).

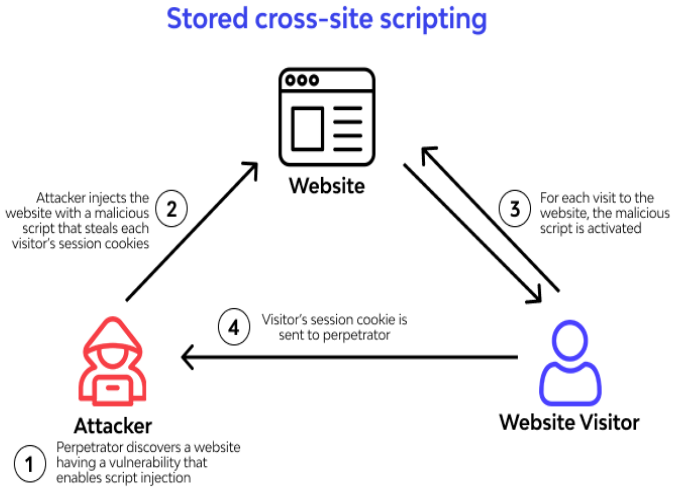


Рисунок 1.3 – XSS-атака [6]

4) Cross Site Request Forgery (CSRF) або міжстайтова підробка запитів. Під час цієї атаки авторизованого користувача вебдодатка змушують виконати небажані дії без його попередньої згоди або усвідомлення. Тобто хакер використовує довірливість вебдодатка до випадкового коду, який він впроваджує від імені справжнього користувача (рис. 1.4).

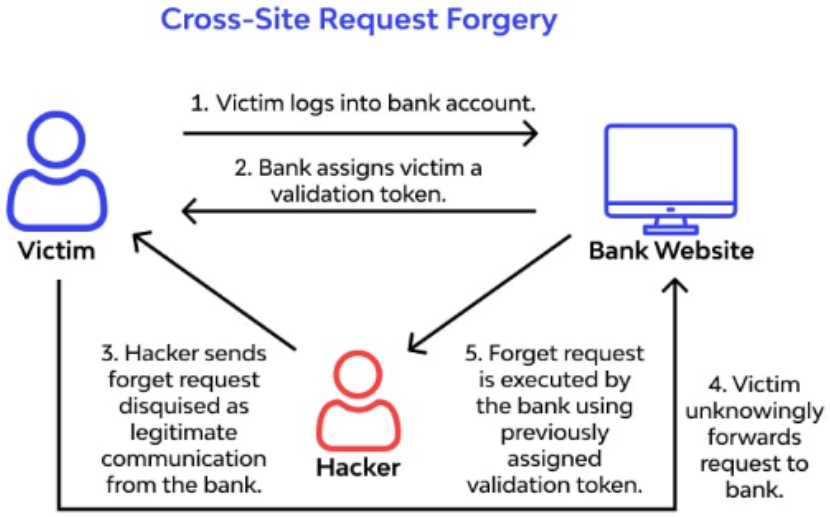


Рисунок 1.4 – Міжстайтова підробка запитів [7]

5) Bruteforce або вибір паролів (рис. 1.5). Це атака на аутентифікацію, коли зловмисник підбирає усі можливі комбінації логінів та паролів, доки не знайде правильний варіант. Завдяки вдалому зламу хакер отримує доступ до облікового запису чи ресурсу [1].

Рисунок 1.5 – Bruteforce або вибір паролів [8]

Використання Інтернет-мережі для зберігання/передавання платіжних та інших конфіденційних даних вимагає наявність надійного каналу зв'язку між комп'ютером користувача та веб-сервером. Утворення такого каналу можливе в разі застосування протоколів шифрування та аутентифікації даних, одним з яких є SSL/TLS.

SSL/TLS належить до класу криптографічних протоколів, покликаних забезпечити безпечний обмін даними між учасниками мережі на основі застосування криптографічних алгоритмів і перетворень.

Спочатку було розроблено SSL (Secure Sockets Layer), останню версію SSL 3.0 якого було опубліковано в 1996 р., але вже за два десятки років було визнано застарілою через наявність багатьох вразливостей і недосконалості архітектури.

На базі SSL 3.0 було створено протокол TLS (Transport Layer Security), актуальна версія якого на даний момент часу - TLS 1.3. Оскільки абревіатура SSL у колі фахівців набула широкого поширення, її все ще використовують, хоча технічна реалізація, як правило, ґрунтується вже на TLS, як більш надійному

варіанті [2].

На основі використання SSL було створено HTTPS, який «упаковує» дані, що передаються, у «криптографічний формат» SSL. Для сервера як ідентифікатор застосування шифрування в протоколі гіпертексту виступає номер використовуваного ним порту – 443. Це спрощує налагодження зв'язку при створенні нового з'єднання між сервером і клієнтом.

SSL є сполучною ланкою між одним із прикладних протоколів (HTTP, TELNET, FTP тощо) і протоколом транспортного рівня, виконуючи водночас роль своєрідного фільтра, що захищає дані.

SSL, як і будь-який інший криптографічний протокол, покликаний виконувати такі основні функції:

– формування, обмін і поділ ключів;

– автентифікація взаємодіючих сторін;

– забезпечення конфіденційності та цілісності даних;

– забезпечення цілісності з'єднання.

Для реалізації кожної з перерахованих функцій застосовується низка алгоритмів.

Для роботи з ключами в основному використовують алгоритми Diffie-Hellman, SRP, ECDH і вже застарілий RSA.

Для аутентифікації – ECDSA, RSA та деякі інші.

Для шифрування – DES, RC4, AES та інші.

Для хеш-функцій, які забезпечують перевірку цілісності даних – MD5, SHA [2].

Процес утворення захищеного каналу із сервером

Утворенню захищеного каналу між клієнтом і сервером передує процедура підтвердження зв'язку, квітування SSL або, як кажуть, «рукостискання», під час якої відбувається узгодження параметрів встановлення захищеного з'єднання. Як клієнт зазвичай виступає браузер користувача, який заходить на веб-сайт, розміщений на сервері. Кроки цієї процедури :

З боку клієнта надходить запит до сервера на встановлення захищеного з'єднання;

1) Серверу передається інформація про технології шифрування, які доступні клієнтській стороні;

2) Сервер повідомляє клієнту про те, які з доступних клієнту алгоритмів кодування даних були ним обрані;

3) Сервер передає цифровий сертифікат разом із відкритим ключем для можливості його автентифікації;

4) На клієнтській стороні відбувається перевірка переданого сертифіката на предмет його автентичності на основі наявних на стороні клієнта кореневих сертифікатів перевірених центрів сертифікації;

5) У разі успішної перевірки генерується загальний секретний сеансовий ключ, який необхідний для кодування сесії. Ключ може бути згенерований з використанням одного з наведених вище алгоритмів, наприклад, RSA або Діффі-Хеллмана, як найбільш надійного [2].

У разі успішного завершення зазначеного процесу між його учасниками відбувається встановлення безпечного з'єднання з утворенням каналу, готового для передавання закодованих даних із використанням методів симетричного шифрування. Канал існуватиме доти, доки з'єднання не буде завершено.

У разі якщо сторони з якихось причин «не домовляться», з'єднання не буде встановлено, і процес буде завершено з видачею відповідної помилки.

Щоб переконатися в автентичності власника відкритого ключа, протокол SSL/TLS задіює можливості цифрового сертифіката, який може бути як самопідписаним, так і виданим одним із довірених центрів сертифікації. Цей документ може «працювати» на рівні домену (Domain Validation), організації (Organization Validation) або проводити повну або розширену перевірку (Extended Validation).

З'ясувати, який тип сертифіката використовується для того чи іншого сайту, можна, клацнувши на зображення замка, зображеного ліворуч від адресного рядка браузера.

Рівень використовуваного на сайті сертифіката дає змогу оцінити ступінь довіри до нього з боку користувачів. Тому дуже часто банківські установи використовують сертифікати найвищого рівня – Extended Validation. У цьому

разі сертифікаційні центри здійснюють не тільки формальну перевірку домену, а й ідентифікацію його власника – юридичної або фізичної особи, що значно підвищує рівень довіри до сайту [2].

CSP (Content Security Policy) – це механізм безпеки браузера, який спрямований на запобігання XSS-атакам (міжсайтовий скриптинг) та іншим атакам. Вона працює шляхом обмеження ресурсів (наприклад, скриптів і зображень), які сторінка може завантажувати, а також контролює, чи може сторінка бути вставлена в рамку іншими сторінками [3].

Для ввімкнення CSP у відповідь сервера необхідно додати HTTP-заголовок Content-Security-Policy зі значенням, що містить політику. Сама політика складається з одного або декількох директив, розділених крапкою з комою.

Однак необхідно бути обережними, дозволяючи скрипти з зовнішніх доменів. Якщо зловмисник може впливати на контент, що надається з цього домену, він може використати його для атаки. Наприклад, CDN (мережі доставки контенту), які не використовують URL-адреси для кожного клієнта, такі як ajax.googleapis.com, не слід довіряти, оскільки сторонні особи можуть завантажувати свій контент на такі домени.

CSP дозволяє також використовувати два додаткові методи для визначення надійних ресурсів.

Нонси (nonces). Директива CSP може вказувати нонс (випадкове значення), який повинен бути використаний у тегах завантаження скриптів. Якщо значення не збігаються, скрипт не буде виконано. Щоб цей метод був ефективним, нонс повинен бути згенерований безпечно для кожного завантаження сторінки та недоступний для прогнозування зловмисниками.

Хеші. CSP-директива може містити хеш вмісту надійного скрипта. Якщо хеш реального скрипта не збігається із зазначеним у директиві, скрипт не буде виконано. У разі змінення вмісту скрипта потрібно буде оновити значення хешу в директиві [3].

CSP зазвичай блокує ресурси, такі як скрипти. Однак багато політик дозволяють запити до зовнішніх серверів для завантаження зображень. Це може створити ризик витоку, наприклад, токенів CSRF через елементи <img>.

Деякі браузери, як-от Chrome, мають вбудований захист від некоректної розмітки, блокуючи запити, що містять певні символи, наприклад, необроблені символи нового рядка або кутові дужки.

Ряд політик CSP може бути більш суворими, забороняючи всі зовнішні запити. Однак навіть такі обмеження можна обійти, якщо залучити користувача до виконання певних дій. Наприклад, можна вставити HTML-елемент, який при кліці користувачем збере та надішле інформацію на зовнішній сервер [3].

При використанні криптографічних механізмів : PKI, SSL/TLS, системи захисту веб-додатків не відповідають вимогам реального часу. Атаки типу аналізу часу виконання, активування механізмів обміну ключами, або експлуатація моментів ротації ключів, вказують на обмеження традиційних рішень, які зазвичай мають статичну структуру.

Більшість реалізацій полягає у активності ключів надто довго, що підвищує ризик компрометації. Деякі дослідження (NIST Recommendations for Key Management [12]) рекомендують зменшити «життєвий цикл» ключів та автоматизувати їх заміну, але у веб-додатках така реалізація застосовується рідко.

Шифрування в реальному часті часто не підтримується або реалізується на рівні HTTPS. Таким чином виникає проблема забезпечення динамічного управління криптографічними ключами та шифруванням в реальному часі у веб-додатках.

Вирішенням проблеми слугує створення гнучкої системи, яка адаптується до середовища, автоматично змінює ключі, а також реалізує реальний захист проти атак на часові залежності.

1.2 Аналіз методів управління криптографічними ключами та їх ефективність

Управління криптографічними ключами є одним із найважливіших етапів забезпечення інформаційної безпеки, оскільки саме ключі забезпечують доступ до зашифрованих даних та виконання критичних операцій. Надійність всієї системи захисту залежить не лише від криптостійкості алгоритмів, а й від

правильності генерації, розподілу, зберігання та оновлення ключів.

Традиційно ключі створюються під час ініціалізації з'єднання (наприклад, під час встановлення SSL/TLS-сесії), після чого залишаються актуальними протягом усієї сесії. Цей підхід є прийнятним у багатьох випадках, однак він створює ризик: у разі компрометації сесійного ключа зловмисник отримує можливість дешифрувати всі передані дані [12].

Існує кілька підходів до управління ключами. Один із найпоширеніших – централізоване управління за допомогою сервісів, таких як AWS KMS, Azure Key Vault або HashiCorp Vault. Ці рішення дозволяють зберігати ключі у захищеному середовищі, налаштовувати політики доступу, створювати резервні копії та здійснювати аудит дій. Наприклад, HashiCorp Vault підтримує механізми динамічного створення ключів і має інтеграцію з PKI [27].

Попри свої переваги, такі сервіси мають і низку обмежень. По-перше, вони потребують додаткової інфраструктури або облікових записів у хмарних середовищах, що не завжди прийнятно для ізольованих систем або систем з підвищеним рівнем безпеки. По-друге, централізоване зберігання ключів створює єдину точку відмови: у разі атаки на саму систему управління – компрометуються всі ключі [28].

Також застосовуються гібридні підходи: наприклад, локальна генерація ключів із подальшим короткочасним зберіганням у пам'яті та періодичною ротацією. Стандарт NIST SP 800-57 рекомендує обмежувати термін дії криптографічного ключа залежно від типу алгоритму та рівня загроз. Для симетричних ключів, що використовуються в онлайн-трафіку, рекомендований термін ротації становить від кількох хвилин до однієї години [29].

У сучасних системах управління безпекою все частіше впроваджується концепція динамічної ротації ключів, коли новий ключ створюється після кожної сесії, транзакції або навіть при кожному запиті. Такий підхід реалізується за допомогою вбудованих механізмів генерації ключів у реальному часі (on-the-fly) і дозволяє не зберігати ключі довго, що значно зменшує ризик їх втрати.

У розробленій системі було реалізовано адаптивну схему: ключ генерується при першому запиті та зберігається в оперативній пам'яті

обмежений час (30 секунд), після чого оновлюється автоматично або вручну при виникненні інциденту. Ключі додатково фіксуються в базі даних для забезпечення журналювання. Такий підхід дозволяє забезпечити стійкість до атак повторного використання ключа, а також підвищити рівень прозорості за рахунок обліку кожної ротації.

Таким чином, аналіз показує, що динамічне управління ключами, особливо із підтримкою адаптивної ротації, забезпечує вищу гнучкість і безпеку в порівнянні з класичними схемами. Обрана стратегія є компромісом між захищеністю, швидкодією та інтеграційною простотою.

Буде зосереджено увагу на традиційних підходах до управління ключами, таких як централізовані методи з використанням центрів розподілу ключів (KDC) і дистрибуція через публічні інфраструктури ключів (PKI). Ці методи мають різний рівень безпеки та гнучкості, що робить їх придатними для різних сценаріїв, починаючи від малих мереж до великих корпоративних систем.

В результаті цього аналізу будуть висвітлені переваги та недоліки кожного методу, а також їх ефективність у контексті сучасних загроз і вимог до захисту веб-додатків.

Потужність криптографії з відкритим ключем незаперечна. Вона вражає своєю простотою і здатністю забезпечити вирішення багатьох, здавалося б, непереборних організаційних проблем. Однак використання криптографії з відкритим ключем на практиці рідко буває таким простим, як здається на перший погляд. По-перше, необхідно вибрати тип криптографії з відкритим ключем, який підходить для вирішення даної організаційної задачі. Широкий вибір вражає: схеми шифрування з відкритим ключем, схеми цифрового підпису, схеми шифрування знаків, схеми ідентифікації, схеми атестації та будь-яка кількість більш спеціалізованих схем з різними функціональними властивостями та гарантіями безпеки.

Після цього організація має розробити інфраструктуру для генерації та розповсюдження автентичних копій відкритих та приватних ключів. Проблеми з розробкою відповідної інфраструктури з відкритим ключем (PKI) широко задокументовані. Навіть важливість і проблеми з генерацією випадкових ключів

широко обговорювалися як у практичній, так і в теоретичній літературі. Один з аспектів генерації ключів в практичній літературі в значній мірі ігнорується: проблема вибору відповідної довжини ключа.

У добре розробленій криптографічній схемі довший ключ означає вищу безпеку (і меншу ймовірність того, що система буде скомпрометована зловмисником). Це також зазвичай означає повільнішу схему. Більшість симетричних криптографічних схем не допускають використання ключів різної довжини. Якщо дизайнер хоче запропонувати симетричну схему, яка забезпечує різні рівні безпеки в залежності від розміру ключа, то дизайнер повинен сконструювати різні варіанти центральної конструкції, в яких використовується різна попередньо визначена довжина ключа. (Іншими словами, хоча AES-128 і AES-256 засновані на одних і тих же принципах, вони мають різні внутрішні компоненти, які були обрані для доповнення різної довжини ключа). Зокрема, допустимі довжини ключів вибирає конструктор схеми. Схеми з відкритим ключем відрізняються тим, що зазвичай мають одну конструкцію, яку можна використовувати з ключами будь-якої довжини (без додаткового втручання від розробника схем). Отже, користувач, а не дизайнер, повинен визначити відповідну довжину ключа для використання зі схемою.

Це часто залишає складне математичне рішення в руках математично недосвідченого менеджера з безпеки. Найбільш поширеним механізмом визначення відповідної довжини ключа є звернення до стандарту. Існує кілька стандартів, які дають рекомендації щодо довжини ключа для поширених технологій з відкритим ключем (наприклад, NIST, 2007; ECRYPT-II, 2010). Щоб допомогти менеджерам з безпеки приймати розумні рішення, ці стандарти зазвичай надають довжину ключа з точки зору кількості часу, протягом якого ключ буде «працювати», тобто кількість часу, протягом якого дані, захищені ключем цієї довжини, можуть вважатися безпечними. Математичне джерело чисел у цих стандартах та їх практичне значення часто розуміють неправильно.

Методи управління криптографічними ключами є центральним аспектом забезпечення інформаційної безпеки, особливо у контексті захисту веб-додатків. Ключі використовуються для забезпечення конфіденційності, цілісності та

автентичності даних. Вибір і впровадження відповідного методу управління ключами залежить від особливостей системи, ризиків і потреб бізнесу.

Одним із перших підходів до управління ключами є централізована модель, заснована на використанні центрів розподілу ключів (KDC). У цій моделі KDC виконує функції створення та передачі сесійних ключів між сторонами, які обмінюються інформацією. Цей підхід ефективний у закритих системах, однак централізація створює ризики: компрометація KDC означає потенційний доступ до всієї системи.

Іншим підходом є публічна інфраструктура ключів (PKI), яка базується на сертифікаційних центрах (CA). PKI дозволяє надійно перевіряти автентичність публічних ключів через сертифікати. Впровадження PKI передбачає підтримку сертифікаційної ієрархії та забезпечення надійної роботи CA, що часто потребує значних ресурсів.

Ефективність криптографічної системи залежить від правильної довжини ключа. Для симетричних алгоритмів, таких як AES, довжина ключів зазвичай варіюється від 128 до 256 біт. Для асиметричних алгоритмів, таких як RSA або ECC, необхідна більша довжина ключів, наприклад, 2048 або 3072 біт, щоб забезпечити стійкість до сучасних атак. Утім, довший ключ вимагає більше ресурсів для обробки, що може вплинути на продуктивність системи.

Сучасні веб-додатки часто використовують методи динамічного управління ключами, такі як ротація ключів, що дозволяє регулярно оновлювати ключі без переривання роботи системи. Наприклад, протоколи SSL/TLS забезпечують динамічний обмін ключами між клієнтами та серверами для шифрування веб-трафіку.

Окрім класичних та сучасних підходів, важливе місце займають гібридні системи управління ключами, які поєднують симетричне та асиметричне шифрування. Наприклад, асиметричні алгоритми (RSA, ECC) використовуються для безпечного розподілу сесійних ключів, які, своєю чергою, застосовуються у симетричних алгоритмах (AES) для шифрування великих обсягів даних. Це дозволяє досягти балансу між безпекою та продуктивністю.

Дослідження показують, що гібридні системи ефективно працюють у

хмарних середовищах, де безпека переданих даних та зберігання є ключовими викликами. Завдяки автоматизації обміну ключами та інтеграції з PKI, такі системи підтримують безпеку навіть у випадку фізичних загроз або атак на мережу .

Для захисту великих мереж, зокрема IoT-систем, використовуються розподілені системи управління ключами (DKMS). Вони передбачають зберігання та управління ключами на декількох вузлах, що забезпечує стійкість до атак типу "єдина точка відмови" (SPOF). Такі системи підтримують адаптивність до змін середовища та підвищують масштабованість, що критично важливо для сучасних динамічних додатків .

Оцінка ефективності методів управління ключами базується на кількох критеріях: стійкість до криптоаналізу, швидкість обробки, зручність у впровадженні та адаптація до нових загроз. Наприклад, методи на основі PKI забезпечують високу стійкість до атак, але потребують значних ресурсів для налаштування і підтримки .

Протоколи, такі як Kerberos, дозволяють мінімізувати ризики атак повторного використання ключів (replay attacks), завдяки короткочасним квиткам та автоматизованій перевірці автентичності. Водночас новітні підходи, такі як квантово-стійкі алгоритми, готуються до протистояння загрозам, пов'язаним із розвитком квантових обчислень, які потенційно можуть зламати сучасні асиметричні криптографічні схеми .

На основі проведеного аналізу можна зробити висновок, що оптимальний метод управління ключами залежить від специфіки застосування. Для веб-додатків із високим навантаженням найкраще підходять гібридні системи, які забезпечують баланс між швидкістю обробки та стійкістю до атак. Для захищених мереж і IoT-рішень доцільно використовувати розподілені системи DKMS із динамічною ротацією ключів.

Одним із сучасних підходів, який набуває популярності, є динамічна ротація криптографічних ключів. Цей метод передбачає автоматичну зміну ключів через певні проміжки часу або після кожної сесії. Така практика підвищує безпеку, оскільки навіть у випадку компрометації ключ стає непридатним для

подальшого використання. Динамічна ротація ключів широко використовується в хмарних системах, які потребують високого рівня захисту даних, особливо в системах із великим числом транзакцій .

Ефективність динамічної ротації також залежить від правильного налаштування алгоритмів генерації ключів. Наприклад, для асиметричних систем важливо забезпечити генерацію ключів із використанням криптографічно стійких генераторів випадкових чисел, що гарантує унікальність і непередбачуваність нових ключів.

Цей підхід дозволяє ефективно реагувати на атаки в реальному часі. Одним із прикладів є протоколи SSL/TLS, які використовують так званий "perfect forward secrecy", забезпечуючи, щоб навіть у випадку компрометації одного ключа інші залишалися захищеними.

Довжина криптографічного ключа є важливим фактором, що визначає рівень безпеки та продуктивність. З одного боку, довші ключі забезпечують вищу стійкість до криптоаналізу, з іншого – вони вимагають більше ресурсів для обчислень. Наприклад, для алгоритму RSA мінімальна рекомендована довжина ключа становить 2048 біт, але при цьому швидкість операцій значно знижується порівняно з використанням ключів у 1024 біти.

Зростання популярності хмарних технологій призвело до активного розвитку сервісів для управління криптографічними ключами. Такі платформи, як AWS Key Management Service (KMS) або Microsoft Azure Key Vault, надають користувачам зручні інструменти для генерації, зберігання та ротації ключів. Однією з головних переваг цих сервісів є інтеграція з іншими хмарними продуктами, що дозволяє автоматизувати процеси захисту даних.

Водночас використання хмарних сервісів супроводжується ризиками, пов'язаними з конфіденційністю. Наприклад, при компрометації облікових даних доступ до ключів може бути отриманий зловмисниками. Тому рекомендується застосовувати багатофакторну автентифікацію та обмежувати доступ до управління ключами.

Автоматизація управління криптографічними ключами є важливим напрямом розвитку технологій безпеки. Використання машинного навчання для

аналізу поведінки користувачів і виявлення аномалій відкриває нові можливості для підвищення захищеності систем. Наприклад, сучасні платформи можуть автоматично ідентифікувати спроби несанкціонованого доступу та оперативно змінювати ключі для запобігання атакам.

Перспективи розвитку включають впровадження квантово-стійких алгоритмів, які забезпечують захист від майбутніх загроз, пов'язаних із квантовими обчисленнями. Ці алгоритми потребують нових підходів до управління ключами, зокрема через значне збільшення їх довжини.

Ці тенденції вказують на важливість постійного вдосконалення методів управління ключами та їх адаптації до змін у технологічному середовищі.

Розглянуті методи управління криптографічними ключами демонструють широкий спектр підходів, кожен із яких має свої переваги та обмеження. Вибір оптимального методу залежить від специфіки системи, рівня ризиків і ресурсів, доступних організації. У майбутньому подальший розвиток технологій, зокрема впровадження квантово-стійких систем, вимагатиме ще більш гнучких і автоматизованих рішень.

1.3 Оцінка існуючих реалізацій захисту веб-додатків з динамічним управлінням криптографічними ключами

Проблематика ефективного управління криптографічними ключами у веб-додатках давно перебуває у фокусі дослідників у галузі кібербезпеки. Класичні реалізації переважно базуються на фіксованому ключі сесії або використовують TLS як єдиний канал захисту, що само по собі не забезпечує достатнього рівня захисту у випадку компрометації або атаки на канальному рівні. З огляду на це, в останні роки з’явились концепції систем з динамічним розподілом ключів, ротацією в режимі реального часу та реактивним шифруванням.

Одним із прикладів таких підходів є системи шифрування у потокових протоколах типу DTLS, де ключі періодично оновлюються в обхід основного SSL-каналу. Аналогічно, у деяких рішеннях, зокрема в Zero Trust-архітектурах, реалізуються мікросесійні ключі, що генеруються для кожної транзакції або навіть API-запиту [29]. Проте більшість таких систем орієнтовані на розподілені

хмарні середовища та складні корпоративні ландшафти, і не підходять для локальних або ізольованих розгортань.

Серед реалізацій, що демонструють спробу впровадження автономного захисту на рівні додатку, можна відзначити open-source проєкти типу Keycloak, OpenAM або Spring Security з власною підтримкою шифрування. Однак у них акцент зроблено переважно на автентифікації та авторизації, а не на ізольованому криптозахисті з динамічною зміною ключів.

Існуючі бібліотеки, як-от libsodium, OpenSSL, пропонують потужні інструменти для генерації ключів, але не забезпечують готових механізмів їх автоматичної ротації, фіксації в журналах безпеки та динамічного управління в обхід користувацького втручання [30].

У системах, орієнтованих на веб-додатки, також практично відсутні модулі, які дозволяють інтегрувати адаптивну зміну ключа під час роботи з REST API без потреби у зовнішньому KMS. Це створює серйозну проблему для систем, які мають працювати в умовах високої динаміки, з частими перевірками цілісності, вимогами до мінімізації часу обробки та потребою у внутрішній автономії захисту.

У наявних реалізаціях криптографічні ключі залишаються активними протягом тривалого періоду, що підвищує ризик їх компрометації. Навіть при використанні механізмів TLS, шифрування в реальному часі не забезпечується на рівні логіки застосунку – ключ фіксується на сесію, а не змінюється відповідно до кожного запиту або інциденту.

Дослідження показують, що статичні схеми шифрування вразливі до атак, які ґрунтуються на аналізі часу виконання операцій. Наприклад, атаки типу timing attack або cache-based side-channel дозволяють за певних умов відновити частину зашифрованих даних [31]. Це особливо небезпечно в системах з високочастотною взаємодією між клієнтом і сервером, де навіть кількасекундна компрометація здатна спричинити витік великого обсягу інформації.

У розробленій системі розв’язання цих проблем реалізовано шляхом генерації нового криптографічного ключа кожні 30 секунд, з автоматичним зберіганням в базі даних, обліком ротації та викликом захищених функцій

шифрування та дешифрування із перевіркою хешу. Такий підхід дозволяє адаптуватися до загрозового середовища в реальному часі, мінімізуючи ризики, пов’язані з фіксованими схемами та уразливістю на етапах передачі ключів.

У сучасних веб-додатках захист конфіденційних даних є одним із найважливіших аспектів інформаційної безпеки. Динамічне управління криптографічними ключами дозволяє забезпечити підвищений рівень безпеки шляхом регулярної ротації ключів, обмеження доступу до них та їхнього використання лише в конкретних контекстах .

Цей підхід допомагає мінімізувати ризики компрометації даних навіть у випадку порушення безпеки окремих компонентів системи. У цьому підрозділі буде проведено аналіз існуючих підходів і реалізацій, які використовуються для захисту веб-додатків із застосуванням динамічного управління ключами та шифрування в реальному часі.

Служба керування ключами AWS (AWS KMS) – це керована служба AWS, яка полегшує створення та керування ключами шифрування, які використовуються для шифрування ваших даних. Ключі AWS KMS, які ви створюєте в AWS KMS, захищені сертифікованими апаратними модулями безпеки (HSM) FIPS 140-2. Вони ніколи не залишають AWS KMS незашифрованим. Щоб використовувати свої KMS-ключі або керувати ними, ви взаємодієте з AWS KMS [9].

Коли ви шифруєте дані, вам потрібно захистити свій ключ шифрування. Якщо ви шифруєте свій ключ, вам потрібно захистити його ключ шифрування. Зрештою, ви повинні захистити ключ шифрування найвищого рівня (відомий як кореневий ключ) в ієрархії, яка захищає ваші дані. Саме тут на допомогу приходить AWS KMS.

AWS KMS захищає ваші кореневі ключі. Ключі KMS створюються, керуються, використовуються та видаляються повністю в AWS KMS. Вони ніколи не залишають сервіс в незашифрованому вигляді. Щоб використовувати свої KMS-ключі або керувати ними, ви викликаєте AWS KMS.

Крім того, ви можете створювати та керувати ключовими політиками в AWS KMS, гарантуючи, що лише довірені користувачі мають доступ до ключів

KMS.

Регіони AWS, у яких підтримується AWS KMS, перелічені в кінцевих точках та квотах служби керування ключами AWS. Якщо функція AWS KMS не підтримується в регіоні AWS, який підтримує AWS KMS, регіональна різниця описана в розділі про цю функцію.

Як і у випадку з іншими продуктами AWS, використання AWS KMS не вимагає контрактів або мінімальних покупок. Для отримання додаткової інформації про ціни на AWS KMS перегляньте статтю Ціни на послугу керування ключами AWS.

Служба керування ключами AWS підтримується угодою про рівень обслуговування, яка визначає нашу політику доступності послуг [].

Google Cloud KMS або Key Management Service – це хмарний сервіс для управління ключами шифрування для інших служб Google Cloud, які підприємства можуть використовувати для реалізації криптографічних функцій. Google Cloud Key Management Service (KMS) – це хмарна система керування ключами, яка дає змогу створювати, використовувати та керувати криптографічними ключами, а також безпечно виконувати криптографічні операції [9].

Коли дані зберігаються в Google Cloud, за умовчанням вони шифруються в стані спокою. Таким чином, коли користувачі використовують платформу Cloud Key Management Service (Cloud KMS), вони можуть отримати більший контроль над тим, як їхні дані шифруються в стані спокою та як керують ключами шифрування. KMS надає високобезпечне та масштабоване рішення для керування ключами, яке відповідає вимогам широкого спектру додатків та галузей. Він дає змогу створювати та використовувати ключі шифрування для хмарних служб і програм, а також забезпечує захист ваших даних під час зберігання та передачі [10].

Для джерела ключів Cloud KMS надає такі можливості:

– сервер програмного забезпечення Cloud KMS забезпечує гнучкість для шифрування даних за допомогою симетричного або асиметричного ключа, яким можна керувати;

– одулі безпеки хмарного обладнання (HSM);

– ключі шифрування, керовані клієнтом (CMEK): можливість вибору ключів, згенерованих Cloud KMS з іншими службами Google Cloud, і налаштування періоду ротації ключів;

– хмарний диспетчер зовнішніх ключів (EKM);

– ключі шифрування, надані клієнтом (CSEK).

Можливості KMS

1) Керування ключами: KMS дає змогу створювати, керувати та використовувати ключі шифрування для хмарних служб і програм.

2) Ротація ключів: KMS надає політику ротації ключів, яка допомагає регулярно змінювати ключі для підвищення безпеки.

3) Інтеграція: KMS інтегрується з іншими службами Google Cloud Platform, такими як Cloud Storage, BigQuery та Compute Engine.

4) Контроль доступу: KMS забезпечує детальний контроль доступу, дозволяючи контролювати, хто може створювати, використовувати та керувати ключами шифрування.

5) Журнал аудиту: KMS забезпечує ведення журналу аудиту, що дозволяє відстежувати використання ключів і забезпечувати дотримання політик безпеки [10].

Сервер хмарного програмного забезпечення KMS. Більшість KEK для шифрування фрагментів даних генеруються в Keystore, а решта генеруються всередині служб зберігання. Усі KEK генеруються за допомогою загальної криптографічної бібліотеки Google за допомогою генератора випадкових чисел (ГВЧ), створеного Google. Цей ГВЧ заснований на NIST 800–90Ar1 CTR-DRBG і генерує AES-256 KEK.

Використання KEK керується за допомогою списків керування доступом (ACL) у Keystore для кожного ключа з політикою для кожного ключа. Доступ до ключа мають лише авторизовані сервіси Google та користувачі. Використання кожного ключа відстежується на рівні окремої операції, для якої потрібен цей ключ – тому кожного разу, коли користувач використовує ключ, він автентифікується та реєструється. Доступ користувачів до всіх даних підлягає

аудиту в рамках загальної політики безпеки та конфіденційності Google.

Сховище ключів захищено кореневим ключем, який називається майстер-ключем сховища ключів, який обгортає всі KEK у сховищі ключів. Цей майстер-ключ сховища ключів є AES-256 і сам зберігається в іншій службі керування ключами, яка називається Root Keystore; Майстер-ключ сховища ключів зберігається в одноранговій інфраструктурі, яка називається дистрибутивом головних ключів кореневого сховища. Дистриб'ютор кореневого сховища ключів зберігає ключі в оперативній пам'яті лише на тих самих спеціалізованих комп'ютерах, що й Root Keystore, і використовує ведення журналу для перевірки належного використання [10].

DEK: Data Encryption Key – використовується для шифрування даних. KEK: Key Encryption Key – використовується для шифрування або обгортання ключа шифрування даних. Майстер-ключ KMS: ключ, який використовується для шифрування KEK. Цей ключ розподіляється в пам'яті. Майстер-ключ KMS створює резервні копії на апаратних пристроях і відповідає за шифрування ключів користувачів. Root KMS: це внутрішня служба керування ключами Google. Модулі безпеки хмарного обладнання (HSM)

Коли використовується Cloud HSM, дані строго ізольовані від інших клієнтів і служб Google Cloud. API площини даних Cloud HSM, який є частиною API служби керування хмарними ключами, дає змогу програмно керувати ключами, підтримуваними HSM.

Подібно до ключа з підтримкою Cloud HSM, він дає вам можливість створювати резервну копію CMEK за допомогою стороннього хмарного HSM. Це дійсно найкраще для клієнтів, які не довіряють Google Cloud.

Він дозволяє шифрувати ваші дані за допомогою CMEK так само, як і раніше, але щоразу, коли використовується цей ключ, він здійснює виклик API до іншого хмарного HSM для обробки шифрування. KEK зберігається в компанії HMS Third Party.

Приватні ключі не зберігаються в GCP. Клієнт створює обгорнутий ключ із закритим ключем клієнта та публічним ключем GCP, GCE/GCS. Для кожного запиту замовник надає ключ шифрування AES256 для доступу до даних.

Доступно для GCE або GCS (не можна використовувати для інших керованих служб GCP) [10].

Azure Key Vault – це хмарна служба для безпечного зберігання та доступу до секретних даних. Секрет – це все, до чого ви хочете жорстко контролювати доступ, наприклад, ключі API, паролі, сертифікати або криптографічні ключі. Служба Key Vault підтримує два типи контейнерів: сховища та пули керованих модулів безпеки апаратного забезпечення (HSM). Сховища підтримують зберігання програмного забезпечення та ключів, секретів і сертифікатів, захищених HSM. Керовані пули HSM підтримують лише ключі, підтримані HSM. Перегляньте огляд API Azure Key Vault REST для отримання повної інформації [11].

Щоб виконувати будь-які операції зі Сховищем ключів, спочатку потрібно пройти автентифікацію в ньому. Існує три способи автентифікації в Key Vault:

Принципал і секрет служби: хоча для автентифікації в Key Vault можна використовувати реєстраційний запис служби та секрет, ми не рекомендуємо цього робити. Важко автоматично повернути секрет початкового завантаження, який використовується для автентифікації в Key Vault.

Azure Key Vault використовує протокол Transport Layer Security (TLS) для захисту даних під час їх переміщення між сховищем ключів Azure Key Vault і клієнтами. Клієнти домовляються про підключення TLS до Azure Key Vault. TLS забезпечує надійну автентифікацію, конфіденційність і цілісність повідомлень (дозволяючи виявляти фальсифікацію, перехоплення та підробку повідомлень), сумісність, гнучкість алгоритмів і простоту розгортання та використання.

Perfect Forward Secrecy (PFS) захищає з'єднання між клієнтськими системами клієнтів і хмарними службами Microsoft за допомогою унікальних ключів. Для з'єднань також використовується 2 048-бітний ключ шифрування на основі RSA. Ця комбінація ускладнює перехоплення та доступ до даних, які передаються.

Скористайтеся наведеною нижче таблицею, щоб краще зрозуміти, як Key Vault може допомогти задовольнити потреби розробників і адміністраторів безпеки.

Будь-хто, хто має передплату на Azure, може створювати та використовувати сховища ключів. Хоча Key Vault приносить користь розробникам і адміністраторам безпеки, його може впроваджувати та керувати адміністратор організації, який керує іншими службами Azure. Наприклад, цей адміністратор може ввійти за допомогою передплати Azure, створити сховище для організації, у якому зберігатимуться ключі, а потім відповідати за такі операційні завдання:

– створити або імпортувати ключ або секрет;

– відкликати або видалити ключ або секрет;

– надавати користувачам або програмам доступ до сховища ключів, щоб вони потім могли керувати або використовувати його ключі та секретні дані;

– налаштувати використання ключів (наприклад, підписати або зашифрувати);

– слідкувати за використанням клавіш.

Потім цей адміністратор надає розробникам URI для виклику з їхніх програм. Цей адміністратор також надає інформацію журналу використання ключів адміністратору безпеки [11].

Нижче наведено порівняльну таблиці зі всіма існуючими програмами та з розроблювальною системою (табл. 1.1) :

Таблиця 1.1 – Порівняння рішень управління криптографічними ключами для захисту веб-додатків

Назва рішення	Динамічна ротація ключів	Підтримка real-time шифрування	Локальне розгортання	Аудит змін ключів
1	2	3	4	5
AWS KMS	Так (за розкладом)	Ні	Ні	Так
Google Cloud KMS	Так	Ні	Частково	Так
Azure Key Vault	Так	Ні	Так	Так

HashiCorp Vault	Так	Частково	Так	Так
OpenSSL (вручну)	Ні	Так (залежить від реалізації)	Так	Ні
Spring Security	Частково (через сесії)	Ні	Так	Частково
Розроблена система	Так (кожні 30 сек)	Так	Так	Так

На основі проведеного аналізу сформулюємо основні задачі, які необхідно вирішити в процесі розробки системи захисту веб-додатків з динамічним управлінням криптографічними ключами:

– розробити архітектуру програмного модуля, що дозволяє здійснювати генерацію, ротацію та зберігання криптографічних ключів у реальному часі;

– забезпечити можливість адаптивної зміни ключа у відповідь на потенційні загрози або інциденти безпеки;

– реалізувати внутрішню систему перевірки цілісності повідомлень під час взаємодії між клієнтом і сервером;

– розробити механізм ведення журналу подій, що фіксує кожну зміну ключа, операції шифрування та доступу;

– забезпечити незалежність від зовнішніх сервісів управління ключами (KMS) та відповідність вимогам до локальних і автономних середовищ;

– провести тестування продуктивності реалізованого рішення, зокрема перевірку впливу динамічної ротації ключів на затримки та стабільність веб-додатку.

1.4 Висновки до розділу 1

У першому розділі було здійснено аналіз сучасного стану методів і засобів захисту веб-додатків, зокрема в частині управління криптографічними ключами. Показано, що класичні схеми захисту базуються переважно на статичних або малорухомих механізмах шифрування, які залишаються недостатньо ефективними в умовах динамічного змінного загрозового середовища.

Основними обмеженнями таких підходів є довготривалий життєвий цикл ключів, відсутність ротації під час сесії, а також залежність від зовнішніх сервісів (KMS), що не дозволяє повною мірою контролювати захист у локальних реалізаціях.

На основі аналізу було встановлено, що більшість наявних рішень не передбачають адаптивної зміни криптографічних ключів у відповідь на інциденти безпеки, а також не мають механізмів перевірки цілісності інформації під час кожної взаємодії з API. Це створює передумови для реалізації атак, які можуть бути успішними навіть за наявності базового шифрування, зокрема таймінгових атак, атак повторного використання сесій або атак із компрометацією ключа доступу.

Було критично проаналізовано роботу зовнішніх систем типу AWS KMS, HashiCorp Vault, а також відкритих рішень, що працюють у межах Spring Security або OpenSSL. Попри високу гнучкість, ці засоби не вирішують проблему повного внутрішнього контролю за процесами генерації та ротації ключів у реальному часі. Як наслідок, актуальним стає створення власної, ізольованої, динамічної системи управління ключами з підтримкою автоматичного оновлення, шифрування, перевірки цілісності та аудиту.

Таким чином, за результатами теоретичного аналізу було обґрунтовано доцільність розробки системи захисту веб-додатків, у якій динамічне управління ключами реалізується незалежно від зовнішніх інфраструктур, із можливістю негайної реакції на зміну стану безпеки та забезпечення повної прозорості дій у системі. Отримані висновки стали підґрунтям для формування архітектури, описаної у наступному розділі.

2 УДОСКОНАЛЕННЯ МЕТОДІВ ЗАХИСТУ ВЕБ-ДОДАТКІВ З ДИНАМІЧНИМ УПРАВЛІННЯМ КРИПТОГРАФІЧНИМ КЛЮЧАМИ

Цей розділ присвячено розробці та обґрунтуванню удосконалень до існуючих методів захисту веб-додатків із використанням динамічного управління криптографічними ключами. Пояснюється, чому існуючі підходи не задовольняють сучасні вимоги безпеки, та пропонуються нові рішення для їх покращення. Описується структура бази даних, яка забезпечує зберігання і динамічне управління ключами, її властивості та інтеграція в загальну систему. Детально розглядаються розроблені алгоритми динамічного розподілу ключів і їх адаптація до реального часу, з поясненням механізмів шифрування. Розділ завершується описом інтеграції запропонованих рішень у веб-додаток та оцінкою очікуваних результатів щодо покращення безпеки.

2.1 Обґрунтування необхідності удосконалення системи захисту веб-додатків

Світ інформаційних технологій стрімко змінюється, що призводить до вдосконалення як самих веб-додатків, так і методів атак. Актуальність захисту прямо пов’язана із зростанням складності та частоти кіберзлочинів.

Сучасні веб-додатки обробляють та зберігають великі обсяги конфіденційної інформації, включно з персональними даними користувачів, фінансовими операціями та корпоративними секретами. Через це вони стають більш привабливими для кіберзлочинців.

Традиційні методи захисту часто не встигають адаптуватися до нових видів атак, таких як експлуатація нульового дня, складні багатовекторні атаки та атаки на криптографічні механізми (наприклад, квантові обчислення, які можуть загрожувати сучасним алгоритмам шифрування).

Використання ботів, штучного інтелекту та машинного навчання дає змогу зловмисникам швидше знаходити вразливості у веб-додатках.

Незважаючи на постійний розвиток технологій, у багатьох системах захисту все ще є критичні недоліки.

Застосування одного ключа для багатьох транзакцій збільшує ймовірність його компрометації. Якщо ключ буде викрадений, зловмисники отримають доступ до значної кількості даних. Більшість традиційних систем захисту не враховують динамічні зміни умов роботи, що робить їх менш ефективними у випадках нових загроз. Навіть добре захищені веб-додатки можуть бути вразливими до атак, які аналізують час виконання операцій для розкриття конфіденційної інформації.

Веб-додатки часто працюють у динамічному середовищі, де загрози можуть з'являтися миттєво. Наприклад, якщо зловмисник запускає DoS-атаку, система має реагувати негайно, щоб уникнути втрати доступності для легітимних користувачів.

У реальному часі особливо важливо забезпечити безпечну передачу даних між клієнтами та сервером, адже навіть кількасекундний компроміс може спричинити витік даних.

Багато сучасних стандартів, таких як ISO/IEC 27001, GDPR, або PCI DSS, вимагають впровадження сучасних механізмів захисту. Невідповідність цим стандартам може призводити до штрафів, втрати репутації та відтоку клієнтів. Зокрема:

Злам веб-додатка або компрометація даних може спричинити суттєві економічні втрати, включно з зниженням довіри з боку клієнтів та партнерів, судовими позовами через витік конфіденційної інформації, збільшенням витрат на відновлення систем після атаки.

Сучасні веб-додатки мають складну архітектуру, що включає хмарні сервіси, мікросервіси та API. Кожен з цих елементів може бути потенційною точкою атаки, що вимагає більш гнучких і адаптивних підходів до захисту. Традиційні методи, такі як фаєрволи чи статичні алгоритми шифрування, більше не здатні забезпечувати повну безпеку.

Поява квантових обчислень ставить під загрозу існуючі криптографічні алгоритми. Перехід до більш динамічних та адаптивних систем захисту забезпечить стійкість веб-додатків до майбутніх викликів.

Ці фактори створюють необхідність впровадження нових методів захисту,

таких як динамічний розподіл криптографічних ключів та управління ними в реальному часі, які значно підвищать рівень безпеки веб-додатків та забезпечать їхню відповідність сучасним викликам та стандартам.

Удосконалення за допомогою динамічного розподілу криптографічних ключів забезпечує автоматичне оновлення ключів під час кожної транзакції або після певного інтервалу часу. Це значно зменшує ризик компрометації ключа, навіть якщо зловмисник отримає доступ до поточного сеансу.

Захист від атак «людина посередині» (Man-in-the-Middle): Динамічне оновлення ускладнює перехоплення ключа та дешифрування даних.

Навіть у разі викрадення даних зловмисники не зможуть їх розшифрувати через постійну зміну ключів. Використання оптимізованих алгоритмів для генерації та розподілу ключів не вплине на швидкість роботи додатка.

Шифрування та управління ключами в реальному часі дозволяють миттєво реагувати на потенційні загрози. Наприклад, система може змінити ключі або змінити правила доступу, якщо виявлено підозрілу активність.

Здатність змінювати рівень захисту залежно від контексту (тип атаки, ризик компрометації). У разі атаки або компрометації система оперативно блокує вразливі транзакції. Автоматизація процесів зменшує вплив людського фактора.

Розробка системи із вбудованим механізмом вирівнювання часу виконання операцій приховує залежність обчислювального часу від вхідних даних. Це робить веб-додаток стійким до аналізу з боку зловмисників.

Очікувані результати включають в себе :

– виключення можливості розкриття конфіденційної інформації через аналіз часу;

– покращення безпеки алгоритмів шифрування: Додаткова затримка створює ще один рівень захисту.

Поведінкова аналітика дозволяє виявляти аномальні дії користувачів або зловмисників у системі, зокрема при розподілі ключів чи передачі даних.

Наприклад, система може виявити та припинити роботу зловмисника, який намагається здійснити несанкціонований доступу. Легітимні користувачі не

відчують затримок чи блокувань, оскільки аналіз відбувається у фоновому режимі.

Система зможе використовувати сучасні методи шифрування даних, включно з симетричними та асиметричними алгоритмами, з можливістю адаптації до нових стандартів, таких як постквантова криптографія.

Захист від майбутніх загроз, коли традиційні алгоритми стануть уразливими. Можливість вибору відповідних механізмів залежно від обсягу даних чи потреб користувача.

Автоматизація управління ключами та адаптивних механізмів захисту спрощує процеси підтримки безпеки для адміністративного персоналу.

Адміністратори можуть зосередитися на стратегічних завданнях замість рутинних перевірок. Автоматизовані процеси виключають суб’єктивні фактори.

Застосування динамічного шифрування та управління ключами дозволяє дотримуватися суворих вимог стандартів безпеки, таких як PCI DSS, HIPAA та ISO/IEC 27001.

Загалом удосконалення забезпечить всебічний захист веб-додатків, зробить їх більш стійкими до сучасних та майбутніх загроз, збільшить ефективність адміністрування, а також забезпечить відповідність новим стандартам і регламентам.

2.2 Проєктування бази даних для ефективного управління криптографічними ключами

Побудова безпечного та гнучкого механізму зберігання даних є одним із ключових завдань при створенні системи захисту веб-додатків. У контексті обраної теми особливу увагу приділено обробці службової інформації, а саме – збереженню криптографічних ключів, веденню журналу подій та забезпеченню цілісності дій користувачів. Ці компоненти вимагають централізованого зберігання, доступу з обмеженнями та фіксації у хронологічному порядку.

Для реалізації збереження даних у межах проєкту було розглянуто два потенційні підходи: використання класичної реляційної бази даних (наприклад, PostgreSQL) та документоорієнтованої бази (MongoDB [14]). Розглянемо

особливості обох підходів у контексті задачі.

PostgreSQL є потужною реляційною системою з підтримкою складних запитів, транзакцій, зовнішніх ключів і строгого контролю цілісності даних. Такий варіант є виправданим у випадках, коли модель даних є жорстко структурованою, має складні взаємозв'язки та суворі залежності між сутностями. У випадку з динамічними ключами, які швидко генеруються, обробляються й не мають прямої прив'язки до користувача чи транзакцій, ця модель виявилася надмірно громіздкою для поставленої задачі.

Натомість, MongoDB – документоорієнтована СКБД, яка дозволяє зберігати дані у вигляді гнучких структур (документів BSON/JSON), – ідеально відповідає вимогам системи захисту в реальному часі. Переваги MongoDB у даному випадку очевидні:

– гнучкість схеми зберігання: можна швидко змінювати структуру моделей без зміни всієї БД;

– висока швидкодія запису та читання, особливо для коротких транзакцій типу «запис логів» або «створення нового ключа»;

– просте масштабування у випадку збільшення обсягу записів;

– відсутність зайвих перевірок цілісності, які в цій системі не є критичними.

У системі реалізовано дві основні моделі документів – KeyEntry та LogEntry. Перша відповідає за збереження усіх згенерованих криптографічних ключів. Друга – за логування подій: як технічних (генерація ключів, обробка повідомлень), так і користувацьких (авторизація, розшифрування, виявлення порушення цілісності).

Модель KeyEntry містить три поля:

– key – симетричний криптографічний ключ у форматі base64;

– createdAt – час створення ключа;

– rotated – логічна ознака, що вказує, чи було оновлення ключа автоматичним (за розкладом) або ручним (у відповідь на інцидент).

Відповідна реалізація моделі виглядає так:

```
const mongoose = require('mongoose');
```

```
const keyEntrySchema = new mongoose.Schema({
  key: { type: String, required: true },
  createdAt: { type: Date, default: Date.now },
  rotated: { type: Boolean, default: false }
});
module.exports = mongoose.model('KeyEntry', keyEntrySchema);
```

Запис нового ключа відбувається під час кожної ротації (виконується кожні 30 секунд), а також при виявленні несанкціонованої зміни даних (порушення хешу). Функція генерації виглядає наступним чином:

```
const crypto = require('crypto');
const KeyEntry = require('./models/KeyEntry');
function generateAndSaveKey() {
  const key = crypto.randomBytes(32); // AES-256
  const newKey = new KeyEntry({
    key: key.toString('base64'),
    rotated: false
  });
  newKey.save().catch(err => {
    console.error('Помилка збереження ключа:', err);
  });
  return key;
}
```

Ключ після генерації одразу зберігається в базу, а також повертається функції, яка здійснює шифрування.

Протоколювання подій – ще один критичний компонент системи, реалізований через модель LogEntry. Вона фіксує всі ключові події: від спроб авторизації до дешифрування повідомлень і виявлення загроз. Кожна така подія фіксується з урахуванням дати, типу дії, ініціатора та детального опису.

Модель має наступну структуру:

```
const mongoose = require('mongoose');
const logEntrySchema = new mongoose.Schema({
  timestamp: { type: Date, default: Date.now },
  action: String,
  username: String,
  detail: String
});
module.exports = mongoose.model('LogEntry', logEntrySchema);
```

Поле action визначає тип події (LOGIN, ENCRYPT, DECRYPT, DECRYPT_FAIL тощо), username – користувача, який ініціював запит, а detail – короткий опис результату.

Усі дії в системі, пов'язані з ключами та журналюванням, реалізуються

через централізовану архітектуру, що включає REST API, базу даних та окремі модулі логування. Нижче представлена схема взаємодії між компонентами при записі логів та ключів :

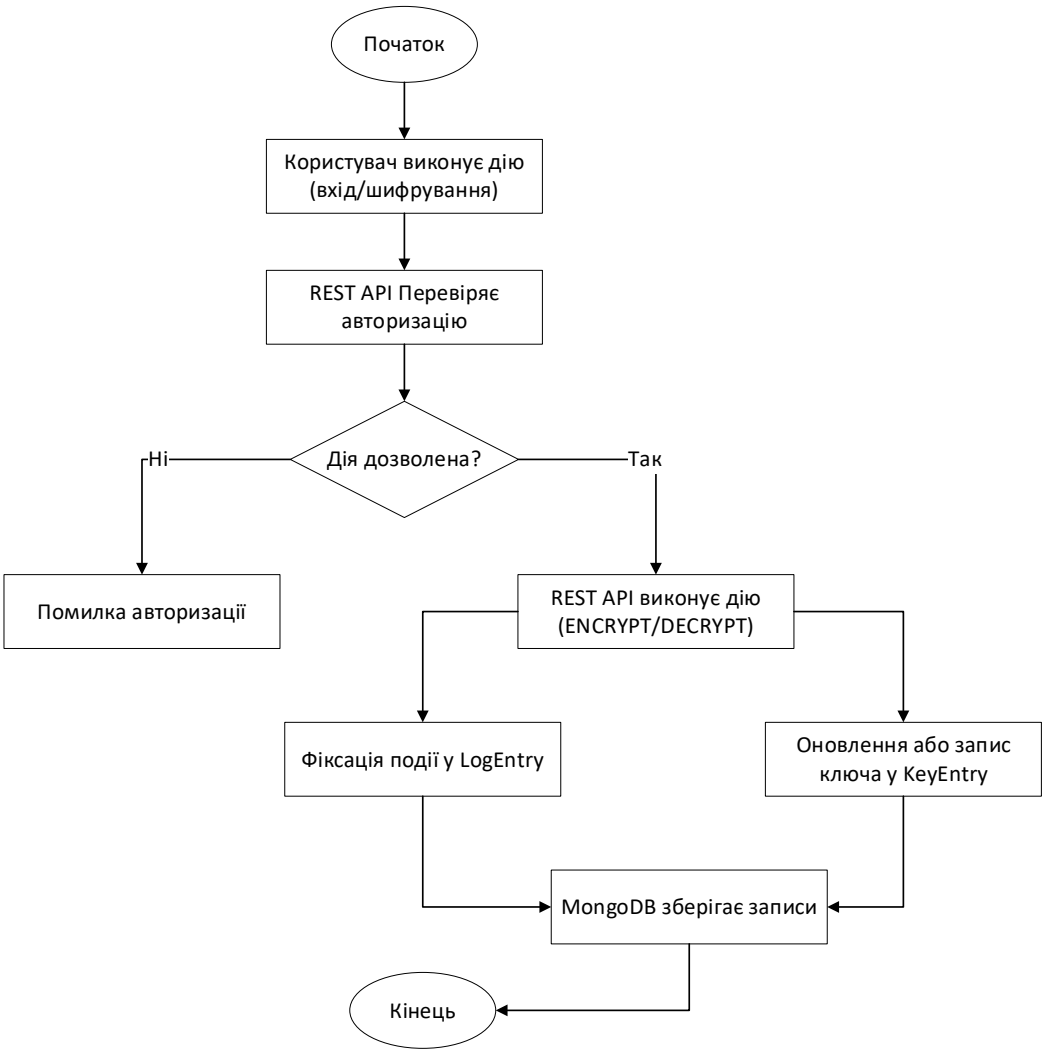


Рисунок 2.1 – Структура зберігання ключів і журналювання подій

Така реалізація забезпечує високу швидкодію та дозволяє системі масштабуватися при збільшенні кількості операцій і користувачів.

Логування подій реалізується централізовано, викликом функції logAction у відповідних API-обробниках. Приклад реалізації:

```
const LogEntry = require('./models/LogEntry');
function logAction(action, username, detail) {
  const entry = new LogEntry({
    action,
    username,
    detail
  });
  entry.save().catch(err => console.error('Помилка логування:', err));
}
```

Ця функція викликається з усіх критичних маршрутів – авторизації, шифрування, дешифрування. У результаті формується ланцюжок дій користувача, який дозволяє здійснювати аудит безпекових подій.

Між моделями KeyEntry і LogEntry відсутній жорсткий зв’язок (foreign key), однак їхній логічний зв’язок реалізується через синхронізацію по часу. Наприклад, запис у лог з дією ENCRYPT може бути зіставлений з активним ключем, що використовувався в цей момент, якщо зіставити час події з полем createdAt у колекції ключів.

Користувачі, взаємодіючи із системою, проходять авторизацію через маршрут /login, після чого отримують можливість використовувати функції /encrypt, /decrypt, /keys, /log. Усі ці дії ведуть до формування журналу подій, а також – за необхідності – викликають генерацію нового ключа.

Цей підхід дозволяє зберігати систему простою, гнучкою та розширюваною. У разі потреби зв’язки можуть бути деталізовані через додаткові поля, наприклад, keyId у логах, або через введення таблиці користувачів, якщо буде потрібно масштабування прав доступу.

Оскільки система працює з критичною інформацією – зокрема, з криптографічними ключами та журналами подій, – особливу увагу приділено питанням збереження цілісності даних, доступності та відновлення у випадку інцидентів.

MongoDB дозволяє реалізовувати рівень безпеки як на стороні доступу (через автентифікацію та контроль ролей), так і через використання окремих середовищ запуску для продуктивної версії системи. Додатково всі збережені ключі можуть бути зашифровані на рівні застосунку перед тим, як потрапляють у колекцію KeyEntry, а самі резервні копії архівуються та зберігаються за межами основного середовища.

Для MongoDB передбачено регулярне створення резервних копій через утиліту mongodump, яка дозволяє зробити бекап усієї бази або окремих колекцій. У проєкті також передбачена можливість автоматизації цього процесу за допомогою Node.js-скриптів [15], які створюють архів бази з шифруванням та записують його у хмарне сховище або локальну директорію адміністратора. Це

дозволяє відновити ключі та логи у разі відмови сервера.

Система побудована з урахуванням можливого зростання навантаження. MongoDB підтримує горизонтальне масштабування за допомогою шардування – розподілу даних між кількома вузлами. У разі перевищення навантаження на один сервер зберігання даних може бути рознесено за типами (ключі та логи на різних вузлах), або за часом створення (архівні дані окремо). Окрім цього, Node.js-платформа дозволяє реалізувати load balancing (балансування запитів) між кількома інстансами застосунку.

У межах підрозділу було здійснено повноцінне проєктування підсистеми зберігання даних для системи захисту веб-додатків. Вибір MongoDB як основної СКБД обґрунтований вимогами до швидкодії, гнучкості та простоти масштабування. Реалізовані моделі KeyEntry та LogEntry забезпечують збереження криптографічних ключів та аудит дій користувачів.

Архітектура зберігання даних дозволяє в режимі реального часу:

– зберігати та оновлювати ключі з високою частотою ротації;

– логувати дії користувачів для аудиту;

– масштабуватись при зростанні навантаження;

– виконувати резервне копіювання та забезпечувати відновлення;

– динамічно розширювати модель для нових вимог без ризику порушення роботи системи.

Таким чином, створена база даних повністю відповідає вимогам захисту інформації, є надійною та придатною для використання у продуктивному середовищі.

2.3 Розробка та опис алгоритмів динамічного розподілу і управління криптографічними ключами

У сучасних умовах забезпечення інформаційної безпеки веб-додатків потребує високої гнучкості, здатності до адаптації та оперативної реакції на зовнішні загрози. Особливо критичною є проблема захисту переданих даних від несанкціонованого доступу, підміни чи компрометації, що вимагає надійного криптографічного захисту. Використання фіксованих ключів у таких системах

поступово втрачає ефективність через ризики повторного використання, вичерпання ентропії або прямої компрометації. Саме тому реалізація динамічного управління криптографічними ключами – тобто їх ротація, оновлення та контроль – є одним з пріоритетних напрямів удосконалення засобів безпеки.

Розроблений у межах дипломної роботи програмний модуль базується на багатокомпонентній логіці, реалізованій на платформі Node.js з використанням архітектури REST API. Такий підхід забезпечує модульність, масштабованість і легкість у розгортанні захисного рішення в межах типового веб-додатку. Кожна взаємодія клієнта з сервером – авторизація, шифрування, дешифрування або доступ до журналів – виконується у вигляді HTTP-запиту, що обробляється окремим маршрутом. Відповідні контролери реалізують повну логіку перевірки облікових даних, вибору чи генерації ключа, виконання криптографічної операції, а також збереження результату в базі даних.

Основна логіка роботи системи представлена як лінійно-детермінований алгоритм, де кожна перевірка або дія виконується у суворій, наперед визначеній послідовності. Така структура дозволяє уникнути неоднозначності в логіці захисту, а також забезпечити повний аудит подій — від моменту ініціації запиту до формування відповіді. Особливу увагу приділено обробці помилкових ситуацій: спроба використання недійсного пароля, виявлення зміни повідомлення або порушення цілісності автоматично призводить до фіксації інциденту у журналі та ініціації додаткових захисних дій. Завдяки цьому система не лише забезпечує криптографічну стійкість, а й підтримує концепцію "живого" захисту, що реагує на загрозу в реальному часі.

Основна логіка роботи системи реалізована як детермінований алгоритм, де кожна операція або перевірка виконується у суворій послідовності. Такий підхід забезпечує передбачуваність поведінки системи, а також спрощує аудит і верифікацію результатів. Схема алгоритму представлена на рисунку 2.2 :

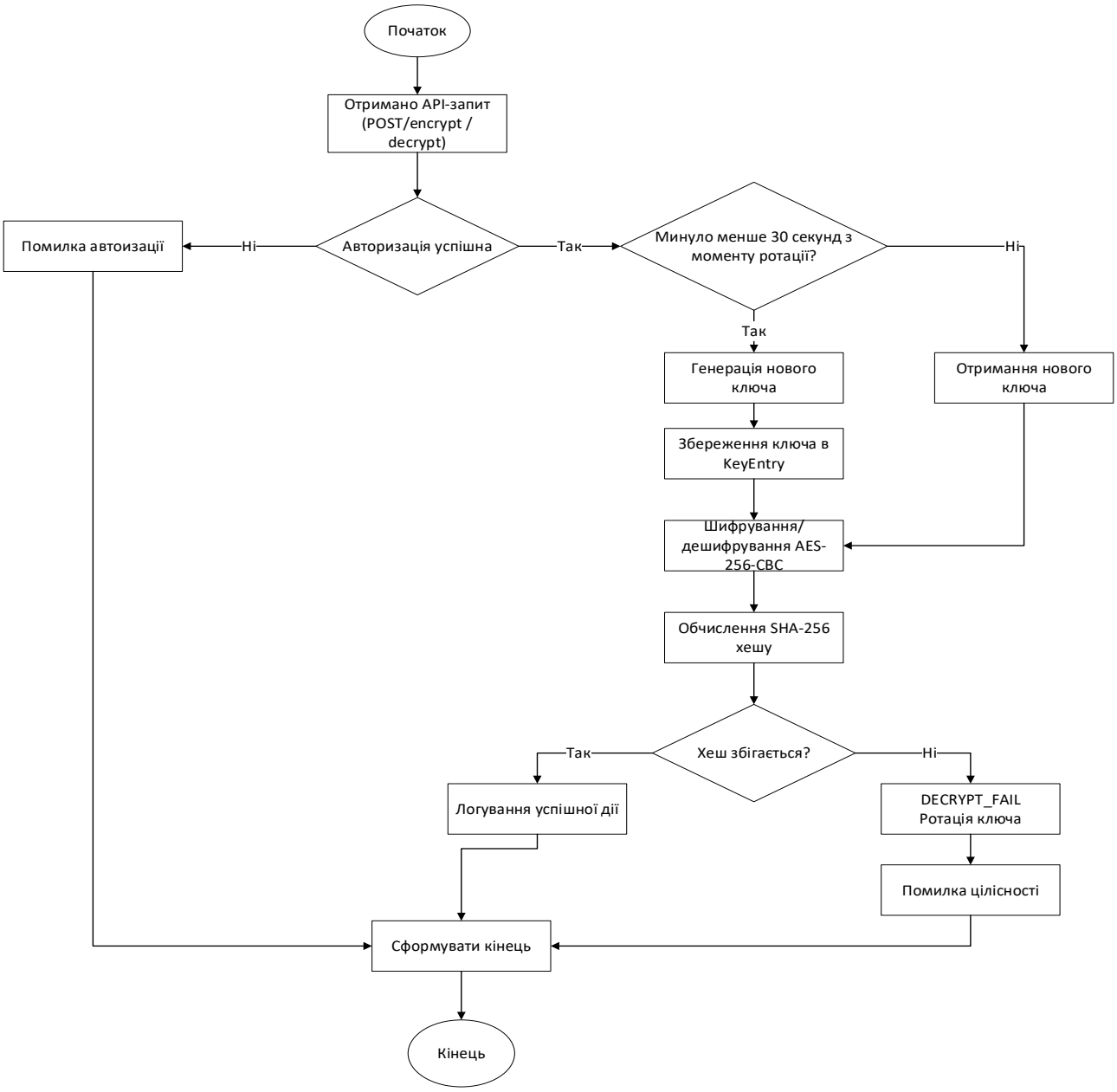


Рисунок 2.2 – Алгоритм обробки запиту з динамічною ротацією ключа

1. Ініціація запиту та автентифікація користувача.

Будь-яка взаємодія з захищеними функціями розпочинається із надсилання API-запиту, який містить облікові дані користувача (логін і пароль). Система перевіряє їх достовірність шляхом порівняння хешу пароля з попередньо збереженим значенням у базі (алгоритм SHA-256). Якщо перевірка не пройдена – відправляється помилка автентифікації (401 Unauthorized), що запобігає подальшій обробці запиту.

2. Аналіз типу запиту.

Після автентифікації система ідентифікує, яку саме дію необхідно виконати: шифрування або дешифрування. У залежності від типу запиту

обирається відповідний маршрут обробки.

3. Перевірка ротації ключа

Перед виконанням будь-яких криптографічних дій відбувається перевірка, чи пройшов встановлений час (30 секунд) з моменту останньої генерації ключа. Якщо так – система автоматично генерує новий симетричний ключ (довжина 256 біт), зберігає його у базу даних KeyEntry, та оновлює його значення у системній пам’яті.

Такий механізм дозволяє мінімізувати ризики, пов’язані з довготривалим використанням одного й того ж ключа, зокрема при атаках типу key reuse або brute-force.

4. Криптографічна обробка повідомлення.

У процесі шифрування:

– повідомлення шифрується за алгоритмом AES-256-CBC;

– генерується унікальний IV (ініціалізаційний вектор);

– результат кодується у формат base64;

– паралельно розраховується SHA-256 хеш повідомлення, що надалі використовується для перевірки цілісності.

Під час дешифрування система використовує переданий IV і ключ для розшифрування, після чого виконує валідацію цілісності через порівняння отриманого хешу з розрахованим.

5. Реакція на порушення цілісності.

У разі, якщо хеш не збігається, система трактує це як можливу атаку (наприклад, підміну повідомлення або спробу маніпуляції вмістом). У відповідь відбувається ручова ротація ключа (незалежно від таймера), а також запис події DECRYPT_FAIL до журналу подій.

Цей механізм дозволяє припинити використання потенційно скомпрометованого ключа ще до того, як зловмисник встигне повторно скористатися каналом.

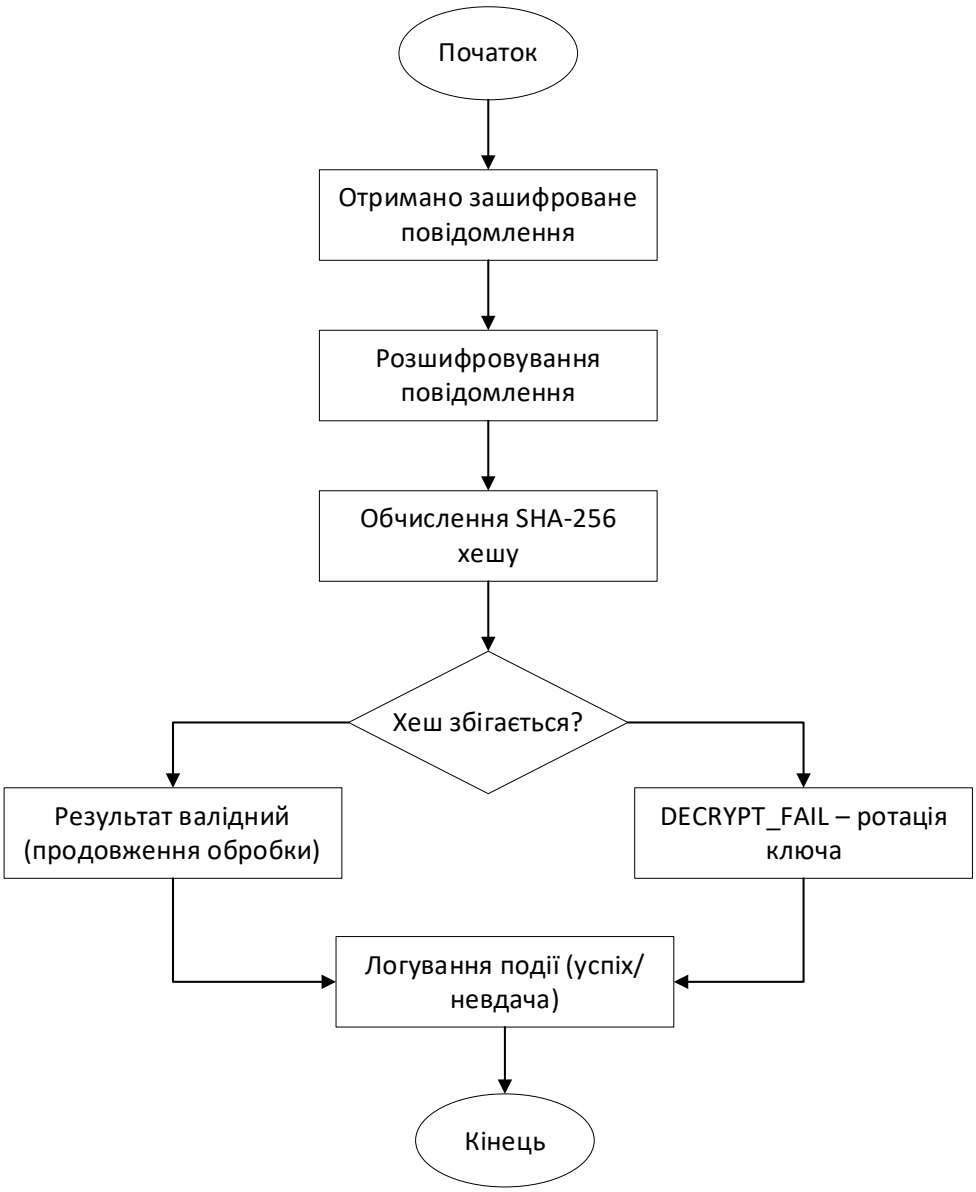


Рисунок 2.3 – Реакція системи на порушення цілісності повідомлення

6. Логування подій

Вся активність користувача – авторизація, криптографічні операції, помилки, зміна ключів – детально фіксується у базі LogEntry. Для кожної події записується:

– точна часова мітка;

– тип дії (LOGIN, ENCRYPT, DECRYPT, DECRYPT_FAIL);

– ім’я користувача;

– опис результату.

Логування реалізовано у вигляді централізованого механізму, викликом функції logAction, що дозволяє обробляти всі записи однаково.

Рисунок 2.4 – Блок-схема логування подій у системі

7. Формування відповіді

Після завершення усіх етапів користувач отримує відповідь у форматі JSON. У разі успішної обробки – результат шифрування або дешифрування, у разі помилки – відповідне повідомлення про неуспішність операції (із зазначенням типу помилки або коду).

8. Загальна архітектура інтеграції

Система побудована таким чином, щоб її можна було легко інтегрувати у будь-який веб-додаток. На рисунку 2.5 наведено архітектурну схему, яка демонструє взаємозв’язок між фронтендом, API, криптографічними модулями, модулями логування та базою даних.

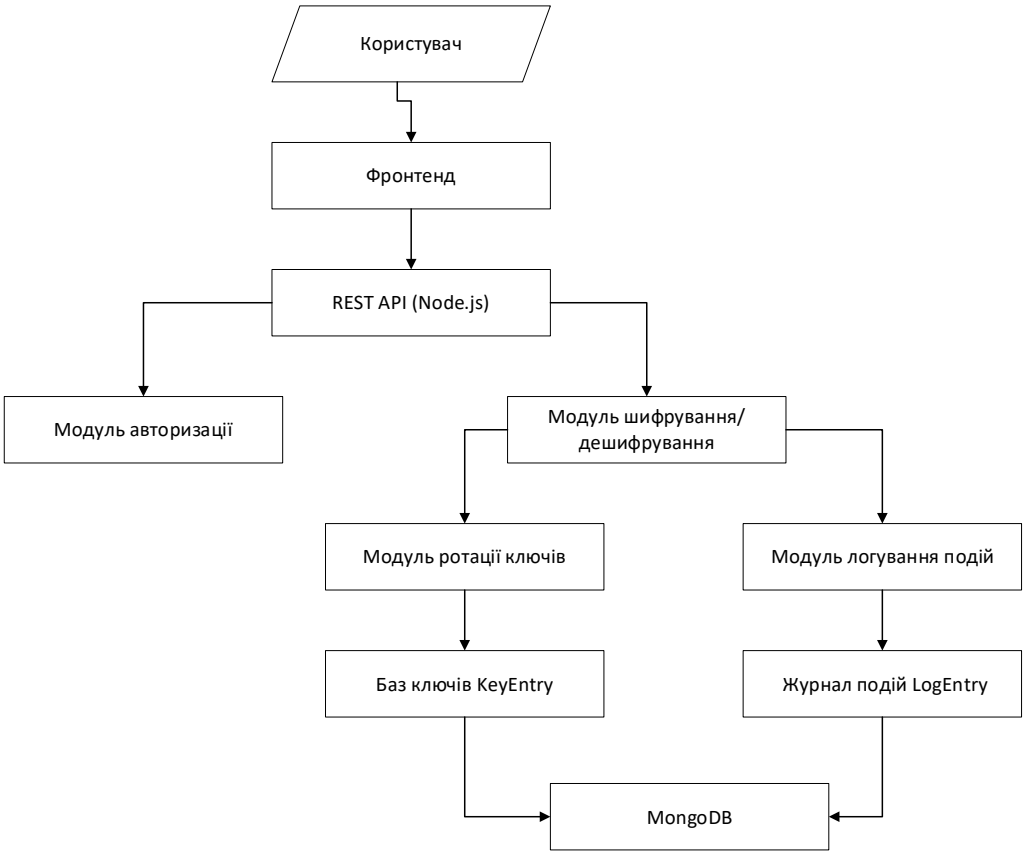


Рисунок 2.5 – Архітектура інтеграції системи захисту у веб-додаток

Запропонований алгоритм побудований з урахуванням принципів адаптивності, масштабованості та стійкості до атак. Завдяки модульному підходу, його легко інтегрувати у вже існуючі системи без суттєвого перепроєктування. Контроль кожного етапу – від авторизації до логування – дозволяє виявляти спроби атак ще на етапі ініціації.

2.4 Висновки до розділу 2

Таким чином, у межах підрозділу було реалізовано та детально описано алгоритм динамічного управління криптографічними ключами, що покладено в основу функціонування програмного модуля захисту веб-додатків. Запропонований підхід забезпечує не лише стандартне шифрування й дешифрування даних, але й включає адаптивні механізми реагування на інциденти безпеки, зокрема автоматичну або ручову ротацію ключів у випадку виявлення ознак компрометації.

Реалізований алгоритм є багатоетапним і охоплює повний цикл захисту даних: від перевірки автентичності користувача, через вибір або створення

ключа, до контролю цілісності та логування результатів. Інтеграція контрольної суми SHA-256 дозволяє виявляти спроби маніпуляцій або порушень цілісності повідомлення. При цьому система негайно реагує на такі загрози, виконуючи додаткову ротацію ключа та фіксуючи відповідну подію в журналі.

Блок-схеми, що супроводжують підрозділ, відображають ключові етапи роботи системи та демонструють її логічну завершеність. Запропонований алгоритм є гнучким, масштабованим та придатним до впровадження в реальні проєкти, з можливістю інтеграції в існуючі інформаційні системи без порушення їх структури. Його застосування дозволяє суттєво підвищити рівень криптографічної безпеки, зменшити ризики витоку інформації та забезпечити прозорий аудит усіх дій користувачів.

3 РЕАЛЗІАЦІЯ СИСТЕМИ ЗАХИСТУ ВЕБ-ДОДАТКІВ

У цьому розділі буде розглянуто практичну реалізацію системи захисту веб-додатків. Обґрунтовано вибір мови програмування та інструментів розробки. Наведено реалізацію алгоритму динамічного управління криптографічними ключами, описано ключові модулі системи. Проведено тестування основних функцій і проаналізовано результати.

3.1 Обґрунтування вибору мови програмування та програмних засобів

При створенні програмного модуля захисту веб-додатків з динамічним розподілом та управлінням криптографічними ключами було поставлено завдання реалізувати надійну, гнучку та продуктивну систему з можливістю обробки даних у реальному часі. Виходячи з цих вимог, було обґрунтовано вибір відповідних мов програмування, інструментів розробки, середовищ виконання та базових технологій, що забезпечили реалізацію поставлених цілей.

Основним інструментом реалізації серверної логіки обрано мову програмування JavaScript [16], з використанням середовища виконання Node.js.

Такий вибір пояснюється кількома факторами:

– універсальність та асинхронність Node.js, що дозволяє обробляти одночасні запити в режимі реального часу з мінімальними затримками;

– великий екосистемний супровід, наявність розвиненої спільноти та безлічі готових бібліотек, які прискорюють розробку;

– пряма сумісність з JSON, що є ключовим форматом для обміну даними між клієнтом і сервером та використовується у базі даних MongoDB.

Мова JavaScript є динамічно типізованою, що дозволило швидко реалізувати логіку системи без надмірної бюрократизації структури даних. Водночас, завдяки підтримці модульної структури у Node.js, кожен елемент системи (модулі авторизації, шифрування, логування, роботи з ключами) було реалізовано як окремий компонент, що полегшує розширення і тестування.

Для реалізації шифрування даних у системі використано вбудований модуль crypto, який входить до стандартної бібліотеки Node.js. Це дозволило

уникнути застосування сторонніх криптографічних бібліотек, що може бути небезпечним у контексті довіри до реалізації алгоритмів. Алгоритм AES-256-CBC [17] згенерований через crypto.randomBytes() застосовується для симетричного шифрування, що забезпечує достатній рівень стійкості до атак навіть у відкритому середовищі.

Крім того, хешування повідомлень для перевірки їх цілісності здійснюється за допомогою алгоритму SHA-256 [18], який також реалізований через crypto. Це дозволяє мінімізувати ризики порушення цілісності при передачі або зберіганні даних.

Зберігання службової інформації (криптографічних ключів, журналів подій) реалізовано на базі MongoDB – документоорієнтованої системи керування базами даних, яка підтримує збереження структурованих об’єктів у форматі BSON/JSON. Для роботи з базою даних у середовищі Node.js використано бібліотеку Mongoose, яка забезпечує високий рівень абстракції при створенні моделей, автоматичну валідацію даних і зручну реалізацію CRUD-операцій.

MongoDB дозволила реалізувати такі функції, як:

– збереження згенерованих ключів у колекції KeyEntry;

– запис кожної дії користувача у LogEntry;

– формування API-запитів до журналу подій та списку ключів;

– масштабованість та можливість розподіленого зберігання.

Для тестування та перевірки запитів до API використовується Postman [13]– середовище для роботи з HTTP-запитами, яке дозволяє швидко перевірити коректність роботи маршрутів /login, /encrypt, /decrypt, /keys, /log. З його допомогою проводились як базові тести функціонування, так і перевірка обробки помилок (наприклад, неправильна авторизація чи порушення хешу повідомлення).

Оскільки проєкт реалізовано як повноцінний серверний застосунок, для його структурування було використано стандартну модульну архітектуру Node.js. Усі функціональні частини реалізовані у вигляді окремих файлів та папок, що відповідають за конкретні аспекти системи:

– server.js – головний файл, що відповідає за запуск сервера, ініціалізацію

підключення до бази даних, реєстрацію маршрутів API та запуск ротації ключів;

– cryptoModule.js – модуль, що відповідає за шифрування, розшифрування та перевірку хешу повідомлень. Реалізовано за допомогою вбудованого crypto;

– keyManager.js – окремий файл, який забезпечує логіку автоматичної та ручної ротації ключів;

– auth.js – базовий модуль для авторизації користувача з порівнянням логіна і пароля (з хешем);

– logger.js – відповідає за фіксацію всіх подій у системі у форматі LogEntry;

– models/ – директорія з описом Mongoose-схем для роботи з MongoDB (KeyEntry.js, LogEntry.js);

– db.js – конфігурація підключення до MongoDB.

Такий підхід дозволяє підтримувати чітке розділення обов'язків (separation of concerns), що є критичним для подальшого розширення або рефакторингу системи.

Для керування залежностями використовується npm – стандартний пакетний менеджер для середовища Node.js. У файлі package.json вказані всі бібліотеки, які використовуються у проєкті, зокрема:

– express – для побудови HTTP-сервера та маршрутизації;

– mongoose – для роботи з MongoDB;

– body-parser – для обробки вхідних JSON-запитів;

– crypto – вбудований модуль Node.js, що не потребує додаткової установки.

Усі ці інструменти забезпечують не лише продуктивність і масштабованість, а й відкритість технологій: система не залежить від закритих або платних сервісів і може бути розгорнута на будь-якому сервері, що підтримує Node.js та MongoDB.

У середовищі розробки використовувалася Visual Studio Code [19], яке надає зручні засоби автодоповнення, підсвітки синтаксису, інтеграції з терміналом, а також підтримку Git [20]. Це дозволило ефективно працювати з кодом, швидко переходити між модулями, тестувати запити та спостерігати за

логами у режимі реального часу.

Таким чином, вибраний стек технологій дозволив реалізувати систему, яка:

– швидко реагує на запити;

– безпечно обробляє дані за допомогою стандартних криптографічних алгоритмів;

– зручно масштабовується;

– дозволяє вести централізований журнал подій;

– має відкрите середовище для модифікацій та адаптації під нові потреби.

Крім технічних переваг, обрані програмні засоби мають важливе стратегічне значення для майбутньої підтримки та розвитку системи. Всі використані технології є відкритими, активно підтримуються розробницькими спільнотами та не вимагають ліцензування, що зменшує залежність проєкту від конкретного виробника ПЗ.

Node.js та MongoDB, як частина сучасного стеку JavaScript-технологій, дозволяють швидко адаптувати розробку до змін у вимогах. Завдяки підтримці модульної архітектури та гнучкій структурі даних система легко розширюється – як у напрямку інтеграції нових функціональних модулів, так і в частині взаємодії з іншими сервісами або мікросервісами.

Крім того, з точки зору безпеки, використання перевірених алгоритмів та стандартних бібліотек (crypto, express, mongoose) є кращим підходом у порівнянні з самописними або нестандартними реалізаціями, які можуть містити помилки або вразливості. Такий вибір значно підвищує рівень довіри до системи та дозволяє уникати несертифікованих методів захисту.

Загалом, використані інструменти забезпечили не лише реалізацію функціональних вимог до системи, але й заклали підґрунтя для її подальшого розвитку, масштабування та інтеграції в інші середовища.

Вибір мови програмування JavaScript, серверного середовища Node.js, бази даних MongoDB та супутніх інструментів був повністю обґрунтований поставленими вимогами до системи. Вони забезпечили реалізацію безпечної, гнучкої та масштабованої архітектури, яка підтримує динамічне управління криптографічними ключами, шифрування, логування та аудит дій користувачів.

Таким чином, підрозділ сформував технологічну основу реалізації системи, що описана в наступних частинах роботи.

3.2 Реалізація алгоритму динамічного розподілу і управління криптографічними ключами

Розроблений у попередньому розділі алгоритм управління криптографічними ключами реалізовано у вигляді модульного програмного рішення, що функціонує на платформі Node.js. Алгоритм передбачає автоматичну генерацію симетричних ключів, їх збереження у базі даних, можливість ручового оновлення в разі виявлення загрози, а також динамічну взаємодію із модулями шифрування, розшифрування та журналювання.

В основі реалізації лежить принцип централізованого керування ключем, який у кожен момент часу зберігається в оперативній пам’яті як активний. Ключ оновлюється відповідно до заданого інтервалу або у разі виявлення порушення цілісності повідомлення. Це дозволяє не лише захистити інформацію під час передачі, а й забезпечити обмежений життєвий цикл ключа.

Основну логіку управління ключами зосереджено у файлі keyManager.js, де реалізовано функції генерації нового ключа, ініціації автоматичної ротації та можливості ручового оновлення ключа у відповідь на інцидент.

Функція генерації нового ключа базується на стандартному криптографічному API Node.js – модулі crypto. Для генерації використовується функція crypto.randomBytes(32), що забезпечує створення 256-бітного ключа, придатного для алгоритму AES-256.

Приклад генерації :

```
const crypto = require('crypto');
function generateKey() {
  return crypto.randomBytes(32);
}
```

Одержаний ключ одразу кодується у формат base64 та зберігається в базу даних для подальшого аудиту. Модель ключа (KeyEntry) передбачає фіксацію часу створення та ознаки, чи була ротація виконана вручну:

```
const mongoose = require('mongoose');
const keyEntrySchema = new mongoose.Schema({
```

```
  key: { type: String, required: true },
  createdAt: { type: Date, default: Date.now },
  rotated: { type: Boolean, default: false }
});
module.exports = mongoose.model('KeyEntry', keyEntrySchema);
```

Збереження нового ключа в базу та оновлення активного значення в оперативній пам’яті реалізується так:

```
const KeyEntry = require('./models/KeyEntry');
let currentKey = generateKey();
function startKeyRotation(intervalMs = 30000) {
  setInterval(() => {
    currentKey = generateKey();
    const keyRecord = new KeyEntry({
      key: currentKey.toString('base64'),
      rotated: false
    });
    keyRecord.save();
  }, intervalMs);
}
```

Таким чином, система постійно підтримує актуальний ключ у пам’яті для шифрування/дешифрування, зберігаючи його історію в базі даних MongoDB. Інтервал ротації задається у мілісекундах та легко змінюється в разі потреби.

Крім автоматичної генерації, реалізовано механізм ручової ротації ключа, який викликається у разі виявлення загрози – наприклад, коли перевірка хешу повідомлення вказує на порушення цілісності:

```
function forceRotateKey() {
  currentKey = generateKey();
  const keyRecord = new KeyEntry({
    key: currentKey.toString('base64'),
    rotated: true
  });
  keyRecord.save();
}
```

Ця функція дає змогу системі оперативно реагувати на потенційні загрози без втрати працездатності або залучення адміністратора.

Шифрування даних реалізовано в окремому файлі cryptoModule.js. Система використовує алгоритм AES-256-CBC з випадковим вектором ініціалізації (IV), що генерується для кожного повідомлення. Такий підхід унеможливлює повторне використання того самого шифрованого блоку, навіть якщо вхідні дані збігаються.

Перед шифруванням повідомлення також обчислюється його хеш за алгоритмом SHA-256. Це дозволяє під час розшифрування перевірити, чи не було змінено вміст – навіть частково. Таким чином, система реалізує базовий механізм контролю цілісності переданих даних.

Операція шифрування реалізується наступним чином:

```
const crypto = require('crypto');
function encrypt(plaintext, key) {
  const iv = crypto.randomBytes(16);
  const cipher = crypto.createCipheriv('aes-256-cbc', key, iv);
  const encrypted = Buffer.concat([
    cipher.update(plaintext, 'utf8'),
    cipher.final()
  ]);
  const hash = crypto.createHash('sha256').update(plaintext).digest('base64');
  return {
    encryptedData: encrypted.toString('base64'),
    iv: iv.toString('base64'),
    hash
  };
}
```

Цей фрагмент створює новий IV, шифрує текст і розраховує хеш. Усі результати повертаються у форматі base64, що зручно для передачі JSON-запитом.

Розшифрування виконується у зворотному порядку: повідомлення декодується, порівнюється хеш, і лише тоді, якщо перевірка успішна, повертається текст.

```
function decrypt(encryptedData, key, iv, expectedHash) {
  const decipher = crypto.createDecipheriv('aes-256-cbc', key, Buffer.from(iv, 'base64'));
  const decrypted = Buffer.concat([
    decipher.update(Buffer.from(encryptedData, 'base64')),
    decipher.final()
  ]);
  const decryptedText = decrypted.toString('utf8');
  const actualHash = crypto.createHash('sha256').update(decryptedText).digest('base64');
  if (actualHash !== expectedHash) {
    throw new Error('Integrity check failed');
  }
  return decryptedText;
}
```

Таким чином, реалізовано повноцінну двосторонню обробку повідомлень, що гарантує збереження конфіденційності та цілісності.

Шифрування та дешифрування відбуваються через виклики API /encrypt та /decrypt, відповідно. Перед цим кожен користувач має пройти авторизацію через POST /login, після чого отримує доступ до решти маршрутів.

Маршрут /encrypt очікує в тілі запиту такі параметри:

– username, password – для перевірки автентичності;

– message – текст, який потрібно зашифрувати.

Після успішної перевірки автентичності викликається функція encrypt(), а результат (зашифрований текст, IV і хеш) передається у відповідь користувачу.

Приклад обробки запиту:

```
app.post('/encrypt', (req, res) => {
  const { username, password, message } = req.body;
  if (!authenticateUser(username, password)) {
    return res.status(401).json({ error: 'Аутентифікація не пройдена' });
  }
  const key = getCurrentKey();
  const result = encrypt(message, key);
  logAction('ENCRYPT', username, `Зашифровано повідомлення: "${message}"`);
  res.status(200).json(result);
});
```

У випадку розшифрування, маршрут /decrypt виконує аналогічну перевірку. Якщо хеш не збігається – викликається ручова ротація ключа, а подія фіксується у журналі.

```
app.post('/decrypt', (req, res) => {
  const { username, password, encryptedData, iv, hash } = req.body;
  if (!authenticateUser(username, password)) {
    return res.status(401).json({ error: 'Аутентифікація не пройдена' });
  }
  try {
    const key = getCurrentKey();
    const decrypted = decrypt(encryptedData, key, iv, hash);
    logAction('DECRYPT', username, 'Успішне розшифрування повідомлення');
    return res.status(200).json({ decrypted });
  } catch (err) {
    forceRotateKey(); // Відповідь на загрозу
    logAction('DECRYPT_FAIL', username, '❌ Цілісність порушено або дані змінено');
    return res.status(400).json({ error: 'Цілісність порушено або неправильні дані' });
  }
});
```

Такий підхід забезпечує не лише обробку запиту, а й автоматичне реагування на загрозу, з негайною зміною криптографічного ключа.

Шифрування даних реалізовано в окремому файлі cryptoModule.js. Система використовує алгоритм AES-256-CBC з випадковим вектором ініціалізації (IV), що генерується для кожного повідомлення. Такий підхід унеможливлює повторне використання того самого шифрованого блоку, навіть якщо вхідні дані збігаються.

Перед шифруванням повідомлення також обчислюється його хеш за алгоритмом SHA-256. Це дозволяє під час розшифрування перевірити, чи не було змінено вміст – навіть частково. Таким чином, система реалізує базовий механізм контролю цілісності переданих даних.

Операція шифрування реалізується наступним чином:

```
const crypto = require('crypto');
function encrypt(plaintext, key) {
  const iv = crypto.randomBytes(16);
  const cipher = crypto.createCipheriv('aes-256-cbc', key, iv);
  const encrypted = Buffer.concat([
    cipher.update(plaintext, 'utf8'),
    cipher.final()
  ]);
  const hash = crypto.createHash('sha256').update(plaintext).digest('base64');
  return {
    encryptedData: encrypted.toString('base64'),
    iv: iv.toString('base64'),
    hash
  };
}
```

Цей фрагмент створює новий IV, шифрує текст і розраховує хеш. Усі результати повертаються у форматі base64, що зручно для передачі JSON-запитом.

Розшифрування виконується у зворотному порядку: повідомлення декодується, порівнюється хеш, і лише тоді, якщо перевірка успішна, повертається текст.

```
function decrypt(encryptedData, key, iv, expectedHash) {
  const decipher = crypto.createDecipheriv('aes-256-cbc', key, Buffer.from(iv, 'base64'));
  const decrypted = Buffer.concat([
    decipher.update(Buffer.from(encryptedData, 'base64')),
    decipher.final()
  ]);
  const decryptedText = decrypted.toString('utf8');
  const actualHash = crypto.createHash('sha256').update(decryptedText).digest('base64');
  if (actualHash !== expectedHash) {
```

```
    throw new Error('Integrity check failed');
  }
  return decryptedText;
}
```

Таким чином, реалізовано повноцінну двосторонню обробку повідомлень, що гарантує збереження конфіденційності та цілісності.

Шифрування та дешифрування відбуваються через виклики API /encrypt та /decrypt, відповідно. Перед цим кожен користувач має пройти авторизацію через POST /login, після чого отримує доступ до решти маршрутів.

Маршрут /encrypt очікує в тілі запиту такі параметри:

– username, password – для перевірки автентичності;

– message – текст, який потрібно зашифрувати.

Після успішної перевірки автентичності викликається функція encrypt(), а результат (зашифрований текст, IV і хеш) передається у відповідь користувачу. Приклад обробки запиту:

```
app.post('/encrypt', (req, res) => {
  const { username, password, message } = req.body;
  if (!authenticateUser(username, password)) {
    return res.status(401).json({ error: 'Аутентифікація не пройдена' });
  }
  const key = getCurrentKey();
  const result = encrypt(message, key);
  logAction('ENCRYPT', username, `Зашифровано повідомлення: "${message}"`);
  res.status(200).json(result);
});
```

У випадку розшифрування, маршрут /decrypt виконує аналогічну перевірку. Якщо хеш не збігається – викликається ручова ротація ключа, а подія фіксується у журналі.

```
app.post('/decrypt', (req, res) => {
  const { username, password, encryptedData, iv, hash } = req.body;
  if (!authenticateUser(username, password)) {
    return res.status(401).json({ error: 'Аутентифікація не пройдена' });
  }
  try {
    const key = getCurrentKey();
    const decrypted = decrypt(encryptedData, key, iv, hash);
    logAction('DECRYPT', username, 'Успішне розшифрування повідомлення');
    return res.status(200).json({ decrypted });
  } catch (err) {
    forceRotateKey(); // Відповідь на загрозу
```

```
    logAction('DECRYPT_FAIL', username, '❌ Цілісність порушено або дані змінено');
    return res.status(400).json({ error: 'Цілісність порушено або неправильні дані' });
  }
});
```

Такий підхід забезпечує не лише обробку запиту, а й автоматичне реагування на загрозу, з негайною зміною криптографічного ключа.

Реалізована система не лише відповідає поточним вимогам щодо безпеки інформації у веб-середовищі, але й створює технічну основу для подальшого розвитку програмного модуля, зокрема впровадження багатофакторної автентифікації, розподіленого керування ключами та інтеграції з зовнішніми інформаційними системами. Отримані результати реалізації є підставою для подальшого тестування програмного засобу, що буде розглянуто в наступному підрозділі.

3.3 Тестування програмного засобу

Тестування маршруту /login. Для перевірки модуля автентифікації користувачів було здійснено серію запитів до маршруту /login. На першому етапі на сервер передавались коректні облікові дані адміністратора. У результаті система виконала успішну перевірку, підтвердила авторизацію та повернула повідомлення про успішний вхід, що підтверджує правильність реалізації логіки перевірки.

На другому етапі тестування було змодельовано ситуацію, коли користувач вводить неправильний пароль. У відповідь система повернула повідомлення про помилку з відповідним HTTP-кодом 401 (Unauthorized), не допустивши доступу до закритих маршрутів.

Також було перевірено реакцію системи на відсутність одного з параметрів у запиті – наприклад, відсутність поля password. У цьому випадку сервер належним чином обробив виняток, надавши повідомлення про помилку авторизації.

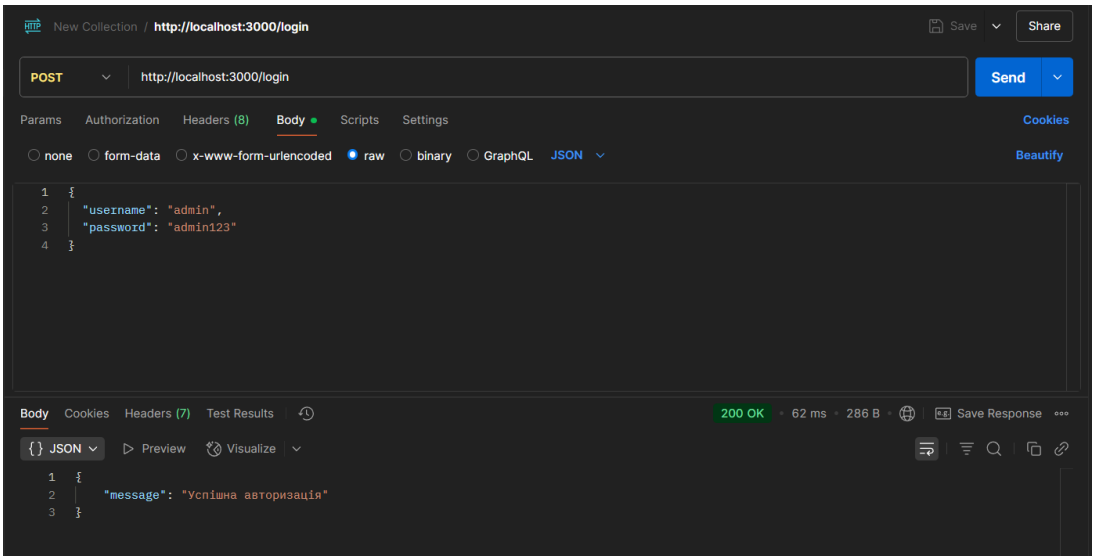


Рисунок 3.1 – Успішна авторизація користувача з правильними обліковими даними

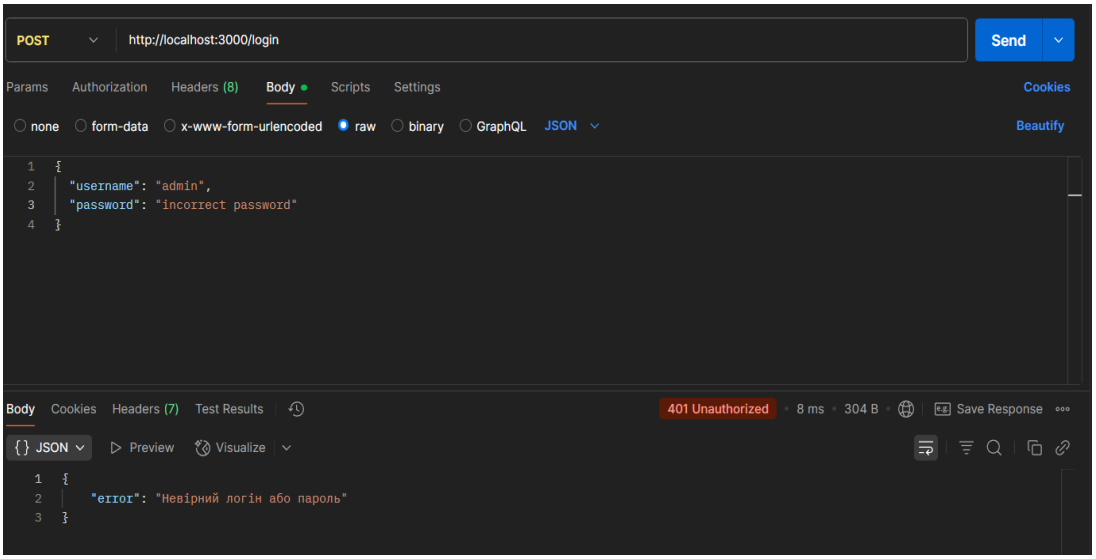


Рисунок 3.2 – Відповідь системи при неправильному паролі користувача

Тестування маршруту /encrypt. Метою тестування маршруту /encrypt було перевірити здатність системи шифрувати передане повідомлення за допомогою активного криптографічного ключа, сформованого відповідно до алгоритму, реалізованого в підрозділі 3.2.

На початковому етапі тестування було виконано авторизований POST-запит до маршруту /encrypt, у тілі якого передавалось коротке текстове повідомлення. За результатами запиту система повернула зашифроване повідомлення у форматі base64, окремо згенерований вектор ініціалізації (IV) та хеш, створений на основі оригінального тексту. Це підтверджує коректність

роботи модуля шифрування та відповідність вихідної структури запиту очікуваному формату.

Додатково було перевірено поведінку системи у випадку відсутності авторизації. При навмисному введенні неправильного пароля або повній відсутності даних аутентифікації система повертала відповідь із помилкою (401 Unauthorized), при цьому шифрування не виконувалось.

Також було здійснено серію повторюваних запитів з ідентичним повідомленням. Кожного разу шифрований результат мав інший вигляд завдяки новому IV, що свідчить про дотримання вимог до криптостійкості.

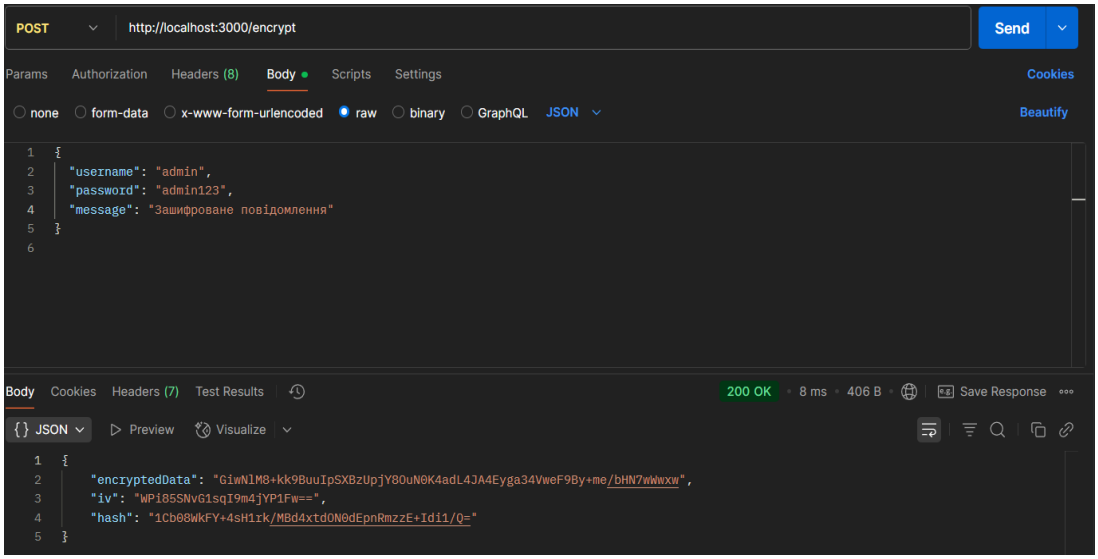


Рисунок 3.3 – Успішне шифрування повідомлення з поверненням IV і хешу

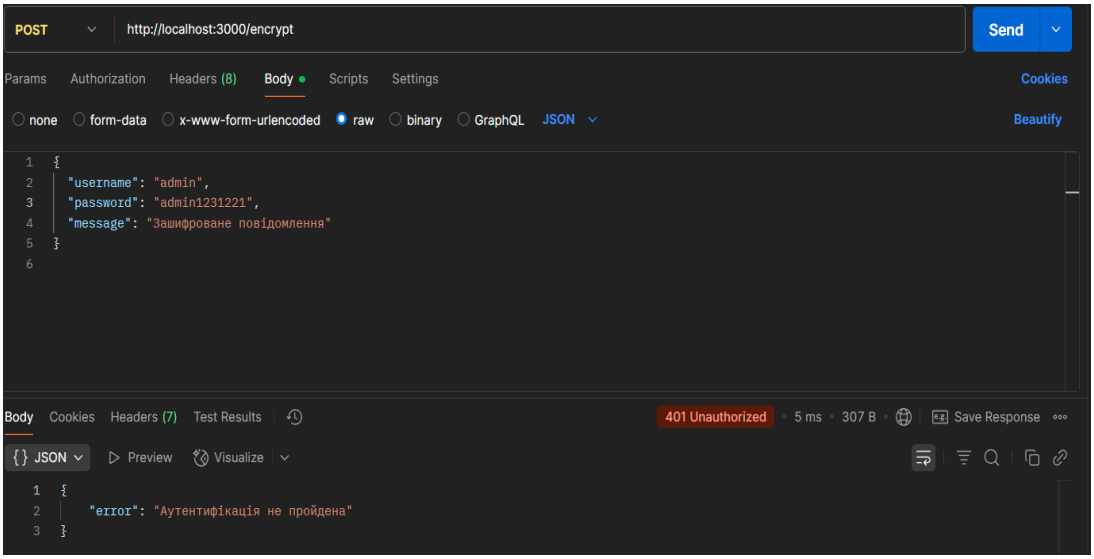


Рисунок 3.4 – Відповідь системи при відсутності авторизації під час спроби шифрування

Тестування маршруту /decrypt. Наступним етапом тестування була перевірка маршруту /decrypt, який відповідає за розшифрування раніше зашифрованого повідомлення з використанням активного ключа, вектора ініціалізації та контрольного хешу.

Перший тест був здійснений на основі даних, отриманих після успішного шифрування. Система прийняла IV, зашифроване повідомлення та хеш, після чого успішно відновила оригінальний текст. Це підтверджує правильність реалізації симетричного шифрування за алгоритмом AES-256-CBC та коректність порівняння хешів.

На другому етапі було змодельовано зміну одного байта у зашифрованому повідомленні. Система виявила невідповідність контрольної суми та повідомила про порушення цілісності даних. Як реакція на інцидент, було автоматично виконано ручову ротацію ключа, що відображено в журналі дій із міткою DECRYPT_FAIL. Така поведінка підтвердила наявність вбудованого захисту від спроб маніпуляції з шифрованими даними.

Також була перевірена реакція системи на відсутність одного з параметрів запиту – наприклад, невказаний хеш або неправильний формат IV. У всіх випадках сервер коректно обробляв помилку та повертав відповідне повідомлення, не допускаючи виконання розшифрування.

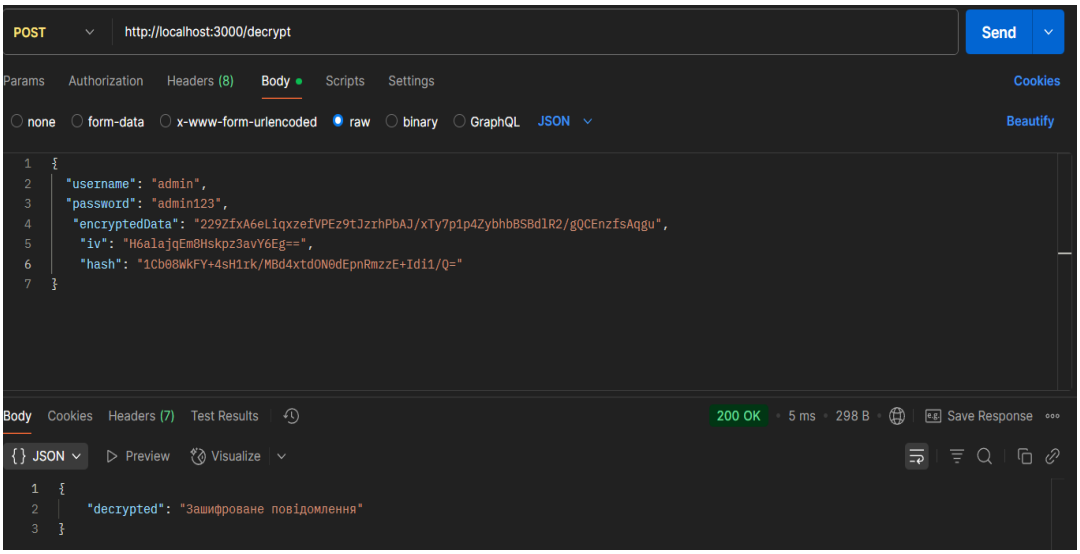


Рисунок 3.5 – Успішне розшифрування повідомлення з коректним хешем

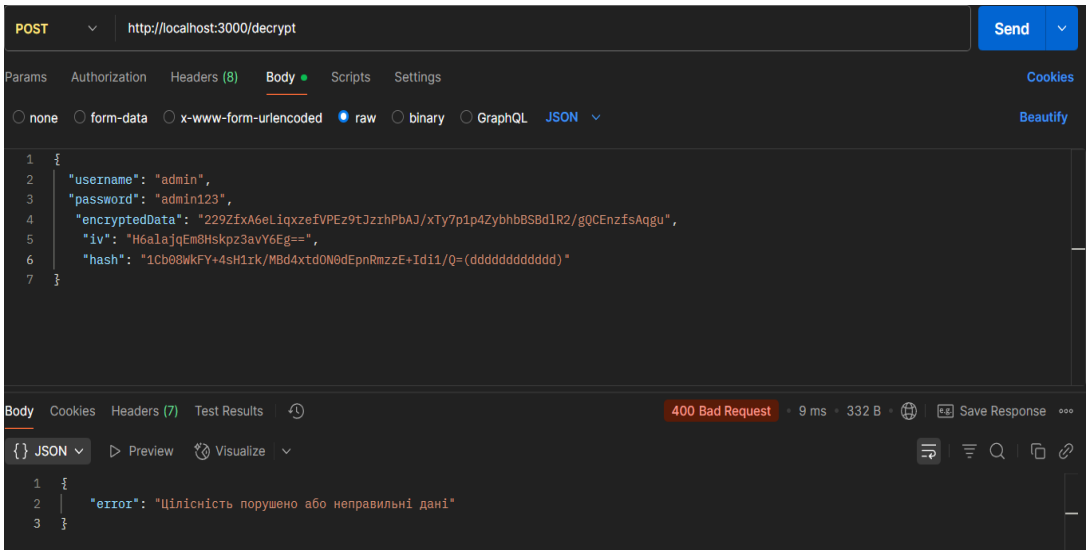


Рисунок 3.6 – Реакція системи на зміну зашифрованого повідомлення: порушення цілісності

Тестування адміністративного доступу: маршрути /log і /keys. Для забезпечення контролю за безпекою та прозорістю функціонування системи було реалізовано два адміністративних маршрути: /log – для перегляду журналу подій, та /keys – для отримання інформації про історію згенерованих криптографічних ключів.

На першому етапі було протестовано доступ до маршруту /log під обліковим записом адміністратора. У результаті сервер повернув повну вибірку журналу дій користувачів у форматі JSON, відсортовану за спаданням дати. Відповідь містила всі важливі поля: тип дії (наприклад, LOGIN, ENCRYPT, DECRYPT_FAIL), ім’я користувача, час та опис події. Це підтвердило, що механізм логування працює коректно та всі події надійно фіксуються.

На другому етапі маршрут /log було викликано від імені звичайного користувача. Система одразу повернула відповідь з кодом 403 (Forbidden) і повідомленням про відсутність прав доступу. Це свідчить про правильне обмеження доступу до конфіденційної інформації відповідно до ролі користувача.

Аналогічні дії було виконано для маршруту /keys. При авторизованому доступі від адміністратора сервер повернув перелік ключів, що містив:

– значення ключа (у форматі base64);

– дату створення;

– ознаку ротації (rotated: true/false).

Усі записи були відсортовані за часом, а структура відповіді була повністю відповідною до схеми бази даних. У разі звернення до маршруту звичайним користувачем система відреагувала так само – повідомленням про заборонений доступ.

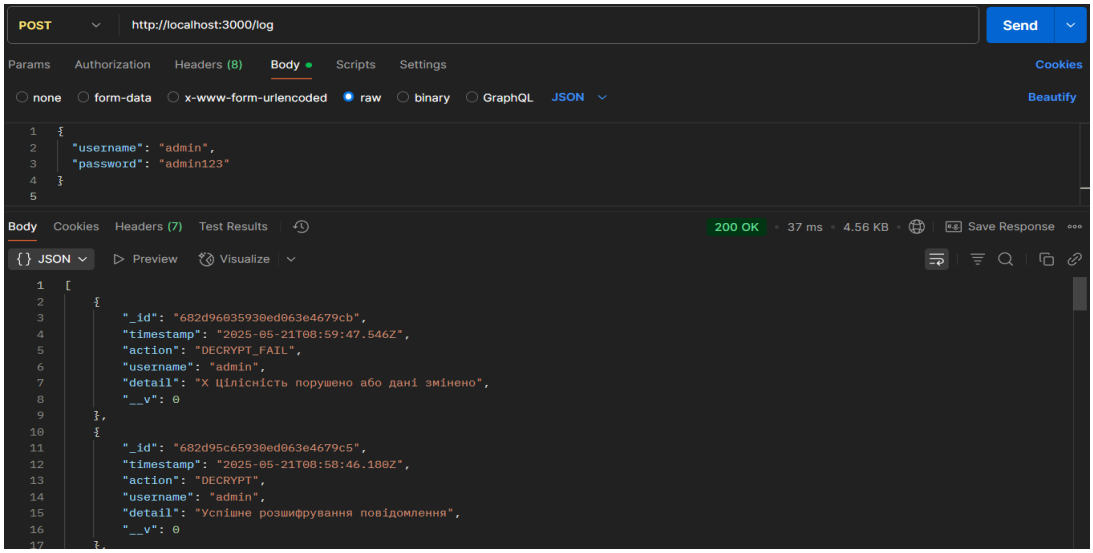


Рисунок 3.7 – Відповідь сервера на запит до /log під обліковим записом адміністратора

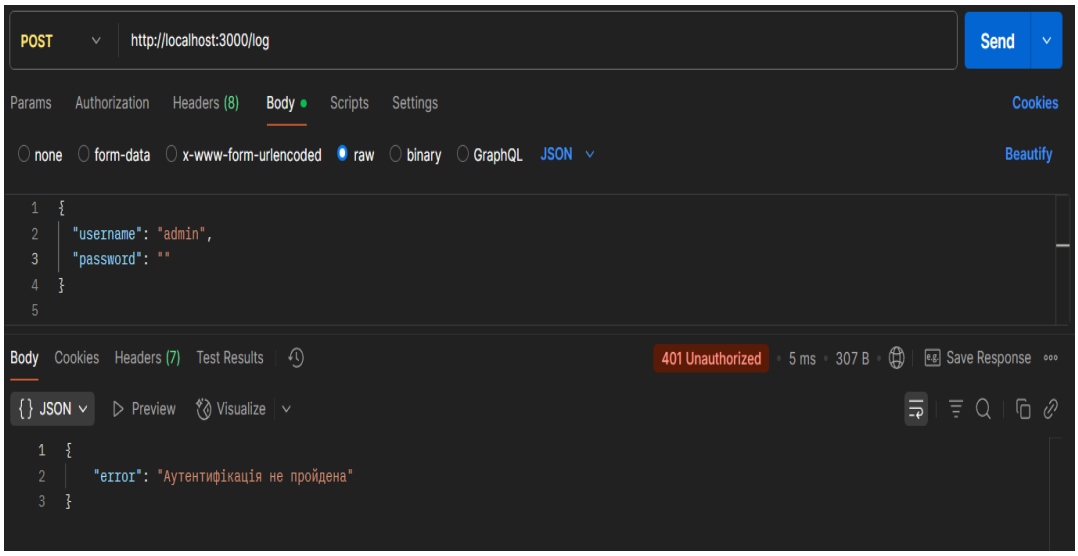


Рисунок 3.8 – Відмова у доступі до /log при спробі неавторизованого доступу

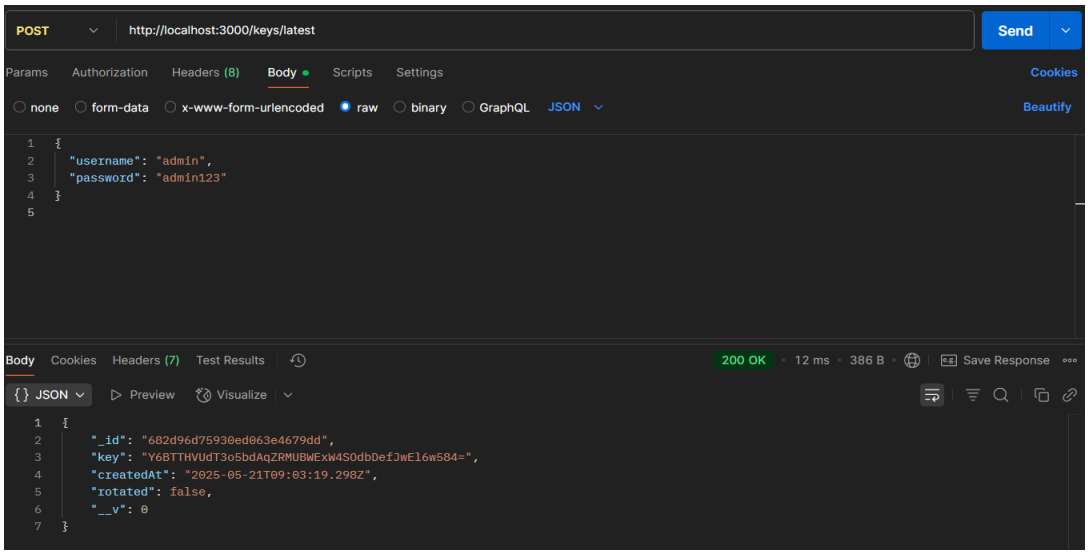


Рисунок 3.9 – Список згенерованих ключів, доступний лише адміністраторам

Тестування автоматичної ротації криптографічних ключів. Однією з особливостей реалізованого програмного модуля є підтримка автоматичної ротації ключів шифрування з фіксованим часовим інтервалом. Відповідно до алгоритму, описаного у розділі 3.2, система генерує новий симетричний ключ кожні 30 секунд, замінюючи активний ключ у пам’яті та записуючи новий запис у колекцію KeyEntry.

Для перевірки цього механізму сервер було запущено у звичайному режимі. У процесі роботи через регулярні інтервали часу (30 секунд) у базі даних з’являвся новий запис із полем createdAt, яке відповідало поточному моменту часу. Поле rotated у таких записах мало значення false, що свідчило про автоматичну, а не ручну зміну.

Було підтверджено, що:

– ключі генеруються без дублювання;

– попередні значення не перезаписуються, а зберігаються з повною історією;

– обробка нових шифрувальних запитів завжди здійснюється з актуальним ключем;

– система не перериває виконання запитів під час оновлення ключа.

Паралельно виконувались шифрування та розшифрування повідомлень. При цьому затримка на зміну ключа не впливала на стабільність роботи.

Жодного випадку переривання сеансу або втрати даних зафіксовано не було.

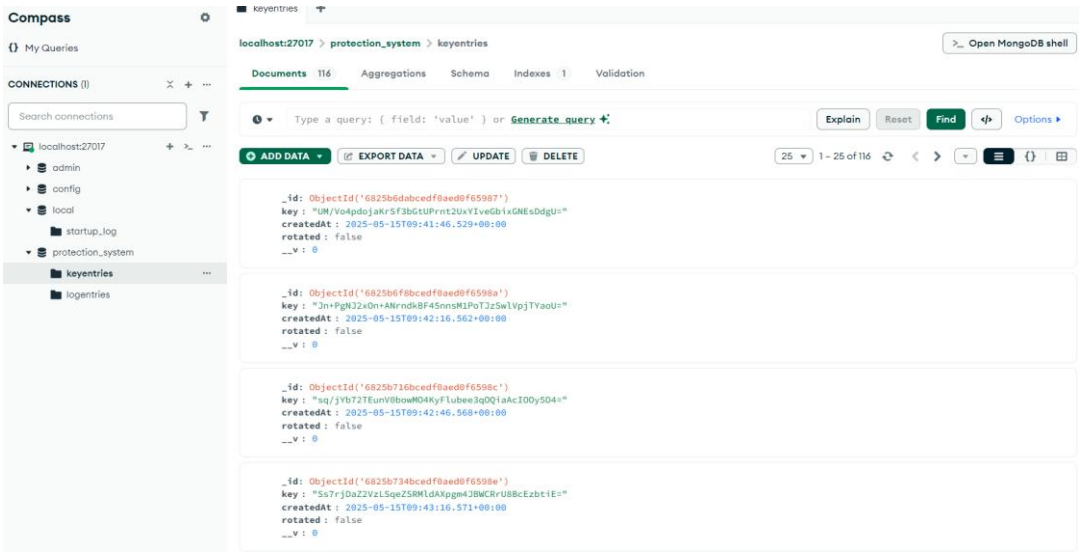


Рисунок 3.10 – Послідовна поява ключів у базі даних з інтервалом 30 секунд

У процесі тестування було перевірено повний цикл роботи реалізованої системи захисту веб-додатків із динамічним управлінням криптографічними ключами.

Результати тестування підтвердили, що система:

– коректно ідентифікує користувача та захищає маршрути на основі прав доступу;

– забезпечує стабільне шифрування з унікальним IV та хешем для контролю цілісності;

– ефективно виявляє порушення цілісності та виконує захисну реакцію у вигляді ротації ключа;

– підтримує стабільну роботу в режимі реального часу без втрати продуктивності;

– веде детальний журнал дій користувача з можливістю адміністративного перегляду;

– генерує нові ключі кожні 30 секунд без збоїв і з фіксацією в базі даних.

Таким чином, тестування підтвердило функціональну готовність програмного модуля до реального використання в умовах, де необхідно забезпечити захист даних, контроль доступу та автоматизовану реакцію на потенційні загрози.

3.4 Висновки до розділу 3

У межах третього розділу було повністю реалізовано програмний модуль захисту веб-додатків, заснований на динамічному розподілі та управлінні криптографічними ключами. Спочатку було обґрунтовано вибір мови програмування, технологій та бібліотек, на яких базується система. Обрані засоби – JavaScript (Node.js), MongoDB, Express, Mongoose та Crypto API – дозволили забезпечити високу швидкодію, гнучкість і масштабованість рішення.

На етапі реалізації було впроваджено механізми генерації симетричних ключів з фіксованим терміном дії, автоматичну та ручову ротацію ключів, перевірку цілісності повідомлень через хешування, а також централізоване логування усіх дій користувачів у системі. Створені API-маршрути забезпечують чітке розділення повноважень між користувачами та адміністраторами, підтримують надійний контроль доступу до службових ресурсів.

Проведене тестування підтвердило стабільність функціонування системи, її відповідність функціональним вимогам та стійкість до помилок. Усі компоненти працюють узгоджено, а система демонструє правильну поведінку як у штатних умовах, так і при виникненні загроз, таких як порушення цілісності або несанкціонований доступ.

Таким чином, третій розділ засвідчив повну технічну реалізацію розробленої системи та її готовність до впровадження у веб-додатки, що потребують сучасного рівня криптографічного захисту.

ВИСНОВКИ

У межах дипломної роботи було розроблено повнофункціональну систему захисту веб-додатків із динамічним розподілом і управлінням криптографічними ключами, що забезпечує конфіденційність, цілісність і контроль доступу до переданої інформації. Проєкт охоплює повний цикл – від теоретичного обґрунтування концепції до реалізації, тестування й оцінки ефективності розробленого рішення.

Розпочато дослідження з аналізу сучасних загроз, які виникають у процесі обміну інформацією у відкритих мережах. У результаті було виявлено, що класичні схеми шифрування з фіксованими ключами не є достатньо стійкими в умовах динамічного загрозового середовища. Це обумовило потребу в створенні механізму захисту, який здатний адаптивно змінювати параметри шифрування в реальному часі та автоматично реагувати на інциденти безпеки.

Під час проєктування архітектури системи особливу увагу було приділено збереженню службової інформації та захисту ключових об’єктів – криптографічних ключів. Запропонована модель бази даних забезпечує надійне зберігання ключів із прив’язкою до часових позначок та фіксацією типу ротації. Крім того, реалізована система журналювання подій дозволяє здійснювати аудит дій користувачів, що підвищує рівень прозорості та контрольованості роботи модуля.

Програмна реалізація здійснена за допомогою платформи Node.js із використанням мови JavaScript, що дозволило забезпечити високу продуктивність, гнучкість і сумісність із сучасними веб-технологіями. Уся логіка реалізована у вигляді окремих модулів, що значно спрощує підтримку й масштабування системи. Алгоритм шифрування заснований на використанні AES-256-CBC, з динамічно згенерованим вектором ініціалізації та контролем хешу повідомлень за допомогою SHA-256. Це дозволяє гарантувати як конфіденційність, так і цілісність переданих даних.

Тестування розробленого рішення довело стабільну роботу всіх функціональних модулів, у тому числі механізмів авторизації, шифрування,

дешифрування, автоматичної та ручної ротації ключів, ведення журналу подій і контролю доступу. Система виявила здатність вчасно реагувати на порушення цілісності, обробляти критичні ситуації та зберігати працездатність у режимі реального часу. Усі тестові сценарії завершились успішно, без виявлення критичних помилок, що підтверджує надійність програмного продукту.

Розроблена система є універсальним і придатним до впровадження рішенням для захисту інформації у веб-додатках, які працюють із чутливими даними. Завдяки модульній структурі, використанню відкритих технологій і підтримці масштабованості, вона може бути адаптована до різних умов використання, розширена новими функціями або інтегрована в складніші архітектури інформаційних систем.

Підсумовуючи, можна зазначити, що всі поставлені в роботі цілі було досягнуто. Проведене дослідження має як теоретичну, так і практичну цінність, а створений програмний засіб є технічно завершеним, перевіреним і готовим до подальшого застосування в реальних умовах експлуатації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Як Web Application Firewall захищає вебдодатки від хакерських атак? URL: https://hub.kyivstar.ua/articles/yak-web-application-firewall-zahyshhaye-vebdodatky-vid-hakerskyh-atak (дата звернення 25.11.2024)

2. Захищений протокол SSL/TLS – принцип роботи та використання. URL: https://hosting.in.ua/ua/articles/domeny/zakhyshchenyi-protokol-ssl-tsl-pryntsyp-roboty-ta-vykorystannia/ (дата звернення 25.11.2024)

3. Content security policy. URL: https://portswigger.net/web-security/cross-site-scripting/content-security-policy (дата звернення 25.11.2024)

4. What is SQL injection? URL: https://www.myrasecurity.com/en/sql-injection/ (дата звернення 25.11.2024)

5. A Brief Overview Of Remote Code Execution (RCE). URL: https://www.xcitium.com/remote-code-execution/ (дата звернення 25.11.2024)

6. What is XSS (cross site scripting) attack ? URL: https://www.wallarm.com/what/what-is-xss-cross-site-scripting (дата звернення 25.11.2024)

7. CSRF Attack Cross-Site Request Forgery. URL: https://www.wallarm.com/what/what-is-cross-site-request-forgery (дата звернення 25.11.2024)

8. Internet-scammers. URL: https://rubryka.com/article/internet-scammers/ (дата звернення 25.11.2024)

9. AWS Key Management Service. URL: https://docs.aws.amazon.com/kms/latest/developerguide/overview.htmlfdffffddddddd (дата звернення 25.11.2024)

10. Encryption principals in GCP. URL: https://medium.com/google-cloud/encryptions-principals-in-gcp-370c2dbbb330 (дата звернення 25.11.2024)

11. Azure Key Vault basic concepts. URL: https://learn.microsoft.com/en-us/azure/key-vault/general/basic-concepts (дата звернення 25.11.2024)

12. NIST SP 800-57 Part 1 Rev. 5. Recommendation for Key Management: Part 1 – General. URL: https://csrc.nist.gov/pubs/sp/800/57/pt1/r5/final (дата звернення

10.05.2025)

13. Postman. URL: https://www.postman.com (дата звернення 20.05.2025)

14. MongoDB Atlas. Fully managed MongoDB in the cloud. URL: https://www.mongodb.com (дата звернення 20.05.2025)

15. NodeJS. URL: https://nodejs.org/en (дата звернення 20.05.2025)

16. Що таке JavaScript. URL: https://cases.media/article/sho-take-javascript?srsltid=AfmBOoooa-4piL31T1yfrr6nslS2U6uU9PosXwGXI2Fu6lyjzoouzk2w (дата звернення 20.05.2025)

17. AES-256. URL: https://www.vpnunlimited.com/ua/help/cybersecurity/aes-256?srsltid=AfmBOoqX78aLWRX_ZxOgw4OUb4mzWG5VKgm4YezqLk1aIbQcZuyunbb- (дата звернення 20.05.2025)

18. Огляд алгоритму шифрування SHA-256. URL: https://www.cryptohuckers.club/2018/09/oglyad-algorytmu-shyfruvannya-sha-256.html (дата звернення 20.05.2025)

19. Visual Studio Code. URL: https://code.visualstudio.com (дата звернення 20.05.2025)

20. Git. URL: https://git-scm.com (дата звернення 20.05.2025)

21. OWASP Foundation. OWASP Top Ten Web Application Security Risks – 2023. URL: https://owasp.org/Top10/ (дата звернення: 23.05.2024).

22. Zeldovich N. et al. Web application security: threats and countermeasures. ACM Computing Surveys, 2022, 54(6), pp. 1–38.

23. Словник термінів з кібербезпеки / за заг. ред. О. В. Копана, Є. Д. Скулиша. Київ : ВБ «Аванпост-Прим», 2012. 214 с.

24. Hubbard, D. W., & Seiersen, R. How to measure anything in cybersecurity risk. John Wiley & Sons, 2023. 368 p.

25. Салієва О. В. Визначення витрат на забезпечення захищеності системи захисту інформації ранжуванням загроз. Матеріали VI Міжнародної науково-практичної конференції «Перспективні напрями захисту інформації», 02–06 вересня 2020 р. Одеса, 2020. С. 83–84.

26. HashiCorp Vault Documentation. URL:

https://developer.hashicorp.com/vault/docs (дата звернення: 11.05.2025).

27. Heninger N., Durumeric Z. et al. "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices". USENIX Security Symposium, 2012.

28. Салієва О. В. Моделі та засоби оцінювання рівня захищеності систем захисту інформації на основі когнітивного моделювання : дис. ... д-ра філософії. Львів, 2021. 208 с.

29. Kindervag J. Building a Zero Trust Network. Forrester Research Report, 2020.

30. OpenSSL Cryptography and SSL/TLS Toolkit. URL: https://www.openssl.org (дата звернення: 23.05.2025).

31. Kocher P. et al. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Advances in Cryptology — CRYPTO'96. Springer, 1996. С. 104–113.

ДОДАТКИ

Додаток А. Технічне завдання

Вінницький національний технічний університет
Факультет менеджменту та інформаційної безпеки
Кафедра менеджменту та безпеки інформаційних систем

ЗАТВЕРДЖУЮ
Голова секції "Управління інформаційною
ю" кафедри МБІС
д.т.н., професор
Юрій ЯРЕМЧУК
" 20 " березня 2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ

до бакалаврської кваліфікаційної роботи на тему:
«Розробка системи захисту веб-додатків із динамічним розподілом і управлінням криптографічними ключами та з можливістю шифрування у реальному часі.»

08-72.БКР.011.00.082.ТЗ

Керівник бакалаврської кваліфікаційної роботи
к.т.н., доцент
Грицак А.В.
"____"

Вінниця – 2025 р.

1. Найменування та область застосування

Програмний засіб захисту веб-додатків із динамічним розподілом та управлінням криптографічним ключами у реальному часі. Область застосування: забезпечення конфіденційності, цілісності та контролю доступу до інформації у веб-інформаційних системах.

2.Підстава для розробки

Розробка виконується на основі наказу ректора ВНТУ № 97 від 20.03.2025 р.

3. Мета та призначення розробки

3.1 Мета розробки: створення системи захисту веб-додатків із автоматичним оновленням ключів, шифруванням у реальному часі та перевіркою цілісності.

3.2 Призначення: реалізація безпечного обміну даними та контролю доступу на основі криптографічних механізмів.

4. Джерела розробки

4.1. Захищений протокол SSL/TLS – принцип роботи та використання. URL: https://hosting.in.ua/ua/articles/domeny/zakhyshchenyi-protokol-ssl-tsl-pryntsyp-roboty-ta-vykorystannia/

4.2. Content security policy. URL: https://portswigger.net/web-security/cross-site-scripting/content-security-policy

4.3. A Brief Overview Of Remote Code Execution (RCE). URL: https://www.xcitium.com/remote-code-execution/

4.4. NIST SP 800-57 Part 1 Rev. 5. Recommendation for Key Management: Part 1 – General. URL: https://csrc.nist.gov/pubs/sp/800/57/pt1/r5/final

5. Вимоги до програми

5.1 Вимоги до функціональних характеристик:

5.1.1 Програмний засіб повинен забезпечувати REST API для шифрування, дешифрування, аутентифікації користувачів, управління ключами та перегляду журналу дій;

5.1.2 Програмний засіб не повинен вимагати спеціального ліцензійного програмного забезпечення;

5.1.3 Система повинна виконувати криптографічний захист даних та перевірку цілісності повідомлень.

5.2 Вимоги до надійності:

5.2.1 Програмний засіб повинен функціонувати стабільно без критичних збоїв. У разі помилок або порушення цілісності даних система має повідомляти про загрозу та автоматично змінювати криптографічний ключ;

5.2.2 Логи та ключі повинні зберігатись у базі даних MongoDB;

5.2.3 Програмний засіб повинен виконувати свої функції.

5.3 Вимоги до складу і параметрів технічних засобів:

– процесор – Pentium 1500 МГц і подібні до них;

– оперативна пам’ять – не менше 512 Mb;

– вільне місце на диску – від 200 Mb;

– середовище функціонування – операційна система сімейство Windows;

– середовище виконання: Node.js (v18 або вище), MongoDB (локально або в хмарі);

– вимоги до техніки безпеки при роботі з програмою повинні відповідати існуючим вимогам та стандартам з техніки безпеки при користуванні комп’ютерною технікою.

6. Вимоги до програмної документації

6.1 Обов’язкова поетапна інструкція для майбутніх користувачів, наведена у пункті 3.4

7. Вимоги до технічного захисту інформації

7.1 Необхідно забезпечити захист розроблюваного програмного засобу від несанкціонованого використання.

7.2 Неможливість отримання доступу незареєстрованих користувачів до інформаційних ресурсів.

8. Техніко-економічні показники

8.1 Цінність результатів використання даного проекту повинна перевищувати витрати на його реалізацію.

8.2 Має бути реалізований таким чином, щоб підходити для використання широкого загалу.

9. Стадії та етапи розробки

Назва етапів бакалаврської кваліфікаційної роботи	Початок	Закінчення
Визначення напрямку бакалаврської роботи, формулювання теми	20.03.2025	23.03.2025
Аналіз предметної області обраної теми	24.03.2025	13.04.2025
Розробка алгоритму роботи	14.04.2025	27.04.2025
Написання бакалаврської роботи на основі розробленої теми	28.04.2025	25.05.2025
Передзахист бакалаврської кваліфікаційної роботи	26.05.2025	27.05.2025
Виправлення, уточнення, корегування бакалаврської дипломної роботи	28.05.2025	02.06.2025
Захист бакалаврської кваліфікаційної роботи	09.06.2025	10.06.2025

10. Порядок контролю та прийому

10.1 До приймання бакалаврської кваліфікаційної роботи надається:

– ПЗ до бакалаврської кваліфікаційної роботи;

- демонстрація результату бакалаврської кваліфікаційної роботи;
- презентація;
- відзив керівника роботи;
- відзив рецензента

ічне завдання до виконання прийняв _____ Лесь В.С.

Додаток Б. Лістинг програми

1. Модель KeyEntry (MongoDB + Mongoose)

```
const mongoose = require('mongoose');
const keyEntrySchema = new mongoose.Schema({
  key: { type: String, required: true },
  createdAt: { type: Date, default: Date.now },
  rotated: { type: Boolean, default: false }
});
module.exports = mongoose.model('KeyEntry', keyEntrySchema);
```

2. Модель LogEntry (MongoDB + Mongoose)

```
const mongoose = require('mongoose');
const logEntrySchema = new mongoose.Schema({
  timestamp: { type: Date, default: Date.now },
  action: String,
  username: String,
  detail: String
});
module.exports = mongoose.model('LogEntry', logEntrySchema);
```

3. Функція генерації криптографічного ключа

```
const crypto = require('crypto');
const KeyEntry = require('./models/KeyEntry');

function generateAndSaveKey() {
  const key = crypto.randomBytes(32); // AES-256
  const newKey = new KeyEntry({
    key: key.toString('base64'),
    rotated: false
  });
  newKey.save().catch(err => {
    console.error('Помилка збереження ключа:', err);
  });
  return key;
}
```

4. Функція логування подій

```
const LogEntry = require('./models/LogEntry');

function logAction(action, username, detail) {
  const entry = new LogEntry({
    action,
    username,
    detail
  });
  entry.save().catch(err => console.error('Помилка логування:', err));
}
```

5. Обробка API-запитів

```
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  if (authenticateUser(username, password)) {
    logAction('LOGIN', username, 'Успішна авторизація');
    return res.status(200).json({ message: 'Успішна авторизація' });
  } else {
    logAction('LOGIN_FAIL', username, 'Невірний логін або пароль');
    return res.status(401).json({ error: 'Невірний логін або пароль' });
  }
});
app.post('/encrypt', (req, res) => {
  const { username, password, message } = req.body;
  if (!authenticateUser(username, password)) {
    return res.status(401).json({ error: 'Аутентифікація не пройдена' });
  }
  const key = getCurrentKey();
  const result = encrypt(message, key);
  logAction('ENCRYPT', username, `Зашифровано повідомлення: "${message}"`);
  res.status(200).json(result);
});
```

Додаток В. Ілюстративний матеріал

Розробка системи захисту веб-додатків із динамічним розподілом і управлінням криптографічними ключами та з можливістю шифрування у реальному часі.

Виконав студент групи 1КІТС-21б Лесь В.С.
Науковий керівник: к.т.н, доц. кафедри МБІС Грицак А.В.

Вінниця - 2025

Актуальність теми

- У сучасному цифровому середовищі веб-додатки стали ключовими елементами обробки персональних, фінансових та корпоративних даних. Саме тому вони є головною ціллю атак типу перехоплення, підміни трафіку, компрометації ключів та повторного використання сесій.
- Традиційні системи захисту, що використовують статичні або довготривалі ключі, не забезпечують належного рівня безпеки у випадку загроз реального часу.
- Особливої актуальності набувають рішення, які підтримують динамічне оновлення криптографічних ключів, адаптивне шифрування та ведення журналу безпеки.
- Розробка такої системи дозволяє не лише захистити дані, а й вчасно реагувати на загрози без зупинки сервісу.

Мета та задачі дослідження

- Метою бакалаврської кваліфікаційної роботи є розробка програмної системи захисту веб-додатків, що забезпечує динамічний розподіл і управління криптографічними ключами, а також шифрування даних у реальному часі з перевіркою цілісності та веденням журналу безпеки.
- Для досягнення поставленої мети були сформульовані такі основні задачі:
- – провести аналіз сучасних методів захисту інформації у веб-додатках;
- – дослідити підходи до управління криптографічними ключами та ротації в реальному часі;
- – сформувати архітектуру системи із шифруванням та логуванням подій;
- – реалізувати захищений API з підтримкою ключових операцій;
- – забезпечити тестування працездатності та реагування на потенційні загрози.

Об’єкт і предмет дослідження

- Об’єктом дослідження є процес забезпечення інформаційної безпеки веб-додатків, зокрема з використанням криптографічних методів шифрування та контролю доступу.
- Предметом дослідження виступають методи та засоби реалізації динамічного розподілу криптографічних ключів, алгоритми шифрування даних у режимі реального часу, а також механізми забезпечення цілісності та фіксації подій безпеки.
- Фокус дослідження спрямований на побудову захисної системи, яка забезпечує ефективне управління ключами, стійке шифрування та адаптивне реагування на спроби атак без необхідності втручання користувача чи адміністратора.

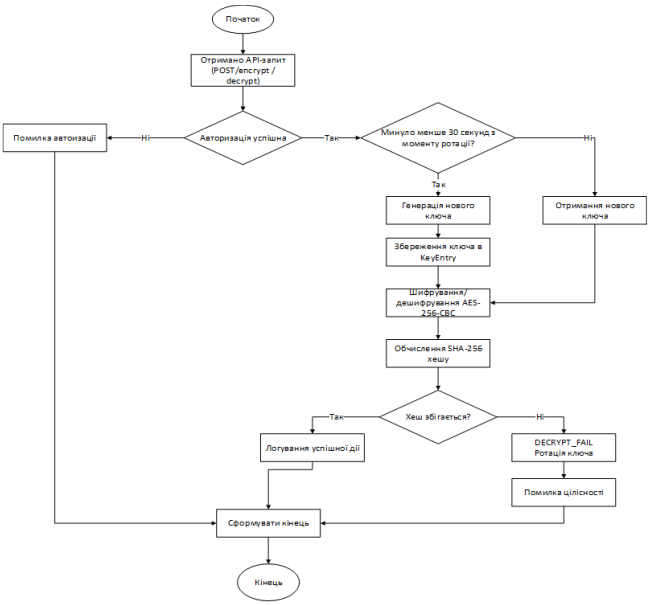
Аналіз існуючих рішень

- У процесі дослідження було розглянуто сучасні рішення щодо зберігання та управління криптографічними ключами, зокрема:
- – AWS Key Management Service (KMS)
- – Google Cloud Key Management
- – Azure Key Vault
- – HashiCorp Vault
- – OpenSSL з ручною генерацією ключів
- Недоліки більшості систем:
- – переважна залежність від хмарної інфраструктури;
- – відсутність або обмежена підтримка шифрування в реальному часі;
- – відсутність динамічної ротації ключів без втрати сесій;
- – складність локальної інтеграції в автономні системи.
- Висновок: існує потреба у незалежній системі, яка підтримує ротацію ключів, адаптивне шифрування та гнучку локальну інтеграцію.

Запропоноване рішення

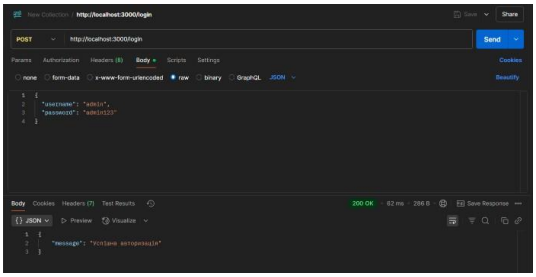
- У межах бакалаврської роботи запропоновано реалізацію власної системи захисту веб - додатків, що забезпечує:
- – динамічний розподіл і ротацію криптографічних ключів кожні 30 секунд;
- – шифрування даних у реальному часі з використанням симетричного алгоритму AES-256 у режимі CBC та генерацією унікального IV;
- – обчислення контрольної суми повідомлень за допомогою SHA-256 для виявлення спроб зміни даних;
- – збереження ключів і журналу безпеки у базі даних MongoDB;
- – REST API для реалізації функцій шифрування, дешифрування, авторизації, ротації ключів і доступу до логів.
- Побудована система дозволяє забезпечити захист даних без залежності від хмарних сервісів, із підтримкою автономної роботи та високою швидкістю обробки запитів.

Алгоритм роботи

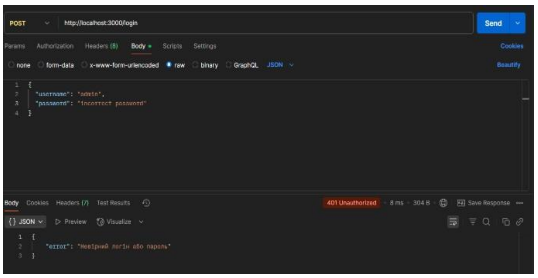


Алгоритм обробки запиту з динамічною ротацією ключа

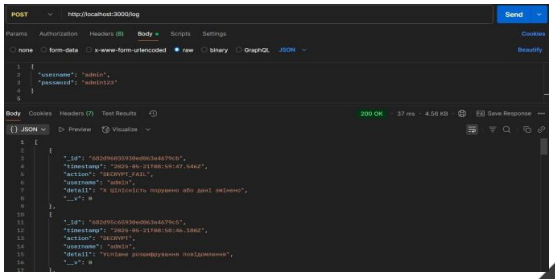
Тестування системи



Вигляд при правильному логіні та паролі



Вигляд при неправильних даних



Список всіх спроб входу

Результати реалізації

- У результаті виконання бакалаврської кваліфікаційної роботи було створено повноцінну систему захисту веб -додатків, що включає:
- 🔐 Реалізовано:
- – Генерацію криптографічних ключів у реальному часі
- – Динамічну ротацію ключів через заданий інтервал часу (30 с)
- – Шифрування та дешифрування повідомлень з перевіркою хеш -цілісності
- – Повноцінну систему логування дій користувачів та адміністраторів
- 🧠 Інтеграція:
- – REST API для взаємодії з клієнтською частиною
- – Збереження конфіденційних операцій у MongoDB
- – Простота розгортання та масштабування
- 📈 Ефективність:
- – Зменшено ризики повторного використання ключів
- – Підвищено стійкість до атак типу Replay, MITM, Key-leakage
- – Підтверджено коректну роботу в процесі тестування
- Система є завершеним та гнучким рішенням, яке можна інтегрувати у реальні веб -додатки.

Висновки

- У ході виконання бакалаврської кваліфікаційної роботи було досягнуто поставленої мети — створено систему захисту веб-додатків, яка забезпечує:
- – надійне шифрування інформації з динамічною ротацією криптографічних ключів;
- – реалізацію алгоритмів перевірки цілісності повідомлень (SHA-256);
- – виявлення потенційних загроз і миттєве реагування шляхом автоматичної заміни ключів;
- – ведення повного журналу безпеки, доступного для адміністратора;
- – незалежність від хмарних сервісів з можливістю автономної локальної інтеграції.
- Запропоноване рішення підвищує безпеку веб-додатків, забезпечуючи як захист даних під час передачі, так і контроль над процесами доступу й обробки інформації.
- Отримані результати можуть бути використані як основа для подальшого вдосконалення захисних механізмів, зокрема — з додаванням багатофакторної автентифікації, ролей доступу та інтеграції із системами моніторингу.

ПРОТОКОЛ ПЕРЕВІРКИ НАВЧАЛЬНОЇ (КВАЛІФІКАЦІЙНОЇ) РОБОТИ

Назва роботи:

зробка системи захисту веб-додатків із динамічним розподілом і управлінням ліптографічними ключами та з можливістю шифрування у реальному часі

Тип роботи: бакалаврська кваліфікаційна робота

Підрозділ Кафедра менеджменту на безпеки інформаційних систем
Факультет менеджменту та інформаційної безпеки
Гр. 1КІТС-21б

Керівник доцент Грицак А.В. _____

Показники звіту подібності

Strike Plagiarism	
Оригінальність	93,26%
Загальна схожість	6,74%

Аналіз звіту подібності (відмітити потрібне)

- ☐ **Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.**
- ☐ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.
- ☐ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Заявляю, що ознайомлений (-на) з повним звітом подібності, який був згенерований Системою щодо роботи (додається)

Автор _____ Лесь В.С.
(підпис) (прізвище, ініціали)

Опис прийнятого рішення

- ☐ **Допустити до захисту**

_____

_____

_____

_____

Особа, відповідальна за перевірку _____ Коваль Н.П.
(підпис) (прізвище, ініціали)

Експерт _____ _____
(за потреби) (підпис) (прізвище, ініціали, посада)