МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**

**НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ КІБЕРБЕЗПЕКИ ТА ЗАХИСТУ ІНФОРМАЦІЇ
КАФЕДРА УПРАВЛІННЯ КІБЕРБЕЗПЕКОЮ ТА ЗАХИСТОМ ІНФОРМАЦІЇ**

КВАЛІФІКАЦІЙНА РОБОТА

на тему: “Методи забезпечення безпеки API корпоративних інформаційних систем та сервісів”

на здобуття освітнього ступеня бакалавра
зі спеціальності 125 Кібербезпека
освітньої програми Управління інформаційною та кібернетичною безпекою

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Назар ЦЮПАК
(підпис) *Ім’я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр. УБД-42

Назар ЦЮПАК

Керівник:
д.т.н., професор Володимир ШУЛЬГА

Рецензент:
к.т.н., доцент Олександр ТУРОВСЬКИЙ

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут кібербезпеки та захисту інформації

Кафедра управління кібербезпекою та захистом інформації

Ступінь вищої освіти бакалавр

Спеціальність 125 Кібербезпека

Освітня програма Управління інформаційною та кібернетичною безпекою

ЗАТВЕРДЖУЮ
Завідувач кафедри УКБЗІ
_____ Світлана ЛЕГОМІНОВА
“____ ”_____2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Цюпаку Назару Юрійовичу
(прізвище, ім’я, по батькові здобувача)

1. Тема кваліфікаційної роботи “Інноваційні технології формування обізнаності й навчання персоналу з інформаційної безпеки”,
керівник кваліфікаційної роботи Шульга Володимир Петрович, д.т.н., професор
(ІПРІЗВИЩЕ, Ім’я,, науковий ступінь, вчене звання)

затверджені наказом Державного університету інформаційно-комунікаційних технологій від “20“ лютого 2026 р. №51

2. Строк подання кваліфікаційної роботи “12” травня 2026р.

3. Вихідні дані до кваліфікаційної роботи: *інформаційна безпека підприємства, методи забезпечення безпеки API, міжнародні стандарти, наукова та технічна література.*

4. Перелік питань, які мають бути розроблені:
4.1. Проаналізувати роль API в корпоративних інформаційних системах та сервісах, та дослідити їх вплив на інформаційну безпеку підприємства.
4.2. Дослідити основні загрози, вразливості, принципи та стандарти забезпечення безпеки API.
4.3. Вивчити процес моделювання загроз та інструментальні засоби тестування безпеки API, розробити практичні рекомендації щодо підвищення рівня їх захищеності.

5. Перелік ілюстративного матеріалу: *презентація PowerPoint*

6. Дата видачі завдання “05“ березня 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Етапи кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1.	Визначення об’єкту, предмету, мети та завдань дослідження.	18.03.2026	
2.	Збір та аналіз літератури.	30.03.2026	
3.	Аналіз ролі API в корпоративних інформаційних системах та дослідження їх впливу на інформаційну безпеку	08.04.2026	
4.	Дослідження методів і технологій забезпечення безпеки API.	15.04.2026	
5.	Вивчення практичних аспектів впровадження, тестування та оцінки ефективності захисту API.	22.04.2026	
6.	Формулювання висновків за результатами проведеного дослідження.	29.04.2026	
7.	Оформлення роботи.	06.05.2026	
8.	Оформлення презентації.	11.05.2026	
9.	Отримання рецензії на роботу.	10.06.2026	
10.	Захист в ДЕК.	__.06.2026	

Здобувач вищої освіти _____ Назар ЦЮПАК

(підпис) *(Ім’я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Володимир ШУЛЬГА

(підпис) *(Ім’я, ПРІЗВИЩЕ)*

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ КІБЕРБЕЗПЕКИ ТА ЗАХИСТУ ІНФОРМАЦІЇ

ПОДАННЯ
ГОЛОВІ ЕКЗАМЕНАЦІЙНОЇ КОМІСІЇ
ЩОДО ЗАХИСТУ КВАЛІФІКАЦІЙНОЇ РОБОТИ
на здобуття освітнього ступеня бакалавра

Направляється здобувач Цюпак Н. Ю. до захисту кваліфікаційної роботи
(прізвище та ініціали)
за спеціальністю 125 Кібербезпека
(код, найменування спеціальності)
освітньої програми Управління інформаційною та кібернетичною безпекою
(назва)
на тему: "Методи забезпечення безпеки API корпоративних інформаційних систем та сервісів"

Кваліфікаційна робота і рецензія додаються.

Директор ННІЗІ _____ Євгенія ІВАНЧЕНКО
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Висновок керівника кваліфікаційної роботи

Здобувач ЦЮПАК Назар у кваліфікаційній роботі проаналізував роль API в корпоративних інформаційних системах та їх вплив на інформаційну безпеку, дослідив основні загрози, вразливості й принципи забезпечення безпеки API, вивчив методи та технології захисту API, а також розробив практичні рекомендації за темою дослідження.
ЦЮПАК Назар показав розуміння проблематики дослідження та бачення основних теоретичних і практичних напрямів її вирішення, довів володіння методами наукового дослідження, проявив себе як організований, відповідальний виконавець. Результати дослідження апробовані на двох конференціях.
Все це дозволяє оцінити кваліфікаційну роботу здобувача ЦЮПАКА Назара на оцінку «добре» та присвоїти йому кваліфікацію бакалавра з кібербезпеки за освітньою програмою «Управління інформаційною та кібернетичною безпекою».

Керівник кваліфікаційної роботи _____ Володимир ШУЛЬГА
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

"____"_____2026 року

Висновок кафедри про кваліфікаційну роботу

Кваліфікаційна робота розглянута. Здобувач Цюпак Н.Ю. допускається до захисту даної роботи в Екзаменаційній комісії.

Завідувач кафедри
управління кібербезпекою та
захистом інформації _____ Світлана ЛЕГОМІНОВА
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

ВІДГУК РЕЦЕНЗЕНТА
на кваліфікаційну бакалаврську роботу

здобувача вищої освіти ЦЮПАКА Назара
на тему «Методи забезпечення безпеки API в корпоративних інформаційних сервісах та системах»

Актуальність. У сучасних корпоративних інформаційних системах API є ключовим механізмом взаємодії між програмними компонентами, вебсервісами, мобільними застосунками та хмарними платформами. Активне використання API сприяє підвищенню ефективності бізнес-процесів, проте водночас створює додаткові ризики для інформаційної безпеки. Помилки в реалізації механізмів автентифікації та авторизації, неналежний захист даних, відсутність контролю доступу та інші вразливості можуть призвести до витоку конфіденційної інформації, компрометації корпоративних ресурсів і значних фінансових збитків. У зв'язку з цим дослідження методів забезпечення безпеки API в корпоративних інформаційних сервісах та системах є актуальним науковим завданням.

Позитивні сторони.

1. У роботі досліджено роль API в корпоративних інформаційних системах, проаналізовано основні загрози, вразливості, принципи та стандарти забезпечення безпеки API.
2. Кваліфікаційна робота оформлена відповідно до встановлених вимог. Матеріал викладено послідовно та логічно, а результати дослідження належним чином узагальнено у висновках. Основні положення роботи проілюстровано рисунками та схемами.
3. Автор опрацював значну кількість наукових і професійних джерел, зокрема сучасні англомовні публікації, стандарти та рекомендації у сфері безпеки API.
4. У роботі досліджено сучасні методи автентифікації та авторизації, засоби технічного захисту API, архітектурні підходи до забезпечення безпеки в мікросервісних і хмарних середовищах, а також розроблено практичні рекомендації щодо підвищення рівня захищеності API.

Недоліки.

Доцільно було б приділити більше уваги практичному порівнянню ефективності окремих інструментів автоматизованого тестування безпеки API та аналізу їх застосування в реальних корпоративних середовищах.
Однак зазначене зауваження не впливає на загальну позитивну оцінку кваліфікаційної роботи.

Висновок: кваліфікаційна робота виконана на належному науково-методичному рівні та заслуговує на оцінку добре, а здобувач заслуговує на присвоєння кваліфікації бакалавра з кібербезпеки за освітньою програмою «Управління інформаційною та кібернетичною безпекою».

Рецензент: _____ Олександр ТУРОВСЬКИЙ
к.т.н., доцент *підпис* Ім’я, ПРІЗВИЩЕ

РЕФЕРАТ

Кваліфікаційна робота присвячена методів забезпечення безпеки API в корпоративних інформаційних сервісах та системах. Робота складається зі вступу, трьох розділів, що містять 4 рисунків, висновків і списку використаних джерел. Загальний обсяг роботи становить 82 аркушів, з яких 4 аркуші займають список використаних джерел та додатки.

Метою роботи є дослідження методів забезпечення безпеки API корпоративних інформаційних сервісів та систем, аналіз основних загроз і вразливостей API, а також розроблення практичних рекомендацій щодо підвищення рівня захищеності API-сервісів.

Об'єктом дослідження є корпоративні інформаційні системи та сервіси, що використовують API для взаємодії між внутрішніми компонентами системи, зовнішніми платформами та іншими сервісами.

Предмет дослідження – методи забезпечення безпеки API в корпоративних інформаційних системах та сервісах, спрямовані на зменшення ризиків несанкціонованого доступу, витоку конфіденційної інформації та інших кіберзагроз.

Методи дослідження. Для вирішення поставлених завдань у роботі використано методи аналізу та синтезу наукових джерел, порівняння, узагальнення, класифікації, системного аналізу, моделювання загроз, а також методи оцінювання ефективності засобів захисту API.

У результаті виконання роботи досліджено роль API в КІС та їх вплив на інформаційну безпеку, проаналізовано основні загрози та вразливості API, розглянуто принципи та стандарти забезпечення безпеки API. Досліджено сучасні методи автентифікації та авторизації, методи технічного захисту, також архітектурні підходи до забезпечення безпеки API в мікросервісних та хмарних середовищах. Розглянуто процес моделювання загроз, проаналізовано інструментальні засоби тестування безпеки API та розроблено практичні рекомендації щодо підвищення рівня захищеності API в корпоративних

інформаційних системах.

Галузь застосування. Результати дослідження можуть бути використані під час проектування, розроблення, модернізації та експлуатації КІС і сервісів, що використовують API, а також у діяльності фахівців з кібербезпеки для підвищення рівня захисту інформаційних ресурсів підприємства.

Ключові слова: API, БЕЗПЕКА API, КОРПОРАТИВНІ ІНФОРМАЦІЙНІ СИСТЕМИ, АВТЕНТИФІКАЦІЯ, АВТОРИЗАЦІЯ, КІБЕРБЕЗПЕКА, ЗАХИСТ ДАНИХ, МІКРОСЕРВІСНА АРХІТЕКТУРА, ХМАРНІ ТЕХНОЛОГІЇ, ТЕСТУВАННЯ БЕЗПЕКИ API.

ABSTRACT

The qualification paper is devoted to the study of API security methods in corporate information systems and services. The paper consists of an introduction, three chapters containing 4 figures, conclusions, and a list of references including sources. The total volume of the paper is 82 pages, of which 4 pages are occupied by the references and appendices.

The purpose of the study is to investigate methods of ensuring API security in corporate information systems and services, analyze the main API threats and vulnerabilities, and develop practical recommendations for improving the security level of API services.

The object of the study is corporate information systems and services that use APIs for interaction between internal system components, external platforms, and other services.

The subject of the study is methods of ensuring API security in corporate information systems and services aimed at reducing the risks of unauthorized access, confidential information leakage, and other cyber threats.

Research methods. To achieve the objectives of the study, methods of analysis and synthesis of scientific sources, comparison, generalization, classification, system analysis, threat modeling and methods for evaluating the effectiveness of API security measures were applied.

As a result of the research, the role of APIs in corporate information systems and their impact on information security were investigated. The main API threats and vulnerabilities were analyzed, and the principles and standards of API security were examined. Modern authentication and authorization methods, technical API protection mechanisms, and architectural approaches to API security in microservice and cloud environments were studied. The process of threat modeling was considered, API security testing tools were analyzed, and practical recommendations for improving API security in corporate information systems were developed.

Field of application. The results of the study can be used in the design,

development, modernization, and operation of corporate information systems and services that utilize APIs, as well as in the activities of cybersecurity specialists aimed at improving the protection of enterprise information resources.

Keywords: API, API SECURITY, CORPORATE INFORMATION SYSTEMS, AUTHENTIFICATION, AUTHORIZATION, CYBERSECURITY, DATA PROTECTION, MICROSERVICE ARCHITECTURE, CLOUD TECHNOLOGIES, API SECURITY TESTING.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ І СКОРОЧЕНЬ

КІС	Корпоративна Інформаційна Система
API	Application Programming Interface
HTTP	HyperText Transfer Protocol
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
TLS	Transport Layer Security
mTLS	Mutual TLS
RBAC	Role-Based Access Control
ABAC	Attribute-Based Access Control
JWT	JSON Web Token
XSS	Cross-Site Scripting
ПЗ	Програмне забезпечення

ВСТУП

Актуальність теми. На сьогодні, API (Application Programming Interface) є невід'ємним елементом корпоративних інформаційних систем та сервісів, оскільки він забезпечує взаємодію між внутрішніми та зовнішніми сервісами, хмарними платформами, мобільними застосунками.

Разом з цим, також підвищуються інформаційні ризики для компанії, оскільки через API здійснюється доступ до конфіденційної інформації, бізнесової логіки та критичних ресурсів підприємства. Неправильна реалізація методів аутентифікації та авторизації, недотримання стандартів забезпечення безпеки API, відсутність методів технічного захисту API підвищують ризики витоку конфіденційної інформації, неавторизованого доступу до сервісів підприємства а також великих фінансових втрат

Мета роботи полягає у дослідженні методів забезпечення безпеки API корпоративних інформаційних сервісів та систем, дослідження загроз та вразливостей, а також практичні рекомендації для підвищення рівня захищеності API сервісів.

Об'єкт дослідження – корпоративні інформаційні системи та сервіси, що використовують API для взаємодії між внутрішніми компонентами системи, зовнішніми платформами та іншими необхідними сервісами.

Предмет дослідження – методи забезпечення безпеки API в корпоративних інформаційних сервісах та системах, спрямовані на зменшення ймовірності вдалих хакерських атак на сервіси підприємства, витоків конфіденційної інформації та інших можливих кіберзагроз.

Методи дослідження. В ході написання роботи, було проведено аналіз основних загроз та вразливостей API, принципів та стандартів забезпечення безпеки API. Також було вивчено методи та технології забезпечення безпеки API, методи технічного захисту API, архітектурні підходи до забезпечення безпеки API в хмарних та мікросервісних середовищах. Також було розглянуто методи тестування та оцінки ефективності захисту API сервісів.

Практичне значення одержаних результатів. Застосування результатів дослідження полягає у можливості використання запропонованих рекомендацій для зниження кібер-ризиків. Також результати дослідження можуть бути використані в розробці, модернізації та впровадженні безпечних API сервісів, що будуть використовуватися компаніями.

Розділ 1 ТЕОРЕТИЧНІ ОСНОВИ БЕЗПЕКИ API В КОРПОРАТИВНИХ ІНФОРМАЦІЙНИХ СИСТЕМАХ

Для всебічного розкриття теми кваліфікаційної роботи необхідно розглянути основи безпеки API в корпоративних інформаційних системах, основні загрози та вразливості, з якими зіштовхуються при розробці, модернізації та використанні API, а також принципи та стандарти забезпечення безпеки API інформаційних систем.

1.1 Роль API в корпоративних системах та їх вплив на інформаційну безпеку

У сучасній комп'ютерній науці API (Application Programming Interface або ж інтерфейс програмування додатків) – це набір специфікацій, протоколів та інструментів, що забезпечують взаємодію між різними програмними компонентами [3]. З точки зору архітектури, API виступає як абстрактний шар, який приховує складність внутрішньої логіки сервісу, і надає розробникам лише методи для доступу до необхідних ресурсів [2]. Сучасна модель API базується на архітектурному стилі REST (Representational State Transfer), який використовує протокол HTTP (Hyper Text Transfer Protocol) та більш легкі формати обміну даними. Це прийшло на зміну протоколу SOAP (Simple Object Access Protocol), який був складнішим у використанні та мав менше можливостей, тому зміна протоколу забезпечила вищу продуктивність та гнучкість інтеграції API у будь-який сервіс [3].

В корпоративних інформаційних системах та сервісах API трансформувалися з допоміжного інструменту в важливий стратегічний актив, який визначає ефективність корпоративної IT-інфраструктури. [1] Роль API в корпоративних інформаційних системах охоплює наступні критичні аспекти:

- Формування екосистем та ланцюжків цінності: API дозволяють корпораціям створювати канали дистрибуції власних цифрових активів для загального використання. Це називають "Ланцюжком цінності API"

(API Value Chain) (рис. 1.1). Тобто компанія визначає для себе цінність бізнесу, яка може бути використана в API, яку цінність це принесе в результаті для кінцевих користувачів та бізнесу загалом, які переваги для себе отримає розробник API (provider) і тд. [3].

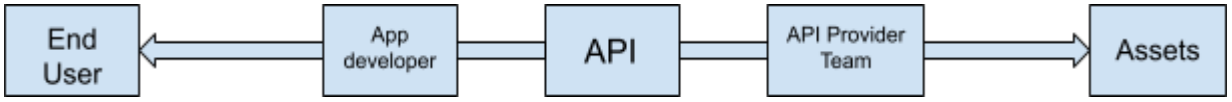

рис. 1.1. схема ланцюжка цінності API (API Value Chain)

- Забезпечення корпоративної гнучкості: Використання внутрішніх API дозволяє деконструювати монолітні системи на мікросервіси. Це дає змогу внутрішнім комндам розробників швидко створювати нові технологічні рішення, використовуючи вже існуючу логіку [1].
- Управління доступом та безпека інформації: У корпоративних інформаційних системах API є головним методом взаємодії між різними сервісами, тому його захист є першочерговим завданням. Це здійснюється наступними методами:

  - Сучасні протоколи авторизації: використання стандарту OAuth2 для делегування прав доступу та OpenID Connect для ідентифікації користувачів [2, 3].

  - Моделі контролю доступу: впровадження рольового (RBAC, Role Based Access Control) та атрибутивного (ABAC, Attribute Based Access Control) типів доступу, які дозволяють динамічно регулювати дозволи на основі контексту запиту [2].

  - Захисні шлюзи (API Gateways): Використання спеціального програмного забезпечення для фільтрації трафіку, обмеження частоти запитів (Rate-limiting) та запобігання кібератакам [2, 3].
- Централізоване управління життєвим циклом: Корпоративні API потребують суворого контролю на кожному етапі – створення, публікація, контроль версій, сповіщення користувачам про зміну внутрішньої логіки API та системи відслідковування технічних проблем. Це забезпечує стабільність корпоративної інформаційної системи та легкість у взаємодії між департаментами організації. [3]

Систематизація прикладних програмних інтерфейсів в корпоративних інформаційних сервісах базується на рівні їх відкритості та цільовій аудиторії, що дозволяє розділити API на публічні (public) та приватні (private) API [3].

- Публічні або зовнішні API (public / external APIs) це програмні інтерфейси, доступні для будь-якого зовнішнього розробника. Це також є можливістю для підприємства інтегруватися в глобальну цифрову екосистему та монетизувати свої цифрові активи [3]. Такі інтерфейси представляють бренд в цифровому середовищі, тому для них є список найвищих вимог щодо стандартизації, якості документації та зручності використання. Для успішного функціонування такого API критично важливим є забезпечення стабільності та тривалої зворотної сумісності, щоб зміни в системі не призводили до збоїв у роботі тисяч зовнішніх додатків, які залежать від цього інтерфейсу [5].
- Приватні або зовнішні API (private / internal APIs) розробляються виключно для споживання всередині корпоративної мережі. Їх впровадження є стратегічним кроком до створення модульної архітектури підприємства (composable enterprise), де IT-команди мають можливість швидко конструювати нові сервіси з існуючих функціональних блоків. У межах такого підходу система складається з окремих незалежних сервісів, які взаємодіють між собою через програмні інтерфейси. Це значно спрощує створення нових компонентів та оновлення вже існуючих модулів, оскільки не потрібно змінювати всю систему. Також приватні API сприяють підвищенню надійності та стабільності корпоративних сервісів. Це дозволяє краще контролювати процеси всередині системи і швидше локалізовувати помилки. Використання приватних інтерфейсів також дозволяє покращити взаємодію та комунікацію всередині підприємства [6].
- Партнерські (partner) API дозволяють контрольований обмін даними та функціональністю між компанією та її бізнес-партнерами. На відміну від публічних API, де доступ має будь-який зовнішній розробник, чи

приватних, де доступ має лише співробітники підприємства, до партнерських інтерфейсів доступ надається лише визначеному колу користувачів, таких як постачальники, дистриб'ютори чи корпоративні клієнти. Однією з основних переваг partner API є спрощення інтеграції з новими партнерами та масштабування партнерської екосистеми. Завдяки стандартизованим API компанії можуть уникати створення окремих інтеграційних рішень для кожного нового партнера, що знижує витрати на підтримку та розвиток інфраструктури. Партнерські API також сприяють виходу компаній на нові ринки та розширенню клієнтської бази. Це дозволяє партнерам використовувати функціонал компанії у власних інформаційних системах, надаючи кінцевим користувачам доступ до послуг без необхідності прямої взаємодії з постачальником програмного інтерфейсу. [6]. (рис. 1.2)

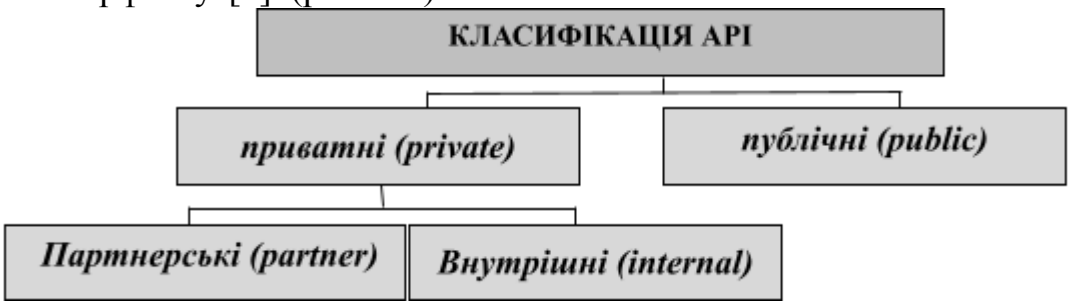

рис. 1.2. класифікація API за типом доступу

Ефективність обраної моделі доступу безпосередньо залежить від технологічної реалізації. Залежно від бізнес-завдань та вимог до продуктивності системи, розробники застосовують різноманітні архітектурні стилі та протоколи, які ми розглянемо далі.

SOAP API (Simple Object Access Protocol) є одним із основним підходів для реалізації вебсервісів, заснованим на використанні XML (eXtensible Markup Language) повідомлень для обміну даними між клієнтом і сервером. Цей протокол має свою формально визначену граматику повідомлень, підтримує стандартизований опис сервісів через WSDL (Web Service Description Language) та має можливість платформно-незалежної взаємодії між різними програмними системами. SOAP інтерфейси використовуються в сервісно-орієнтованій архітектурі (SOA, Service Oriented Architecture), де сервіси мають бути

незалежними, модульними та придатними до інтеграції в корпоративних інформаційних сервісах. Особливо важливими ці API є для інтеграції legacy-систем, баз даних і корпоративних платформ, оскільки це забезпечує доступ до функціональності через мережу. Недоліками SOAP API є складність реалізації та надмірна об’ємність SOAP-повідомлень через велику кількість супровідних стандартів та використання XML. [10]

REST API (Representational State Transfer) є веб-інтерфейсом, який ґрунтується на архітектурному стилі REST, сформульованому Роєм Філдінгом у контексті розвитку та масштабування Всесвітньої павутини. Основним призначенням REST API є забезпечення стандартизованої взаємодії між клієнтськими застосунками та веб сервісами через протокол HTTP. [9] Цей підхід можна назвати як легшу альтернативу SOAP, оскільки складність реалізації цих сервісів є великою. Також відмінність від SOAP полягає в тому, що REST є архітектурним стилем, а SOAP - саме протоколом [10]. Основною характеристикою REST підходу є використання HTTP-методів відповідно до їх призначення. Метод GET застосовується для отримання (читання) даних, HEAD – отримує метадані, POST – створення нових ресурсів чи виконання контролерів, PUT – оновлення ресурсів, DELETE – для видалення ресурсів. Цей підхід забезпечує прозорість та передбачуваність роботи між клієнтом і сервером. Перевагами REST API можна назвати масштабованість, зменшення навантаження на сервер завдяки кешуванню, технологічну незалежність і стандартизацію інтеграції сервісів. Серед обмежень REST API можна виокремити залежність між сервером та клієнтом, оскільки клієнт має підтримувати виконання отриманого коду, це відбувається через впровадження механізму code-on-demand [9].

GraphQL – це мова запитів для прикладних програмних інтерфейсів та середовище виконання для обслуговування цих запитів на основі наявних даних. На відміну від методів описаних вище, GraphQL є декларативною мовою отримання даних, де розробники описують вимоги до інформації, яку хочуть отримати, що усуває проблеми надлишковості трафіку та мінімізує кількість

мережевих запитів. Основною характеристикою цього методу є ієрархічність запиту, його структура завжди відповідає формі даних, що повертаються. Це також можна назвати однією з переваг, оскільки ця характеристика вирішує проблеми надлишкового (overfetching) та недостатнього (underfetching) отримання даних, що притаманне архітектурі REST. Крім того, GraphQL має сувору типізацію: кожен сервіс визначає типи в схемі GraphQL, що є кресленням для даних і забезпечує їх валідацію. Серед недоліків можна виділити глибину запиту (query complexity), це означає, що запит може не мати великої глибини вкладення, але може мати надмірну кількість полів та списків, і це потребуватиме більшої кількості обчислювальних ресурсів. Також, в порівнянні з REST API, де кешування є простішим через прив'язку відповідей до конкретних URL-адрес, у GraphQL всі запити проходять через один endpoint, тому ефективність стандартних методів HTTP-кешування зменшується. Також GraphQL потребує постійного моніторингу безпеки й продуктивності. Для використання GraphQL в корпоративному інформаційному сервісі необхідно використовувати додаткові сервіси моніторингу, як наприклад Apollo Engine. Цей сервіс дозволяє аналізувати роботу GraphQL API в реальному часі. Його задача полягає в зборі статистики щодо виконання запитів та виявленні потенційних проблем продуктивності. Також Apollo Engine може контролювати складність запитів та виявляти потенційно небезпечні та ресурсомісткі операції [7].

gRPC API – це технологія міжпроцесної взаємодії, яка побудована на базі віддалених викликів процедур (RPC, Remote Procedure Call), що використовує Protocol Buffers для опису інтерфейсів та HTTP/2 як транспортний протокол. Важливою рисою цього методу є сувора типізація контрактів, яка чітко визначає формати повідомлень та методи взаємодії. Це суттєво мінімізує ризики виникнення помилок на етапі виконання у складних середовищах. В порівнянні з вище перерахованими методами, gRPC підтримує чотири патерни комунікації: унітарні запити, потокову передачу на стороні сервера, на стороні клієнта, а також двосторонній стрімінг. З переваг цієї технології можна виділити

підвищену продуктивність завдяки бінарному підходу, що підвищує пропускну здатність та знижує затримки, що критично важливо для корпоративних інформаційних систем. Основними недоліками gRPC є обмежена підтримка у веб-браузерах, що вимагає додаткових рішень для забезпечення взаємодії з фронтенд-додатками. Все через те, що обмін відбувається у бінарному форматі, що є незручним для безпосереднього читання людиною, що ускладнює процес налагодження та моніторингу мережевого трафіку порівняно з іншими форматами. У корпоративних інформаційних системах gRPC виконує роль базового протоколу для внутрішньої (east-west) взаємодії між мікросервісами, що забезпечує високу продуктивність. Впровадження цієї технології дає змогу вирішувати питання безпеки через підтримку TLS (Transport Layer Security) та механізмів автентифікації на основі токенів, а також інтегруватися з API-шлюзами (API Gateways), що балансує навантаження на сервіс та дозволяє здійснювати моніторинг за ним [8].

1.2 Основні загрози та вразливості API

В попередньому підрозділі ми розглянули що таке API, їх класифікації та ролі в корпоративних інформаційних системах. Не менш важливим є питання безпеки використання цих інтерфейсів. Оскільки вони забезпечують взаємодію між різними сервісами, застосунками та користувачами, вони стають однією з основних точок потенційних атак на інформаційну систему. У зв’язку з цим, особливо актуальним стає дослідження основних загроз та вразливостей API, які можуть призвести до порушення конфіденційності, цілісності та доступності даних.

Першою категорією вразливих механізмів у корпоративних інформаційних сервісах можна виділити вразливості автентифікації, авторизації та контролю доступу. Забезпечення захищеності сучасних інформаційних систем ґрунтується на концепції AAA (Authentication, Authorization, Accounting), яка відповідно охоплює процеси автентифікації, авторизації та обліку дій користувачів. Автентифікація визначається як процедура підтвердження автентичності

суб’єкта, що має на меті запобігати доступу стороннім особам під виглядом легітимних користувачів. Наступний етап – авторизація, яка полягає у визначенні конкретних прав та дозволів на маніпуляції з ресурсами після успішного підтвердження особи. Контроль доступу є комплексним механізмом, що обмежує взаємодію суб’єктів з об’єктами системи, забезпечуючи конфіденційність та цілісність даних. Існує декілька моделей безпеки систем контролю доступу. Модель дискреційного керування доступом (DAC Discretionary Access Control) є найменш обмеженою. Система базується на праві власника ресурсу самостійно делегувати привілеї іншим учасникам, що забезпечує високу гнучкість, але разом з тим підвищує ризики через людський фактор. Наступною моделлю є мандатне керування доступом (MAC, Mandatory Access Control). Вона є найбільш суворою, але в той же час найбезпечнішою. В основі цього підходу лежить використання міток безпеки для кожного об’єкта та рівнів допуску для кожного суб’єкта. Процес прийняття рішень про доступ є повністю автоматизованим і здійснюється операційною системою на основі порівняння цих параметрів. Користувач не має можливості змінити права доступу навіть до власних файлів, що мінімізує ризики, пов’язані з людським фактором. Це робить MAC незамінною у структурах з високими вимогами до конфіденційності, хоча складність адміністрування та відсутність гнучкості обмежують її використання у загальному корпоративному секторі. Далі розглянемо модель рольового керування доступом RBAC. Вона пропонує логічне структурування прав на основі організаційної структури підприємства. Замість призначення прав конкретним особам, привілеї групуються у ролі (залежно від посади), та згодом призначаються користувачам. Це спрощує управління доступом у великих корпоративних системах: при зміні посадових обов’язків працівника адміністратору достатньо змінити його роль, а не переглядати права доступу до сотень окремих файлів. RBAC забезпечує дотримання принципу мінімальних привілеїв, тому що користувач отримує доступ лише до тих ресурсів, що необхідні для виконання його професійних функцій. Найбільш прогресивною є модель доступу на основі атрибутів ABAC.

На відміну від статичних ролей, ABAC оцінює сукупність атрибутів суб’єкта, об’єкта та середовища. Це дозволяє реалізовувати надзвичайно деталізовані правила доступу. Завдяки своїй адаптивності, ABAC є ідеальним рішенням для сучасних хмарних середовищ та складних екосистем API, де контекст запиту має вирішальне значення для безпеки [11].

Наступною категорією вразливостей API є вразливості, які пов’язані з обробкою та передачею даних. Основними проблемами є можливість несанкціонованого доступу, модифікації або витоку даних у моменти її переміщення між компонентами системи. Характеристики цих загроз описуються через “тріаду проблем” (Trinity of Trouble): зв‘язність (connectivity), розширюваність (extensibility) та складність (complexity). Зв‘язність створює численні шляхи для експлуатації, які раніше не існували, тоді як складність коду прямо пропорційно збільшує кількість дефектів безпеки. На етапі передачі даних найбільшим ризиком для програмних інтерфейсів є ризик перехоплення повідомлень (Man-in-the-Middle), якщо не забезпечено належне шифрування транспортного рівня. Одним з методів захисту API є захист транспортного рівня TLS (Transport Layer Security), який є стандартним галузевим підходом, забезпечує захист усього каналу зв’язку, проте його недоліком є припинення захисту в точці термінації трафіку, що залишає дані вразливими всередині корпоративної мережі. Методи захисту на рівні повідомлень (наприклад, JWS - JSON Web Signature або JWE - JSON Web Encryption) забезпечують наскрізний захист незалежно від транспортного каналу, але впровадження такого методу супроводжується значними обчислювальними витратами та складністю інтеграції, що може негативно вплинути на продуктивність системи. У корпоративних системах роль захисту даних є визначальною для функціонування “економіки API” та мікросервісних архітектур, де сотні сервісів постійно обмінюються чутливою інформацією. Неналежна обробка вхідних даних може призвести до розкриття внутрішньої логіки системи через деталізовані повідомлення про помилки, що полегшує підготовку цілеспрямованих атак. Для прикладу розглянемо атаку на компанію Target у

2013 році. Зловмисникам вдалося інтегрувати шкідливе ПЗ, призначене для перехоплення даних, в систему обробки транзакцій. Це дозволило хакерам отримувати доступ до даних безпосередньо в момент їх обробки. Внаслідок цього було скомпроментовано понад 40 мільйонів платіжних карток клієнтів. Цей кейс ілюструє концепцію “тріади проблем”, де висока зв’язність корпоративних систем та складність їх архітектури створюють приховані загрози всередині мережі [12].

Не менш важливим для корпоративних інформаційних систем, є стійкість до ін’єкцій коду, масового призначення параметрів та надмірного розкриття даних. Дослідження цих деструктивних чинників дозволяє мінімізувати ризики порушення конфіденційності та цілісності інформації в API. Феномен надмірного розкриття даних (Excessive Data Exposure) виникає в архітектурі кінцевих точок інтерфейсу, які призначені для надання інформації клієнтським програмам, коли сервер передає повну структуру об’єкта бази даних замість обмеженого набору властивостей. Основними характеристиками цього дефекту є трансляція чутливих системних або персональних атрибутів, таких як внутрішні ідентифікатори, службові токени та персональні дані, безпосередньо у протоколах HTTP. Перевагою такого методу з точки зору інженерії програмного забезпечення є суттєва оптимізація витрат часу на розробку та підвищення універсальності програмного коду, оскільки єдиний ендпоінт спроможний обслуговувати декілька різнорідних споживачів без необхідності створення кастомізованих представлень на стороні сервера. Проте критичним недоліком є хибне делегування функцій безпеки та фільтрації на сторону клієнта, що дозволяє зовнішньому суб’єкту перехоплювати необроблений трафік за допомогою інструментів проксі-аналізу або вбудованих інструментів розробника в браузері, нівелюючи встановлені інтерфейсні обмеження. Уразливість масового призначення параметрів (Mass Assignment) пов’язана зі стадією обробки даних, коли програмний інтерфейс здійснює автоматичну прив’язку переданих клієнтом параметрів до внутрішніх об’єктних моделей або змінних без попередньої верифікації. Головна характеристика цієї вразливості

зумовлена специфікою сучасних веб-фреймворків, які заохочують розробників автоматично відображати параметри HTTP-запитів на властивості сутностей у базі даних. Перевагою для бізнесу у цьому підході є значне прискорення темпів розробки та скорочення обсягів шаблонного коду, що знижує витрати на супровід системи. Головним недоліком є повна втрата контролю над архітектурною цілісністю внутрішнього стану об'єктів, що дозволяє зловмисникам впроваджувати непередбачені логікою системи атрибути, такі як модифікатори адміністративних ролей чи фінансові показники. У межах корпоративного середовища експлуатація масового призначення призводить до несанкціонованого підвищення привілеїв, фальсифікації критичних бізнес-правил та обходу механізмів комплаєнсу. Ін'єкційні вразливості (Injections) класифікуються як дефекти валідації обробки даних, за яких API передає неперевірені дані від користувача безпосередньо інтерпретаторам у складі команд або пошукових запитів. Ключовою характеристикою процесу є те, що інтерпретувальне середовище розпізнає впроваджені символьні послідовності як керуючі директиви, а не пасивні текстові аргументи. Проектування динамічних запитів надає розробникам перевагу у вигляді високої гнучкості при реалізації складних аналітичних вибірок, однак цей чинник повністю нівелюється катастрофічним недоліком – тотальним руйнуванням меж безпеки між користувацьким введенням та ядром виконання. Роль ін'єкцій у корпоративних системах має винятково руйнівний характер, оскільки їх вектор спрямований на центральні сховища даних та операційні системи серверів. Це спричиняє повну ексфільтрацію корпоративних баз даних або отримання повного контролю над хостом. Практичним прикладом такої атаки можна розглянути компрометацію GraphQL-мутації ImportPaste. Через впровадження спеціальних символів у змінну шляху (PATH) зловмисники отримали можливість виконувати довільні системні команди в середовищі Linux з правами привілейованого користувача root, санкціонуючи читання конфіденційного файлу паролів /etc/passwd [13].

Реалізація загроз безпеці API в корпоративних інформаційних системах

призводить до значних технічних та фінансових наслідків через порушення логіки взаємодії між компонентами інфраструктури. У науковому дискурсі наслідки реалізації загроз визначаються як комплексний деструктивний результат навмисного або ненавмисного втручання в логіку роботи прикладних інтерфейсів, що призводить до порушення цілісності, доступності та конфіденційності корпоративних ресурсів. Основними характеристиками наслідків експлуатації вразливостей API є їхня висока латентність, здатність до каскадного поширення по всій інфраструктурі та безпосередній зв'язок із компрометацією бізнес-логік організації. На відміну від класичних мережевих атак, наслідки яких часто локалізуються на рівні окремих вузлів, несанкціоноване використання інтерфейсів взаємодії дозволяє зловмисникам діяти в межах легітимних запитів та сесій. Через це шкідлива активність може довгий час залишатися непоміченою для традиційних засобів моніторингу, що зумовлює значний масштаб збитків, оскільки одинична помилка чи вразливість контролю доступу може призвести до повного витоку баз даних або несанкціонованого виконання фінансових операцій. У контексті побудови сучасних корпоративних інформаційних систем аналіз потенційних ризиків має чітко виражені переваги й недоліки. До бізнесових та технічних переваг можна віднести забезпечення нормативних вимог щодо захисту персональних даних, збереження ринкових переваг підприємства шляхом недопущення автоматизованого збирання унікального контенту чи блокування сервісів. Проте недоліки впровадження комплексного контролю наслідків пов'язані із суттєвим ускладненням обчислювальної архітектури, виникненням додаткових затримок при обробці запитів через необхідність глибокого аналізу трафіку та високою вартістю інвентаризації та безперервного моніторингу застарілих або незадокументованих сервісів. Роль наслідків реалізації загроз у корпоративних інформаційних сервісах полягає у їх безпосередньому впливі на стратегічну стабільність та життєздатність бізнесу. Оскільки сучасні підприємства функціонують на основі мікросервісної архітектури, де API виступає головним сполучним елементом, успішна атака на один з компонентів через механізми

порушення авторизації об’єктів або функцій здатна дестабілізувати роботу всієї екосистеми. Таким чином, оцінка наслідків переміщує проблему безпеки прикладного коду з площини суто інженерних завдань IT-підрозділу на рівень стратегічного менеджменту організації [14].

1.3 Принципи та стандарти забезпечення безпеки API

Безпека API корпоративних інформаційних сервісів визначається як комплексна сукупність методологій, архітектурних підходів та інструментів, спрямованих на захист програмних інтерфейсів від несанкціонованого доступу, витоку конфіденційної інформації, модифікації критичних даних та інших деструктивних кібервпливів. Основне призначення механізмів безпеки API полягає в забезпеченні конфіденційності, цілісності та доступності інформаційних ресурсів організації, і не менш важливо, у створенні захищеного цифрового середовища, де кожен запит піддається суворій перевірці та легітимізації. Це закладає основу для подальшого проектування систем контролю доступу та моніторингу інформаційних потоків. Забезпечення стійкого захисту інтерфейсів базується на декількох фундаментальних характеристиках та концептуальних моделях. Однією з ключових характеристик є впровадження архітектурної моделі “нульової довіри” (Zero Trust). Її логіку можна описати наступним чином: будь-який суб’єкт або мережевий трафік апріорі вважається потенційно небезпечним, незалежно від його походження – чи надходить запит із зовнішньої мережі, чи з внутрішнього контуру організації. Модель нульової довіри вимагає постійної, безперервної верифікації та аналізу кожного інформаційного потоку.

Іншою важливою характеристикою є принцип тотальної валідації (Validate everything). Він передбачає обов’язкову перевірку структури та вмісту даних на всіх етапах їхнього проходження: у вхідних запитах, вихідних відповідях, процесах взаємодії зі сторонніми сервісами та безпосередньо в базах даних. Валідація обмежує можливість використання гнучких або занадто вільних схем даних, які ускладнюють контроль та створюють передумови для ін’єкцій шкідливого коду. Впровадження структурованої програми безпеки API надає

організаціям суттєві стратегічні та операційні переваги. З бізнесової точки зору, превентивний захист інтерфейсів дозволяє уникнути масштабних фінансових збитків та репутаційних втрат, які можливі через витік комерційних даних чи персональної інформації клієнтів. Зміцнення довіри з боку користувачів та партнерів стає додатковим драйвером для безпечного розгортання цифрових екосистем. Використання перевірених та стабільних бібліотек знижує ймовірність технічних помилок при реалізації бізнес-логіки. Автоматизація безпекового тестування (DevSecOps) дозволяє виявляти дефекти конфігурації та логіки на етапі розробки, підвищуючи загальну стійкість інфраструктури.

Разом з тим, реалізація високих стандартів безпеки супроводжується певними недоліками та викликами. Сучасні корпорації оперують сотнями інтерфейсів, частина з яких помилково вважається внутрішніми, тому звільняється від суворого контролю. Статистика кіберзагроз свідчить, що переважна більшість атак здійснюється саме автентифікованими користувачами, які намагаються обійти обмеження авторизації та отримати доступ до чужих об’єктів або конфіденційної інформації бізнесу. Тому, належна організація безпеки API забезпечує прозорість інфраструктури, унеможливлює появу невідомих інтерфейсів та гарантує, що корпоративні дані не будуть скомпроментовані через недоліки проектування або надмірне відкриття внутрішньої інформації [15].

Важливим аспектом при розробці чи підтримці програмного інтерфейсу є забезпечення захисту передачі та обробки даних. Цей захист визначається як комплекс архітектурних і процедурних заходів, спрямованих на верифікацію, обмеження та криптографічне забезпечення інформаційних потоків, що циркулюють між взаємодіючими вузлами. Головним призначенням цього захисного контуру є нейтралізація загрози несанкціонованого доступу, модифікації чи перехоплення відомостей, а також недопущення компрометації вищих рівнів привілеїв у системі. Ключовою характеристикою захисту обробки даних в API є сувора та безкомпромісна валідація вхідних параметрів на основі чітко задекларованих форматів і обмежень. Цей підхід вимагає використання

жорстко контрольовані схеми даних, що унеможливлює передачу аномальних запитів. Іншою важливою характеристикою є систематичне використання моделей атак і класифікаторів для виявлення дефектів логіки. У межах цієї діяльності стійкість безпеки API описується через протидію наступним категоріям атак:

- Підміна сутностей (Spoofing): перехоплення випадкових локальних портів або сокетів (Port/Socket Squatting) до ініціалізації API, online-перебір ідентифікаторів без штучного уповільнення обробки запитів (backoff) та примусове зниження рівня криптографічного захисту каналу (Downgrade Authentication)
- Атаки витоку інформації (Information Disclosure): перехоплення транзитних даних (Man-in-the-Middle), небезпечне збереження ключів доступу у загальнодоступних файлових директоріях замість апаратних модулів безпеки (HSM) та передбачуваність генераторів випадкових чисел у хмарних середовищах
- Атаки підвищення привілеїв (Elevation of Privilege): динамічне руйнування оперативної пам’яті (Dynamic Corruption), переповнення стеку чи купи через відсутність перевірки довжини параметрів, а також уразливості ланцюжка викликів (Call Chain), коли дочірні процеси або сторонні плагіни отримують несанкціонований доступ до батьківської логіки.

Головною перевагою впровадження зрілих архітектурних рішень із захисту API є можливість виявлення та нейтралізації критичних дефектів (таких як використання статичних паролів або впровадження стороннього коду) на етапі моделювання, що обходиться організації значно дешевше, ніж ліквідація наслідків успішних кібератак у продуктивному середовищі. Безпечно спроектований інтерфейс суттєво мінімізує когнітивне навантаження на програмістів та інженерів з інтеграції, надаючи їм готову безпекову специфікацію і позбавляючи необхідності проходити через складні, заплутані та схильні до помилок процедури виклику. Моделювання загроз обробки

розширює можливості інженерів з контролю якості (QA) та фахівців з тестування на проникнення, забезпечуючи їх детальною картою потенційно вразливих зон системи. Основним недоліком є значне ускладнення процесів проектування архітектури ПЗ на початкових етапах життєвого циклу розробки. Крім того, існує суттєвий ризик виникнення інженерних помилок через суб'єктивні чинники розробників. До них належать "сліпота творця" (creator blindness), коли проектувальники програмних інтерфейсів несвідомо ігнорують загрози на стиках взаємодії підсистем, або схильність приймати помилкові архітектурні рішення щодо надійності середовища обробки без належної верифікації. У середовищі сучасних корпоративних інформаційних систем захист передачі та обробки API відіграє роль критичного бар'єра безпеки, який ізолює внутрішній периметр організації та ядро бізнес-логіки від зовнішніх неконтрольованих сегментів мережі. За умови інтеграції корпоративної інфраструктури з хмарними обчисленнями (IaaS, Infrastructure as a Service) та підключення сторонніх сервісів, захищені інтерфейси запобігають транзитивному перенесенню кіберзагроз, локалізуючи потенційні інциденти всередині окремих ізольованих компонентів. Формалізація життєвого циклу облікових записів (Account Life Cycle) під час обробки запитів до API є базовою умовою для підтримки загальної цілісності корпоративних даних та забезпечення відповідності внутрішнім стандартам безпеки підприємства [16].

Гарантування безпеки на етапах передачі та обробки даних закладає надійний фундамент для захисту внутрішнього периметра інформаційної системи. Проте навіть найсуворіша валідація вхідних параметрів, криптографічний захист каналів та ретельне моделювання загроз на етапі проєктування неспроможні повністю прибрати ризики під час реальної експлуатації інтерфейсу. Будь-яка статична архітектурна модель стикається з динамікою виробничого середовища, де виникають непередбачувані аномалії трафіку, нові вектори атак та каскадні збої у ланцюжках залежностей. Саме тому логічним і необхідним продовженням комплексної стратегії захисту є перехід від превентивного моделювання до безперервного контролю системи у режимі реального часу. Це

актуалізує розгортання засобів моніторингу та управління безпекою API, які забезпечують спостережуваність інфраструктури, оперативну ізоляцію помилок та динамічне реагування на безпекові інциденти безпосередньо на етапі виконання. Ключовими характеристиками таких інструментів є спостережуваність (observability) та інтегрованість у децентралізовану інфраструктуру. Збір інформації здійснюється через інструментування середовища виконання, зокрема шляхом перехоплення даних на шлюзах API (API gateways), або через аналіз проєктної та розробницької інфраструктури. До основних функціональних властивостей належать автоматизоване звітування про помилки на різних точках взаємодії, трасування розподіленого трафіку для ізоляції збоїв, а також динамічне відображення безпекових подій за допомогою виробничих дашбордів (production-level dashboarding). Технічні та бізнесові переваги використання засобів моніторингу полягають у підвищенні загальної стійкості розподіленої системи та здатності оперативно реагувати на аномалії без створення операційних бар'єрів. Це дозволяє компаніям приймати обґрунтовані управлінські рішення на основі точних метрик, оптимізувати виділення обмежених ресурсів та мінімізувати каскадні збої по ланцюжках залежності. Недоліком є висока координаційна та фінансова вартість побудови ідеальної системи збору даних, оскільки забезпечення абсолютної повноти інформації вимагає значних бюджетних інвестицій та ресурсів. Крім того, зі збільшенням масштабів корпоративної екосистеми централізований контроль та ручний моніторинг стають неефективними, що створює ризик втрати видимості важливих процесів, якщо інструменти автоматизації не передані безпосередньо командам розробників. У великомасштабних корпоративних системах ці засоби трансформують модель управління від централізованого командного контролю до розподіленої відповідальності. Моніторинг є основою безпеки в умовах високої швидкості змін, коли нові компоненти додаються децентралізовано. Вони дозволяють перенести функції відстеження інцидентів безпосередньо на рівень крос-функціональних команд розробки, надаючи їм необхідну аналітичну підтримку та інструменти для захисту специфічних бізнес-доменів

[17].

У контексті постійного ускладнення архітектури корпоративних інформаційних систем та зростання ролі API як базового механізму інтеграції цифрових сервісів, особливого значення набуває стандартизація підходів до забезпечення їхньої безпеки. Розподілений характер сучасних API-орієнтованих середовищ, велика кількість міжсервісних взаємодій і висока динаміка змін інфраструктури формують передумови для виникнення нових кіберзагроз, пов'язаних із порушенням механізмів автентифікації та авторизації, несанкціонованим доступом до ресурсів, витоком конфіденційних даних і експлуатацією помилок конфігурації. За таких умов ефективне функціонування програмних інтерфейсів неможливе без використання міжнародних стандартів та рекомендацій, які визначають уніфіковані принципи управління безпекою API, регламентують механізми контролю доступу, захисту передавання даних та організації безперервного безпекового контролю в межах корпоративної цифрової інфраструктури. Одним із найбільш важливих міжнародних стандартів у сфері інформаційної безпеки є стандарт International Organization for Standardization ISO/IEC 27001, який визначає вимоги до створення та функціонування системи управління інформаційною безпекою. Положення цього стандарту формують загальні принципи захисту інформаційних систем, що мають безпосереднє значення для безпеки програмних інтерфейсів. Зокрема, стандарт передбачає необхідність реалізації контролю доступу, управління ризиками, використання криптографічних механізмів захисту та постійного моніторингу подій безпеки. Для API особливо важливими є принципи мінімальних привілеїв та сегментації доступу, оскільки вони дозволяють обмежити можливості несанкціонованої взаємодії із сервісами та зменшити потенційні наслідки компрометації облікових даних [18].

Важливу роль у формуванні практичних підходів до захисту API відіграють рекомендації організації OWASP Foundation, представлені в документі OWASP API Security Top 10. У межах цього документа систематизовано найбільш поширені та критичні вразливості API, які становлять загрозу для сучасних

інформаційних систем. До таких загроз належать порушення контролю авторизації, недостатній рівень автентифікації, надмірне розкриття даних, відсутність належного управління активами API та недостатній моніторинг безпеки. Значення рекомендацій OWASP полягає в тому, що вони орієнтовані не лише на виявлення потенційних вразливостей, але й на впровадження конкретних механізмів захисту, серед яких застосування багатофакторної автентифікації, перевірка прав доступу, обмеження кількості запитів та журналювання подій безпеки. Крім того, рекомендації OWASP акцентують увагу на необхідності врахування вимог безпеки ще на етапі проєктування програмних інтерфейсів, що відповідає концепції Secure by Design [19].

Суттєвий внесок у стандартизацію механізмів автентифікації та авторизації API зробила організація Internet Engineering Task Force шляхом розроблення протоколу OAuth 2.0. Цей протокол є міжнародним стандартом делегованої авторизації, який забезпечує можливість надання обмеженого доступу до ресурсів без передачі користувацьких облікових даних стороннім застосункам. Використання OAuth 2.0 дозволяє реалізувати безпечний механізм взаємодії між клієнтськими застосунками, серверами авторизації та ресурсними серверами, що є особливо актуальним для хмарних сервісів та мікросервісних архітектур. У межах цього стандарту визначаються механізми видачі токенів доступу, правила перевірки автентичності клієнтів та способи обмеження доступу до ресурсів залежно від встановлених прав. Завдяки цьому OAuth 2.0 став одним із найбільш поширених стандартів забезпечення безпеки API у сучасних корпоративних інформаційних системах [20].

Таким чином, міжнародні стандарти та рекомендації у сфері безпеки API формують комплексний підхід до захисту сучасних інформаційних систем від актуальних кіберзагроз. Використання положень ISO/IEC 27001 забезпечує створення системного підходу до управління інформаційною безпекою, рекомендації OWASP дозволяють своєчасно виявляти і усувати критичні вразливості API, а застосування протоколу OAuth 2.0 сприяє реалізації безпечних механізмів автентифікації та авторизації. Сукупне впровадження

зазначених стандартів і рекомендацій дозволяє підвищити рівень захищеності корпоративних інформаційних систем, а також мінімізувати ризики несанкціонованого доступу до цифрових ресурсів.

Висновки до розділу 1

Отже, API у сучасних корпоративних інформаційних системах є не лише технічним інструментом інтеграції сервісів, а й стратегічним компонентом цифрової інфраструктури підприємства. Їх використання забезпечує взаємодію між внутрішніми та зовнішніми системами, підтримує мікросервісну архітектуру, сприяє масштабованості бізнесу та формуванню цифрових екосистем. Водночас зростання ролі API супроводжується збільшенням кількості кіберзагроз, пов’язаних із порушенням механізмів автентифікації та авторизації, витоком даних, ін’єкційними атаками, надмірним розкриттям інформації та експлуатацією вразливостей бізнес-логіки.

Проведений аналіз показав, що безпека програмних інтерфейсів повинна розглядатися як комплексний процес, який охоплює всі етапи життєвого циклу API: від проєктування та розробки до моніторингу й експлуатації. Ефективний захист API базується на використанні сучасних моделей уонтролю доступу, криптографічного захисту каналів передачі даних, принципів Zero Trust та Secure by Design, а також механізмів постійного моніторингу та аналізу подій безпеки. Особливе значення мають міжнародні стандарти та рекомендації, зокрема ISO/IEC 27001, OWASP API Security Top 10 і OAuth 2.0, які формують уніфіковані підходи до забезпечення захищеності корпоративних інформаційних систем.

Таким чином, забезпечення безпеки API є критично важливою умовою стабільного функціонування сучасних корпоративних інформаційних систем, оскільки саме програмні інтерфейси виступають основним механізмом взаємодії між сервісами, користувачами та цифровою інфраструктурою підприємства. Зростання кількості кіберзагроз зумовлює необхідність впровадження комплексних методів і технологій захисту API. У зв’язку з цим особливого значення набуває дослідження сучасних підходів до автентифікації

та авторизації, технічних механізмів захисту передачі даних і контролю запитів, а також архітектурних рішень для забезпечення безпеки програмних інтерфейсів у мікросервісних та хмарних середовищах. Саме ці аспекти будуть детально розглянуті у наступному розділі.

Розділ 2 МЕТОДИ ТА ТЕХНОЛОГІЇ ЗАБЕЗПЕЧЕННЯ БЕЗПЕКИ API

2.1 Методи автентифікації та авторизації API

У сучасних умовах побудови корпоративних інформаційних систем безпека взаємодії між різнорідними компонентами покладається на чітко визначені інженерні протоколи перевірки суб'єктів. Процес автентифікації є фундаментальним елементом захисту програмних інтерфейсів. Він полягає у перевірці автентичності заявленої ідентичності користувача, клієнтського застосунку чи стороннього сервісу. Ще можна описати цю процедуру як підтвердження того, що суб'єкт, який намагається виконати запит до обчислювального ресурсу, дійсно є тим, за кого себе видає, шляхом надання специфічних доказів, таких як криптографічні ключі, цифрові сертифікати або інші секретні маркери. Основна мета перевірки особи користувача або сервісу в інфраструктури API полягає у мінімізації ризиків несанкціонованого доступу та запобіганні компрометації конфіденційних даних. В умовах розгортання хмарних обчислень та сервіс-орієнтованих архітектур, де компоненти системи взаємодіють через відкриті мережеві канали, автентифікація виступає першим критичним рубежем захисту периметра безпеки. Без надійної автентифікації неможливо забезпечити підзвітність дій в системі, оскільки сервіс-провайдер повинен точно фіксувати, який саме клієнт ініціював конкретну транзакцію чи запит на модифікацію стану даних [21]. У межах комп'ютерних наук важливим теоретичним аспектом є чітке розмежування визначень ідентифікації та автентифікації, які часто помилково ототожнюються у прикладних дискусіях. Ідентифікація є початковим процесом, під час якого суб'єкт лише заявляє про свою унікальність у системі за допомогою певного ідентифікатора, наприклад, імені користувача або унікального ідентифікатора клієнта (Client ID). На противагу цьому, автентифікація передбачає верифікацію цього твердження за допомогою пред'явлення автентифікаційних даних, які підтверджують володіння секретом, пов'язаним із цим ідентифікатором [22]. Таким чином, ідентифікація відповідає на питання про те, ким заявляє себе суб'єкт, тоді як автентифікація доводить правдивість цієї заяви за допомогою технічних засобів

перевірки. Типові сценарії використання автентифікації в API охоплюють широкий спектр міжсистемних взаємодій, зокрема автентифікацію клієнтських застосунків перед отриманням доступу до захищених ресурсів. У традиційних веб-архітектурах конфіденційні серверні застосунки проходять автентифікацію на кінцевих точках авторизаційного сервера (Token Endpoints) з використанням секретів клієнта (Client Secrets) або взаємного шифрування транспортного рівня (Mutual TLS) [21]. У мобільних та односторінкових браузерних застосунках, які класифікуються як публічні клієнти через неможливість безпечного збереження статичних секретів, використовуються динамічні криптографічні рішення для підтвердження автентичності даних на отримання токенів доступу [22].

Після успішного виконання процедури автентифікації та встановлення ідентичності суб’єкта, архітектура безпеки API ініціює наступний логічний етап контролю доступу, яким є авторизація. На відміну від перевірки автентичності, авторизація визначається як процес надання суб’єкту прав на виконання певних дій або отримання доступу до конкретних інформаційних ресурсів у системі. Відповідно до специфікацій сучасних стандартів контролю доступу, авторизація оперує поняттями дозволів, областей дій та політик безпеки, що дозволяє регулювати операції читання, створення, оновлення чи видалення об’єктів даних через програмні інтерфейси [22]. Перевірка прав доступу до ресурсів API здійснюється безпосередньо на рівні серверів ресурсів або шлюзів (API Gateways), які аналізують атрибути запиту та супровідні мандати доступу. У сучасних хмарних інфраструктурах для цього застосовуються токени доступу (Access Tokens), які інкапсулюють інформацію про дозволені області дії (Scopes) та специфічні бізнес-атрибути, представлені у вигляді тверджень [21]. Сервер ресурсів інтерпретує ці токени для визначення легітимності операції, зіставляючи запитувану дію клієнта з переліком дозволів, зафіксованих у структурі токена [22]. Проектування ефективних систем авторизації базується на принципі мінімальних привілеїв, який вимагає, щоб будь-який суб’єкт системи володів лише тим обмеженим набором прав, який є абсолютно необхідним для виконання його поточного функціонального завдання.

Реалізація цього принципу у середовищах інфраструктур мікросервісів мінімізує потенційні збитки від сожливої компрометації маркерів доступу. Замість надання універсальних адміністративних повноважень, архітектура розподілених токенів дозволяє динамічно звужувати права доступу за допомогою механізму обміну токенів (Token Exchange), забезпечуючи трансляцію широких зовнішніх дозволів у вузькоспеціалізовані внутрішні мандати для міжсервісної взаємодії [21]. Взаємозв'язок між автентифікацією та авторизацією має характер послідовної та нерозривної технологічної залежності, де один процес виступає логічним попередником для іншого. Автентифікація завжди передує авторизації, оскільки неможливо ухвалити обґрунтоване рішення про надання прав доступу без попереднього точного встановлення особи чи сервісу, який ці права запитує. Проте, успішна автентифікація автоматично не гарантує отримання доступу до ресурсів, оскільки остаточне рішення залежить від матриці доступу та відповідності запиту чинним політикам безпеки, що підкреслює автономність та специфіку етапу авторизації в загальному життєвому циклі обробки мережевого запиту [21].

Проектування систем захисту API вимагає дотримання комплексу архітектурних вимог, серед яких найважливіше місце посідає забезпечення конфіденційності інформаційних потоків. Конфіденційність у контексті контролю доступу гарантує, що конфіденційні автентифікаційні дані, секрети клієнтів та маркери доступу захищені від несанкціонованого перехоплення чи ознайомлення третіми сторонами під час їх транспортування та збереження. Для виконання цієї вимоги сучасні стандарти безпеки суворо регламентують обов'язкове використання шифрування транспортного рівня для всіх кінцевих точок авторизації та обміну даними, що виключає можливість передачі критичних даних у відкритому текстовому форматі [22]. Не менш важливою вимогою є забезпечення цілісності даних, яка полягає у запобіганні несанкціонованій або випадковій модифікації інформації під час її передачі між клієнтом і сервісом. Механізми контролю доступу повинні гарантувати, що самі

маркери авторизації, такі як структуровані токени JWT, захищені від підробки чи перевстановлення зловмисниками. Застосування цифрових підписів на основі асиметричної криптографії дозволяє серверам ресурсів математично верифікувати, що вміст токена не був змінений після його генерації довіреним сервером авторизації, забезпечуючи тим самим високий рівень довіри до інкапсульованих у ньому прав доступу [21]. Безпечне управління обліковими даними є наріжним каменем стійкості всієї системи безпеки, що передбачає суворі правила генерації, збереження, оновлення та відкликання ідентифікаційних токенів і ключів. Специфікації безпеки забороняють передачу конфіденційних маркерів у URL-адресах через ризик їх логування проміжними серверами, а також вимагають впровадження механізмів негайного відкликання (Token Revocation) у разі виявлення ознак компрометації або при закритті сесії користувачем. Крім того, архітектура безпеки повинна підтримувати обмежений час життя токенів доступу та використання захищених оновлювальних маркерів (Refresh Tokens) для автоматичного продовження сесій без повторного запиту облікових даних користувача [22]. Масштабованість та централізоване управління доступом є критичними вимогами для сучасних корпоративних екосистем, де кількість сервісів та обсяги трафіку постійно зростають. Децентралізована перевірка прав на кожному окремому мікросервісі призводить до ускладнення підтримки коду та підвищує ймовірність виникнення помилок у конфігураціях безпеки. Рішенням такої проблеми є побудова архітектури на основі концепції нульової довіри (Zero Trust Architecture), де функції автентифікації користувачів та первинної валідації запитів делегуються централізованому серверу авторизації та шлюзам API. Такий підхід забезпечує єдину точку застосування політик безпеки, що дозволяє гнучко масштабувати систему, спрощує проведення аудиту доступу та гарантує консистентність захисту інформаційних ресурсів організації [21].

Для широкого розуміння та порівняння процесів авторизації та автентифікації, необхідно розглянути кожен з них детально. Є чимало методів автентифікації API, кожен з яких має свою структуру, свою систему та свої особливості.

Автентифікація за допомогою API-ключів являє собою один із найбільш фундаментальних механізмів ідентифікації клієнта. В основі цього підходу лежить використання унікального ідентифікатора, який генерується сервером та передається клієнту для подальшого використання у кожному запиті. Цей ключ діє як довгостроковий ідентифікатор суб’єкта, що дозволяє системі розмежувати запити та застосовувати відповідні політики доступу. Ефективність такого методу безпосередньо залежить від впровадження надійних фреймворків управління ключами, оскільки скомпрометований ключ надає необмежений доступ до ресурсів до моменту його відкликання. Хоча простота впровадження робить цей метод привабливим для швидкої інтеграції, він часто вимагає додаткових засобів захисту під час передачі, оскільки сам по собі ключ не забезпечує конфіденційності запиту [23]. Наступним етапом розвитку методів доступу є механізм Basic Authentication. Відповідно до принципів архітектури сучасних веб-систем, цей метод базується на передачі облікових даних у заголовках HTTP-запиту, зазвичай закодованих за допомогою алгоритму Base64. Незважаючи на свою універсальність та відповідність базовим протоколам передачі даних, Basic Authentication має суттєві недоліки, пов’язані з безпекою, оскільки облікові дані фактично передаються з кожним запитом, що створює ризики перехоплення інформації у випадку відсутності надійного шифрування каналу зв’язку, наприклад, через TLS [24]. У сучасних розподілених системах значного поширення набули методи, що базуються на використанні спеціальних токенів (Token Based Authentication). Основна ідея такого підходу полягає у відокремленні процесу автентифікації від процесу отримання доступу до ресурсів. Після успішної перевірки ідентичності користувача, сервер видає токен, який клієнт використовує для наступних викликів API протягом певного терміну дії. Окремим технологічно досконалим стандартом у цій групі є JWT (JSON Web Token). Він дозволяє передавати інформацію про об’єкт автентифікації у компактному, безпечному для URL-адрес форматі, що складається з трьох частин: заголовка, корисної інформації та підпису. Використання цифрового підпису гарантує цілісність переданих даних,

дозволяючи отримувачу перевірити легітимність токена без необхідності звернення до бази даних користувачів при кожному запиті, що суттєво оптимізує продуктивність системи [25]. Більш складним та гнучким інструментом управління правами доступу є протокол OAuth 2.0. Він пропонує ієрархічний підхід, де клієнт отримує доступ до захищених ресурсів від імені власника, використовуючи “маркери доступу” (access tokens), а не облікові дані користувача. Цей протокол чітко розмежовує ролі клієнта, власника ресурсу, сервера авторизації та сервера ресурсів, що дозволяє реалізовувати сценарії делегованого доступу, де треті сторони отримують обмежений доступ до даних без передачі паролів [20]. Надбудовою над протоколом OAuth 2.0 є OpenID Connect, який розширює функціонал авторизації, додаючи івень ідентифікації користувача. Це дозволяє клієнтським програмам отримувати інформацію про профіль користувача, що успішно пройшов автентифікацію через сервер авторизації, базуючись на даних ID Token. Така синергія OAuth 2.0 та OpenID Connect створює надійну екосистему для федеративної ідентифікації, яка стала стандартом де-факто для інтеграції сервісів у сучасних корпоративних інформаційних системах [26]. Важливим елементом у посилені захищеності доступу є багатофакторна автентифікація (MFA). Вона передбачає поєднання кількох незалежних методів підтвердження особи: того, що користувач знає (пароль), того, що має (смартфон, токен) або того, чим він є (біометричні дані). Впровадження MFA в архітектуру API є превентивним заходом проти несанкціонованого доступу, оскільки значно ускладнює компрометацію облікового запису навіть у випадку витоку пароля, вимагаючи додаткового фактора верифікації для завершення сесії. Застосування вказаних методів у сукупності створює багаторівневий периметр безпеки, адаптований до викликів сучасного кіберпростору [23].

Надійна автентифікація є лише першим етапом контролю, який дозволяє однозначно перевірити справжність заявленої ідентичності суб’єкта. Проте успішне встановлення особи користувача або сервісу не дає відповіді на критичне питання щодо обсягу його повноважень у межах конкретної сесії

інформаційного обміну. Тому наступним логічним шаром безпеки виступає авторизація API – динамічний процес розмежування прав доступу, покликаний гарантувати, що автентифікований суб’єкт зможе виконувати лише ті операції та отримувати доступ виключно до тих ресурсів, які безпосередньо необхідні для виконання його функціональних завдань. Нижче наведено аналіз сучасних архітектурних моделей та політик авторизації, що дозволяють реалізувати гранульований контроль доступу в розподілених інформаційних системах. У сучасній практиці проектування безпеки API концепція управління доступом на основі ролей RBAC посідає одне з центральних місць серед механізмів забезпечення конфіденційності та цілісності інформаційних ресурсів. В основі цієї моделі лежить принцип абстрагування конкретного суб’єкта доступу від його безпосередніх повноважень через введення проміжного конструкту – ролі, що описує функціональні обов’язки користувача або сервісу в межах організаційної структури підприємства [27]. Формалізація цього підходу дозволяє мінімізувати складність адміністрування прав доступу, оскільки замість зіставлення індивідуальних дозволів для кожного окремого ідентифікатора (User ID) операції авторизації групуються навколо визначених функціональних профілів. З огляду безпеки API, кодній ролі відповідає чітко окреслений набір дозволених методів передачі даних (наприклад, GET, POST, PUT, DELETE) для окремих кінцевих точок (endpoints), що мінімізує ймовірність помилок конфігурації у великих розподілених системах [17]. На практиці реалізація цієї моделі передбачає ієрархічне або дескриптивне структурування користувачів відповідно до їх виробничої діяльності та потребах у доступі до інформаційних масивів. Класичним прикладом розмежування повноважень між такими типовими ролями, як “Адміністратор”, “Менеджер” та “Клієнт”. Роль адміністратора наділяється максимальним спектром повноважень, що охоплює не лише читання й модифікацію даних, а й деструктивні операції, а також конфігурування параметрів самої системи через відповідний адміністративний API [17]. На противагу цьому, роль менеджера зазвичай обмежується доступом до бізнес-ресурсів (наприклад, генерація звітів,

редагування карток товарів), але позбавлена прав на системне налаштування. Клієнтська роль репрезентує кінцевого користувача з мінімальним набором прав, орієнтованих суто на персональні дані та базовий функціонал системи. У складніших сценаріях розробки існують також спеціалізовані ролі розробників та автоматизованих сервісних облікових записів (Service Accounts), вони мають свій унікальний рівень доступу до відповідного функціонального шар програмного забезпечення [17]. Головна наукова та практична цінність застосування RBAC полягає у забезпеченні масштабованого та централізованого управління політиками безпеки. Замість хаотичного розподілу прав прямо в коді окремих мікросервісів, архітектура системи передбачає наявність єдиного диспетчера авторизації, що спрощує процедуру аудиту та верифікації відповідності нормативним вимогам безпеки. У разі зміни організаційної структури або звільнення співробітника адміністратору необхідно всього відкликати або змінити роль в централізованому репозиторії ідентифікаторів, після чого оновлені правила автоматично застосуються до всіх інтегрованих API. Це суттєво зменшує так звану поверхню атаки та нівелює ризики, пов’язані з наявністю надлишкових, забутих або неактуальних прав доступу у користувачів, що є критично важливим для забезпечення стійкості корпоративних інформаційних систем [17].

Незважаючи на високу ефективність RBAC у статичних середовищах, стрімкий розвиток динамічних хмарних сервісів виявив обмеженість цієї моделі, зокрема так званий “рольовий вибух” (role explosion), коли для забезпечення специфічних умов доступу доводиться створювати сотні штучних підролей. На зміну або на доповнення їй прийшла парадигма контролю доступу на основі атрибутів ABAC, яка пропонує більш контекстно-залежний підхід. У моделі ABAC рішення про надання доступу приймається в режимі реального часу на основі оцінки сукупності характеристик, які поділяються на чотири базові характеристики: атрибути суб’єкта (посада, рівень допуску), атрибути ресурсу (тип даних, рівень конфіденційності інформації), атрибути дії (читання, запис, виконання) та атрибути середовища (зовнішні умови здійснення транзакції)

[27]. Ключовою відмінністю та науково-технічною перевагою ABAC є здатність здійснювати глибокий аналіз контексту кожного окремого до API:

- Місцезнаходження: Система авторизації здатна перевіряти мережеву адресу (IP-адресу) або географічні координати відправника запиту, блокуючи спроби доступу до конфіденційного API з-за меж корпоративного периметра або з країн з високим рівнем кіберзагроз.
- Час: Авторизаційні політики можуть обмежувати виконання критично важливих операцій (наприклад, проведення великих фінансових транзакцій через API) виключно межами офіційного робочого часу організації, запобігаючи несанкціонованим діям у нічний період.
- Тип пристрою: Доступ до ресурсів диференціюється залежно від того, чи здійснюється запит з верифікованого корпоративного пристрою з оновленим антивірусним захистом, чи з персонального пристрою користувача, для якого діють жорсткі обмеження.
- Рівень ризику: Сучасні інтелектуальні системи ABAC інтегруються з модулями аналітики поведінки суб'єктів, розраховуючи динамічний індекс ризику. Якщо поведінка користувача виглядає аномальною для його профілю, API автоматично вимагатиме додаткової верифікації або заблокує сесію [27].

Використання логічних виразів та булевої алгебри для опису правил доступу в ABAC забезпечує гнучкість формування безпекового контуру підприємства. Політика безпеки може бути сформована у вигляді складного інтегрального правила: "Дозволити доступ до методу модифікації бази даних лише за умови, що суб'єкт має роль менеджера, підключений через захищений корпоративний VPN, здійснює запит у робочі години, а рівень компрометації його пристрою визначено як нульовий". Така дескриптивна модель дозволяє адаптувати систему захисту до будь-яких бізнес-сценаріїв без необхідності внесення змін до архітектури самого API або переписування програмного коду сервісів, що суттєво підвищує загальний рівень кіберрезистентності інформаційної інфраструктури [27].

У контексті міжсистемної взаємодії та інтеграції сторонніх додатків загальноприйнятим стандартом авторизації є OAuth 2.0. Одним із фундаментальних механізмів обмеження прав у межах цього фреймворку є концепція областей доступу, відомих як “scopes” [17]. Область доступу (scope) є текстовим маркером, який клієнтський застосунок запитує у сервера авторизації під час ініціалізації процедури автентифікації користувача. Scopes не визначають права самого користувача всередині системи, а натомість декларують набір дозволів, які користувач свідомо та явно делегує конкретному сторонньому ПЗ для виконання визначених завдань [29]. Практичне впровадження scope-based авторизації дозволяє реалізувати принцип найменших привілеїв на рівні взаємодії між різними прикладними інтерфейсами. Сервер ресурсів (Resource Server), отримуючи маркер доступу (Access Token), в обов’язковому порядку валідує наявність відповідних міток всередині токена [17]. Наприклад, якщо клієнтський додаток має маркер з міткою read:profile, він зможе успішно виконати запит до кінцевої точки профілю користувача, проте спроба здійснити запис або надіслати POST-запит до цієї ж чи суміжної адреси буде негайно відхилена сервером з поверненням HTTP-статусу 403 Forbidden через відсутність іншої необхідної мітки, наприклад write:profile [29]. Такий підхід унеможливлює ситуації, коли скомпрометований сторонній інтеграційний модуль отримує повний і безконтрольний доступ до всієї екосистеми API. Науковий інтерес у контексті OAuth 2.0 викликає саме механізм безпечного делегування повноважень без передачі конфіденційних автентифікаційних даних третім сторонам. Користувач виступає в ролі власника ресурсу (Resource Owner), який авторизує сторонній сервіс діяти від свого імені виключно в чітко окреслених межах. Маркер доступу, що випускається сервером авторизації після підтвердження користувачем, стає тимчасовим цифровим мандатом, чинність якого обмежена в часі і за спектром доступним дій через механізм scopes. Завдяки цьому забезпечується високий рівень довіри та безпеки у відкритих веб-середовищах, де взаємодіють програмні продукти від різних незалежних розробників та

вендорів [17].

Сучасний ландшафт кіберзагроз, що характеризується розмиванням класичного мережевого периметра через масовий перехід організацій на хмарні обчислення та мікросервісні архітектури, зумовив перехід до нової фундаментальної парадигми безпеки – концепції “Нульової довіри” (Zero Trust). Відповідно до базових засад, сформульованих Національним інститутом стандартів і технологій США (NIST) у спеціальній публікації SP 800-27, архітектура нульової довіри (ZTA) повністю відкидає застаріле припущення про те, що будь-який суб’єкт або пристрій всередині локальної мережі підприємства є апріорі безпечним і заслуговує на довіру. Будь-яка транзакція, незалежно від її походження - з глобального інтернету чи з внутрішнього дата-центру – розглядається як потенційно ворожа до моменту повної та всебічної перевірки [27]. Реалізація принципів Zero Trust на рівні авторизації програмного інтерфейсу вимагає безперервного, динамічного контролю кожного вхідного запиту. Традиційний підхід, за якого автентифікація відбувається лише один раз на початку сесії (наприклад, при отриманні сесійної куки), вважається невідповідним стандартам ZTA. Кожне звернення до будь-якої кінцевої точки API має супроводжуватися передачею криптографічно захищеного токена доступу, який підлягає миттєвій валідації на предмет актуальності повноважень, цілісності структури та незмінності контексту середовища. Більше того, тривалість життя таких токенів мінімізується, а процедури їх оновлення вимагають регулярної невидимої для користувача перевірки поточного стану безпеки пристрою та облікового запису [27]. У межах мікросервісних систем, концепція нульової довіри трансформується у вимогу повної ліквідації неконтрольованої горизонтальної взаємодії (lateral movement). Сервіси більше не довіряють один одному лише тому, що вони розгорнуті в одному віртуальному кластері або підключені до спільної внутрішньої мережі [27]. Кожен мікросервіс виступає як ізольований сервер ресурсів, який вимагає від суміжного сервісу пред’явлення унікального сертифіката взаємної автентифікації (mTLS) та токена авторизації, що підтверджує право на

виконання конкретної міжсервісної операції. Це дозволяє локалізувати потенційну компрометацію: навіть якщо зловмисник отримає контроль над одним із другорядних вузлів системи, він не зможе автоматично отримати доступ до інших критичних компонентів інфраструктури [17]. Для сучасних хмарних інфраструктур, що динамічно масштабуються і оперують тисячами контейнерів, впровадження Zero Trust та інтегрованих методів авторизації (таких як ABAC та OAuth 2.0 scopes) є єдиним дієвим способом забезпечення інформаційної безпеки. Розподіл архітектури на площину управління (Control Plane) та площину даних (Data Plane) дозволяє централізовано транслювати політики безпеки на рівні шлюзів API або спеціалізованих сервісних мереж (Service Mesh). Такий рівень авторизації та гранульованого контролю гарантує безперервний захист корпоративних активів в умовах постійної видозміни хмарного ландшафту та високої інтенсивності інформаційних потоків [27].

Дослідження сучасних парадигм захисту корпоративних інформаційних систем передбачає ретельний аналіз методів розмежування доступу та ідентифікації суб’єктів у мережевому просторі. Систематизація вище розглянутих технологічних засад функціонування інтерфейсів API створює підґрунтя для їхнього детального порівняльного аналізу. Об’єктивна верифікація ефективності цих засобів захисту спирається на баланс між криптографічною стійкістю, складністю інженерної реалізації та сукупною вартістю володіння інфраструктурою, які безпосередньо впливають на стійкість інформаційних систем. Ключове місце при розробці посідає вибір оптимального механізму перевірки автентичності та повноважень суб’єктів взаємодії. Ключі API (API Keys) представляють собою статичні текстові рядки, які передаються у заголовках запитів, або як параметри URL-адреси, забезпечуючи найпростішу ідентифікацію клієнтського застосунку без прив’язки до конкретного користувача. Головним недоліком цього підходу є відсутність внутрішнього контексту безпеки та незмінність ключа протягом тривалого часу, що робить його компрометацію критичною для всієї системи [29]. На відміну від статичних ключів, веб-токени JSON є структурованими автономними

об’єктами, що містять у собі підписаний набір тверджень про суб’єкта, його права й термін дії самого документа. Це дозволяє серверам ресурсів самостійно перевіряти автентичність токена за допомогою криптографічних ключів без додаткових звернень до централізованої бази даних [30]. Протокол OAuth 2.0 виступає комплексним індустріальним стандартом делегованої авторизації, який не просто визначає формат токена, а описує чіткі ролі та взаємодії між власником ресурсу, клієнтським застосунком, сервером авторизації та сервером ресурсів [29, 30]. Завдяки такій архітектурі OAuth 2.0 забезпечує чітке розмежування процесів автентифікації та безпосередньої авторизації, мінімізуючи ризики передачі конфіденційних облікових даних користувача третім особам [30]. Аналіз стійкості зазначених методів до поширених векторів кібератак демонструє суттєві відмінності у рівнях захисту, які вони здатні забезпечити. Механізм API Key є найбільш вразливим до перехоплення в процесі передачі даних, оскільки зловмисник, отримавши статичний ключ, отримує необмежений у часі доступ до ресурсів від імені легітимного клієнта. Токени JWT мають значно вищий рівень захисту завдяки обов’язковому використанню цифрових підписів, проте вони залишаються чутливими до акак типу перехоплення токена, якщо не забезпечено належне шифрування транспортного рівня, а також до компрометації секретних ключів підпису на стороні сервера. Критичною загрозою для JWT є витік токенів з безмежним або надто тривалим терміном дії, що дозволяє неавторизованим сутностям безперешкодно виконувати уособлення легітимних користувачів [30]. Протокол OAuth 2.0 демонструє найвищу стійкість до складних атак завдяки впровадженню обмежених у часі токенів доступу та механізмів їх оновлення, а також використанню специфічних профілів безпеки для різних типів клієнтів. Наприклад, при використанні OAuth 2.0 у мобільних застосунках існує ризик перехоплення коду авторизації через специфіку обробки зворотніх викликів операційною системою пристрою. Для нівелювання цієї вразливості та захисту від подібних маніпуляцій застосовується розширення RFC 7636, відоме як ключ підтвердження обміну кодом, що забезпечує криптографічну верифікацію того,

що токен запитує саме той застосунок, який ініціював процес авторизації [30].

З точки зору початкового впровадження та швидкості розробки, використання ключів API характеризується мінімальним порогом входження, оскільки вимагає лише генерації унікального рядка на стороні сервера та його прямого порівняння при кожному публічному HTTP-запиті. Це робить цей метод ідеальним для публічних інформаційних сервісів або простих внутрішніх систем, де швидкість інтеграції сторонніх розробників є пріоритетом. Реалізація JWT вимагає складнішої логіки, яка включає підключення спеціалізованих криптографічних бібліотек для формування заголовків, кодування корисного навантаження, обчислення цифрових підписів та валідації терміну дії на стороні сервера ресурсів [30]. Найбільш трудомістким для розгортання є протокол OAuth 2.0, оскільки розробникам необхідно спроектувати та налаштувати повноцінну інфраструктуру, яка підтримує різноманітні сценарії авторизації, обробку запитів на автентифікацію, управління життєвим циклом токенів доступу та токенів оновлення, а також взаємодію з інтерфейсами надання згоди користувача [29, 30]. Ефективність масштабування архітектури безпосередньо залежить від обраного способу управління станом сесії безпеки. Використання традиційних API Key часто змушує сервер виконувати запити до централізованого середовища або бази даних при кожному зверненні, щоб перевірити активність та права ключа, що створює значне навантаження на систему при зростанні трафіку. Веб-токени JWT вирішують цю проблему завдяки своїй автономній природі, оскільки вся необхідна інформація про користувача та його повноваження вже міститься всередині самого токена. Це дозволяє реалізувати таку архітектуру серверів додатків, яка може бути масштабованою без потреби в синхронізації стану або постійних зверненнях до центральної бази даних для кожної операції верифікації [30]. У межах інфраструктури OAuth 2.0 висока масштабованість досягається за рахунок концептуального та фізичного відокремлення сервера авторизації, який бере на себе все обчислювальне навантаження з автентифікації користувачів та видачі маркерів, від серверів ресурсів, які лише валідують отримані токени та

безпосередньо обробляють бізнес-логіку запитів [30]. Розгортання систем захисту API накладає специфічні вимоги до технічного стеку та архітектури корпоративного середовища. Для ключів API інфраструктурні вимоги мінімальні та обмежуються наявністю захищеного каналу передачі даних і сховища для збереження відповідності ключів ідентифікаторів клієнтів. Для роботи з JWT необхідно забезпечити надійні механізми розподілу та оновлення криптографічних ключів, а також узгодженість алгоритмів шифрування між усіма компонентами системи, що взаємодіють з токенами [30]. Найбільш жорсткі та комплексні вимоги висуває протокол OAuth 2.0, який функціонує як повноцінна екосистема. Корпоративна інфраструктура в такому випадку повинна включати виділені сервери авторизації, інтегровані з внутрішніми службами каталогів або провайдерами ідентифікації для перевірки облікових даних. Крім того, виникає потреба в автоматизації процесів реєстрації клієнтських застосунків, моніторингу відкликання токенів, а також у впровадженні додаткових захисних шлюзів, які здатні координувати складні потоки автентифікації та авторизації на межі корпоративного контуру [30].

Процес архітектурного проектування корпоративних систем вимагає диференційованого підходу до вибору засобів захисту залежно від цільової аудиторії та призначення програмних інтерфейсів. Для зовнішніх публічних API, метою яких є швидке залучення сторонніх розробників та надання базового доступу до відкритих інформаційних ресурсів або сервісів монетизації, раціональним вибором часто стає використання ключів API у поєднанні з обмеженням частоти запитів [29, 30]. Якщо підприємство будує відкриту платформу партнерської взаємодії, яка передбачає інтеграцію зі складними сторонніми бізнес-застосунками та обробку конфіденційних даних користувачів, обов’язковим є впровадження протоколу OAuth 2.0 для забезпечення безпечного делегування повноважень [29, 30]. Для внутрішніх інтеграційних рішень, де взаємодія відбувається суто між довіреними компонентами корпоративної мережі, використання JWT дозволяє побудувати швидку систему обміну контекстом безпеки без надлишкових інфраструктурних

ускладнень [30]. Сучасні тенденції розвитку корпоративного програмного забезпечення характеризуються переходом від монолітних рішень до розподілених сервісних архітектур, що безпосередньо впливає на стратегію захисту програмних інтерфейсів. У мікросервісному середовищі передача статичних ключів між сервісами є неефективною та небезпечною практикою, оскільки компрометація одного внутрішнього компонента ставить під загрозу безпеки всієї мережі. Оптимальним архітектурним шаблоном тут стає використання концепції єдиного входу на основі зв'язки протоколу OAuth 2.0 та розширення OpenID Connect, яке виступає стандартом для автентифікації користувачів. Сервер авторизації здійснює первинну перевірку суб'єкта, після чого генерує автономний токен доступу. Надалі цей токен передається між мікросервісами, дозволяючи кожному з них самостійно і без затримок інтерпретувати права доступу клієнта, мінімізуючи мережеві затримки та усуваючи єдину точку відмови в особі центрального сервера автентифікації [30]. Прийняття остаточного рішення щодо впровадження конкретного механізму безпеки в корпоративних системах завжди супроводжується пошуком компромісу між рівнем захищеності та швидкодією системи. Підвищення рівня безпеки шляхом додавання багатоетапних перевірок, шифрування та динамічної валідації неминуче створює додаткове обчислювальне навантаження та збільшує час відгуку програмного інтерфейсу. Для оптимізації продуктивності корпоративних API часто застосовується механізм кешування даних, який дозволяє суттєво знизити кількість повторних звернень до сховищ та прискорити обробку запитів. Водночас архітектори повинні враховувати, що кешування вимагає значних обсягів оперативної пам'яті та має певні обмеження, оскільки воно ефективне лише тоді, коли однакові набори даних запитуються багаторазово, і є неприпустимим для динамічних операцій з високими вимогами до конфіденційності. У критично важливих бізнес-транзакціях, де ціна помилки або несанкціонованого доступу є надзвичайно високою, архітектура повинна жертвувати продуктивністю на користь максимального захисту. У таких сценаріях доцільно впроваджувати

динамічну перевірку контексту безпосередньо в момент виконання операції, включаючи механізми покрокової або посиленої автентифікації, які вимагають від користувача повторного підтвердження особи перед виконанням чутливих дій, мінімізуючи ризики викоритсання перехоплених або застарілих токенів доступу [30].

2.2 Методи технічного захисту API: шифрування, обмеження запитів (rate limiting), валідація даних

Стрімка еволюція та розширення ландшафтів використання API в корпоративних інформаційних системах супроводжується виникненням специфічних ризиків, які роблять їх пріоритетною мішенню для кібератак. Найбільш критичними деструктивними чинниками є загрози, спрямовані на експлуатацію вразливостей логіки додатків та механізмів автентифікації на рівні кінцевих точок. Зокрема, значну небезпеку становлять атаки, орієнтовані на компрометацію легітимних сесій, модифікацію параметрів запитів та несанкціоноване отримання доступу до конфіденційних об’єктів даних. Іншим поширеним вектором залишаються ін’єкційні атаки, які виникають внаслідок відсутності або недостатності суворої валідації вхідних даних на стороні сервера, що дозволяє зловмисникам виконувати шкідливі сценарії [31]. Окрім цього, критичною загрозою для інфраструктури є автоматизовані атак, спрямовані на вичерпання обчислювальних ресурсів за допомогою надмірної кількості запитів, що призводить до дестабілізації або повної відмови від застарілих підходів до безпеки та впровадження спеціалізованх системно-технічних інструментів контролю трафіку додатків [32]. Інтенсифікація процесів мікросервісної інтеграції та перехід організацій до хмарних обчислень зумовлюють об’єктивну потребу у застосуванні багаторівневих засобів технічного захисту API. Забезпечення безпеки виключно на рівні мережевого периметра за допомогою традиційних міжмережевих екранів є малоефективним, оскільки трафік інтерфейсів прикладного програмування використовує легітимні протоколи і зазвичай сприймається мережевим обладнанням як безпечний. Відповідно, виникає необхідність у

глибокому аналізі семантики запитів та перевірці їх відповідності встановленим схемам безпосередньо на рівні прикладних програмних інтерфейсів [32]. Окрім архітектурних передумов, необхідність впровадження технічних засобів захисту чітко регламентується сучасними міжнародними нормативно-правовими актами у сфері кібербезпеки та захисту інформації. Зокрема, Загальний регламент про захист даних (GDPR) встановлює імперативні вимоги щодо реалізації концепцій “безпека за замовчуванням” (Security by Default) та “безпека за проектуванням” (Security by Design), що зобов’язує розробників та операторів систем інтегрувати захисні механізми на кожному етапі життєвого циклу програмного інтерфейсу, який оперує персональними даними. Недотримання цих технічних стандартів створює передумови для масштабних витоків інформації, що тягне за собою суворі правові санкції, фінансові втрати та довгострокові репутаційні ризики для установ [33]. Забезпечення стійкості та цілісності сучасних корпоративних систем безпосередньо залежить від ефективності інтегрованих захисних механізмів, які здійснюють безперервний моніторинг та фільтрацію інформаційних потоків. Головну роль у цій архітектурі відіграють спеціалізовані рішення класу API Security та міжмережеві екрани для захисту веб-додатків (Web Application Firewalls), які виконують валідацію вхідних і вихідних повідомлень на відповідність специфікаціям, блокуючи несанкціоновані або аномальні запити до внутрішніх сегментів мережі [32]. Важливою складовою технічного захисту КІС є впровадження строгих криптографічних протоколів для шифрування даних під час транзиту, а також використання надійних методів автентифікації та контролю доступу суб’єктів до системних ресурсів. Для запобіганню перевантаженню інфраструктури та нівелювання загроз типу “відмова в обслуговуванні” захисні механізми забезпечують лімітування частоти запитів (Rate Limiting), що гарантує стабільну пропускну здатність системи навіть в умовах підвищеної мережевої активності [31]. Комплексна інтеграція технічних інструментів захисту з централізованими системами аудиту подій безпеки дозволяє корпораціям оперативно ідентифікувати потенційні інциденти, забезпечувати повну

відповідність правовим нормам регуляторів та мінімізувати ймовірність компрометації критично важливих бізнес-процесів організації [33].

Забезпечення захисту сучасних інтерфейсів прикладного програмування є першочерговим завданням в архітектурі розподілених інформаційних систем, оскільки через ці шлюзи передаються значні обсяги критично важливих даних. Фундаментальним базисом для побудови безпечного каналу взаємодії між клієнтом та сервером виступає криптографічний протокол TLS. Він функціонує в поєднанні з протоколом HTTP, утворюючи захищене розширення HTTPS. Основна мета впровадження даного технологічного стеку полягає в автентифікації взаємодіючих сторін, забезпеченні конфіденційності трансляції інформаційних пакетів та гарантуванні цілісності переданих повідомлень шляхом запобігання несанкціонованому перехопленню або модифікації даних зловмисниками [2]. Сучасний протокол TLS версії 1.3 суттєво оптимізує процес встановлення зв’язку (handshake) порівняно зі своїми попередниками, мінімізуючи часові затримки та відсікаючи застарілі, потенційно вразливі криптографічні алгоритми [34]. Процедура ініціалізації сесії починається з надсилання клієнтом запиту ClientHello, що містить перелік підтримуваних шифронаборів (cipher suites) та параметри ключів. Сервер відповідає повідомленням ServerHello, обираючи оптимальний набір шифрування, та надає свій цифровий сертифікат для перевірки автентичності. Важливою архітектурною особливістю TLS 1.3 є обов’язкове використання механізмів диференційованого обміну ключами на основі алгоритму Діффі-Гелмана на еліптичних кривих (Elliptic Curve Diffie-Hellman Ephemeral, ECDHE), що забезпечує властивість прямої секретності (Perfect Forward Secrecy). Це гарантує, що навіть у разі компрометації довгострокового закритого ключа сервера в майбутньому, зловмисник не зможе розшифрувати сесії, записані раніше [13]. Для повноцінного функціонування інфраструктури HTTPS критичне значення має застосування цифрових сертифікатів (Certificate Authorities), які підтверджують відповідність публічного ключа конкретному доменному імені сервера [2]. У високонадійних архітектурах API стандартного

одностороннього TLS-з’єднання, де автентифікація лише сервер, часто буває недостатньо. У таких сценаріях впроваджують механізм взаємної автентифікації mTLS. Він вимагає від клієнта надати свій цифровий сертифікат для перевірки сервером. Застосування mTLS дозволяє реалізувати суворий контроль доступу на транспортному рівні, суттєво знижуючи ризики несанкціонованого підключення до компонентів API та виключаючи можливість проведення атак типу “людина посередині” (Man-in-the-Middle) [35]. Окремим аспектом безпеки є механізм прикріплення сертифікатів (Certificate Pinning), який зазвичай реалізується на рівні мобільних або десктопних клієнтських додатків, що взаємодіють з API. Ця технологія передбачає жорстке кодування відбитка легітимного сертифіката або публічного ключа безпосередньо в коді додатка, завдяки чому клієнт ігнорує системне сховище довірених сертифікатів операційної системи. Такий підхід нівелює загрози, пов’язані з компрометацією локальних центрів сертифікації або навмисним встановленням шкідливих сертифікатів на пристрої користувача з метою перехоплення HTTPS-трафіку. Водночас адміністрування HTTPS вимагає постійного моніторингу конфігурацій, оскільки використання слабких шифронаборів або застарілих версій протоколу, таких як TLS 1.0 чи TLS 1.1, створює вектори для реалізації відомих криптографічних атак [13]. Незважаючи на високу ефективність протоколу TLS, захист інформаційних потоків у середовищі API не повинен обмежуватися лише транспортним рівнем, оскільки дані можуть проходити через низку проміжних вузлів, таких як балансувальники навантаження, зворотні проксі-сервери, шлюзи API та мережі доставки контенту (CDN). На цих проміжних етапах TLS-з’єднання часто термінується (de-encapsulation), внаслідок чого інформація тимчасово перебуває у відкритому текстовому вигляді в оперативній пам’яті транзитних пристроїв. Для нейтралізації пов’язаних із цим ризиків внутрішнього перехоплення або витоку даних застосовують прикладне шифрування безпосередньо на рівні повідомлень (Message-Level Encryption), що забезпечує захист даних незалежно від безпеки каналів зв’язку [13]. У контексті веб-орієнтованих архітектур, що

використовують формати обміну даними JSON (JavaScript Object Notation), стандартом де-факто для прикладного захисту є специфікації проєкту JOSE (JSON Object Signing and Encryption), розроблені інженерною радою Інтернету (IETF). Специфікація JSON Web Encryption (JWE) описує стандартизовану структуру для представлення зашифрованого вмісту, що дозволяє шифрувати як усе корисне навантаження (payload) API-запиту, так і окремі чутливі поля об'єкта. Процес формування JWE передбачає генерацію випадкового симетричного ключа шифрування контенту (Content Encryption Key, CEK), яким безпосередньо зашифровуються дані за допомогою стійких алгоритмів, наприклад, AES у режимі Galois/Counter Mode (AES-GCM), після чого сам CEK зашифровується публічним ключем одержувача з використанням асиметричних алгоритмів типу RSA або криптографії на еліптичних кривих (ECC) [35]. Для архітектур, що базуються на протоколі SOAP та форматі XML, аналогічні завдання вирішуються за допомогою стандартів W3C, зокрема XML Encryption та XML Signature. Ці специфікації забезпечують високий рівень гранулярності, дозволяючи зашифровувати лише певні елементи XML-документа, залишаючи загальну структуру придатною для маршрутизації проміжними серверами без необхідності повного доступу до вмісту. Проте впровадження прикладного шифрування суттєво підвищує складність розробки та супроводу програмного забезпечення, оскільки вимагає від обох сторін інформаційного обміну інтеграції відповідних криптографічних бібліотек та узгодження схем управління ключами [2]. Шифрування під час передачі також тісно пов'язане з безпекою токенів автентифікації, які передаються в HTTP-заголовках, таких як Authorization Bearer. Використання форматів JSON Web Token вимагає не лише підписування токенів JWS, а й використання JWE, якщо токен містить конфіденційні ідентифікатори користувача. Крім того, під час проектування RESTful API необхідно враховувати, що параметри, передані в URL-рядку запиту (Query Parameters), не захищаються шифруванням прикладного рівня і часто логуються веб-серверами у відкритому вигляді, що обумовлює необхідність передачі всіх критичних даних виключно в зашифрованому тілі

запиту (Request Body) [13]. Захист конфіденційної інформації в програмних інтерфейсах охоплює комплекс технічних та організаційних заходів, спрямованих на запобігання несанкціонованому доступу, модифікації або витоку персональних даних користувачів, фінансових транзакцій, комерційної таємниці та автентифікаційних маркерів. На відміну від захисту даних у русі, безпека конфіденційної інформації вимагає забезпечення її захищеності також у стані спокою (Data at Rest) на серверах баз даних та у сховища конфігурацій. Це досягається шляхом комбінування криптографічних методів, суворого розмежування прав доступу на основі ABAC або RBAC, а також застосування систем автоматизованого управління життєвим циклом криптографічних ключів [35]. Одним із ключових аспектів є правильний вибір криптографічний примітивів для різних категорій конфіденційних даних. Для інформації, яка не потребує зворотнього дешифрування, наприклад, паролів користувачів, застосовують механізми незворотного криптографічного хешування з обов’язковим додаванням унікальної криптографічної “солі” для запобігання атакам за словниками та райдужними таблицями. При цьому викоритсання застарілих алгоритмів типу MD5 або SHA-1 є неприпустимим. Сучасні стандарти вимагають застосування адаптивних функцій розтягування ключів, таких як Argon2, bcrypt або PBKDF2, які штучно збільшують обчислювальні витрати, нівелюючи переваги використання спеціалізованого апаратного забезпечення зловмисниками при спробах підбору [34]. Для обробки та зберігання конфіденційних даних, які потребують подальшого відновлення первинного вигляду, застосовують симетричне шифрування на основі стандарту AES з довжиною ключа 256 біт. Безпека такої схеми критично залежить від ізоляції ключів шифрування від самих зашифрованих даних. Компрометація сервера додатків API не повинна призводити до компрометації даних у сховищі, що досягається за рахунок використання апаратних модулів безпеки (Hardware Security Modules) або хмарних сервісів KMS, які виконують криптографічні операції у захищеному вигляді. Крім того, впроваджуються механізми регулярної ротації ключів для обмеження обсягу даних, зашифрованих одним і

тим самим ключем [34]. Додатковим інструментом захисту конфіденційної інформації виступає технологія токенізації та маскування даних (Data Masking). Токенізація передбачає заміну чутливого елемента даних (наприклад, номера банківської картки) випадково згенерованим еквівалентом (токеном), який не має математичного зв'язку з оригіналом і є марним для зловмисника поза контекстом конкретної платіжної системи або внутрішньої захищеної бази відповідностей. Маскування даних, у свою чергу, застосовується під час виведення інформації через API для некритичних процесів (наприклад, відображення лише останніх чотирьох цифр рахунку в інтерфейсі), що мінімізує ризики випадкового витоку інформації через лог-файли, аналітичні системи або помилки розробників [13]. Впровадження криптографічних методів захисту в інфраструктуру програмних інтерфейсів забезпечує низку фундаментальних переваг, які дозволяють будувати надійні, стійкі до компрометації розподілені системи. Перш за все, криптографія гарантує тріаду інформаційної безпеки: конфіденційність, цілісність та автентичність. Завдяки шифруванню перехоплені зловмисником дані перетворюються на шифротекст, який неможливо аналізувати без знання відповідного ключа. Механізми цифрового підпису та кодів автентифікації повідомлень (MAC) дозволяють однозначно ідентифікувати відправника запиту API та гарантувати, що вміст повідомлення не був змінений або підроблений під час транспортування мережею, що є критично важливим для фінансових та юридично значущих транзакцій [34]. Іншою важливою перевагою є забезпечення відповідності нормативним вимогам та міжнародним стандартам безпеки, важливим для збереження персональних та платіжних даних. Застосування стандартизованих криптографічних протоколів, таких як TLS та компоненти JOSE, спрощує інтеграцію між системами різних розробників, оскільки забезпечує сумісність на рівні алгоритмів та форматів даних. Крім того, концепція наскрізного шифрування (End-to-End Encryption) дозволяє організаціям використовувати публічні хмарні платформи для розгортання API, мінімізуючи ризики доступу провайдера хмари до конфіденційної інформації клієнтів [13]. Незважаючи на

очевидні переваги, криптографічний захист має суттєві архітектурні та експлуатаційні обмеження, які необхідно враховувати під час проектування програмного інтерфейсу. Основним обмеження є висока обчислювальна складність криптографічних операцій. Процедури асиметричного шифрування, генерації цифрових підписів та встановлення TLS-з'єднань створюють значне навантаження на центральний процесор серверів, що може призвести до збільшення часу затримки (latency) обробки API-запитів та зниження загальної пропускної здатності системи. Для нівелювання цього фактору доводиться інвестувати в додаткові апаратні ресурси, такі як спеціалізовані плати прискорення або балансувальники з функцією TLS Offloading [34]. Другим критичним обмеженням є проблема управління ключами (Key Management Lifecycle), яка вважається однією з найскладніших інженерних задач у сфері безпеки. Надійність будь-якої криптосистеми базується на припущенні про абсолютну таємність ключів: якщо зловмисник отримує доступ до сховища ключів внаслідок помилки конфігурації, вразливості в коді або соціальної інженерії, вся система шифрування втрачає сенс [35]. Наостанок, криптографія не захищає від логічних помилок в архітектурі API, таких як вразливості Broken Object Level Authorization (BOLA) або ін'єкції шкідливого коду, де зловмисник може надсилати повністю легітимні з погляду шифрування запити, але отримувати несанкціонований доступ до чужих даних через недоліки в логіці авторизації додатка. Таким чином, криптографічний захист є обов'язковим, але не єдиним компонентом комплексного забезпечення безпеки API [13].

В сучасній архітектурі корпоративних інформаційних систем механізм обмеження частоти запитів, в науковій практиці більш відомий як rate limiting, становить один із фундаментальних інструментів забезпечення відмовостійкості, безпеки та раціонального розподілу обчислювальних ресурсів [13]. За своєю сутністю обмеження запитів визначається як системна політика штучного стримування та регулювання кількості транзакцій або мережевих звернень, які конкретний споживач, автентифікований користувач, клієнтський додаток чи ідентифікована за певними ознаками адреса інтернет-протоколу (IP)

має право здійснювати до програмного інтерфейсу вебсервера протягом чітко встановленого дискретного часового інтервалу. Дане обмеження примусово впроваджується на рівні постачальника послуг за допомогою спеціалізованого програмного забезпечення, вебсерверів, міжмережевих екранів (firewalls) або шлюзів API, які виконують роль координатора та фільтра вхідного трафіку [14]. Функціонування зазначеного механізму базується на безперервному моніторингу вхідних потоків даних та порівнянні поточної інтенсивності звернень із заздалегідь сконфігурованими пороговими значеннями, перевищення яких автоматично активує захисні інструменти платформи [14]. У разі виявлення аномальної активності або перевищення ліміту, інформаційна система відхиляє подальшу обробку запитів, повертаючи клієнту відповідний статус помилки, найчастіше HTTP-код 429 (Too Many Requests), що супроводжується специфічними заголовками метаданих, які вказують на час відновлення доступу [36]. Складність реалізації сутності обмеження запитів у сучасних архітектурах, зокрема RESTful API, полягає в необхідності ідентифікації споживача без порушення принципу stateless, що вимагає залучення унікальних маркерів автентифікації або криптографічних токенів, які передаються з кожною транзакцією для верифікації прав доступу та ведення обліку утилізації системного ліміту [13]. Таким чином, rate limiting виступає як критично важливий архітектурний бар’єр, що трансформує хаотичний і потенційно деструктивний зовнішній потік запитів у прогнозований, керований та безпечний внутрішній процес обробки даних. Практичне функціонування відкритих мережевих інтерфейсів без систем контролю інтенсивності трафіку неминуче призводить до виникнення критичних вразливостей, пов’язаних із недостатністю обчислювальних потужностей та деградацією сервісів під впливом надмірних навантажень [13]. За відсутності жорстких лімітів на кількість запитів інфраструктурний комплекс постачальника послуг піддається ризику миттєвого вичерпання ресурсів центрального процесора, оперативної пам’яті, дискової підсистеми або пропускної здатності каналів зв’язку, що призводить до аварійного завершення процесів та переходу системи в стан

повної або часткової недоступності, відомий як стан відмови в обслуговуванні (Denial of Service) [14]. Особливу небезпеку для сучасних веб-сервісів становлять розподілені атаки на відмову в обслуговуванні (DDoS), спрямовані на прикладний рівень OSI (Layer 7), де зловмисники імітують легітимну поведінку користувачів шляхом генерації мільйонів високозатратних транзакцій з метою виведення з ладу серверів баз даних або бізнес-логіки додатка [13]. Впровадження алгоритмів обмеження запитів дозволяє нейтралізувати руйнівний потенціал таких векторів атак на ранніх етапах взаємодії, нівелюючи можливість зловмисного чи випадкового створення лавиноподібного навантаження. Окрім прямого захисту від деструктивних кібератак, механізми rate limiting виконують функцію оптимізації операційних витрат в умовах сучасних хмарних інфраструктур, які застосовують технології автоматичного вертикального чи горизонтального масштабування (auto-scaling). У разі відсутності обмежень неконтрольований потік запитів змушує хмарну платформу безперервно розгортати нові обчислювальні вузли для покриття штучно створеного попиту, що тягне за собою експоненціальне і фінансово обтяжливе зростання витрат на утримання інфраструктури, яке можна запобігти за допомогою превентивної фільтрації [14]. У сучасній практиці проектування API розроблено кілька фундаментальних алгоритмічних підходів до реалізації rate limiting, кожен з яких має унікальні характеристики точності та споживання пам’яті [13]. Поширеним рішенням є алгоритм маркерного кошика (Token Bucket), де абстрактне сховище фіксованої місткості безперервно заповнюється умовними маркерами. Кожна транзакція вилучає токен із кошика, а за його відсутності запит відхиляється. Головною перевагою підходу є здатність гнучко обробляти короткочасні сплески трафіку за рахунок накопичених маркерів [14]. Натомість алгоритм дірявого кошика (Leaky Bucket) фокусується на підтримці суворо стабільної швидкості вихідного потоку. Запити акумулюються в черзі, звідки надходять на обробку з фіксованим темпом. Якщо швидкість вхідного трафіку перевищує пропускну здатність, черга переповнюється, а нові звернення блокуються, що забезпечує жорсткий контроль навантаження [14].

простішим у реалізації, але менш точним є алгоритм фіксованого вікна (Fixed Window Counter), який закріплює лічильник транзакцій за статичними часовими відрізками. Його недоліком є вразливість до ефекту накладання на межах вікон (boundary burst), коли споживач може здійснити подвійну норму запитів на стику двох періодів [14]. Для усунення цієї проблеми застосовують підхід ковзного вікна (Sliding Window Log), який динамічно обчислює кількість транзакцій за останні N секунд відносно поточного моменту. Завдяки безперервному зміщенню часових меж досягається абсолютна точність обмеження, хоча це й вимагає більших обчислювальних ресурсів для збереження стану лічильників [13]. Впровадження політик обмеження запитів забезпечує комерційну та операційну стабільність програмних інтерфейсів, впливаючи на архітектуру монетизації та якість послуг [14]. У комерційному контексті обмеження запитів виступає інструментом диференціації тарифних планів, дозволяючи обмежувати безкоштовні рівні підписки та надавати вищі ліміти корпоративним клієнтам [13]. Це гарантує справедливий розподіл апаратних ресурсів дата-центру між усіма споживачами. З інженерної точки зору інтеграція обмежень потребує проектування інтерфейсів зворотного зв’язку за допомогою стандартних HTTP-заголовків [36]. API-сервіс інформує клієнтів про стан їхніх лімітів через службові метадані, такі як x-rate-limit, x-rate-limit-remaining або заголовок Retry-After [14]. Це дозволяє розробникам клієнтського ПЗ впроваджувати алгоритми експоненціального відкладання запитів (exponential backoff) і запобігати збоям. Таким чином, rate limiting є комплексним архітектурним та бізнес-рішенням, що гарантує високу доступність розподілених хмарних платформ.

На ряду з обмеженням запитів API важливим елементом захисту архітектури є механізм валідації даних, який запобігає компрометації внутрішньої логіки серверних застосунків та мінімізує ризики виникнення вразливостей. Процес перевірки вхідних даних становить найбільш критичний рубіж оборони будь-якого веб-орієнтованого API. За своєю сутністю, валідація даних інформаційних потоків є систематичним процесом зіставлення отриманих від

клієнта структур даних із заздалегідь визначеними формальними специфікаціями, схемами або бізнес-правилами. На етапі обробки запиту сервером підсистема валідації здійснює верифікацію типу даних, довжини рядків, діапазону числових значень, а також обов'язковості наявності конкретних полів у структурі переданого об'єкта, найчастіше у форматах JSON або XML. Необхідність суворої перевірки зумовлена парадигмою проектування, згідно з якою будь-які зовнішні дані за замовчуванням вважаються ненадійними та потенційно деструктивними. Реалізація семантичного та синтаксичного аналізу на початковому етапі життєвого циклу запиту дозволяє відкидати некоректно сформовані пакети даних ще до того, як вони будуть передані на рівень бізнес-логіки або збережені у сховищі даних. Таким чином, дескриптивна валідація не лише гарантує стабільність функціонування програмних модулів, але й виступає превентивним засобом проти логічних збоїв, викликаних невідповідністю типів або переповненням буфера [19]. Ефективна перевірка вхідних параметрів безпосередньо корелює з рівнем захищеності інформаційної системи від найбільш поширених векторів кібератак, зокрема впровадження стороннього коду. Однією з найбільш деструктивних загроз для реляційних та нереляційних баз даних залишаються SQL ін'єкції (SQL Injection), які виникають внаслідок динамічного формування запитів до СУБД шляхом з'єднання неперевірених користувацьких рядків. Для нейтралізації цієї вразливості на рівні API застосовується архітектурний підхід, заснований на використанні параметризованих запитів (Prepared Statements) та об'єктно-реляційного відображення запитів (ORM), що відокремлює код інструкцій SQL від безпосередніх даних [36]. Паралельно з цим, програмні інтерфейси, які обслуговують веб-клієнтів, транслюють дані, що згодом можуть відображатися в браузері кінцевого користувача, створюючи умови для реалізації атак міжсайтового скриптингу (Cross-Site Scripting, XSS). Відповідно до методологічних рекомендацій OWASP, у контексті безсерверних та мікросервісних архітектур API захист від XSS-атак покладається на детальну перевірку вмісту, який приймається від клієнта, та суворе обмеження виконання

стороннього коду [19]. Якщо зловмисник спробує передати через програмний інтерфейс шкідливий сценарій, належно налаштований компонент сервера повинен ідентифікувати та заблокувати таку спробу. В NIST зазначається, що безпечна обробка відповідей сервера та екранування вихідних потоків унеможливлюють постійне збереження скрипту в базі даних або його подальшу регенерацію у HTTP-відповідях іншим клієнтам [36]. Окрім класичної валідації, яка функціонує за принципом бінарного рішення щодо допуску або відхилення запиту, вагоме значення для забезпечення стійкості API мають процеси фільтрації та санітизації (очищення) даних. Фільтрація є процесом вилучення небажаних або надлишкових елементів із вхідного потоку, що дозволяє, наприклад, ігнорувати невідомі поля в JSON- об’єктах, захищаючи сервер від атак типу масового призначення властивостей (Mass Assignment). Санітизація, в свою чергу, спрямована на трансформацію потенційно небезпечних даних у безпечну форму без зміни їх первинного інформаційного змісту [37]. Типовим прикладом санітизації є екранування спеціальних символів, конвертація HTML-сутностей або видалення керуючих послідовностей із текстових рядків перед їх збереженням чи виведенням [36]. Застосування санітизації дозволяє нівелювати загрозу інтерпретації даних як виконуваного коду в тих випадках, коли сувора валідація є неможливою через необхідність прийняття специфічних символів від користувача. Таким чином, синергетичне поєднання суворої превентивної валідації, контекстно-залежної санітизації та глибокої фільтрації на рівні сервера дозволяє сформувати надійний та відмовостійкий контур безпеки програмного інтерфейсу, здатний ефективно протидіяти актуальним загрозам у динамічному кіберпросторі.

Детальний порівняльний аналіз технічних методів захисту API дозволяє виявити низку фундаментальних компромісів між рівнем безпеки, архітектурною складністю та загальною продуктивністю інформаційної системи. Rate limiting характеризується високою ефективністю захисту від перевантаження інфраструктури, проте їхнє впровадження супроводжується ризиком виникнення хибнопозитивних спрацьовувань (False Positives). Це може

призвести до випадкового блокування легітимних корпоративних споживачів або інтегрованих партнерських систем, що виконують інтенсивну пакетну передачу даних, генеруючи помилки типу 429 Too Many Requests і знижуючи загальний рівень доступності сервісу [13]. Що стосується засобів глибокого аналізу вмісту запитів, то їхньою перевагою є здатність виявляти складні сигнатури атак безпосередньо в тілі повідомлень. Однак критичний недолік полягає в суттєвому уповільненні обробки транзакцій через необхідність розбору великих масивів даних. Навіть використання надійного криптографічного захисту транспортного шару створює додаткове обчислювальне навантаження на процесорні потужності під час процедури TLS handshake та безпосереднього кодування потокових даних, що в архітектурах із високими вимогами до мінімальної затримки (low latency) може стати стримуючим фактором продуктивності [12]. Крім того, налаштування статичних правил фільтрації трафіку часто виявляється неефективним проти цілеспрямованих атак, які імітують поведінку звичайних користувачів, що змушує розробників шукати гнучкіші динамічні рішення. Зважаючи на те, що жоден з вище перерахованих методів захисту не здатен забезпечити абсолютну безпеку програмних інтерфейсів самостійно, сучасна парадигма проектування інформаційних систем базується на принципі ешелонованої оборони (Defense in Depth). Комплексний підхід передбачає створення багаторівневої системи захисту, де кожен наступний рубіж компенсує функціональні обмеження попереднього, а безпека інтегрується безпосередньо на етапі проектування архітектури, а не як тимчасове налаштування під час розгортання. Центральним компонентом такої системи є спеціалізований шлюз API (API Gateways). Він виступає в ролі єдиної точки входу для всього вхідного трафіку та функціонує як точка забезпечення політики безпеки (Policy Enforcement Point). На рівні шлюзу реалізується первинна фільтрація, оркестрація запитів, а також централізована перевірка лімітуючих правил [12]. У межах комплексної архітектури шлюз API тісно взаємодіє із системами превентивного обмеження інтенсивності потоків даних для стримування автоматизованих атак та

Dos-хвиль. Для підвищення стійкості до специфічних прикладних загроз комплексний захист розширюється за допомогою брандмауерів веб-додатків (WAF), які здійснюють глибокий аналіз вмісту пакетів, дозволяючи виявити складні аномалії бізнес-логіки та блокувати спроби обходу встановлених обмежень швидкості [12, 13]. Додатково інфраструктура підсилюється наскрізним логуванням, аудитом та збором метрик на рівні шлюзу, що дозволяє передавати статистичні дані в системи моніторингу подій безпеки для ретроспективного аналізу інцидентів та оперативного виявлення цілеспрямованої шкідливої активності. Таким чином, синергетична інтеграція засобів криптографічного захисту каналів, динамічного керування трафіком та інтелектуального аналізу вмісту на периметрі мережі дозволяє побудувати надійний контур безпеки [12]. Отже, проведений аналіз свідчить, що забезпечення стійкості API є комплексним завданням, яке виключає ефективність універсальних однокомпонентних засобів. Дієвість таких механізмів, як обмеження частоти запитів або валідація даних, безпосередньо залежить від специфіки бізнес-логіки додатка, що змушує балансувати між рівнем безпеки та продуктивністю інфраструктури. Науково обґрунтованим вектором розвитку є перехід до ешелонованої оборони на базі шлюзу API, який консолідує функції лімітування, фільтрації та аудиту трафіку [12]. Перспективи подальших досліджень полягають в інтеграції адаптивних методів захисту на основі аналізу поведінки користувачів для мінімізації хибних спрацювань та автоматизації протидії загрозам у режимі реального часу.

2.3 Архітектурні підходи до забезпечення безпеки API в мікросервісних та хмарних середовищах

Трансформація сучасних інформаційних систем та перехід від монолітних архітектур до мікросервісних і хмарних рішень зумовили докорінну зміну парадигми забезпечення інформаційної безпеки. Особливості мікросервісних і хмарних архітектур полягають у високому рівні децентралізації, динамічному масштабуванні компонентів, використанні контейнеризації та підходу, коли окремі сервіси розробляються з використанням різних стеків технологій.

Подібна децентралізація призводить до суттєвого розширення поверхні потенційних атак, оскільки кожен ізольований інтерфейс фактично стає самостійною точкою доступу, що потребує індивідуального захисту. Окрім того, динамічна природа хмарних обчислень, де контейнери постійно створюються та знищуються, нівелює ефективність класичного захисту на рівні статичного мережевого периметра та ускладнює процеси автентифікації, верифікації довіри між компонентами і централізованого керування політиками доступу [38]. Незважаючи на технологічні переваги розподілених систем, функціонування великої кількості взаємопов'язаних інтерфейсів породжує серйозні ризики безпеки в розподілених середовищах. До найбільш критичних загроз, що систематизовані в межах галузевих стандартів, належать порушення авторизації на рівні об'єктів та функцій, що дозволяє зловмисникам отримувати несанкціонований доступ до конфіденційних даних через маніпуляції ідентифікаторами в API-запитах. Поширена практика використання автентифікаційних токенів на базі JWT для передачі контексту користувача між сервісами створює додаткові ризики, пов'язані з можливим компрометуванням ключів підписання або некоректною валідацією токенів на стороні мікросервісів, що вимагає впровадження криптографічних протоколіи взаємної перевірки автентичності, таких як взаємний захист транспортного рівня (mTLS). Відсутність належного обмеження обсягу ресурсів і швидкості обробки запитів (Rate Limiting) робить інфраструктуру вразливою до DDoS атак, а ускладнена трасованість запитів суттєво знижує ефективність оперативного виявлення інцидентів кібербезпеки через дефіцит наскрізного логування та моніторингу. Таким чином, комплексне забезпечення безпеки програмних інтерфейсів потребує інтеграції архітектурних підходів, що поєднують строгу фільтрацію із децентралізованими механізмами захисту внутрішнього хмарного простору [38].

Перехід від монолітних програмних комплексів до мікросервісної архітектури докорінно змінює топологію розподілу даних та інформаційну взаємодію між компонентами системи. У традиційних монолітних архітектурах комунікація

між бізнес-логікою відбувається всередині єдиного адресного простору пам’яті, що мінімізує мережеві ризики, тоді як у мікросервісному середовищі кожен автономний компонент розглядається як окремий мережевий вузол [38, 39]. Будь-яка операція користувача трансформується у складну серію послідовних або паралельних запитів між сервісами, які проходять через відкриті або внутрішні мережеві канали. У зв’язку з цим прийнято виділяти два типи мережевого трафіку, кожен з яких потребує специфічних засобів захисту. Трафік напрямку “північ-південь” (North-South) представляє собою вхідні запити від зовнішніх клієнтів (веб-додатків, мобільних пристроїв або сторонніх інтеграцій) до внутрішнього контуру системи, тоді як трафік напрямку “схід-захід” (East-West) охоплює виключно взаємодію між самими мікросервісами всередині захищеного периметру [39]. Основним інструментом структурування та захисту вхідного трафіку на межі системи є шлюз API (API Gateway). Шлюз виконує роль єдиної точки входу, ізолюючи внутрішню інфраструктуру від безпосереднього зовнішнього впливу та дедуплікуючи рутинні безпекові операції. Замість реалізації механізмів автентифікації, перевірки сертифікатів та обмеження частоти запитів (throttling) на рівні кожного окремого сервісу, ці функції делегуються шлюзу, що дозволяє розробникам зосередитися виключно на реалізації бізнес-функціоналу [38, 39]. Шлюз API перехоплює зовнішній запит, здійснює маршрутизацію до відповідного внутрішнього мікросервісу, а також може трансформувати протоколи передачі даних. Окрім того, шлюз забезпечує централізований моніторинг та аудит вхідних потоків, запобігаючи несанкціонованим спробам сканування внутрішньої мережі архітектури [39]. Для реалізації взаємодії “схід-захід” найчастіше використовуються протоколи прикладного рівня, зокрема REST API на базі HTTP/HTTPS та високопродуктивний фреймворк gRPC, який базується на HTTP/2 [38, 39]. Проектування таких інтерфейсів потребує суворого дотримання підходу, керованого документацією (Documentation-driven development), де спочатку створюється та валідується точна специфікація API, наприклад, за допомогою стандартів OpenAPI (Swagger), і лише після цього здійснюється розробка

клієнтської та серверної частин. Такий підхід мінімізує інтеграційні збої та дозволяє автоматизовано перевіряти структуру повідомлень, що циркулюють між сервісами, на відповідність схемам даних Валідація схем є критично важливою для безпеки, оскільки вона запобігає атакам типу ін’єкцій та переповнення буфера безпосередньо під час сервісного обміну [38]. Проте, незважаючи на структурну коректність, передача даних через мережеві інтерфейси в розподіленій системі створює ризики перехоплення, модифікації або підміни повідомлень, що вимагає впровадження комплексних підходів до ідентифікації та авторизації кожного учасника мережевого обміну [39]. Розподілений характер мікросервісів унеможливлює використання традиційних сесій, притаманних монолітним архітектурам, де контекст користувача зберігається в спільній пам’яті. Коли запит долає межу шлюзу API та починає мігрувати між внутрішніми компонентами, виникає потреба в підтвердженні як ідентичності самого мікросервісу-відправника (автентифікація сервісу), так і повноважень кінцевого користувача, від імені якого виконується операція. Для вирішення завдання міжсервісної автентифікації на транспортному рівні стандартом де-факто є застосування обопільного захисту транспортного шару mTLS. На відміну від стандартного HTTPS, де лише сервер підтверджує свою автентичність за допомогою сертифіката, mTLS вимагає від клієнтського мікросервісу також надати свій цифровий сертифікат для перевірки сервером. Це дозволяє криптографічно гарантувати, що запити надходять виключно від легітимних компонентів системи, які отримали сертифікати від довіреного центру сертифікації (CA). Життєвий цикл таких сертифікатів зазвичай є дуже коротким, а процедури їхньої ротації автоматизуються, щоб мінімізувати ризики у випадку компрометації приватних ключів [39]. Автентифікація на транспортному рівні через mTLS вирішує проблему автентифікації вузлів, проте вона не надає інформації про те, який саме користувач ініціював транзакцію, та які права він має в межах конкретної бізнес-операції. Для безпечної передачі та верифікації контексту користувача між мікросервісами застосовуються децентралізовані криптографічні токени, серед яких найбільшого поширення

набув стандарт JWT. Ці токени генеруються сервером авторизації після успішної автентифікації користувача на межі системи за протоколами OAuth 2.1 або OpenID Connect [38, 39]. Автономність токенів дозволяє мікросервісам самостійно верифікувати їхню валідність та цілісність підпису за допомогою публічного ключа, уникаючи синхронних запитів до центральної бази даних. Обов'язкова валідація кожного запиту включає перевірку терміну дії JWT та контроль цільової аудиторії, що унеможливлює використання токена в непередбачених інтерфейсах [39]. Розмежування доступу реалізується на макрорівні бізнес-логіки шляхом аналізу контексту користувача (User Context) для запобігання несанкціонованій модифікації даних. Важливим аспектом є безпечна трансляція токенів між доменами довіри (Trust Boundaries) через механізми їхнього поширення (token propagation), що мінімізує надлишкове розкриття повноважень користувача [38]. Надійна безпека API потребує суворої архітектурної та інфраструктурної ізоляції компонентів, яка запобігає каскадному зламу системи та горизонтальному переміщенню зловмисника у разі компрометації одного з мікросервісів. Ізоляція будується за багаторівневим принципом і передбачає відокремлення баз даних для кожного сервісу з доступом до них виключно через захищені програмні інтерфейси [38]. Мережевий захист забезпечується сегментацією підмереж у Virtual Private Cloud (VPC) та налаштуванням міжмережевих екранів, що обмежують проходження трафіку лише з довірених IP-адрес. На рівні контейнеризації (Docker, Kubernetes) діє принцип незмінності образів, який забороняє зберігання секретів у файловій системі контейнера та вимагає їхнього динамічного інжектування під час завантаження [38, 39]. Найбільш гнучким інструментом ізоляції є сервісні сітки, де інфраструктурні проксі-сервери керують mTLS-з'єднаннями, верифікацією токенів та трансляцією контексту без модифікації вихідного коду бізнес-логіки [38]. Через високу розподіленість та динамічність мікросервісів традиційна концепція захищеного параметра втрачає ефективність, поступаючись парадигмі нульової довіри (Zero Trust). У межах цієї моделі кожен мікросервіс розглядає будь-який вхідний запит як

потенційно ворожий [38]. Реалізація Zero Trust спирається на чотири засади: безперервну автентифікацію та авторизацію кожного суб'єкта через повторну перевірку JWT та mTLS на рівні кінцевого сервісу. Суворе дотримання принципу найменших привілеїв (Least Privilege) з деталізованою конфігурацією ролей та scopes; тотальне шифрування даних під час транспортування через HTTPS/mTLS; а також постійний аудит і динамічний аналіз логів безпеки централізованими системами моніторингу для своєчасного виявлення аномальної поведінки [38, 39].

У хмарних екосистемах класичне розподілення мереж на зовнішні і внутрішні втрачає ефективність, оскільки загрози можуть виходити з будь-якого сегмента інфраструктури. Окрему небезпеку становлять специфічні вразливості, пов'язані з порушенням механізмів авторизації на рівні об'єктів (BOLA) або функцій (BFLA), які виникають через архітектурні прорахунки та призводять до витоку конфіденційних даних. Застосування шлюзів API в поєднанні з методологією зміщення безпеки ліворуч (Shift-Left Security), дозволяє інтегрувати захисні інструменти на ранніх етапах розробки, мінімізуючи ймовірність потрапляння вразливостей у хмарне середовище [15]. Для мінімізації ризиків витоку інформації через клієнтські застосунки застосовується патерн фантомних токенів (phantom token pattern). Зовнішні клієнти оперують виключно непрозорими токенами, які не містять системних атрибутів користувача [21]. Хмарний шлюз API виконує процедуру інтроспекції цього токена та обмінює його на безпечний, підписаний токен у форматі JWT для передачі внутрішнім мікросервісам [15, 21]. На рівні контейнеризованої інфраструктури для ідентифікації обчислювальних навантажень впроваджуються технології робочих ідентифікаторів (JWT Workload Identities), що дозволяє сервісам динамічно отримувати токени без жорсткого кодування статичних паролів [21]. Захист каналів зв'язку та обмеження прав доступу є фундаментом безпеки хмарних інтерфейсів. На транспортному рівні обов'язковим є використання двостороннього протоколу mTLS, який забезпечує криптографічне шифрування даних у процесі передачі та взаємну

автентифікацію між клієнтом і сервером за допомогою цифрових сертифікатів. Для підвищення стійкості токенів до перехоплення застосовуються механізми обмеження відправника (sender-constrained tokens), які прив’язують токен до конкретного захищеного mTLS-каналу, унеможливлюючи його повторне використання сторонніми особами [15, 21]. Авторизація на рівні прикладного коду хмарних API реалізується через перевірку тверджень та областей доступу, що містяться всередині валідованих JWT-токенів [21]. Перевірка підпису JWT за допомогою сильних алгоритмів є критично важливою, оскільки її ігнорування створює передумови для обходу автентифікації. Контроль доступу кінцевих користувачів базується на двох моделях: RBAC, який оперує статично визначеними ролями, та ABAC, який приймає рішення про дозвіл на базі динамічного аналізу контекстних характеристик суб’єкта та середовища в реальному часі [15]. Забезпечення безперервної видимості функціонування хмарної інфраструктури покладається на інструменти спостережуваності API (API Observability). Належна організація спостережуваності є необхідною для своєчасного реагування на збої, недопущення тривалих періодів недоступності сервісів та запобігання втраті даних. Тріада спостережуваності включає збір та аналіз структурованих журналів подій (logs), розподілених трасувань (traces) та метрик продуктивності систем (metrics). Фіксація кастомних подій безпеки дозволяє виявляти аномалії в поведінці користувачів і спроби експлуатації логічних помилок [15]. Основним інструментом протидії атакам на мережевому та прикладному рівнях є міжмережеві екрани веб-застосунків, які аналізують вхідний трафік на наявність сигнатур відомих нападів. Оскільки зловмисники здатні застосовувати методи маскування шкідливого коду для обходу цих екранів, системи моніторингу мають додатково аналізувати атаки, що базуються на вхідних даних (input-based attacks). Також підлягають детекції загрози, пов’язані зі зловживанням аномальною частотою запитів і нейтралізуються впровадженням політик обмежень швидкості [15].

На архітектурному рівні організація безпеки базується на двох парадигмах, а саме на централізованому та децентралізованому підходах. Розподіл між ними

безпосередньо залежить від характеру мережевих потоків, які поділяються на зовнішній трафік (від клієнта до системи) та внутрішній трафік (між сервісами всередині системи) [4]. Централізований підхід орієнтований на створення єдиного захищеного периметра на межі мережі, де всі функції автентифікації, авторизації та фільтрації покладаються на API Gateway. Шлюз виступає в ролі зворотного проксі-сервера, який приймає всі зовнішні запити, перевіряє ключі арі або токени доступу та обмежує інтенсивність потоків даних для захисту внутрішньої інфраструктури [1]. У такій моделі безпека внутрішніх компонентів часто делегується шлюзу, який транслює перевірені ідентифікатори користувачів далі по ланцюжку виконання, мінімізуючи необхідність повторної перевірки на рівні окремих бізнес-доменів [4]. Децентралізований підхід передбачає розподіл безпекових політик між компонентами системи та застосовується переважно для захисту внутрішніх комунікацій. Головним інструментом реалізації цієї моделі є Service Mesh, яка використовує локальні проксі-сервери (патерн sidecar), розгорнуті разом із кожним мікросервісом. Децентралізація дозволяє відмовитися від концепції довіреного внутрішнього периметра, оскільки кожен мікросервіс самостійно автентифікує викликаючу сторону за допомогою взаємного шифрування транспортного рівня, забезпечуючи ізоляцію та захист від внутрішніх загроз [4]. Централізована архітектура на базі шлюзу API спрощує розробку клієнтських додатків та ефективно керує зовнішнім доступом. Централізована архітектура на базі шлюзу арі спрощує розробку клієнтських додатків та ефективно керує зовнішнім доступом. Використання шлюзу, як єдиної точки входу дозволяє приховати внутрішню топологію мережі, забезпечити наскрізний моніторинг і трансформацію протоколів [4]. Винесення перевірки сертифікатів та інтеграції з OAuth2 на рівень шлюзу звільняє мікросервіси від рутинних завдань безпеки [1]. Крім того, централізований вузол успішно оркеструє обмеження частоти запитів, захищаючи систему від DDoS-атак [4]. Головним недоліком централізованого підходу є ризик виникнення єдиної точки відмови, коли збій шлюзу зупиняє роботу всієї системи. Також існує небезпека накопичення на

шлюзі надлишкової логіки трансформації даних, що перетворює його на застарілу шину ESB та руйнує модульність. До того ж цей підхід неефективний проти внутрішніх загроз, оскільки трафік за периметром межі мережі часто залишається незахищеним [4]. Децентралізовані рішення, зокрема сервісні мережі, гарантують високу безпеку внутрішнього трафіку завдяки взаємному шифруванню та автентифікації кожного міжсервісного з'єднання. Це мінімізує ризики перехоплення даних та локалізує наслідки потенційної компрометації окремого мікросервісу. Проте недоліками децентралізації є суттєве ускладнення експлуатації інфраструктури, зростання витрат обчислювальних ресурсів на утримання проксі-серверів, а також додаткові затримки при передачі даних [4]. Процес вибору архітектури захисту для корпоративних інформаційних систем повинен базуватися на обов'язковому попередньому моделюванні загроз (threat modeling), наприклад за методологією STRIDE, що дозволяє чітко визначити вектори потенційних атак на кожному етапі руху інформаційних потоків. Для великих інфраструктур найбільш раціональним є впровадження гібридної моделі, яка гармонійно поєднує централізовані та децентралізовані механізми захисту відповідно до типу мережевого трафіку [4]. При проектуванні систем ідентифікації та авторизації розробникам слід чітко розділяти контекст безпеки кінцевих користувачів та внутрішніх систем, уникаючи змішування або суміщення їхніх ключів доступу [4]. Для інтеграції зовнішніх розробників та партнерів доцільно розгортати окремі ізольовані екземпляри шлюзів API, щоб запобігти каскадним збоям та обмежити доступ до критично важливих внутрішніх сервісів [1]. Будь-які конфігурації мають підлягати регулярному автоматизованому аудиту, а бізнес-логіка повинна залишатися відокремленою від інфраструктурних рівнів шлюзу чи сервісної мережі для забезпечення довгострокової еволюції та масштабованості корпоративної системи [4].

Висновки до розділу 2

В цьому розділі було проведено аналіз методів автентифікації, авторизації, технічного захисту та архітектурних підходів до забезпечення безпеки API в корпоративних інформаційних сервісах. Дослідження показало, що надійний

контроль доступу базується на суворому дотриманні принципу мінімальних привілеїв, де застарілі підходи поступаються токен-орієнтованим технологіям (JWT, OAuth 2.0, OpenID Connect), які забезпечують гнучку рольову (RBAC) та контекстну (scope-based) авторизацію. Захист інтерфейсів потребує багаторівневого технічного бар'єра, що поєднує криптографічне шифрування трафіку за допомогою протоколів HTTPS/TLS, механізми обмеження запитів (rate limiting) для запобігання DDoS-атакам і перевантаженням, а також сувору серверну валідацію, фільтрацію та санітизацію вхідних даних для нейтралізації ін'єкцій та XSS-вразливостей. У контексті мікросервісних та хмарних архітектур обґрунтовано необхідність переходу від концепції захищеного периметра до парадигми нульової довіри (Zero Trust), де ефективна координація міжсервісної взаємодії та централізація захисних функцій досягається шляхом впровадження шлюзів API (API Gateways) у поєднанні з автоматизованими системами управління секретами, безперервного моніторингу й журналювання подій. Загалом доведено, що максимальна захищеність корпоративних API досягається виключно через системну інтеграцію криптографічних засобів, сучасних протоколів контролю доступу та централізованих архітектурних рішень, що створює теоретичний базис для подальшого практичного проектування системи безпеки.

Розділ 3 ПРАКТИЧНІ АСПЕКТИ ВПРОВАДЖЕННЯ ТА ОЦІНКИ ЕФЕКТИВНОСТІ ЗАХИСТУ API

У даному розділі буде розглянуто практичні аспекти забезпечення безпеки API на прикладі умовної корпоративної інформаційної системи, реалізованої за принципами клієнт-серверної та мікросервісної архітектури. Система використовує REST API для взаємодії між вебзастосунком, серверними сервісами та базою даних, а також забезпечує обробку й передачу конфіденційної інформації користувачів. У межах практичного дослідження розглядаються основні загрози безпеці програмного інтерфейсу, процес моделювання потенційних атак, а також методи технічного захисту, зокрема механізми автентифікації, авторизації, обмеження кількості запитів та валідація вхідних даних. Окрему увагу буде приділено використанню сучасних інструментів тестування безпеки API та оцінці ефективності впроваджених захисних механізмів. Практична частина дослідження спрямована на демонстрацію комплексного підходу до забезпечення безпеки API в корпоративних інформаційних системах та визначення рекомендацій щодо підвищення рівня їх захищеності.

3.1. Моделювання загроз та процесу забезпечення безпеки API

Ефективне забезпечення інформаційної безпеки сучасних корпоративних інформаційних систем неможливе без детального аналізу архітектурних особливостей їхньої взаємодії. Сучасні тенденції проектування програмного забезпечення орієнтовані на перехід від монолітних структур до розподілених мікросервісних та клієнт-серверних архітектур. У контексті дослідження розглядається умовна корпоративна система, яка функціонує на основі мікросервісного підходу, де кожен окремий функціональний блок (наприклад, управління користувачами, системи оплати, аналітика і тд) ізольований та розгорнутий у вигляді самостійного сервісу. Головним інструментом, що забезпечує інтеграцію цих компонентів між собою, а також їх взаємодію із зовнішніми клієнтами (веб-додатками, мобільними клієнтами, сторонніми партнерськими сервісами), є API. Застосування інтерфейсів типу REST,

GraphQL чи gRPC дозволяє стандартизувати обмін даними, проте одночасно створює розширену поверхню атаки, оскільки кожен відкритий кінцевий пункт (endpoint) потенційно стає точкою входу для зловмисника [40]. Через архітектурні компоненти API безперервно циркулюють значні обсяги чутливої інформації, яка потребує суворого дотримання критеріїв конфіденційності, цілісності та доступності. До таких даних належать автентифікаційні та авторизаційні параметри (сесійні токени, ключі API, паролі), персональні дані користувачів, фінансова інформація, комерційні таємниці та критичні системні конфігурації. Особливість програмних інтерфейсів полягає в тому, що вони за своєю природою надають прямий доступ до логіки додатку та баз даних, оминаючи традиційні інтерфейси користувача, які раніше виконували роль додаткового захисного бар'єру. Порушення безпеки на цьому рівні може призвести до масштабних компрометацій усієї інфраструктури, що зумовлює необхідність детального аналізу ключових активів системи. Основними активами досліджуваної КІС, є інформаційні, програмні та системні ресурси. Інформаційні активи охоплюють бази даних із профілями користувачів, транзакційні логи та корпоративну звітність. Програмні активи включають безпосередньо бізнес-логіку мікросервісів та механізми генерації маркерів доступу, таких як JSON Web Tokens. Системні активи репрезентовані обчислювальними потужностями серверів та пропускною здатністю мережевих каналів. Потенційні загрози цим активам мають різноманітний характер і можуть реалізовуватися як зовнішніми, так і внутрішніми порушниками. Серед базових загроз безпеці API виділяють несанкціонований доступ до функціоналу, умисний витік або модифікацію даних, перехоплення та підробку токенів доступу, ін'єкційні атаки на рівні СУБД чи інтерпретаторів операційної системи, а також деструктивний вплив на доступність сервісів шляхом вичерпання системних ресурсів. Для систематизації та детального розуміння природи цих ризиків доцільно проаналізувати типові вектори атак, спираючись на актуальні аналітичні звіти та методологічні рекомендації консорціуму OWASP, зокрема документ OWASP API Security Top 10 [19]. Експлуатація

вразливостей програмних інтерфейсів зловмисниками найчастіше починається з розвідки та аналізу структури запитів. Одним з найбільш критичних векторів є порушення авторизації на рівні об’єктів (BOLA), коли атакуючий змінює ідентифікатор ресурсу в URI або тілі запиту, отримуючи доступ до чужих даних через відсутність належної перевірки прав власності на об’єкт з боку сервера. Аналогічно, загроза порушення авторизації на рівні функцій (BFLA) дозволяє зловмиснику шляхом маніпуляцій з HTTP-методами (наприклад, заміна GET на DELETE або PUT) виконувати адміністративні дії без відповідних привілеїв [16, 19].

Крім логічних помилок авторизації, суттєвий вектор атак пов’язаний із масовим призначенням параметрів (Mass Assignment), де вразливість виникає через автоматичне зв’язування вхідних даних клієнта з внутрішніми моделями об’єктів без фільтрації дозволених полів. Це дозволяє зловмиснику оновити приховані властивості, наприклад, змінити статус облікового запису на “admin”. Сценарії атак також часто орієнтовані на неконтрольоване споживання ресурсів (Unrestricted Resource Consumption), коли відсутність rate limiting або розмір завантажуваних пакетів призводить до відмови в обслуговуванні (DoS) або значних фінансових витрат у хмарних інфраструктурах. Реалізація ін’єкційних атак (SQL, NoSQL, Command Injection) через параметри API залишається поширеним інструментом компрометації баз даних, якщо вхідні параметри не проходять суворої типізації та валідації. Наслідки успішної реалізації зазначених векторів варіюються від репутаційних втрат і штрафів за витік персональних даних до повної втрати контролю над корпоративною системою [16].

З метою превентивного виявлення, класифікації та мінімізації описаних ризиків у процесі розробки та експлуатації КІС застосовується методологія моделювання загроз (Threat Modeling). Цей структурований підхід дозволяє ідентифікувати потенційні вразливості на ранніх етапах життєвого циклу ПЗ [19]. Для побудови ефективної моделі загроз API в межах даного дослідження розглянемо три концептуальні методології: STRIDE, DREAD та PASTA.

Методологія STRIDE, розроблена корпорацією Microsoft, фокусується на аналізі системи за шістьма категоріями загроз:

- Спуфінг (Spoofing) – підміна автентифікаційних даних користувача або мікросервісу;
- Тампериng (Tampering) – несанкціонована модифікація даних у канаді зв'язку або сховищі;
- Репудіація (Repudiation) – заперечення фактів виконання операцій через відсутність журналювання;
- Витік інформації (Information Disclosure) – отримання доступу до конфіденційних даних;
- Відмова в обслуговуванні (Denial of Service) – блокування легітимного доступу до програмного інтерфейсу;
- Ескалація привілеїв (Elevation of Privilege) – отримання несанкціонованих прав адміністратора [41, 42].

Для оцінки та пріоритезації виявлених за допомогою STRIDE ризиків використовується модель DREAD, яка дозволяє обчислити інтегральний показник небезпеки за п'ятьма критеріями, кожен з яких оцінюється за шкалою від 1 до 10:

- Damage (Потенційні збитки);
- Reproducibility (Відтворюваність атаки);
- Exploitability (Легкість експлуатації);
- Affected users (Кількість постраждалих користувачів);
- Discoverability (Легкість виявлення вразливості) [42].

На відміну від суто технічно-орієнтованих підходів, методологія PASTA (Process for Attack Simulation and Threat Analysis) є ризик-орієнтованою фреймворк-структурою, яка тісно пов'язує технічні загрози API з бізнес-цілями організації. Вона включає сім послідовних етапів, починаючи від визначення бізнес-завдань та окреслення меж технічних компонентів КІС, до аналізу вразливостей, симуляції атак та розробки стратегій захисту [43]. Застосування PASTA до досліджуваної мікросервісної архітектури дозволяє чітко визначити,

які саме кінцеві точки програмного інтерфейсу становлять найбільший фінансовий чи операційний ризик для компанії. Інтеграція результатів моделювання загроз у процес проектування дозволяє обґрунтовано обрати та впроваджувати комплексні механізми захисту API, формуючи ешелоновану систему оборони. Основним захисним бар'єром на периметрі КІС є шлюз API (API Gateway). Він виконує функцію єдиної точки входу, забезпечуючи маршрутизацію запитів, централізоване керування трафіком та ізоляцію внутрішніх мікросервісів від прямого доступу із зовнішньої мережі. На рівні API Gateway реалізується механізм обмеження частоти запитів (Rate LImiting), що мінімізує ризики атак типу DoS та brute-force штучним обмеженням кількості дозволених звернень з однієї IP-адреси або від одного токена за одиницю часу [12]. Для забезпечення конфіденційності та цілісності даних під час їх транспортування є обов'язковим використання протоколу TLS останніх версій з криптографічно стійкими шифронаборами, що унеможливлює проведення атак класу "людина посередині" (MitM). Процеси автентифікації та авторизації реалізуються на основі сучасних стандартів OAuth 2.0 та OpenID Connect із використанням токенів JWT. Ці маркери повинні мати обмежений термін дії, підписуватися асиметричними алгоритмами криптографії (наприклад, RS256) та обов'язково валідуватися кожним мікросервісом на предмет цілісності підпису та актуальності часових міток [12, 25].

Невід'ємною складовою архітектури безпеки є наскрізний моніторинг та журналювання подій (Logging and Monitoring). Реєстрації підлягають усі факти невдалих спроб автентифікації, запити до адміністративних функцій, а також аномальні сплески активності. Інтеграція логів із системами класу SIEM (Security Information and Event Management) дозволяє в режимі реального часу ідентифікувати ознаки цілеспрямованих атак та оперативно реагувати на інциденти. Кореляція між виявленим під час моделювання загрозами та впровадженими механізмами захисту наведена в таблиці 3.1.

Таблиця 3.1.

Матриця відповідності загроз та механізмів захисту API

Категорія загрози (STRIDE)	**Специфічний ризик API (OWASP)**	**Механізм захисту (Mitigation)**
Spoofing	Broken Authentication	OAuth 2.0, OpenID Connect, криптографічні JWT
Tampering	Injection / Mass Assignment	Валідація вхідних даних за схемою, використання TLS
Repudiation	Improper Assets Management	Наскрізне журналювання, аудит дій у SIEM
Information	Broken Object Level Authorization (BOLA)	Контроль доступу на рівні об’єктів (RBAC/ABAC)
Denial of Service	Unrestricted Resource Consumption	Rate Limiting на API Gateway, фільтрація трафіку
Elevation of Privilege	BOLA	Валідація прав доступу на кожному мікросервісі

Таким чином, розроблена теоретична модель загроз та визначений комплекс захисних механізмів закладають концептуальний базис безпеки корпоративної системи. Проте практична ефективність запропонованих рішень та точність конфігурації захисних бар’єрів потребують регулярної перевірки. Для автоматизації процесів пошуку проблем в захисті та верифікації стійкості інтерфейсів доцільно використовувати спеціалізоване ПЗ. Це обумовлює необхідність переходу до аналізу практичних інструментів аналізу захищеності, що буде детально розглянуто в наступному підрозділі.

3.2. Інструментальні засоби тестування безпеки API

Ефективна протидія загрозам, що виникають у процесі функціонування програмних інтерфейсів, вимагає не лише впровадження превентивних механізмів захисту, детально розглянутих в попередньому підрозділі, а й

систематичного оцінювання їхньої практичної ефективності. Роль інструментального тестування у загальному життєвому циклі розробки ПЗ полягає у верифікації реалізованих політик безпеки та виявленні прихованих дефектів коду чи конфігурацій ще до моменту розгортання системи в продуктивному середовищі [13, 44]. Основними завданнями є перевірка стійкості логіки додатків, контроль за належним виконанням бізнес-правил, а також виявлення специфічних для API архітектурних слабкостей [13, 19]. Необхідність регулярного проведення аудиту безпеки зумовлена динамічним характером розробки, коли кожне оновлення або зміна специфікації може призвести до появи нових векторів атак. Оскільки зловмисники постійно вдосконалюють методи експлуатації вразливостей, автоматизований та напівавтоматизований пошук недоліків дозволяє мінімізувати ризики несанкціонованого доступу до конфіденційних даних [44]. Застосування інструментальних засобів на етапі контролю якості забезпечує безперервний моніторинг захищеності, трансформуючи теоретичні вимоги безпеки у практично підтверджену стійкість програмного інтерфейсу.

Контроль доступу є першим і найважливішим контуром захисту будь-якого API, тому тестування механізмів ідентифікації та розмежування повноважень посідає ключове місце в процесі аудиту. Для аналізу коректності реалізації цих функцій застосовують спеціальні інструменти перехоплення та модифікації трафіку, серед яких провідне місце посідають проксі-сервери Burp Suite та OWASP ZAP [13]. Дослідження стійкості маркерів доступу, ключів API та токенів JWT здійснюється шляхом їх навмисного спотворення, підробки або повторного використання скомпрометованих сесій [2, 13].

Під час аналізу JWT-токенів за допомогою інтегрованих модулів Burp Suite або розширень OWASP ZAP перевірці підлягають такі параметри:

- криптографічний підпис токена (спроби підробки за допомогою зміни алгоритму на “none”);
- термін дії маркера (валідація параметрів exp (expiration) та nbf (not before));

- стійкість секретних ключів до атак методом повного перебору (brute-force).

Паралельно з цим, інструменти на кшталт SoapUI та Postman дозволяють автоматизувати процеси надсилання запитів із різними рівнями повноважень. Це необхідно для виявлення критичних помилок конфігурацій, пов’язаних із порушеннями авторизації BOLA та BFLA [13, 19]. Типові архітектурні прорахунки, які фіксуються під час такого тестування, охоплюють:

- відсутність належної перевірки прав власності на ресурс у базі даних (коли зміна ідентифікатора в URI дозволяє отримати доступ до чужих профілів);
- передачу конфіденційних токенів через незахищені URL-параметри;
- збереження працездатності сесії після явного виходу користувача із системи [2, 19].

Порівняльний аналіз показує, що якщо Postman є ефективним для швидкої перевірки контрактів та базових сценаріїв контролю доступу, то Burp Suite надає значно глибші можливості для маніпуляцій з низькорівневими заголовками HTTP, що є критично важливим для комплексного тестування логіки автентифікації [13].

Вхідні параметри, що надходять до API від клієнтських додатків, є основним джерелом потенційних загроз, якщо система не здійснює їх належної фільтрації та типізації. Ретельна перевірка структури даних є критично важливою, оскільки саме через дефекти валідації зловмисники здатні реалізувати ін’єкційні атаки (SQL, NoSQL, OS Command Injections) або порушити логіку обробки об’єктів у пам’яті сервера [13, 19]. Для авторизації цього процесу використовуються інструменти фазинг-тестування (fuzzing), які генерують великі масиви аномальних, надлишкових або некоректних даних з метою викликати збій у роботі системи або отримати непередбачувану відповідь від БД [13]. Функціональні можливості інструментів Swagger (OpenAPI) дозволяють здійснювати валідацію на відповідність схемі проєкту. Співставлення структури реальних запитів із задекларованою специфікацією

допомагає виявити вразливості масового призначення, коли надсилання додаткових полів у JSON-об’єкті призводить до несанкціонованої зміни системних атрибутів у базі даних [13, 45]. В свою чергу, сканери безпеки OWASP ZAP та модулі Burp Suite дозволяють автоматично підставляти шкідливі навантаження (payloads) у змінні запиту, заголовки та тіло повідомлень. Це забезпечує виявлення таких недоліків, як:

- некоректне відображення деталізованих системних блоків (stack trace), що полегшує зловмиснику розвідку архітектури;
- відсутність обмежень на довжину або тип символів у полях введення;
- вразливість до впровадження стороннього коду через неправильну інтерпретацію спецсимволів сервером додатків [13, 19].

Спільне використання специфікацій OpenAPI для контрактного тестування та динамічних сканерів для пошуку ін’єкцій забезпечує перехресний контроль, який мінімізує ймовірність компрометації внутрішніх систем через вхідні потоки даних [44, 45].

Механізм обмеження частоти запитів (rate limiting), призначення якого було обґрунтовано в межах формування загальної стратегії захисту в попередньому підрозділі, вимагає обов’язкової практичної валідації на стійкість до стресових навантажень. Цей контур безпеки призначений для запобігання відмовам в обслуговуванні (DoS/DDoS), блокування автоматизованого перебору паролів або маркерів, а також для захисту обчислювальних ресурсів сервера від виснаження через надмірну активність клієнтів. Тестування обмежених запитів полягає в імітації інтенсивного потоку звернень до конкретних кінцевих точок (endpoints) API за короткий проміжок часу [2, 19]. Для реалізації таких сценаріїв застосовують утиліти навантажувального тестування та спеціалізовані скрипти в середовищах Postman або SoapUI, які дозволяють генерувати тисячі послідовних або паралельних HTTP-запитів [13]. Під час аналізу оцінюється реакція системи на перевищення встановлених лімітів: коректно налаштований API повинен повертати стандартний HTTP-статус “429 Too Many Requests” та містити відповідні заголовки керування чергою (наприклад, Retry-After) [2].

Аналіз логів під час тестування часто виявляє помилки, коли обмеження накладаються лише на основі IP-адреси клієнта. Це дозволяє обходити захист шляхом ротації проксі-серверів або підміни заголовків сімейства X-Forwarded-For, що свідчить про необхідність прив’язки лімітів до унікальних ідентифікаторів сесій чи токенів автентифікації [2, 13].

Синтез даних, отриманих у результаті застосування комплексу інструментальних засобів, дозволяє сформувати цілісну картину поточного стану захищеності програмного інтерфейсу. Узагальнення результатів роботи Burp Suite, OWASP ZAP, Postman та засобів валідації контрактів Swagger дає змогу класифікувати виявлені дефекти за категоріями критичності та співвіднести їх з актуальними класифікаторами загроз, зокрема з переліком OWASP API Security Top 10 [13, 19]. Системний підхід до аналізу звітів автоматизованого сканування дозволяє відсіяти хибнопозитивні спрацьовування (false positives) та сфокусувати увагу на реальних архітектурних і програмних вразливостях. У процесі інструментального аудиту зазвичай виявляються ключові категорії вразливостей, наведені в таблиці 3.2.

Таблиця 3.2

Класифікація вразливостей API за результатами інструментального тестування

Категорія вразливості	**Основні інструменти виявлення**	**Наслідки для безпеки API**
Порушення логіки авторизації (BOLA/BFLA)	Burp Suite, OWASP ZAP, SoapUI	Несанкціонований доступ до даних інших користувачів, виконання адміністративних функцій
Дефекти валідації та ін’єкції	Burp Intruder, Swagger (OpenAPI)	Витік або пошкодження даних в СУБД, виконання довільних команд на сервері
Некоректна конфігурація безпеки	OWASP ZAP, Postman	Використання слабких шифрів, витік системної інформації через HTTP-заголовки

Категорія вразливості	**Основні інструменти виявлення**	**Наслідки для безпеки API**
Недостатнє обмеження швидкості	Postman, спеціалізовані скрипти навантаження	Сприйнятливість до DoS-атак, автоматизований збір даних (scraping)

Значення інструментального тестування для підвищення рівня захищеності API полягає в переведенні процесу безпеки з реактивного стану в проактивний. Своєчасне виявлення описаних дефектів до виходу системи в реліз дозволяє розробникам оперативно внести зміни в код, скоригувати налаштування API Gateways та оптимізувати конфігурацію веб-серверів, що мінімізує ймовірність успішного проведення атак у майбутньому [13, 44]. Таким чином, використання сучасного інструментарію для тестування безпеки API – від проксі-серверів аналізу трафіку до засобів контрактної валідації специфікацій – є невід'ємною умовою стоверння захищених інформаційних систем. Комплексне поєднання методів динамічного аналізу, фазингу вхідних даних та перевірки контролю доступу дозволяє ефективно локалізувати критичні вразливості на ранніх етапах. Отримані під час тестування аналітичні дані та результати класифікації дефектів складають практичну основу для розробки цільових контрзаходів. Це зумовлює необхідність переходу до формування конкретних інженерних рішень та архітектурних шаблонів, які будуть детально розглянуті в наступному підрозділі.

3.3. Практичні рекомендації щодо забезпечення безпеки API

На основі проведеного аналізу вразливостей та результатів моделювання загроз стає очевидним, що забезпечення стійкості API-інфраструктури вимагає переходу від фрагментарних заходів до цілісної стратегії захисту. Практична реалізація безпеки повинна базуватися на ієрархічній системі контролю, де кожен рівень нівелює специфічні ризики, ідентифіковані під час оцінки ландшафту загроз. Ефективне управління доступом є першочерговим етапом нейтралізації ризиків, пов'язаних із несанкціонованим доступом до об'єктів та функцій програмних інтерфейсів. Враховуючи критичність атак на рівні логіки

доступу, доцільним є впровадження сучасних протоколів автентифікації та авторизації, зокрема OAuth 2.0 та OpenID Connect. Використання OAuth 2.0 дозволяє делегувати права доступу без розкриття облікових даних користувача, що суттєво обмежує поверхню атаки [25]. Для забезпечення цілісності та компактної передачі даних про повноваження рекомендується застосування токенів формату JWT. Проте його безпека залежить від коректного використання алгоритмів цифрового підпису та обмеженого терміну дії токенів, що запобігає тривалій експлуатації викрадених ідентифікаторів. Реалізація багатофакторної автентифікації є обов'язковою для критичних вузлів системи, оскільки вона створює додатковий бар'єр проти компрометації облікових записів [45]. Ключовим принципом архітектурної безпеки на цьому рівні є дотримання концепції мінімальних привілеїв, яка передбачає надання суб'єктам лише тих прав, що є необхідними для виконання конкретних операцій. Це дозволяє ефективно запобігати горизонтальному та вертикальному підвищенню привілеїв, що було визначено як одна з найбільш небезпечних загроз у межах OWASP Top 10 API Security [19]. Технічний захист API потребує комплексного застосування інструментів фільтрації та криптографічного захисту даних під час їх транзиту. Використання протоколу TLS останніх версій у поєднанні з HTTPS є базовою вимогою для забезпечення конфіденційності та запобігання перехопленню інформації. Для протидії атакам типу "відмова в обслуговуванні" та спробам підбору паролів критично важливо впроваджувати механізми обмеження інтенсивності запитів. Ці заходи дозволяють контролювати навантаження на систему та ідентифікувати аномальну активність на ранніх етапах. Централізованим вузлом управління безпекою має виступати API Gateway, який забезпечує виконання політик безпеки, оркестрацію трафіку та приховування внутрішньої структури мікросервісів [36]. Застосування підходу Security By Design на етапі проектування гарантує, що механізми безпеки будуть інтегровані в архітектуру API як фундаментальні властивості, а не як надбудови, що значно знижує ймовірність появи логічних вразливостей у майбутньому [45].

Забезпечення тривалого життєвого циклу безпечного програмного інтерфейсу неможливе без безперервного моніторингу та регулярного аудиту. Журналювання подій безпеки повинно охоплювати всі спроби автентифікації, зміни прав доступу та виявлені помилки валідації, забезпечуючи достатній рівень деталізації для подальшого розслідування інцидентів. Системи моніторингу мають працювати в режимі реального часу, аналізуючи поведінкові патерни для виявлення ознак масового витоку даних або нетипового використання API-ендпоінтів [19]. Регулярне тестування на вразливості за допомогою інструментів статичного (Static Application Security Testing, SAST) та динамічного (Dynamic Application Security Testing, DAST) аналізу дозволяє виявляти недоліки коду та конфігурацій до їх потрапляння в production середовище. Інтеграція автоматизованих перевірок безпеки в процеси безперервної інтеграції та доставки (Continuous Integration / Continuous Delivery, CI/CD) у межах концепції DevSecOps забезпечує стабільну якість захисту при кожному оновленні системи. Такий підхід мінімізує вплив людського фактору та дозволяє швидко реагувати на нові вектори атак, що постійно з’являються в динамічному середовищі корпоративних інформаційних систем [36]. Формування цілісної моделі безпеки API вимагає синергії організаційних регламентів, архітектурних рішень та технічних засобів контролю. Узагальнюючи результати дослідження, можна стверджувати, що лише багаторівневий підхід, де кожен наступний етап захисту дублює та посилює попередній, здатний забезпечити стійкість до складних цілеспрямованих атак. Архітектурна безпека закладає фундамент через правильну сегментацію та вибір протоколів, технічні засоби реалізують оперативний контроль трафіку, а моніторинг забезпечує прозорість функціонування всієї інфраструктури. Побудова безпечної API-інфраструктури повинна базуватися на постійному циклі вдосконалення, де дані про нові загрози використовуються для коригування політик доступу та налаштування засобів захисту. Таким чином, інтегрована система безпеки стає не просто набором обмежень, а стратегічним активом, що гарантує надійність корпоративних сервісів та збереження

критично важливої інформації.

Висновки до розділу 3

У цьому розділі було розглянуто практичні аспекти забезпечення безпеки API, що охоплюють процес моделювання загроз, використання інструментальних засобів тестування та формування практичних рекомендацій щодо підвищення рівня захищеності програмних інтерфейсів. Проведений аналіз підтвердив, що ефективне забезпечення безпеки API потребує комплексного підходу, який поєднує попередню оцінку ризиків, впровадження відповідних механізмів захисту та їх регулярну перевірку за допомогою спеціальних засобів тестування. У процесі моделювання загроз було визначено основні активи, що потребують захисту під час функціонування API, а також проаналізовано найбільш поширені вектори атак, спрямовані на порушення конфіденційності, цілісності та доступності інформації. Розгляд сучасних підходів до моделювання загроз дозволив встановити, що своєчасне виявлення потенційних ризиків на етапі проектування та розроблення API сприяє підвищенню ефективності подальших заходів захисту та зменшенню ймовірності успішної реалізації атак.

На основі проведеного аналізу було сформовано практичні рекомендації щодо забезпечення безпеки API, які передбачають впровадження сучасних механізмів автентифікації та авторизації, використання криптографічних засобів захисту, реалізацію контролю інтенсивності запитів, перевірку вхідних даних, а також організацію постійного моніторингу та аудиту подій безпеки. Встановлено, що найбільш ефективним є багаторівневий підхід до захисту, за якого окремі механізми безпеки доповнюють один одного та забезпечують протидію різним категоріям загроз.

Отже, результати проведеного дослідження свідчать, що забезпечення безпеки API має розглядатися як безперервний процес, який охоплює всі етапи життєвого циклу програмного інтерфейсу – від проєктування до експлуатації та супроводу. Поєднання моделювання загроз, регулярного тестування безпеки та впровадження сучасних захисних механізмів створює необхідні умови для підвищення рівня захищеності корпоративних інформаційних систем і забезпечення стійкості API до сучасних кіберзагроз.

ВИСНОВКИ

Досліджено теоретичні основи забезпечення безпеки API в корпоративних інформаційних системах. Встановлено, що API є одним із ключових компонентів сучасної цифрової інфраструктури підприємств, забезпечуючи взаємодію між внутрішніми та зовнішніми сервісами, вебзастосунками, мобільними платформами та хмарними середовищами. Водночас зростання ролі API супроводжується підвищенням рівня кіберризиків, пов'язаних із несанкціонованим доступом до даних та сервісів.

Проаналізовано основні загрози та вразливості API. Визначено, що найбільшу небезпеку становлять порушення механізмів автентифікації та авторизації, надмірне розкриття даних, ін'єкційні атаки, помилки конфігурації, недостатній контроль доступу та недоліки управління API. Встановлено, що значна частина успішних атак на корпоративні системи пов'язана саме з помилками реалізації та експлуатації API.

Досліджено принципи та стандарти забезпечення безпеки програмних інтерфейсів. Визначено, що ефективний захист API повинен базуватися на принципах конфіденційності, цілісності та доступності інформації, принципі мінімальних привілеїв, підходах Secure by Design та Zero Trust. Встановлено важливість використання міжнародних стандартів і рекомендацій для формування комплексної системи захисту API.

Проаналізовано сучасні методи автентифікації та авторизації API. Визначено особливості застосування API-ключів, токенів доступу, протоколів OAuth 2.0, OpenID Connect та інших механізмів контролю доступу. Встановлено, що використання багаторівневих механізмів автентифікації та гнучких моделей авторизації суттєво знижує ризик несанкціонованого доступу до ресурсів корпоративних систем.

Досліджено методи технічного захисту API, зокрема шифрування даних, обмеження частоти запитів, валідацію вхідних даних, моніторинг подій безпеки та механізм журналювання. Встановлено, що комплексне використання зазначених засобів дозволяє підвищити стійкість API до поширених кіберзагроз і забезпечити належний рівень захисту інформаційних ресурсів.

Проаналізовано архітектурні підходи до забезпечення безпеки програмних інтерфейсів у мікросервісних та хмарних середовищах. Визначено роль API Gateway, Service Mesh, централізованого управління політиками безпеки та механізмів сегментації мережевої взаємодії. Встановлено, що впровадження таких підходів сприяє підвищенню керованості та масштабованості систем захисту API.

Досліджено процес моделювання загроз для програмних інтерфейсів та визначено його значення для своєчасного виявлення потенційних векторів атак. Встановлено, що використання моделей загроз дає змогу систематизувати

ризики, визначити критичні активи та обґрунтувати вибір необхідних заходів захисту ще на етапі проектування системи.

Проаналізовано сучасні інструментальні засоби тестування безпеки API. Визначено можливості спеціалізованих інструментів для виявлення вразливостей, перевірки механізмів автентифікації та авторизації, аналізу конфігурацій і проведення автоматизованого тестування; Встановлено, що регулярне тестування є необхідною складовою підтримання належного рівня безпеки API протягом усього життєвого циклу їх використання.

Обґрунтовано практичні рекомендації щодо забезпечення безпеки API в корпоративних інформаційних системах. Запропоновано використовувати комплексний підхід, який передбачає впровадження надійних механізмів автентифікації та авторизації, застосування шифрування даних, обмеження запитів, регулярне тестування безпеки, моніторинг подій, дотримання принципів безпечної розробки та використання актуальних стандартів і рекомендацій у сфері захисту API.

Загалом мету роботи досягнуто, а поставлені завдання виконано в повному обсязі. Отримані результати мають практичне значення для розробників, системних адміністраторів та фахівців з кібербезпеки й можуть бути використані під час проєктування, модернізації та експлуатації корпоративних інформаційних систем. Подальші дослідження доцільно спрямувати на вивчення застосування технологій штучного інтелекту для автоматизованого виявлення загроз API, удосконалення методів захисту в багатохмарних середовищах та розроблення нових підходів до забезпечення безпеки мікросервісних архітектур.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Jacobson D., Brail G., Woods D. APIs: A Strategy Guide. O'Reilly Media, 2012. 182 p.

2. Madden N. API Security in Action. Manning Publications, 2020. 560 p.

3. De B. API Management: An Architect's Guide to Developing and Managing APIs for Your Organization. Apress, 2017. 210 p.

4. Gough J., Bryant D., Auburn M. Mastering API Architecture: Design, Operate and Evolve API-Based Systems. Sebastopol : O'Reilly Media, 2022. 312 p.

5. Jin B., Sahni S., Shevat A. Designing Web APIs: Building APIs That Developers Love. Sebastopol : O'Reilly Media, 2018. 202p.

6. Boyd M. Developing the API Mindset. Nordic APIs, 2015. 96 p.

7. Porcelo E., Banks A. Learning GraphQL: Declarative Data Fetching for Modern Web Apps. 1st ed. O'Reilly Media, 2018. 292 p.

8. Indrasiri K. Kuruppu D. gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes. O'Reilly Media, 2020. 237 p.

9. Massé M. REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces. O'Reilly Media, 2011. 112 p.

10. Kalin M. Java Web Services: Up and Running. 2nd ed. O'Reilly Media, 2013. 331 p.

11. Ciampa M. CompTIA Security+ Guide to Network Fundamentals. 7th ed. Cengage Learning, 2022. 638p.

12. Siriwadena P. Advanced API Security: OAuth 2.0 and Beyond. 2nd ed. Apress, 2020. 638 p.

13. Ball C. J. Hacking APIs: Breaking Web Application Programming Interfaces. San Francisco : No Starch Press, 2022. 364 p.

14. Domonofey C. Defending APIs: Uncover advanced defense techniques to craft secure application programming interfaces. Birmingham : Packt Publishing, 2024. 360 p.

15. Peralta J. H. Secure APIs: Design, build, and implement. Shelter Island : Manning Publications Co., 2026. 360 p.

16. Shostack A. Threat Modeling: Designing for Security. Indianapolis : John Wiley & Sons, Inc., 2014. 614 p.

17. Continuous API Management: Making the Right Decisions in an Evolving Landscape / M. Medjaoui, E. Wilde, R. Mitra, M. Amundsen. Sebastopol : O'Reilly Media, 2018. 261 p.

18. International Organization for Standardization. ISO/IEC 27001:2022 Information security, cybersecurity and privacy protection — Information security management systems — Requirements. Geneva : ISO, 2022. 52 p.

19. OWASP Foundation. OWASP API Security Top 10 – 2023. URL: https://owasp.org/www-project-api-security/ (дата звернення: 01.05.2026).

20. Hardt D. The OAuth 2.0 Authorization Framework. RFC 6749. Internet Engineering Task Force, 2012. 76 p. URL: https://datatracker.ietf.org/doc/html/rfc6749 (дата звернення: 01.05.2026).

21. Archer G., Kahrer J., Trojanowski M. Cloud Native Data Security with OAuth: A Scalable Zero Trust Architecture. First Edition. Sebastopol : O'Reilly Media, Inc., 2025. 680 p.

22. Parecki A. The Little Book of OAuth 2.0 RFCs. Second Edition. Aaron Parecki, 2022. 412 p.

23. Laurens J., Rhee K. H. A Secure API Key Management Framework for Cloud-Based Web Services. International Journal of Information Security. 2021. Vol. 20, No. 4. P. 485–501.

24. Fielding R. T., Taylor R. N. Principled Design of the Modern Web Architecture. ACM Transactions on Internet Technology. 2002. Vol. 2, No. 2. P. 115–150.

25. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT). RFC 7519. Internet Engineering Task Force, 2015. 31 p. URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 01.05.2026).

26. Sakimura N., Bradley J., Jones M. OpenID Connect Core 1.0 incorporating errata set 1. OpenID Foundation, 2014. 85 p. URL: https://openid.net/specs/openid-connect-core-1_0.html (дата звернення: 01.05.2026).

27. Rose S., Borchert O., Mitchell S., Connelly S. Zero Trust Architecture. Gaithersburg : National Institute of Standards and Technology, 2020. 59 p. (NIST Special Publication 800-207). URL: https://doi.org/10.6028/NIST.SP.800-207 (дата звернення: 01.05.2026).

28. Ponelat J. S., Rosenstock L. L. Designing APIs with Swagger and OpenAPI. New York : Manning Publications, 2022. 340 p.

29. Preibisch S. API Development: A Practical Guide for Business Implementation Success. Richmond : Apress, 2018. 195 p.

30. Lodha S., Singhal A. Architectural patterns for securing application programming interfaces in cloud-native environments. Journal of Cyber Security and Information Technologies. 2024. Vol. 9, No. 2. P. 45–58.

31. Farhi D., Aleks N. Black Hat GraphQL: Attacking Next-Generation Web Applications. San Francisco : No Starch Press, 2023. 288 p.

32. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (General Data Protection Regulation). Official Journal of the European Union. 2016. L 119. P. 1–88.

33. Aumasson J. P. Serious Cryptography: A Practical Introduction to Modern Encryption. San Francisco : No Starch Press, 2018. 312 p.

34. Rescorla E. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. Internet Engineering Task Force (IETF), 2018. 132 p. URL: https://datatracker.ietf.org/doc/html/rfc8446 (дата звернення: 01.05.2026).

35. Chandramouli R. Security Strategies for Microservices-based Application Systems. NIST Special Publication 800-204B. National Institute of Standards and Technology, Gaithersburg, MD, 2019. 41 p.

36. Fielding R., Reschke J. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231. Internet Engineering Task Force (IETF), 2014.

177 p. URL: https://datatracker.ietf.org/doc/html/rfc7231 (дата звернення: 01.05.2026).

37. Siriwardena P., Dias N. Microservices Security in Action. Shelter Island : Manning Publications Co., 2020. 594 p.

38. Peralta J. H. MIcroservice APIs: Using Python, Flask, FastAPI, OpenAPI and more. Shelter Island : Manning Publications Co., 2023. 420 p.

39. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine, 2000. 180 p.

40. Lipner S. The Trustworthy Computing Security Development Lifecycle. Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC ‘04). IEEE, 2004. P. 14–25.

41. Meier J. D. et al. Improving Web Application Security: Threats and Countermeasures. Redmond : Microsoft Corporation, 2003. 434 p.

42. Ucedavéles T., Morana M. M. Risk Centric Threat Modeling: Process for Attack Simulation and Threat Analysis. John Wiley & Sons, 2015. 400 p.

43. Standard on API Security Testing and Vulnerability Assessment. NIST Special Publication 800-214. National Institute of Standards and Technology, 2024. 65 p.

44. OpenAPI Specification. Swagger Open Source Community, 2025. URL: https://swagger.io/specification/ (дата звернення: 01.05.2026).

45. Richer J., Sanso A. OAuth 2 in Action. Shelter Island : Manning Publications, 2017. 360 p.