Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра інформатики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: **«МЕХАНІЗМИ БЕЗПЕКИ У МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ»**

Виконав: студент 4-го року навчання,
Спеціальності
122 Комп'ютерні науки
Чалюк Андрій Олександрович
Керівник Андрощук М. В.,_____
Рецензент Глибовець А. М._____
(прізвище та ініціали)

Кваліфікаційна робота захищена з
оцінкою _____
Секретар ЕК _____
"___" _____ 20____ р.

Київ – 2023

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри інформатики,
к.ф-м.н., доц. Гороховський С.С
_____
(підпис)
“_____” _____2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на кваліфікаційну роботу

студенту Чалюку Андрію факультету Інформатики 4 року навчання
ТЕМА **«МЕХАНІЗМИ БЕЗПЕКИ У МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ»**

Зміст текстової частини до кваліфікаційної:

Зміст
Анотація
Вступ
Проблеми безпеки мікросервісної архітектури
Рекомендації щодо безпеки
Розробка застосунку на основі мікросервісної архітектури із використанням запропонованих механізмів безпеки
Висновки по роботі
Джерела інформації

Дата видачі “___” _____ 2022р.

Керівник
Андрощук М. В _____
(підпис)

Завдання отримав
Чалюк А. О _____
(підпис)

Тема: Механізми безпеки у мікросервісній архітектурі

Календарний план виконання роботи:

№ п/п	Перелік робіт	Термін виконання	Примітка
1.	Отримання завдання на кваліфікаційну роботу	26.12.2022	
2.	Огляд літератури за темою роботи	29.12.2022	
3.	Виконання аналізу сучасних рішень	18.01.2023	
4.	Вибір інструментів для розробки програмної частини	21.01.2023	
5.	Написання вступу та змісту	01.02.2023	
6.	Створення програмної частини	11.03.2023	
7.	Написання першого розділу	07.04.2023	
8.	Написання другого розділу	17.04.2023	
9.	Написання третього розділу	27.04.2023	
10.	Узгодження попередньої версії роботи з керівником	08.05.2023	
11,	Внесення змін до кваліфікаційної роботи відповідно до зауважень наукового керівника	11.05.2023	
12.	Попередній захист	12.05.2023	
13.	Корегування роботи за результатами попереднього захисту	22.05.2023	
14.	Захист кваліфікаційної роботи	30.05.2023	

Науковий керівник _Андрощук М. В.__

Виконавець кваліфікаційної роботи _Чалюк А. О.___

“___” _____

ЗМІСТ

СПИСОК СКОРОЧЕНЬ

STS – Security Token Service

HTTPS – HyperText Transfer Protocol Secure

SSL – Secure Sockets Layer

JSON – JavaScript Object Notation

RBAC – Role-Based Access Control

MITM – Man-In-The-Middle

DoS – Denial-of-Service

PEP – Policy Enforcement Point

SQL – Structured Query Language

XSS – Cross-Site Scripting

API – Application Programming Interface

MFA – Multi-Factor Authentication

OTP – One Time Password

FIDO – Fast Identity Online

SQS – Simple Queue Service

mTLS – Mutual Transport Layer Security

JWT – JSON Web Token

OIDC – OpenID Connect

DCT – Docker Content Trust

OWASP – Open Web Application Security Project

EKL – Elasticsearch Kibana Logstash

EKS – Elastic Kubernetes Service

CI/CD – Continuous Integration/Continuous Delivery

DTO – Data Transfer Object

RCE – Remote Code Execution

АНОТАЦІЯ

У рамках роботи було розглянуто вразливості, пов'язані з комунікацією між мікросервісами, а також забезпеченням аутентифікації та авторизації для доступу до сервісів. Проаналізовано можливі ризики, пов'язані з управлінням конфігурацією та моніторингом мікросервісів. Були розглянуті питання безпеки при використанні контейнерів та оркестраторів, таких як Docker і Kubernetes. Були запропоновані заходи для забезпечення ізоляції контейнерів, обмеження привілеїв та моніторингу активності контейнерів. Крім того, розглянуто можливість атак на цілісність даних у мікросервісних системах, а також засоби для забезпечення цілісності та шифрування даних, щоб запобігти несанкціонованому доступу та модифікації інформації.

Ключові слова: мікросервісна архітектура, безпека застосунків, механізми захисту, захист від атак, моніторинг, керування доступом

ВСТУП

Актуальність. Мікросервісна архітектура користується все більшою популярністю в розробці програмного забезпечення. Це зумовлюється її модульністю – сервіси можна компонувати, змінювати та повторно використовувати; стійкістю до відмов - у разі виникнення збоїв у роботі одного з компонентів, інші продовжують роботу, що дозволяє програмі працювати навіть без частини функціоналу; масштабованість – кожен сервіс можна масштабувати окремо у разі нестачі його поточної потужності без необхідності зміни інших компонентів програми. Проте цей архітектурний підхід, крім класичних загроз безпеки, що притаманні архітектурі моноліту, також містить нові проблеми, адже з’являється новий потенційно вразливий шар комунікації між сервісами.

Незважаючи на те, що тема дослідження не є новою і досить детально описана десятками авторів, вона все ще є актуальною через безперервну «гонку озброєнь» за участі розробників та хакерів. Сама ж архітектура є досить популярною, вона користується попитом серед великих компаній, які потребують високонавантажених і масштабованих систем (Netflix, Amazon, Uber, Spotify, SoundCloud) [1].

Мета дослідження. Огляд механізмів протидії проблемам безпеки та створення застосунку з використанням найсучасніших механізмів безпеки для їх демонстрації.

Завдання дослідження. Дослідити та реалізувати механізми захисту від загроз безпеки.

Об’єкт дослідження. Мікросервісна архітектура.

Предмет дослідження. Механізми безпеки, інструменти для їх впровадження.

Джерела досліджень. Електронні версії друкованої літератури, електронні ресурси, документація.

У цій роботі досліджуватимуться проблеми безпеки, пов’язані з архітектурою мікросервісів, і будуть запропоновані найактуальніші стратегії їх пом’якшення.

Робота складається з трьох розділів.

Перший розділ присвячено розгляду ряду найпоширеніших загроз безпеці у мікросервісній архітектури.

В другому досліджуються сучасні методи та засоби захисту від загроз, надано рекомендації щодо безпеки у мікросервісній архітектурі.

В третьому розділі описується практичне використання механізмів безпеки на прикладі створення мікросервісного застосунку.

РОЗДІЛ 1. ПРОБЛЕМИ БЕЗПЕКИ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

У мікросервісній архітектурі можуть виникати різноманітні проблеми з безпекою, оскільки вона базується на розподіленій системі з великою кількістю компонентів, які взаємодіють між собою. Безпека є складним завданням, здебільшого через саму природу архітектури. На відміну від монолітного додатку, у якому взаємодія між компонентами відбувається у межах одного процесу, в архітектурі мікросервісів ці компоненти, розробляються як окремі, незалежні сервіси. Крім того, кожен мікросервіс тепер незалежно приймає запити або має власні точки входу, що значно розширює поверхню можливих атак. Види вразливостей можуть бути пов’язані з контролем доступу, розгортанням, мережною комунікацію, управлінням конфігураціями, витоком чутливої інформації тощо.

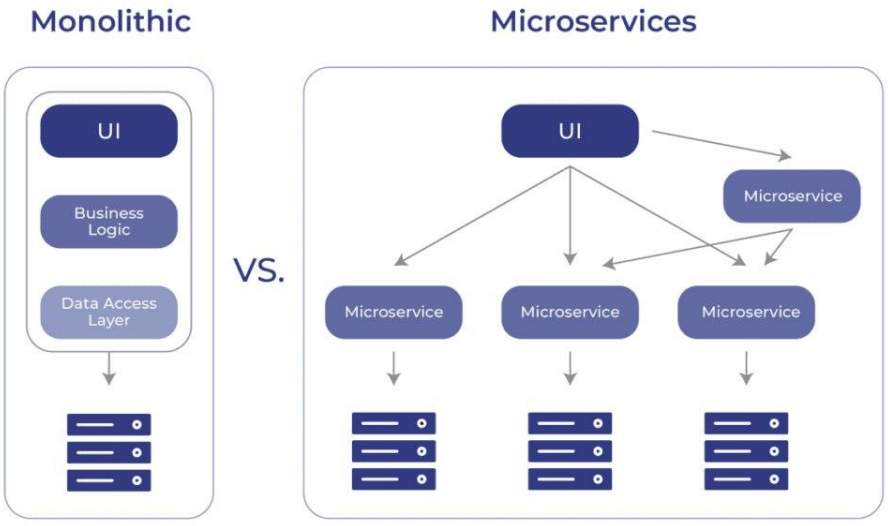


Рисунок 1.1 Порівняння монолітної та мікросервісної архітектур

1.1 Контроль доступу

Проблеми з контролем доступу входять до головних проблем безпеки у мікросервісній архітектурі та в основному виникають через неправильну аутентифікацію та авторизацію, які є важливими складовими безпеки: аутентифікація полягає у перевірці ідентифікаційних даних користувача або сервісу, що намагається отримати доступ до захищених ресурсів, а авторизація, у свою чергу, – у визначенні рівня цього доступу. Цей процес може включати в

себе визначення ролей користувача (при використанні RBAC), за якими здійснюється визначення переліку можливих дій у системі. Ролі використовуються для групування користувачів і визначення ресурсів і списку дій, до яких вони можуть мати доступ. Без належної роботи системи аутентифікації та авторизації зловмисники можуть порушуватись цілісність даних та конфіденційність інформації, або навіть завдати шкоди усій програмі тощо.

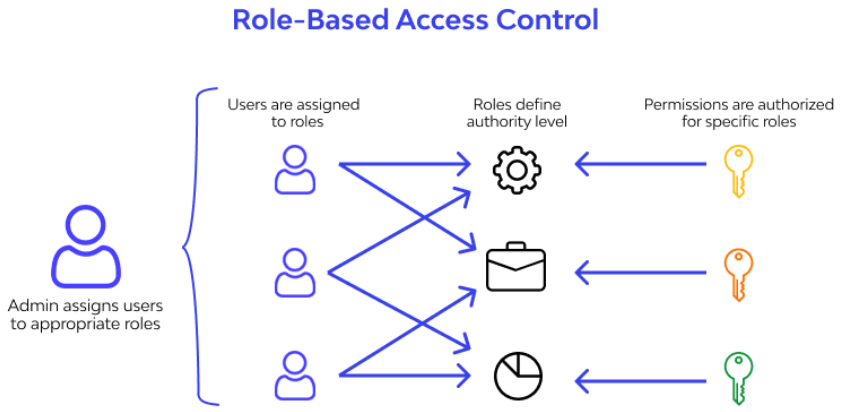


Рисунок 1.2 Опис підходу RBAC

Крім того, існує вразливість підвищення привілеїв, суть якої полягає в отриманні зловмисниками доступу до чутливої інформації та виконання не дозволених за дизайном системи дій завдяки отриманню привілеїв з використанням облікового запису з обмеженим рівнем доступу. Прикладом такої вразливості може слугувати створення облікового запису для адміністратора з допомогою акаунта звичайного користувача без привілеїв.

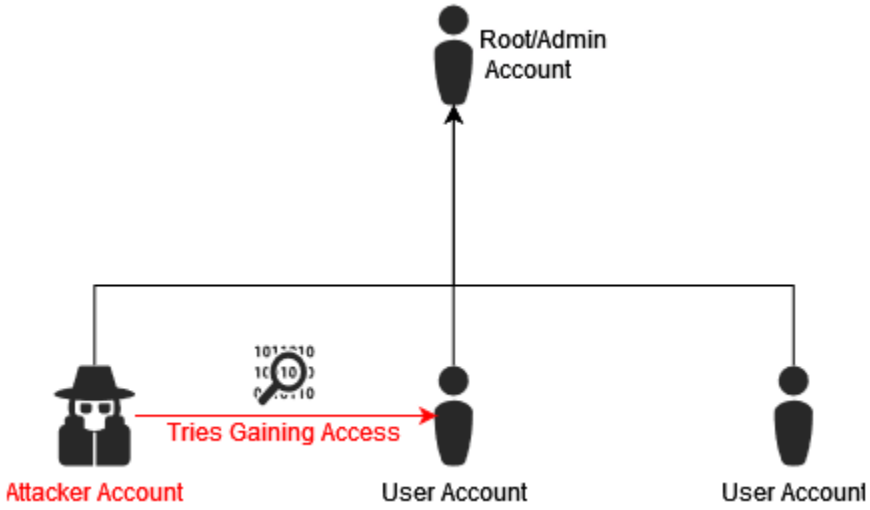


Рисунок 1.3 Схема атаки підвищення привілеїв

Також варто зазначити, що у мікросервісній архітектурі у кожного компонента може бути власна система контролю доступу. Це значно ускладнює управління, може призвести до неузгодженості, підвищує ризик помилкового доступу, може спричинити складнощі у розробці та викликатиме додаткове навантаження на систему для перевірки прав користувача на кожному сервісі. Оскільки мікросервіси взаємодіють між собою через мережу і не залежать від реалізації конкретного сервісу, а лише від його інтерфейсу, тому у великих комерційних проєктах різні сервіси створюються різними командами розробників, часто на різних мовах програмування та своєму стеку технологій для реалізації. Такий підхід робить захист системи значно складнішим та дає зловмисникам простір для маніпуляцій [3].

У застосунку з монолітною архітектурою перевірка безпеки виконується один раз і запит надсилається до відповідного компонента, у мікросервісному ж підході часто наявні декілька перевірок на точці входу кожного мікросервісу. Крім того, при перевірці запитів, може знадобитись підключення до віддаленого сервісу токенів безпеки (STS). Такі постійні перевірки безпеки можуть призвести до затримок і значно погіршити продуктивність системи [4].

1.2 Розгортання системи

З кожним роком програми та сервіси стають все більшими. Керування таким масштабним розгортанням програм з тисячами сервісів було б

надзвичайно складним без автоматизації цього процесу. На допомогу приходить контейнеризація, яка дозволяє упакувати рішення та їх залежності у відокремленні контейнери. Вони забезпечують середовище для запуску програм та переносимість між різними областями використання, що сприяє швидкому та простому розгортанню. Проте, при розгортанні систем варто турбуватись про безпеку. Для належного функціонування контейнери потребують доступу до ресурсів, які потрібно захистити, щоб уникнути несанкціонованого доступу до конфіденційних даних. Однак, якщо контейнери отримують дозволи, вони можуть отримати доступ до чутливої інформації або систем за межами контейнера та змінювати їх, тому важливо обмежити дозволи контейнера лише тим, що мінімально необхідно для роботи системи. Це допомагає уникнути непотрібних ризиків та зменшити потенційні вразливості [4].

Також безпека контейнерів тісно пов’язана з управлінням доступу до віддалених контейнерів. Недостатній захист може створити можливості для зловмисників у отриманні доступу до контейнерів. Крім того, необхідно враховувати ризик replay-атак, при яких зловмисники відтворюють старі метадані разом зі застарілим образами Docker, що можуть містити вразливості [4].

Безпека розгортання мікросервісів повинна розглядатись в контексті системи оркестрування, а не лише як ізольована безпека контейнерів. У таких системах як Kubernetes потрібно потурбуватись про надійне зберігання конфіденційної інформації та контроль доступу [3].

1.3 Мережна комунікація

Безпека мережі є однією з ключових умов для створення якісної мікросервісної архітектури, адже сервіси, з яких складається додаток, взаємодіють між собою через мережу. Мережева безпека передбачає захист мереж та їхніх сервісів від несанкціонованого доступу, модифікації або знищення. Вона також має на меті гарантувати, що мережа працює за призначенням без будь-яких непередбачуваних наслідків або шкідливих

побічних ефектів [2]. Серед найпоширеніших загроз на мережу виділяють man-in-the-middle (MITM), denial-of-service (DoS), різноманітні атаки на сервіс, що відповідає за виявлення інших сервісів, тощо.

Якщо говорити більш детально про атаку MITM, вона являє собою перехоплення зловмисниками конфіденційної інформації та підміну даних, що передаються між сервісами у системі. Серед можливих заходів запобігання – встановлення зв'язку між компонентами лише через зашифровані канали та використання безпечних протоколів, наприклад, HTTPS і SSL [3].

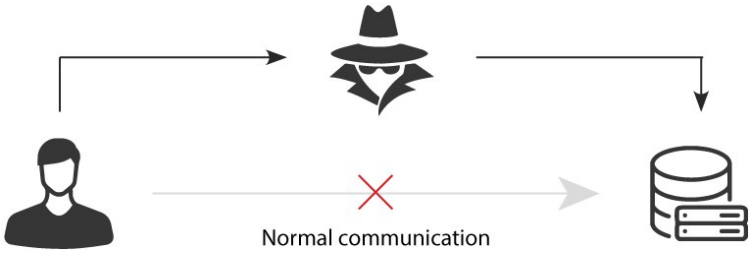


Рисунок 1.4 Схема атаки MITM

Іншим видом атаки на мережу є DoS, що може переповнити ресурси мережі (певний сервіс чи саму мережу), внаслідок чого застосунок перестає бути доступним для звичайних користувачів. Захист від таких атак може бути побудований на різних рівнях: на рівні програми найкращим рішенням буде відхиляти недійсні запити. Багаторівнева архітектура безпеки допомагає спроєктувати кожен рівень з урахуванням різних типів атак і відхилити запити зловмисника на зовнішньому рівні. Брандмауер – один з варіантів захисту від DoS чи DDoS атак. Крім того, всі заходи захисту від цього типу атак не є специфічними для мікросервісів. Для забезпечення захисту всього застосунку будь-яка точка входу, що має доступ до інтернету повинна бути убезпечена від DoS / DDoS-атак [3].

Зловмисники можуть використовувати вразливості в механізмах сервісу, що відповідає за виявлення інших сервісів, щоб перенаправляти трафік на шкідливі сервіси або запускати такі атаки, як DoS, MITM або атаки направлені на підміну сервісу. Крім того, такі атаки можуть бути використані для зламу

інших мікросервісів у мережі, для обходу засобів контролю безпеки та отримання доступу до конфіденційних даних.

Дуже поганою практикою вважається надання користувачу прямого доступу до окремих сервісів, їхніх ендпоїнтів чи портів сервісів. Замість цього рекомендується використовувати шлюз, як точку входу для викликів. Для цього потрібно налаштувати мережу таким чином, щоб мікросервіси приймали лише трафік, який пройшов через шлюз. Також він може діяти у якості Policy Enforcement Point (PEP), що можна застосовувати у всіх службах [3].

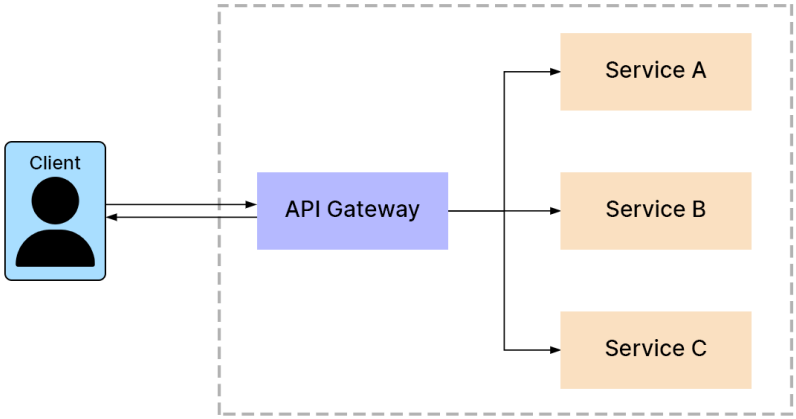


Рисунок 1.5 Схема використання шлюзу API

1.4 Витік чутливої інформації

Загрози пов’язані з витоком чутливої інформації є однією з головних проблем в мікросервісній архітектурі, оскільки мікросервіси покладаються на зв’язок між декількома службами. Конфіденційна інформація (облікові дані користувача, токени аутентифікації тощо) може передаватись через мережу, збільшуючи ризик витоку даних і кібератак. Серед найбільш поширених проблем безпеки пов’язаних з конфіденційною інформацією можна виділити загрози незахищеного зберігання даних, недостатнього шифрування та згадані раніше проблеми з контролем доступу, слабкими механізмами аутентифікації та незахищеною комунікацією сервісів через мережу.

В архітектурі мікросервісів між сервісами немає спільного доступу до ресурсів або є лише до дуже обмеженого набору ресурсів і дані користувача повинні передаватись явно від одного мікросервісу до іншого для перевірки

прав доступу. Тому потрібен механізм для перевірки того, що контекст користувача не був навмисно змінений. Одним з популярних способів обміну даними користувача між сервісами є використання JSON-токенів (JWT) , що допомагає передавати набір атрибутів користувача у закодованому форматі, що захищений від підробки секретним ключем [4].

Мікросервіси можуть зберігати чутливу інформацію у різних місцях: логи, кеш-пам’ять та, звісно, бази даних. До того ж, кожен окремий сервіс може мати власне сховище, що значно ускладнює керування даними та може призвести до помилок. Ці сховища можуть бути вразливими до різного роду атак. Наприклад, SQL-ін’єкції, XSS-атаки та атаки переповнення буферу. Вони можуть призвести до розкриття, зміни та втрати конфіденційної інформації [3].

Шифрування чутливої інформації є критично важливим аспектом безпеки даних, проте його може бути важко ефективно налаштувати в мікросервісному підході, оскільки з кількома сервісами та сховищами даних може бути складно забезпечити послідовне та надійне шифрування в мережі.

Крім того, особливу увагу необхідно приділити безпеці при зберіганні паролів користувачів. Зберігання паролів у відкритому вигляді є неприпустимим, вони повинні зберігатись у захищеному форматі. Одним з варіантів є зберігання паролів у вигляді хешу, для уникнення загрози його розкриття. Також досить часто використовують salt, що додається до пароль перед його хешуванням для ускладнення/унеможливлення підбору з використанням райдужних таблиць, словникових атак тощо. Для перевірки правильності такого паролю система здійснює хешування введеного паролю разом з salt й отриманий хеш порівнюється з тим, що зберігається у базі.

Ще одним аспектом, що вартий уваги, є помилкова конфігурація, що може призвести до виникнення вразливості, таких як неконтрольований доступ до сервісів та даних, розкриття чутливої інформації, яке може виникнути через відсутність шифрування критичних даних. Помилки конфігурації часто виникають через людський фактор і їх може бути дуже важко виявити через велику кількість сервісів з власними налаштуваннями [3].

1.5 Моніторинг

Логування – необхідна частина системи, оскільки на його основі можна робити висновки про внутрішній стан програми. Його відсутність є проблемою безпеки.

Логом може бути будь-яка подія, яку ви реєструєте і яка відповідає певному сервісу. Наприклад, висока кількість невірних запитів на годину може вказувати на те, що систему атакують або що захист першого рівня слабкий. На основі логів також може будуватись система захисту. Наприклад, якщо показник кількості спроб доступу певного мікросервісу виходить за встановленого порогового значення, система може викликати попередження та навіть заблокувати його на певний час для виявлення причин.

Трасування допомагає відстежувати запит від моменту, коли він потрапляє до системи, до моменту, коли він її залишає. Процес трасування значно ускладнюється у мікросервісній архітектурі порівняно з монолітом, оскільки запит може потрапити в систему через один мікросервіс і пройти через декілька інших перш ніж покинути систему [4].

РОЗДІЛ 2. РЕКОМЕНДАЦІЇ ЩОДО БЕЗПЕКИ У МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

2.1 Загальні механізми безпеки

Безпека передбачає створення рівнів доступу та реалізацію принципу найменших привілеїв, коли користувачам надається лише необхідний мінімальний доступ до мережі та привілеї для виконання усіх передбачених дій. Для досягнення цього можна розділити сервіси на дві зони: публічні та приватні. Публічні повинні вирішувати конкретні завдання і знаходитись за шлюзом з власною службою аутентифікації. Приватна зона ж у свою чергу повинна бути захищеною та доступною тільки через певний порт, а також мати власний шлюз і службу аутентифікації. Крім того, необхідно, щоб усі дані програми знаходились у межах приватної зони і були доступними лише для мікросервісів цієї зони. Цей підхід вирішує багато проблем, пов’язаних з контролем доступу та допомагає захистити конфіденційну інформацію [3].

Ще одним не менш важливим заходом безпеки є мінімізація кількості відкритих мережевих портів, що зменшить поверхню атаки, а відповідно, і ризик порушень безпеки у архітектурі мікросервісів. Важливо відкривати вхідний та (або) вихідний доступ лише до тих портів, які необхідні для належного функціонування сервісу. Це стосується моніторингу та агрегації логів – захищати потрібно не лише порти вхідного доступу, а й вихідного, оскільки вони також можуть бути використані для витоку даних, якщо сервіс буде скомпрометовано. Тому важливо обмежити доступ до мережевих портів, як у публічних, так і у приватних зонах API, щоб забезпечити належний захист мікросервісів [3].

Шлюз API – це інтерфейс REST API, який забезпечує центральну точку доступу до групи мікросервісів та/або визначених сторонніх API. Шлюз API є точкою входу до мікросервісів, який перевіряє всі вхідні повідомлення на безпеку. Ключова роль шлюзу API у розгортанні мікросервісів полягає у наданні доступу до вибраного набору мікросервісів у вигляді API. Також на нього покладається роль перевірки токенів безпеки, які постачаються з кожним

запитом до API. Не обов'язково надавати усі сервіси через шлюз - деякі мікросервіси можуть використовуватися лише всередині системи. Крім того, API-шлюз може впроваджувати корпоративні політики контролю доступу, проте більш тонкі налаштування доступу застосовуються безпосередньо мікросервісом. На нього також часто покладаються додаткові функції: аутентифікація і авторизація (найчастіше за допомогою Auth 2.0), захист від DoS атак, ін’єкцій коду тощо та моніторинг вхідного та вихідного трафіку [3].

Виявлення сервісів є критично важливим для мікросервісної архітектури. Це дозволяє швидко масштабувати по горизонталі кількість екземплярів сервісів, що працюють у середовищі. Крім того, воно допомагає підвищити стійкість додатків до відмов. У разі якщо мікросервіс стає недоступним або несправним, більшість механізмів виявлення сервісів видаляють цей екземпляр зі свого внутрішнього списку доступних сервісів.

2.2 Аутентифікація та авторизація

Потрібно потурбуватись про те, як аутентифікувати систему та користувача, що намагаються отримати доступ до мікросервіса. Для цього можна, наприклад, використовувати OAuth 2.0. OAuth 2.0 – протокол безпеки на основі токенів, який описує шаблони для надання авторизації, але не визначає, як насправді виконувати аутентифікацію. Замість цього він дозволяє користувачам аутентифікуватись за допомогою стороннього сервісу, який виконує роль постачальника ідентифікаційних даних. Якщо користувач успішно пройшов аутентифікацію, він отримує токен, який повинен надсилатись з кожним запитом. Цей токен потім може бути підтверджений службою аутентифікації. OAuth 2.0 використовується для делегованого контролю доступу, який є рекомендованим підходом для захисту, коли одна система хоче отримати доступ до API від імені іншої системи чи користувача без необхідності надавати свої облікові лані кожному сервісу, що обробляє його запит. Він широко розповсюджений і використовується, наприклад, Facebook для захисту своїх API. OAuth 2.0 підтримує кілька типів грантів – схем аутентифікації. Кожен тип гранту визначає протокол, який вказує, як

клієнтський додаток отримає токен доступу до API. Загалом існує чотири типи грантів OAuth 2.0: пароль, облікові дані клієнта, код авторизації і неявні. OpenID Connect (OIDC) – це надбудова над протоколом OAuth 2.0, яка забезпечує аутентифікацію та надає інформацію про профіль користувача, який увійшов до системи [6].

Keycloak – це рішення для управління ідентифікацією та доступом з відкритим кодом, основною метою якого є полегшення захисту сервісів. Він централізує аутентифікацію та використовує підхід єдиного входу для неї. Також він підтримує двофакторну аутентифікацію для більшої надійності. Keycloak сервер перевіряє облікові дані користувачів, які намагаються отримати доступ, і якщо вони дійсні, надає токен аутентифікації, який можна передавати між сервісами. Самі ж захищені ресурси можуть зв’язатись з сервером, щоб перевірити валідність токену і отримати роль користувача у системі [3].

Крім того, можна запросити ім’я користувача та пароль разом з іншим фактором для багатофакторної аутентифікації (MFA). Проте, звісно, не кожен додаток потребує MFA, це залежить від того, наскільки чутливими є дані або важливими бізнес активи. Найпопулярнішою формою MFA є одноразовий пароль (OTP), надісланий через SMS чи електронну пошту. Існують і інші значно надійніші форми MFA, що включають біометричні дані, сертифікати та швидку ідентифікацію в Інтернеті [4].

2.3 Мережевий захист

Варто також потурбуватись про захист даних під час зберігання та передачі. Використання протоколу TLS є одним з найпопулярніших способів захисту даних для забезпечення конфіденційності під час передачі. HTTPS також використовує TLS, а це означає, що тільки мікросервіс для якого призначена інформація зможе переглядати дані у відкритому вигляді. Захист даних під час зберігання ж в основному складається з шифрування інформації, щоб захистити її від зловмисників, які отримують прямий доступ до системи. Крім того, більшість систем керування базами даних надають функцію

автоматичного шифрування. Ще одним варіантом є шифрування на рівні програми, при якому програма сама шифрує дані перед передачею їх до кінцевого місця зберігання. Проте шифрування є ресурсоємною операцією і може сповільнити систему, тому варто шифрувати лише конфіденційні дані, які цього дійсно потребують [7].

Модель безпеки, яка розробляється для захисту взаємодії між сервісами повинна враховувати як відбувається фактична комунікація між мікросервісами: синхронно чи асинхронно. У більшості випадків синхронний зв'язок відбувається через HTTP або HTTPS. Асинхронний зв'язок може відбуватись через будь-яку систему обміну повідомленнями RabbitMQ, Kafka, ActiveMQ чи Amazon Simple Queue Service (SQS).

У архітектурі мікросервісів виділяють три основні способи захисту зв'язку між службами: підхід довіри до мережі, mTLS і JWT. Підхід довіри до мережі являє собою модель, у якій не забезпечується безпека при передачі даних між сервісами. Ця модель повністю покладається на безпеку на мережевому рівні, з цієї причини вона вважається антипатерном. Діаметрально протилежним до неї є мережевий підхід нульової довіри. Він передбачає, що кожен запит повинен бути аутентифікований і авторизований на кожному вузлі, перш ніж буде прийнятий для подальшої обробки [5].

Ще одним популярним способом захисту є взаємний TLS. Безпека на транспортному рівні захищає комунікації між двома сторонами для забезпечення конфіденційності і цілісності. При цьому підході кожен мікросервіс повинен мати пару публічних/приватних ключів і використовувати її для аутентифікації користувача. TLS забезпечує конфіденційність та цілісність даних, що передаються, і допомагає клієнту ідентифікувати сервіс. Проте за допомогою TLS (одностороннього) мікросервіс одержувач не може перевірити ідентичність клієнтського сервісу. У свою ж чергу mTLS дозволяє кожному сервісу в комунікації ідентифікувати інші [4].

JSON веб-токен – третій підхід для захисту зв'язку між сервісами. JWT вирішує дві основні проблеми безпеки мікросервісів: захист міжсервісних

комунікацій і передача контексту кінцевого користувача через мікросервіси. Як і у випадку з mTLS, при аутентифікації на основі JWT кожен мікросервіс повинен мати власну пару ключів, і відповідний приватний ключ використовується для підписання JWT. Коли ідентичність мікросервісу не має значення, а ідентичність кінцевого користувача має, слід надавати перевагу використанню JWT, а не mTLS. Проте у переважній більшості випадків аутентифікація на основі JWT працює через TLS для побудови другого рівня захисту. Таким чином JWT забезпечує аутентифікацію, а TLS забезпечує конфіденційність та цілісність даних. Використання окремого JWT для кожної взаємодії між мікросервісами є більш безпечним підходом, ніж використання JWT зі спільним ключем для всіх мікросервісів [4].

2.4 Kafka

У мікросервісній архітектурі сервіс, який приймає запити від клієнтської програми, стає тригером для решти сервісів, що беруть участь у потоці виконання. Він може ініціювати, як синхронні так і асинхронні події для інших мікросервісів для запуску їх виконання. Мікросервіс, який безпосередньо підключений до іншого сервісу через ієрархію залежностей призводить до неефективності масштабування. Для асинхронних подій можна створити реактивні мікросервіси, які здатні реагувати на події, що відбуваються у системі, що зменшує залежність між мікросервісами. Одним з поширених інструментів для створення асинхронної комунікації є Apache Kafka – розподілена потокова платформа, яка працює подібно до черги повідомлень. У мікросервісній архітектурі Kafka використовується як посередник для доставки подій від мікросервісів-джерел до мікросервісів-цілей асинхронно. За допомогою реактивної моделі усуваються прямі зв’язки між екземплярами мікросервісів та підвищується операційна ефективність. Ще ця модель скорочує час відгуку мікросервісів для користувачів і забезпечує зручну платформу для впровадження нової функціональності в систему. Для шифрування повідомлень між мікросервісами і Kafka використовується TLS. Для роботи Kafka потрібен сервер ZooKeeper. Apache ZooKeeper – це централізований сервіс, який надає

широкі можливості для управління та запуску розподілених систем. Деякі з них включають розподілену синхронізацію, служби групування, іменування тощо. Kafka використовує mTLS, щоб контролювати, яким мікросервісам дозволено підключатися до неї, а також ідентифікувати клієнтів, що намагаються це зробити [4].

2.5 Захист чутливої інформації

Зберігати конфігурації програми (особливо конфігурації, що залежить від середовища) найкраще повністю окремо від інших сервісів. Конфігурація програми не повинна розгортатись разом з екземпляром служби, адже у такому випадку при необхідності зміни конфігураційного файлу певного мікросервісу, який був кілька разів масштабований, потрібно буде повторно розгортати кожен екземпляр, а це не тільки затратно по часу, але й може призвести до проблем з конфігурацією між сервісами. Замість цього інформація повинна передаватись або як змінні оточення, або зчитуватись з централізованого сховища під час запуску сервісу. Проте зберігання конфігурації у файловій системі може бути непрактичним для хмарних додатків. Одним з підходів, який користується популярністю, є використання серверу конфігурацій разом з репозиторієм контролю версій, наприклад, Spring Boot Configuration Server з Git [3]. Одним з інструментів, який дозволяє отримати безпечний доступ до секретів – будь-якої інформації, доступ до якої необхідно обмежити, є HashiCorp Vault. Для його налаштування у сервісі Spring Config, потрібно додати профіль Vault. Цей профіль забезпечує інтеграцію з Vault і дозволяє безпечно зберігати властивості мікросервісів. Секрети Vault можна створювати, як з допомогою інтерфейсу так і через CLI команди [7].

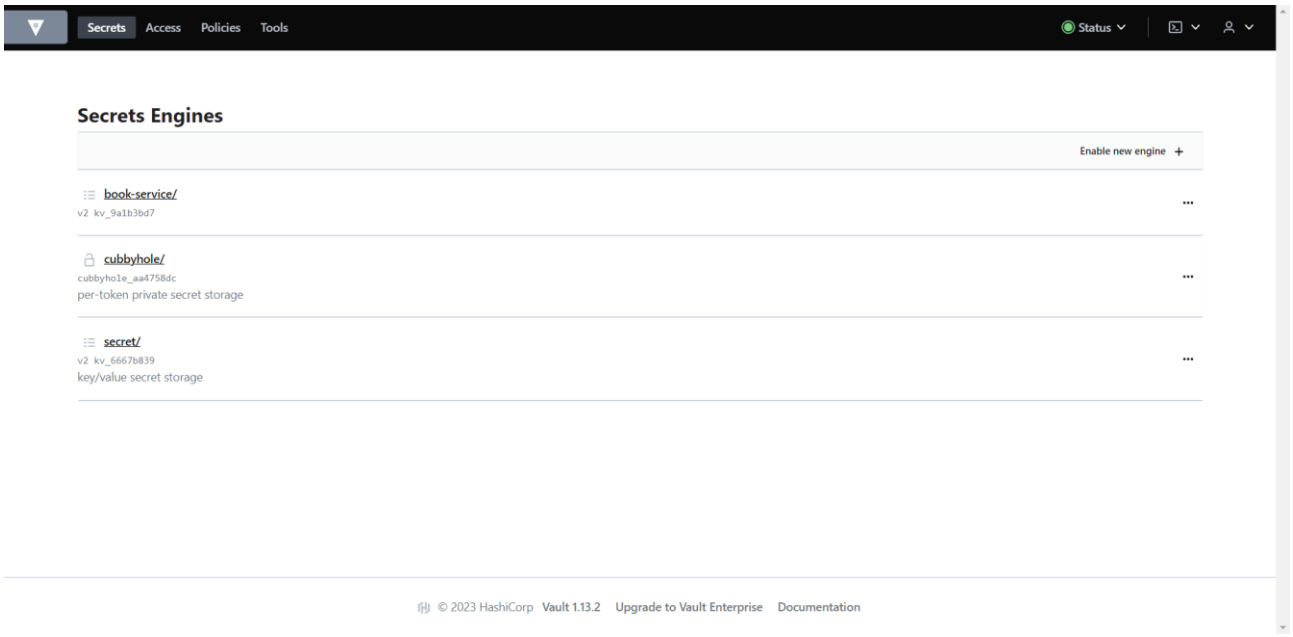


Рисунок 2.1 Інтерфейс управління секретами Vault

Шифрування конфіденційних конфігураційних даних є важливим заходом безпеки. Spring Cloud Config підтримує використання як симетричного – з спільним ключем, так і асиметричного (з публічними/приватними ключами) шифрування. Асиметричне шифрування є більш безпечним, ніж симетричне, оскільки використовує більш складні алгоритми. У сервері конфігурації Spring Cloud Config Server симетричний ключ шифрування – це вибраний рядок символів, який можна задати у файлі конфігурації або передавати сервісу через змінну оточення ENCRYPT_KEY. Хорошою практикою є використання окремого ключа шифрування для кожного середовища, в якому система розгортається, а ключ генерується випадково. Крім того, у жодному разі можна не записувати ключі шифрування у Docker-файли. Замість цього краще посилатись на ENCRYPT_KEY як на змінну оточення всередині Docker-файлу [3].

Назва проекту	Опис	Характеристики
Eureka	Використовується як для виявлення сервісів, так і для управління чутливою інформацією. Розроблено компанією Netflix	Розподілене сховище ключів Гнучке, але відносно важке для налаштування
Zookeeper	Пропонує можливості розподіленого блокування. Від Apache	Найстаріше, найскладніше у використанні
Spring Cloud Configuration Server	Пропонує загальне управління конфігурацією з різними бекендами	Проста інтеграція з сервісами Spring Може використовувати декілька бекендів для зберігання даних конфігурації, включаючи спільну файлову систему, Eureka або Git

Таблиця 2.1 Основні інструменти для управління конфігураціями

2.6 Моніторинг

Мікросервісні застосунки у порівняні з монолітними є дуже розрізненими. Так у монолітній системі ймовірність того, що виклик функції не відбудеться через зовнішні фактори є дуже низькою, оскільки функції при такому підході знаходяться у одному процесі. Якщо у ньому відбудеться збій, застосунок вийде з ладу як єдине ціле, що зменшує ймовірність часткових збоїв. У мікросервісній архітектурі ж у свою чергу запит, зроблений клієнтом, може пройти через декілька мікросервісів і кожен з них може працювати з високим завантаженням, бути недоступним (наприклад, якщо процес завершився аварійно або був зупинений), мати затримки з доступом до сторонніх сервісів, баз даних тощо. Саме тому моніторинг є критично важливим фактором для мікросервісної архітектури.

Метрики – це набір значень даних, які реєструються протягом певного періоду часу. Це здебільшого числові значення, які постійно фіксуються у вигляді мінімуму, максимуму та середнього значення. Зазвичай метрики використовуються для вимірювання ефективності програмного процесу,

наприклад, використання пам’яті певним процесом, середнє навантаження, кількість запитів на годину. Такі системи, як Kubernetes відстежують ці показники для виконання автоматичного відновлення та автоматизації масштабування. Серед недоліків метрик виділяють те, що вони корисні лише для моніторингу певного мікросервісу на основі обмеженого набору атрибутів [4].

Логування – це процес запису певної події з позначкою часу. Воно потрібне для виявлення помилок у роботі, які не можуть бути автоматизовано усунені і потребують втручання розробника та моніторингу важливих подій у системі.

Трасування – це процес запису послідовних пов’язаних розподілених подій. Кожний запис має унікальний ідентифікатор, який дозволяє співставити дані одного запиту, який охоплює різні компоненти системи. Він також може містити важливу інформацію, таку як часові мітки, інформацію про затримку та будь-які інші дані, які можуть бути корисними для виявлення джерела запиту або для усунення несправності тощо [4].

Є безліч готових рішень для вирішення завдання моніторингу: Micrometer, Zipkin, Prometheus, Kibana та Graphana. Micrometer – бібліотека метрик, яка дозволяє отримувати метрики JVM, такі як пули пам’яті, збір сміття тощо. Prometheus – це система моніторингу та оповіщення, яка зберігає усі дані у вигляд часових рядів. Graphana – платформа для аналізу метрик, яка дозволяє запитувати, візуалізувати та розгорнути дані незалежно від того, де вони зберігаються. Ще однією популярною технологією для візуалізації статистики, пов’язаної з моніторингом є Kibana. Zipkin - одне з найпопулярніших рішень для трасування [7].

2.7 Розгортання

Пакування, розповсюдження та тестування мікросервісів у різних середовищах перед запуском в ефективний, менш схильний до помилок спосіб є дуже важливим. Docker - один з найпопулярніших інструментів для цього. Важливо, щоб жодні конфіденційні дані не були вбудовані в образи Docker,

найкраще їх винести назовні – програмні рішення для оркестрування, такі як Kubernetes і Docker Swarm, надають способи керування конфіденційними даними в контейнерному середовищі. Безпека розгортання мікросервісів повинна розглядатись в контексті системи оркестрування контейнерів, а не лише як ізольована безпека контейнерів [3]. Для підписання і перевірки образів Docker використовується Docker Content Trust (DCT). DCT переконується, що всі відповідні образи, які публікуються у реєстрі Docker, підписані і що з реєстру Docker буде витягнуто лише підписані образи. Зловмисник може виконати replay атаку, відтворюючи раніше дійсні метадані довіри разом зі старим образом Docker. Він може містити вразливості, які виправлені в останньому опублікованому випуску. Якщо зловмиснику вдасться відтворити старі файли метаданих у системі, власник не матиме доступу до найновіших. DCT запровадив файл метаданих з міткою часу, щоб виправити цю проблему. Кожного разу, коли видавець публікує образ Docker, DCT створює файл з оновленою міткою часу і версією. Ключ мітки часу підписує цей файл, який містить хеш файлу snapshot.json. А сам snapshot має хеш оновленого (або нового) образу Docker. Щоразу, коли образ Docker оновлюється на стороні клієнта, DCT завантажує найновіший файл timestamp.json з відповідного сховища. Потім він перевірить підпис завантаженого файлу і чи версія в завантаженому файлі мітки часу більша за версію у поточному файлі мітки часу. Якщо завантажений файл, який відтворюється зловмисником, має старішу версію, DCT перерве операцію оновлення і захистить систему від атаки такого типу. За замовчуванням контейнери запускаються від імені root користувача, проте з точки зору безпеки це не є правильним, оскільки надає значно більше привілеїв, що може зашкодити. Одним з підходів це змінити – визначити користувача, від імені якого запускається контейнер, у самому Docker-файлі або передавати його як аргумент команді запуску докера. Інший підхід полягає у використанні можливостей для обмеження того, що користувач може робити всередині контейнера. Docker Bench for Security перевіряє розгортання Docker на відповідність найкращим практикам, визначеним Center for Internet Security в

Docker Community Edition Benchmark, а потім перевіряє саме середовище Docker. Крім нього, для аналізу вразливостей у контейнерах використовуються Anchore і Clair [4].

Kubernetes є найпопулярнішою платформою для оркестрування контейнерів. Програмне забезпечення для оркестрування контейнерів дозволяє розгортати, керувати та масштабувати контейнери у високорозподіленому середовищі. Kubernetes використовує ConfigMaps для екстерналізації конфігурації з коду програми, яка працює у контейнері, але це не є правильним способом екстерналізації чутливих даних у Kubernetes. Безпечнішим способом зберігання конфіденційної інформації у середовищі Kubernetes є використання Secrets, Kubernetes зберігає значення секрету у розподіленому сховищі у форматі ключ-значення у зашифрованому вигляді. Kubernetes надсилає секрети лише тим подам, які їх використовують, і навіть у таких випадках секрети ніколи не записуються на диск, а лише зберігаються у пам’яті [4]. За замовчуванням, кожен под монтується з токеном-секретом у вигляді JWT. Контейнер може використовувати цей стандартний токен-секрет для зв'язку з сервером API Kubernetes. Kubernetes має два типи облікових записів: користувацькі та службові. Облікові записи користувачів не створюються і не управляються Kubernetes, на відміну від службових. За замовчуванням, кожен под пов’язано зі службовим обліковим записом (з іменем default), і кожен обліковий запис служби має власний секретний токен. Рекомендується завжди мати різні службові облікові записи для різних подів (або їх групи). Це одна з найкращих практик безпеки, якої слід дотримуватись при розгортанні Kubernetes. Також рекомендується не надавати секретний токен до подів, якому не потрібен доступ до сервера API. Для забезпечення контролю доступу на сервері API використовується Roles / ClusterRole та RoleBindings / ClusterRoleBindings [9].

2.8 OWASP

Open Web Application Security Project (OWASP) – це проект з відкритим вихідним кодом, спрямований на запобігання розгортання потенційно

вразливих API. За версією OWASP найпоширенішими вразливостями безпеки виступають: порушена авторизація на рівні об’єктів, порушена аутентифікація, надмірна відкритість даних, нестача ресурсів та обмеження швидкості, порушення авторизації на рівні функцій, mass assignment, неправильна конфігурація безпеки, ін’єкції, недостатнє логування та моніторинг [8]. Порушення авторизації на рівні об’єктів – це вразливість, яка виникає при використанні послідовних ідентифікаторів для отримання інформації з API. Щоб зменшити загрозу, потрібно або не передавати ідентифікатор користувача у запиті або використовувати випадковий ідентифікатор, який не можливо вгадати, наприклад Global Unique Identifier (GUID) об’єкта. Порушена аутентифікація – це вразливість, яка виникає, коли схема аутентифікації API недостатньо надійна або реалізована неправильно. Ще однією загрозою є відкритість даних. API повинні повертати лише ті дані, які є релевантними та необхідними для їхніх користувачів. Також програмні системи або веб-сайти не повинні розкривати технології або їх версії, на яких вони працюють. Ця рекомендація дуже часто не дотримується розробниками, що дозволяє зловмисникам знаходити вразливості використаних технологій і атакувати систему, використовуючи цю інформацію. Часто надмірне розкриття інформації відбувається при обробці помилок, яка забезпечує повне відстеження стеку у відповіді API на внутрішню помилку. Для цього використовують екранування винятків – замість того, щоб передавати помилку клієнтській програмі, потрібно екранувати помилку її кодом. Проблеми нестачі ресурсів виникають, якщо API не накладають обмеження на кількість запитів, які обслуговуються за певний час, а також не обмежується кількість даних, що повертається. Це може призвести до того, що у зловмисників з’являється можливість здійснити DoS/DDoS-атаки, які роблять систему недоступною. Запобігання таким атакам зазвичай здійснюється шляхом налаштування максимальної кількості запитів, які можна обробити. Mass assignment – це вразливість, яка виникає, якщо API сліпо зв’язується з JSON-об’єктами, отриманими від клієнтів, не вибираючи атрибути, з якими вони зв’язуються.

Якщо API зберігає дані, наприклад, ролі користувачів, зчитуючи їх з JSON-повідомлення, будь-хто, хто має права на додавання або зміну користувачів, може призначати ролі. Щоб уникнути таких помилок, API повинно вибирати поля з вихідного повідомлення, щоб призначати своїм сутностям. Ін’єкції виникають, коли API приймають дані і передають їх інтерпретаторам для виконання як частину запиту або команди без додаткових обробок. Щоб запобігти таким типам атак, дані, що вводяться користувачем, завжди слід перевіряти та обробляти [8].

Зазвичай при збільшенні кількості запитів, які потрібно обробити, виникає необхідність масштабувати компоненти додатку. Масштабування має дві основні моделі: вертикальне і горизонтальне. Вертикальне масштабування збільшує обчислювальну потужність, на якій працює програмне забезпеченням. Горизонтальне ж полягає у додаванні віртуальних машин (або комп’ютерів) до пулу ресурсів, на яких працює програмне забезпечення, що дозволяє виконувати роботу паралельно. Проте кожна модель потребує ресурсів. Це призводить до того, що потрібно обмежувати кількість запитів, що обслуговуються системою. Це обмеження також може допомогти захистити систему від DoS та DDoS-атак.

РОЗДІЛ 3. СТВОРЕННЯ МІКРОСЕРВІСНОГО ЗАСТОСУНКУ

3.1 Опис предметної області

Створено мікросервісний застосунок із замовлення книг у мережі бібліотек. Основна мета сервісу – забезпечити платформу, на якій клієнтам буде доступний такий функціонал: перегляд наявних книжок, вибір і замовлення книг у мережі бібліотек. Кожна книга може мати ISBN, назву, опис, ціну, дату видання, авторів, видавництво.

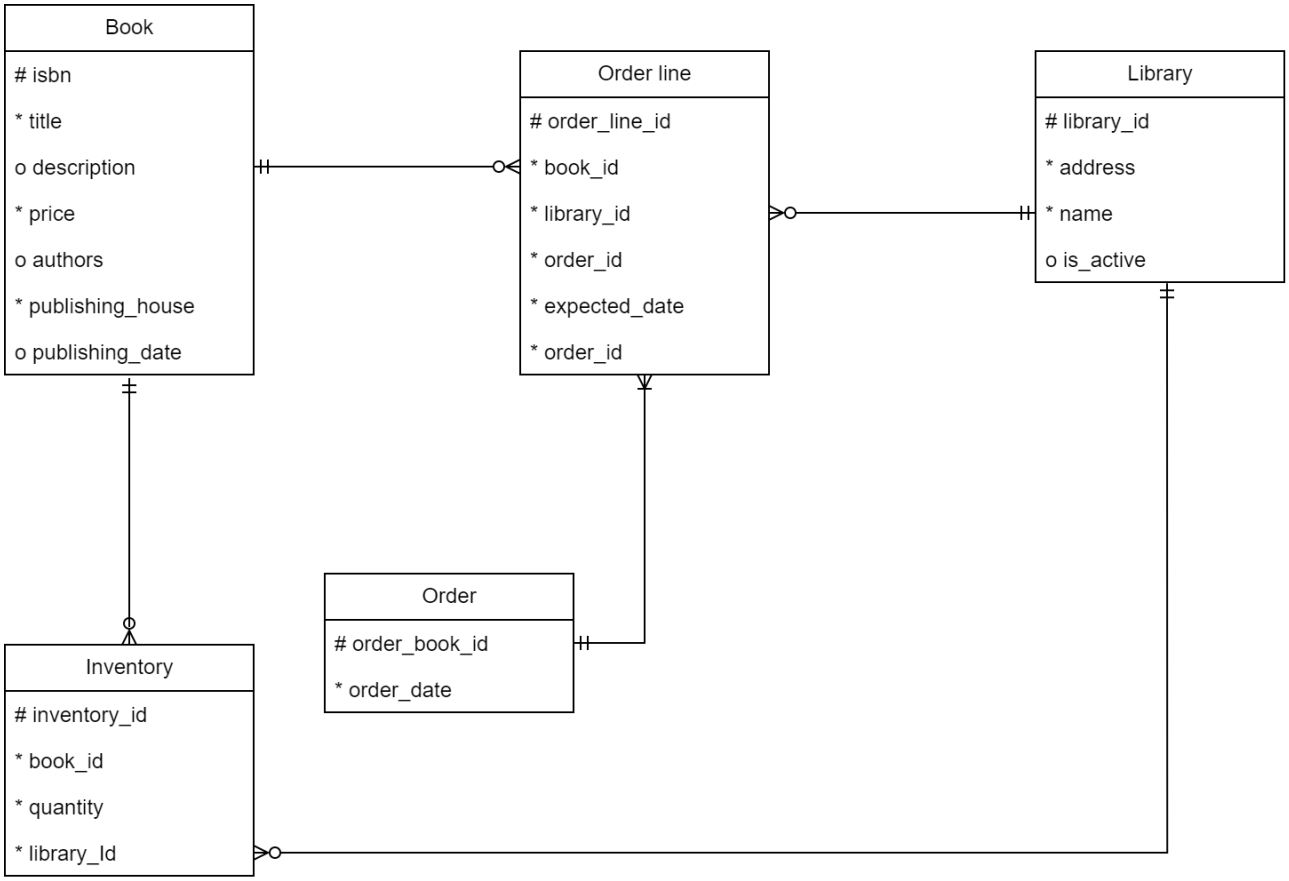


Рисунок 3.1 ER-модель бази даних

Сервіс повинен бути здатним обробляти велику кількість одночасних користувачів і забезпечувати надійний і безпечний процес транзакцій. Ця предметна область яскраво демонструє усі переваги мікросервісної архітектури і слугує чудовим прикладом для імплементації різних підходів до безпеки.

3.2 Інструменти розробки

Для розробки системи була вибрана мова програмування Java. Її вибір зумовлений великою кількістю доступних фреймворків і бібліотек, що значно

полегшують розробку. При створенні системи використовувались бібліотеки Spring Boot та Spring Cloud.

Spring Boot було обрану з огляду на такі переваги:

- вбудований Tomcat сервер для простого розгортання програми;
- екстерналізація конфігурації;
- простота розробки та наявність Spring Security, що полегшує захист програми, тощо. [3]

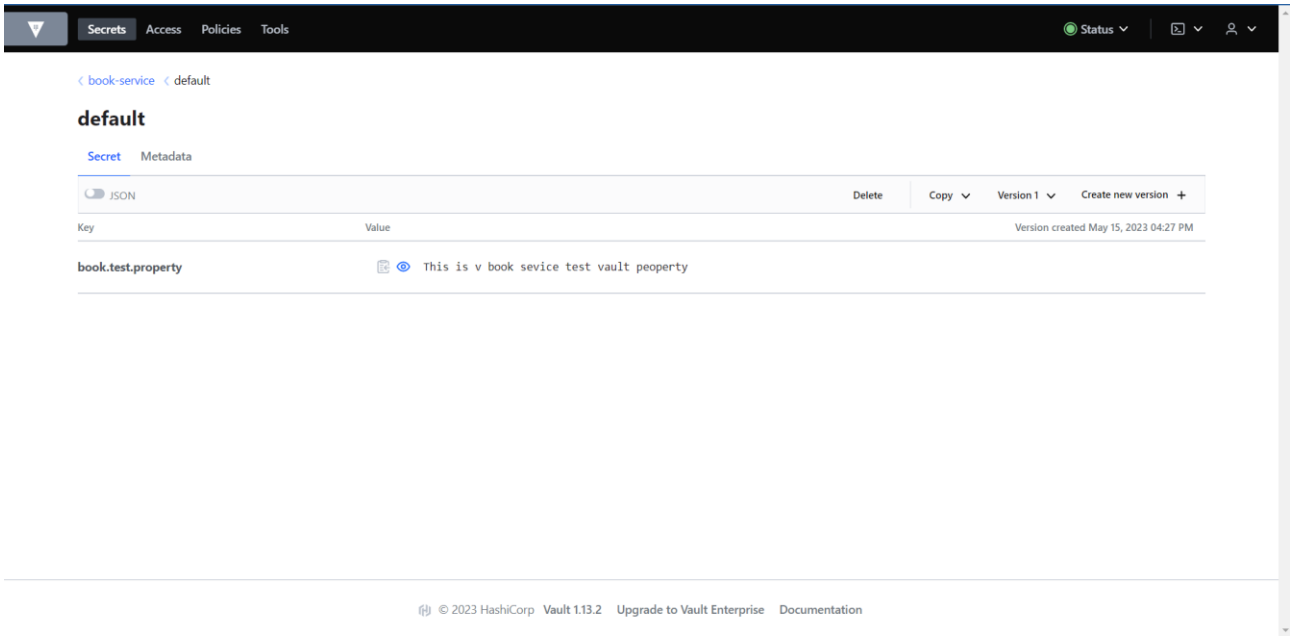
Бібліотека Spring Cloud спрощує розробку та розгортання мікросервісів у приватній та публічних хмарах. Він допомагає в управлінні конфігурацією, спрощує знаходження сервісів, надає функціонал автоматичних вимикачів та інтелектуальної маршрутизації.

Крім цього, використовувався бібліотека Spring Data для забезпечення інтеграції зі сховищами даних: для зберігання даних використовувалась база даних PostgreSQL (розгорнута в Docker).

Для написання спрощеної аутентифікації та авторизації, а також для автоматизованої підтримки запобігання поширених атак на безпеку, використано Spring Security. Це є одним з варіантів вирішення проблеми з контролем доступу.

Були використані Spring Boot Actuator та стек EKL, які надають ряд функцій для моніторингу, та Spring Boot Config Server, який надає загальне керування конфігураціями, дає можливість легко шифрувати конфіденційну

інформацію та містить зручну інтеграцію з Vault для зберігання секретів.



Таблиця 3.2 Приклад секрету Vault

3.3 Загальний опис

Система являє собою мікросервісний застосунок, написаний з використанням Spring Boot, Spring Cloud та інших корисних і сучасних технологій.

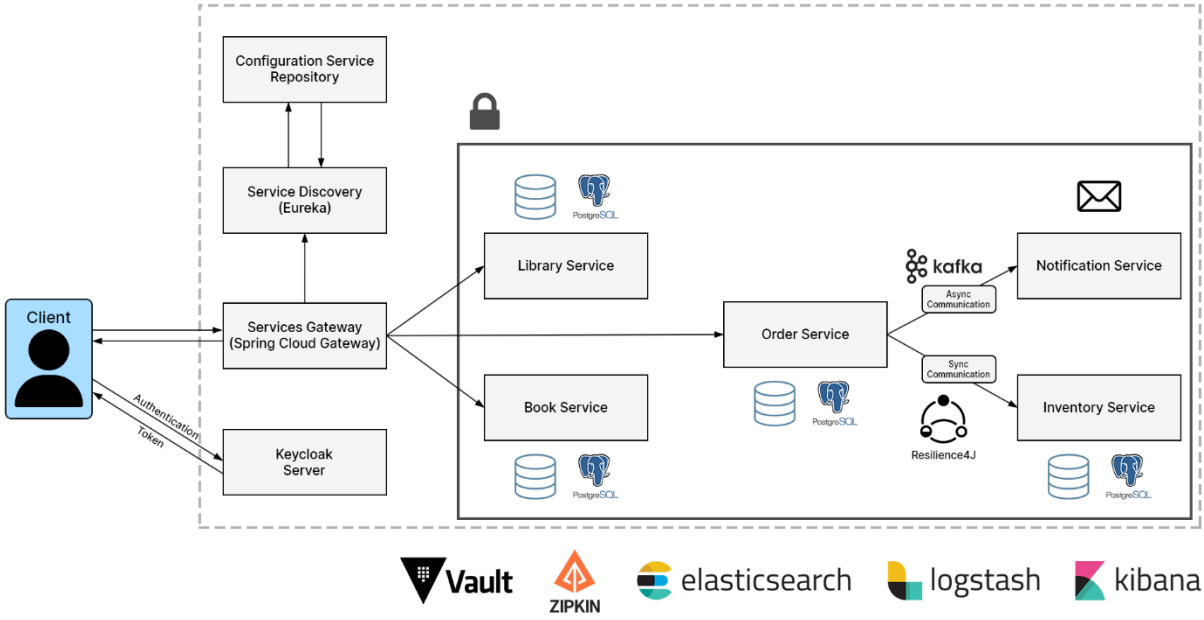


Рисунок 3.3 Архітектура розробленого застосунку

Програма містить декілька взаємопов’язаних сервісів: сервіс бібліотек, книг, складу, замовлень та сповіщень. Сервіс сповіщень отримує асинхронні виклики від сервісу замовлень (реалізовано за допомогою подій Kafka) для відправлення сповіщень по успішне замовлення. Сервіс складу отримує синхронні виклики від сервісу замовлень (з використанням Resilience4J) і використовується для перевірки наявності книг. Роль каталогів бібліотек, які знаходяться у мережі, і книг відіграють однойменні сервіси. Кожен з цих сервісів має доступ до PostgresSQL бази даних та використовує тришарову архітектуру з класами Service, Repository і Controller. Крім цього, застосунок містить Keycloak сервер, який потрібен для отримання токену аутентифікації.

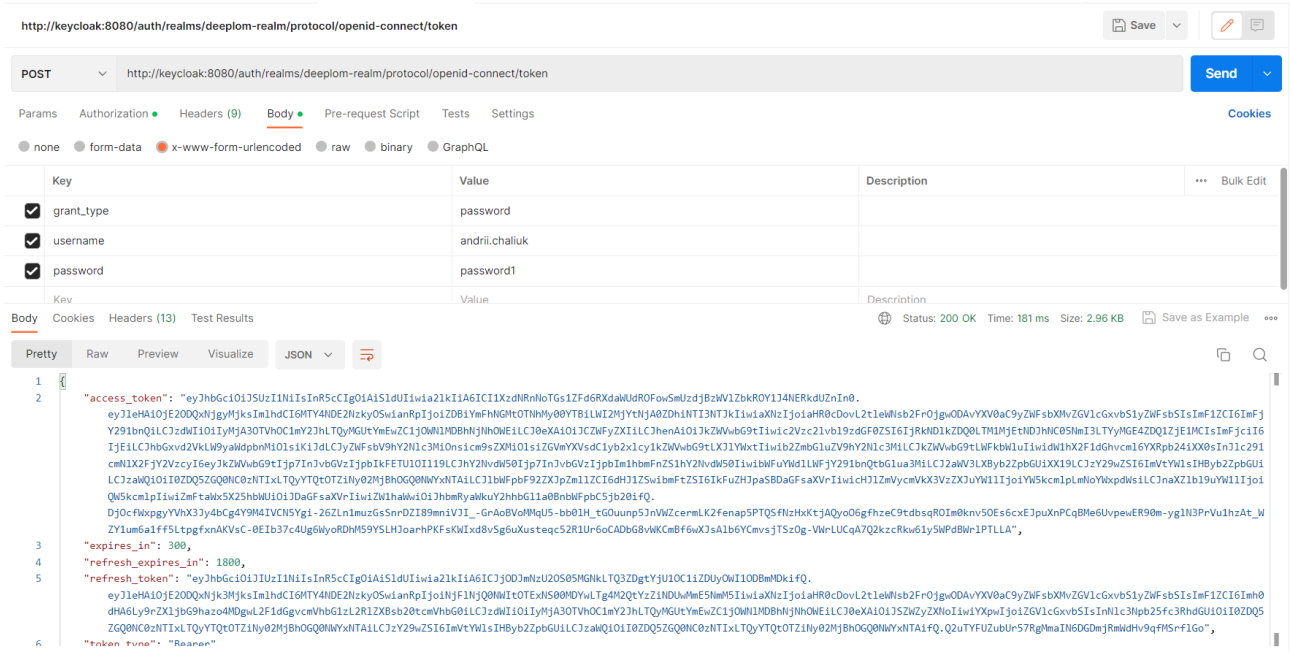


Рисунок 3.4 Отримання токену з допомогою Keycloak

Keycloak дозволяє створювати ролі, облікові записи користувачів та управляти ними. Застосунок має дві ролі: ADMIN (має права перегляду, створення, видалення та оновлення інформації) і USER (має лише права перегляду та

оформлення замовлення).

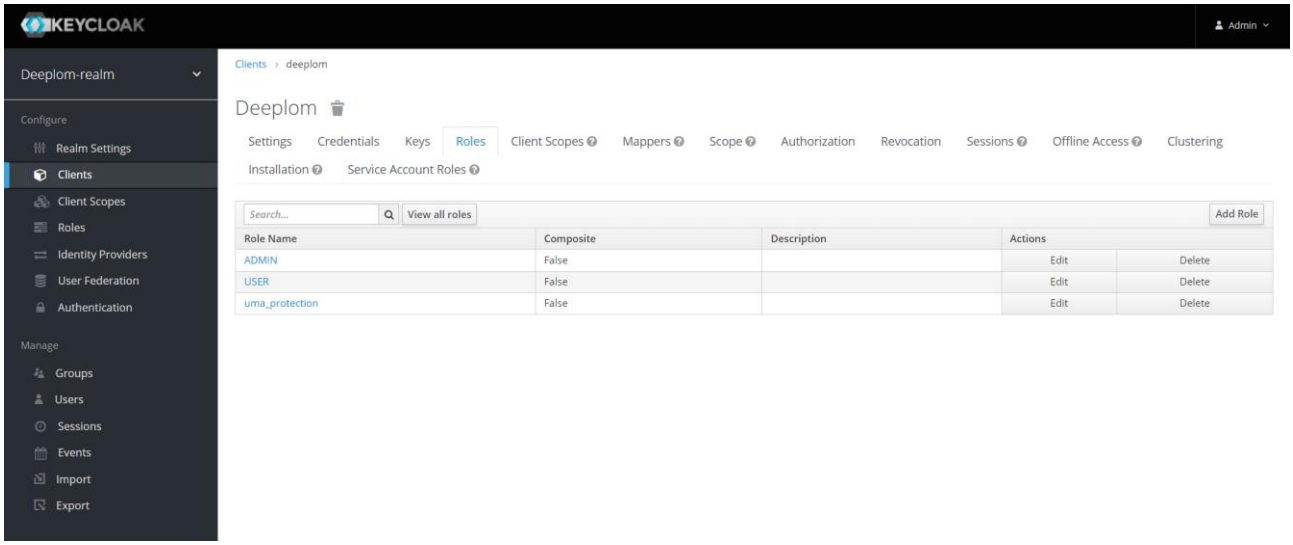


Рисунок 3.5 Ролі користувачів у застосунку

Для аутентифікації використовується протокол безпеки OAuth 2.0 з паролем як типом гранту. Вищезгадані інструменти допомагають вирішити проблему з контролем доступу.

Gateway сервіс – вхідна точка для всієї архітектури. Для його реалізації був використаний Spring Cloud Gateway. Він взаємодіє з Eureka сервером, який відповідає за знаходження інших сервісів і потім робить виклик на потрібний. Використання такого підходу допомагає зменшити поверхню атаки.

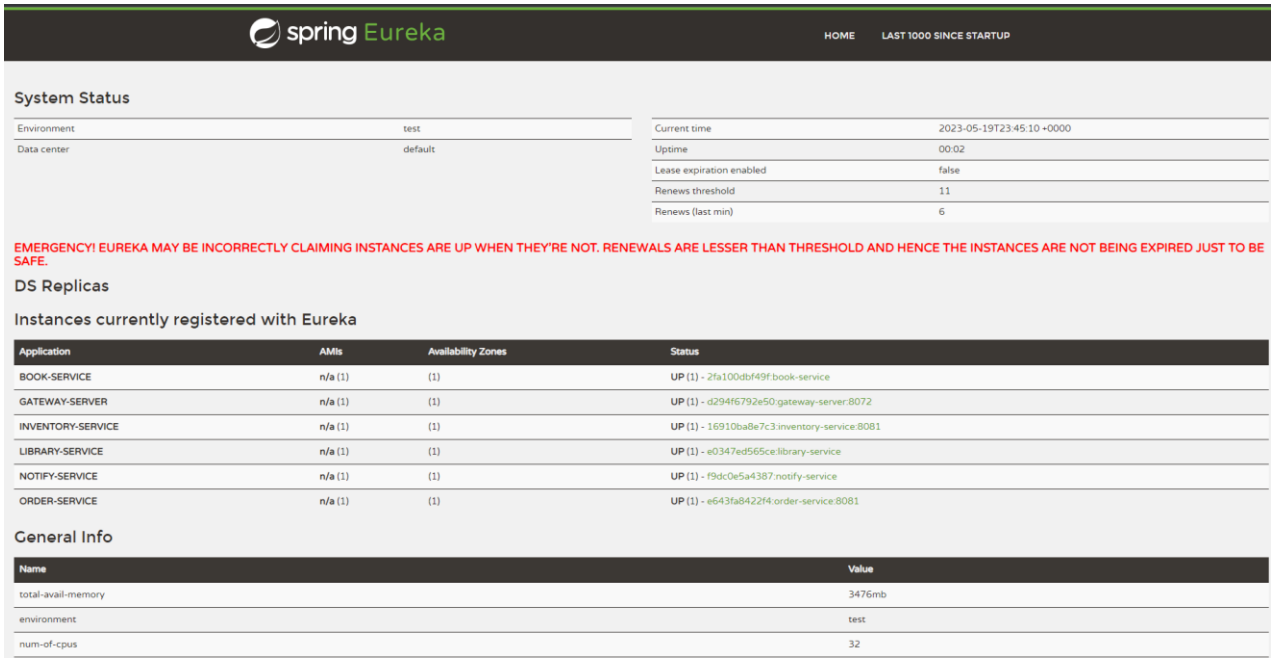
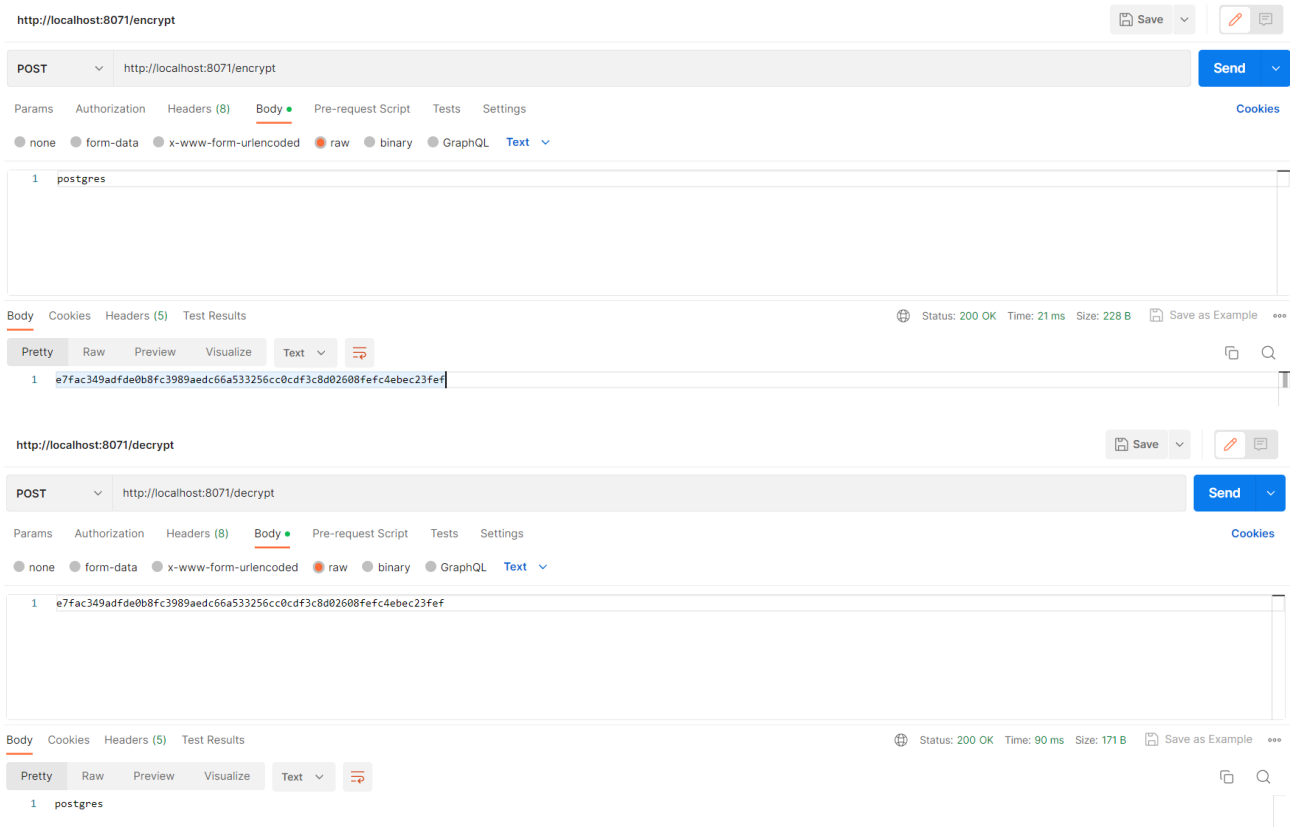


Рисунок 3.6 Сервіси знайдені Eureka

Config сервіс, який використовує Spring Boot Configuration Server, містить файли конфігурації для належної роботи програми. Конфіденційна інформація шифрується за допомогою симетричного ключа. Під час запуску Spring Cloud Configuration Server перевіряє, що змінна оточення ENCRYPT_KEY або відповідна властивість у файлі конфігурації встановлені та автоматично додає 2 нових ендпоїнта, до служби Spring Cloud Config, які потрібні для шифрування і дешифрування з використанням заданого ключа відповідно.



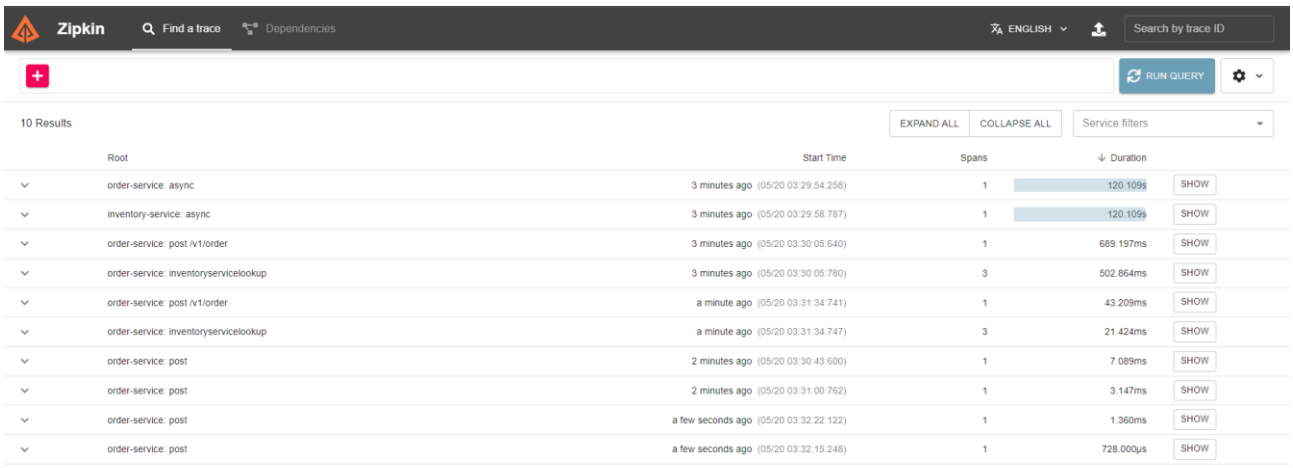
Рисунки 3.7 і 3.8 Шифрування та дешифрування з використанням Spring Cloud Configuration Server

Також сервер конфігурації вимагає, щоб всі зашифровані у такий спосіб властивості починались зі значення {cipher}. Воно повідомляє серверу конфігурації, що він має справу із зашифрованими даними.

Config сервер у поєднанні зі згаданим раніше Vault призначений для захисту чутливої інформації.

Для моніторингу використано ряд інструменті. Також архітектура використовує Zipkin – інструмент візуалізації даних, який показує потік

транзакцій через декілька сервісів. Він дозволяє розбити транзакцію на складові частини і визначити, де можуть бути проблемні місця.



Рисунки 3.9 Інтерфейс Zipkin

Крім цього, використовується стек ELK, який поєднує у собі три інструменти з відкритим вихідним кодом: Elasticsearch, Logstash і Kibana, які дозволяють аналізувати, шукати та візуалізувати логи в режимі реального часу. Elasticsearch – це розподілений аналітичний інструмент для всіх типів даних (структурованих та неструктурованих, числових, текстових тощо). Logstash – це інструмент для обробки даних на стороні сервера, який дозволяє додавати та отримувати дані з декількох джерел одночасно і трансформувати їх перед тим, як вони будуть проіндексовані Elasticsearch. Kibana – це інструмент візуалізації та управління даними в Elasticsearch. Він надає графіки, карти та гістограми в реальному часі. Служби взаємодіють через TCP з Logstash для надсилання даних логів. Logstash фільтрує, трансформує і передає дані до центрального сховища даних (в даному випадку Elasticsearch). Elasticsearch індексує та зберігає дані у зручному форматі, щоб їх пізніше можна було запросити у Kibana. Після того, як дані збережено, Kibana використовує шаблони індексів з Elasticsearch для пошуку даних. Zipkin – це розподілена платформа трасування, яка дозволяє відстежувати транзакції, що проходять через ланцюжок викликів сервісів. Це дозволяє графічно зобразити, скільки часу займає транзакція і скільки часу окремо витрачено на кожному мікросервісі, залученому до виклику. Zipkin дає можливість надсилати дані через RabbitMQ або Kafka. При

HTTP-трасуванні Zipkin використовує асинхронний потік для надсилання даних про продуктивність. Використання RabbitMQ або Kafka для збору даних трасування має перевагу: у разі, якщо сервер Zipkin з якоїсь причини не працює, повідомлення трасування, надіслані до Zipkin, будуть записуватись у чергу доти, доки Zipkin не зможе забрати дані.[7] Крім цього, використовувалась Spring Boot Actuator – це бібліотека, яка надає інструменти моніторингу та адміністрування для REST API у досить простий спосіб. Вона організує та демонструє список доступних ендпоїнтів REST, які дозволяють отримати доступ до різної моніторингової інформації для перевірки стану сервісів [3].

Захист мережевої взаємодії між сервісами здійснюється за допомогою mTLS та JWT. TLS сертифікати були підписані за допомогою OpenSSL. Демон Docker підтримує прослуховування трьох типів: UNIX, TCP і FD. Увімкнення сокета TCP дозволяє клієнту Docker спілкуватись з демоном віддалено. Для цього використано Socat, який запускається як маршрутизатор трафіку на тій самій машині, на якій запущено демон Docker. Він прослуховує TCP-трафік і перенаправляє його на UNIX-сокет демона Docker. Nginx діє як зворотний проксі для Socat. Всі зовнішні Docker-клієнти повинні звертатися до демона Docker через Nginx. Контейнери підписувались за допомогою DCT [4].

Усі сервіси (за винятком бази даних) розгорнуті у вигляді контейнерів Docker і працюють всередині одновузлового кластеру Elastic Kubernetes Service (EKS).

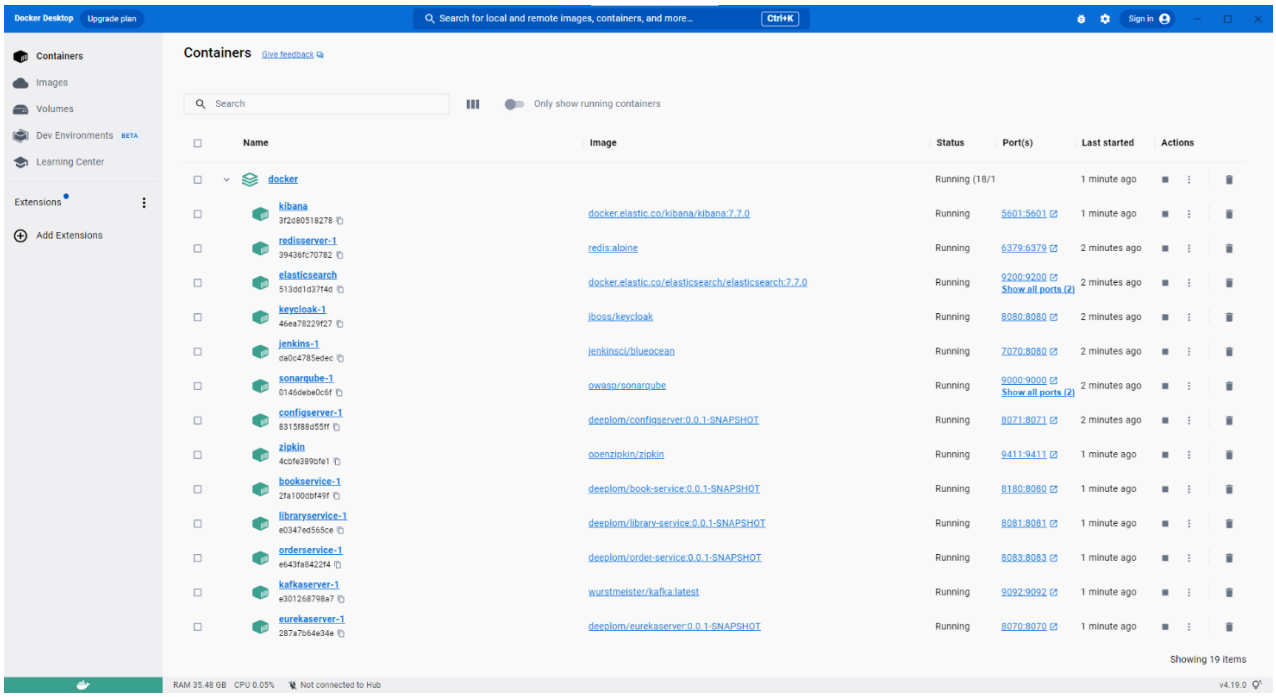


Рисунок 3.10 Розгорнуті сервіси у Docker

EKS конфігурує та налаштовує сервери, необхідні для запуску Docker кластера. EKS також може відстежувати стан контейнерів, що працюють в Docker, і перезапустити сервіс, якщо він вийде з ладу. Використовується група безпеки Amazon, щоб дозволити тільки порту шлюзу бути доступним. Усі сервери, включно з сервером Kafka, недоступні для зовнішнього світу через відкриті Docker-порти. Вони доступні лише всередині контейнера EKS. Для роботи з Amazon Web Services використовуються Kubectl – інструмент для взаємодії з кластером Kubernetes, IAM Authenticator – інструмент, що забезпечує аутентифікацію на кластері Kubernetes, Eksctl – проста утиліта командного рядка для керування та створення кластерів AWS EKS в обліковому записі AWS [10]. Для розгортання бази даних використовується Amazon Relational

Database Service (Amazon RDS).

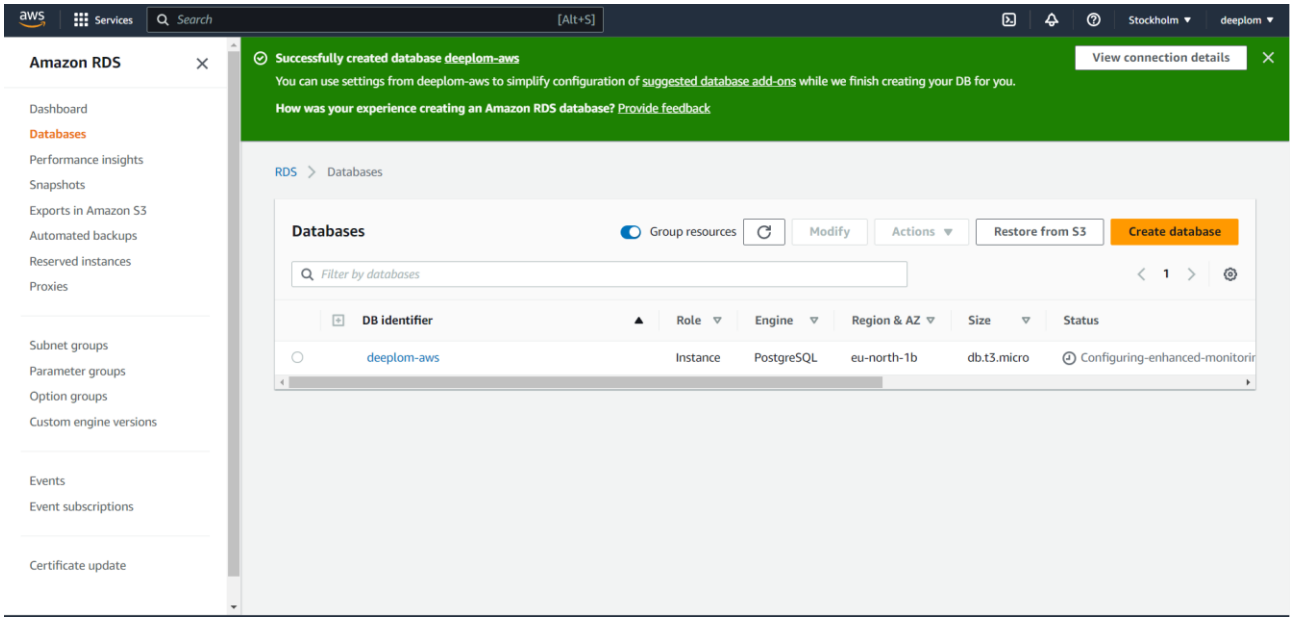


Рисунок 3.11 Розгорнута база даних PostgreSQL у Amazon RDS

Сервіси Elasticsearch, Logstash і Kibana розгорнуті в екземплярі EC2 – це віртуальний сервер у хмарі Amazon Compute Cloud (Amazon EC2).

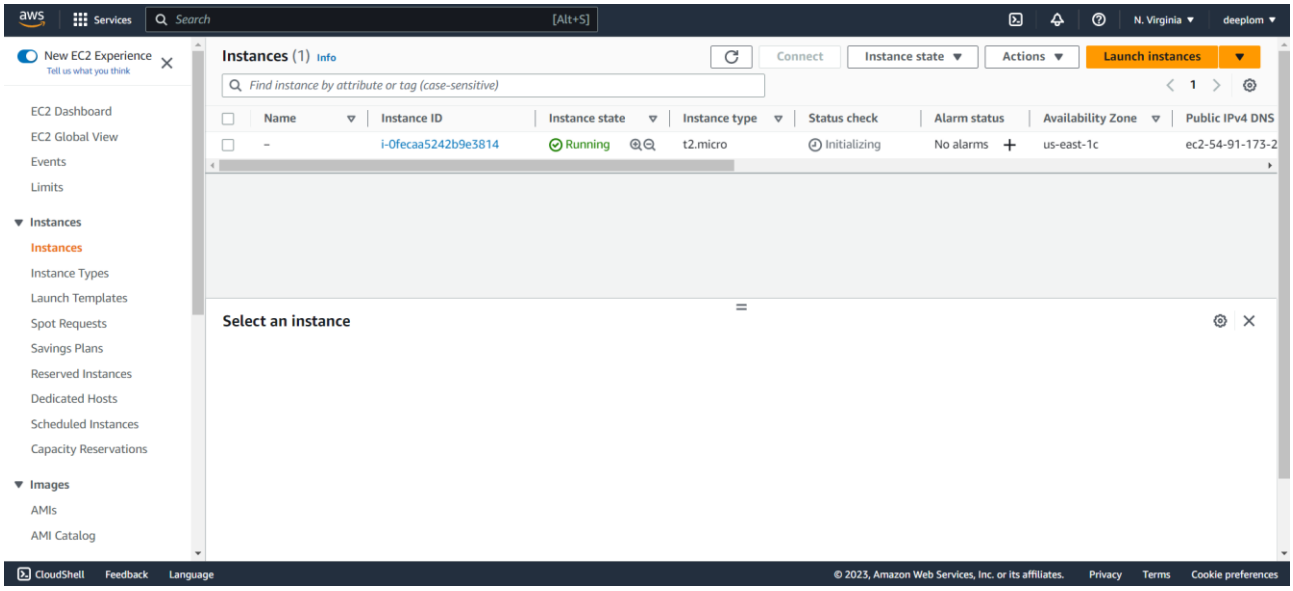


Рисунок 3.12 Розгорнутий EC2 сервер

Виконувався статичний аналіз коду за допомогою SonarQube. SonarQube – це інструмент з відкритим кодом, який допомагає сканувати код на наявність вразливостей безпеки, поганого коду та помилок. Він може бути інтегрований з процесом збірки, щоб сканувати код безперервно (при кожній збірці). Він також може бути інтегрований з автоматизованими інструментами збірки,

такими як Jenkins. Jenkins – це сервер автоматизації з відкритим вихідним кодом. Він надає плагіни для підтримки багатьох процесів збірки, розгортання та автоматизації. SonarQube дозволяє сканувати код на наявність вразливостей до того, як відбудеться збірка, що є частиною CI/CD [12]. Також SonarQube надає шаблони підходів для вирішення проблем.

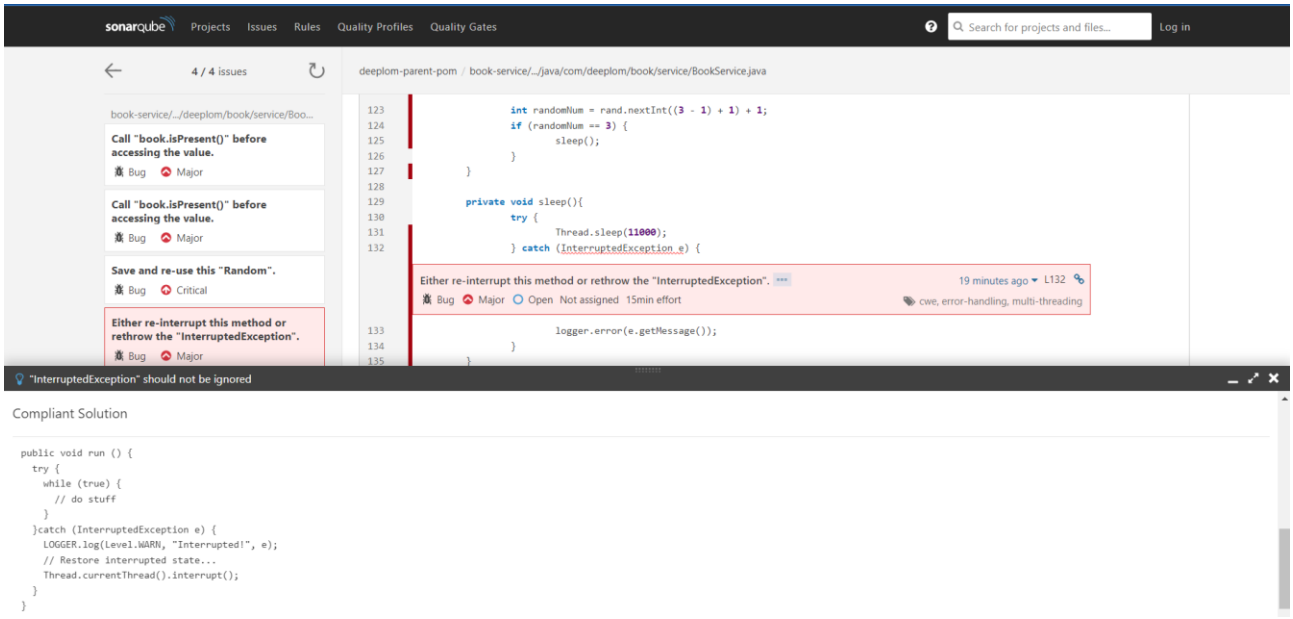


Рисунок 3.13 Приклад рішення, запропонованого SonarQube

Після першого аналізу коду, система знайшла досить багато неякісного коду (в основному пов’язаного з наявністю закоментованого коду та його дублювання), декілька багів і вразливостей, що пов’язані з використанням Entity з бази даних замість Data Transfer Object (DTO). Переважну більшість з них було виправлено. Також SonarQube складає перелік потенційно вразливих місць для ручної перевірки розробником їх серйозності. Цей інструмент може виявляти загальні вразливості безпеки у коді, такі як SQL ін’єкції, XSS-атаки, path traversal, RCE. Він може також виявити незахищені конфігурації (слабкі паролі, секрети записані у код чи неналежним чином захищені засоби контролю доступу) та криптографічні вразливості тощо [14].

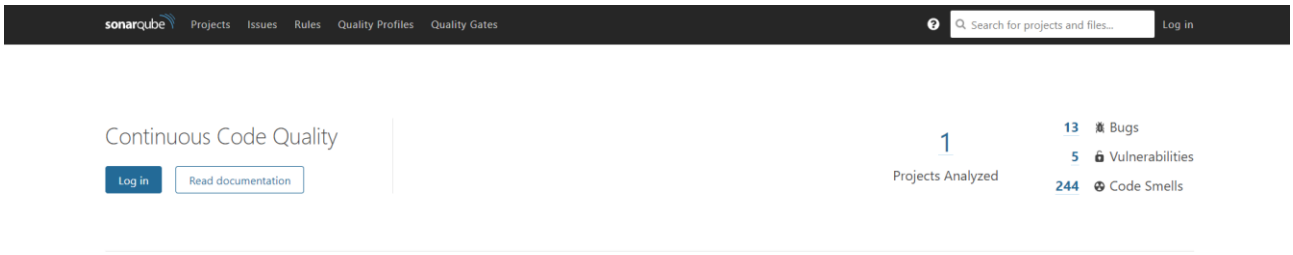


Рисунок 3.14 Результати аналізу після першого запуску SonarQube

Динамічний аналіз коду перевіряє код за допомогою автоматизованих і ручних процесів під час його виконання. Динамічний аналіз додатку виконувався за допомогою OWASP ZAP. ZAP (Zed Attack Proxy) – це інструмент з відкритим кодом, який допомагає знаходити вразливості безпеки у веб-додатках. Цей процес тестує різні шляхи виконання програми, автоматично генеруючи різні типи вхідних параметрів. ZAP – інструмент, який діє як проксі-сервер між клієнтською програмою (веб браузером) і сервером, аналізуючи запити і відповіді, щоб виявити потенційні вразливості в додатку.

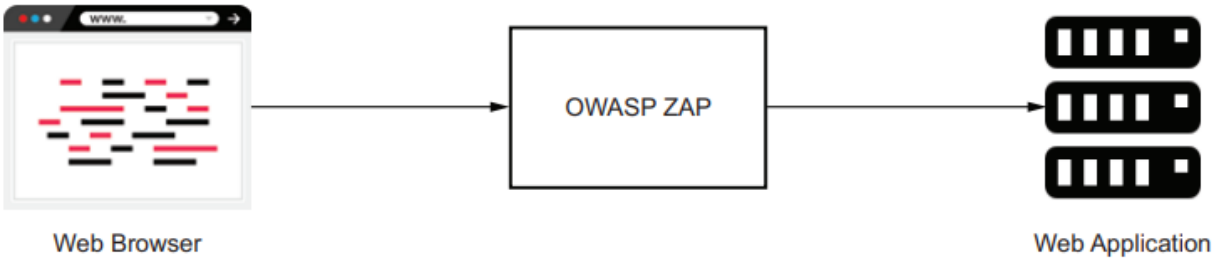


Рисунок 3.14 Схема роботи OWASP ZAP

ZAP може виявляти поширені помилки ін’єкцій (SQL-ін’єкції, XSS. RCE тощо), підробку міжсайтових запитів, порушення аутентифікації, неправильні конфігурації безпеки (відкрита конфіденційна інформація, незахищені конфігурації тощо), розголошення інформації та небезпечні криптографічні практики тощо [13].

Пентест – це процес перевірки комп’ютерної системи, мережі або веб-застосунку на наявність вразливостей, які можуть бути експлуатовані для завдання шкоди. Однією зі складових пентесту є пасивне сканування – це сканування, яке шукає відомі вразливості у трафіку, що проходить. Воно є

цілком безпечним, адже не втручається у роботу системи, а лише аналізує її. Однак, воно є не досить ефективним, бо проводиться лише аналіз можливих вразливостей без спроб їх експлуатації. Прикладом пасивного сканування є пошук використаних версій програмного забезпечення та його перевірка на відомі вразливості у публічних базах даних. Активне сканування, з іншого боку, набагато ефективніше, оскільки навмисно намагається проникнути у систему, використовуючи відомі методи для пошуку вразливостей [11].

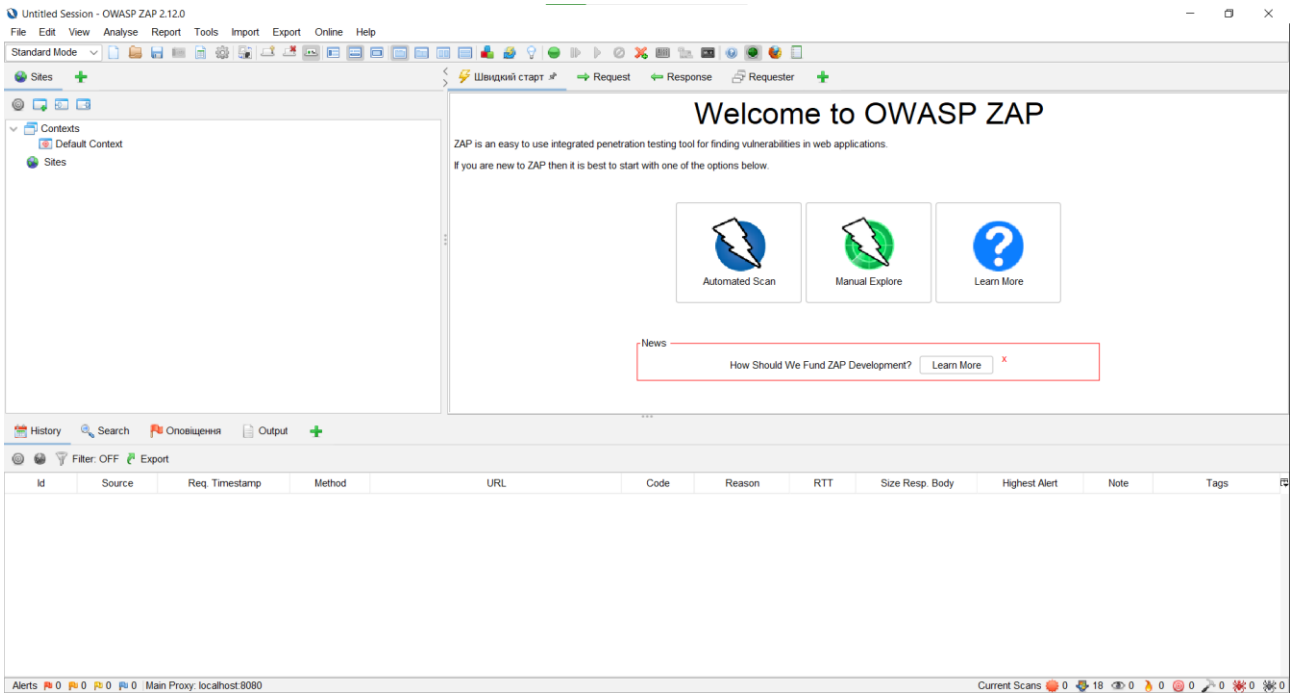


Рисунок 3.15 Інтерфейс інструменту OWASP ZAP

Оскільки автоматичне тестування ZAP призначене для веб-застосунків з інтерфейсом користувача, провести його не вдалось у повній мірі.

ВИСНОВКИ ПО РОБОТІ

У рамках роботи був проведений аналіз різних механізмів безпеки у системах із мікросервісною архітектурою, були розглянуті проблеми безпеки, а також запропоновані рекомендації щодо захисту від загроз в таких системах. Були ідентифіковані основні проблеми безпеки, зокрема проблеми з контролем доступу, розгортанням, мережною комунікацією, витоком чутливої інформації та відсутністю логування у мікросервісах. Ці проблеми можуть ставити під загрозу систему і призводити до негативних наслідків. Крім цього, були розглянуті загальні механізми безпеки, такі як: аутентифікація та авторизація, мережевий захист, моніторинг тощо. Правильна імплементація рекомендацій може значно підвищити рівень безпеки мікросервісних систем:

- Подбати про надійний контроль доступу до застосунку за допомогою аутентифікації та авторизації
- Захистити мережну комунікацію за допомогою mTLS, JWT чи їх поєднання
- Конфіденційну інформацію потрібно зберігати та передавати у зашифрованому вигляді. Для цього можна використовувати спеціальні інструменти
- Під час розгортання сервісів потрібно звернути увагу на дані конфігурації – вони не повинні розгортатись разом з сервісами, для їх зберігання пропонується використовувати, наприклад, Secrets у Kubernetes. Контейнери потрібно підписувати за допомогою DCT та надавати їм лише ті дозволи, які потрібні для нормального роботи.
- Моніторинг є важливою частиною безпеки, тому необхідне використання систем, які його забезпечують (Zipkin, стек EKL, Graphana, Prometheus), оскільки вони можуть вказати на слабкі місця та попередити про атаки.

Було описано практичне використання механізмів безпеки на прикладі створення мікросервісного застосунку за допомогою Spring Boot та Spring Cloud. Також розглядаються інструменти для полегшення імплементації

захищеної мікросервісної архітектури та тестування застосунку на можливі вразливості.

Отже, на основі проведеного аналізу можна зробити висновок, що забезпечення безпеки у мікросервісній архітектурі є критичним завданням і вимагає комплексного підходу. Використання рекомендацій, запропонованих у даній роботі, допоможе забезпечити належний рівень безпеки в мікросервісних застосунках та зменшити ймовірність виникнення небажаних ситуацій і порушень безпеки.

СПИСОК ЛІТЕРАТУРИ

1. 10 companies using microservices. who is using them? | code & pepper. Code & Pepper. URL: https://codeandpepper.com/companies-using-microservices/ (date of access: 17.03.2023).

2. Cultural Communications. URL: http://www.cultural.com/web/security/infosec.glossary.html (date of access: 21.03.2023).

3. Carnell J. Spring microservices in action. Manning Publications, 2017. 384 p.

4. Wajjakkara Kankanamge Anthony Nuwan Dias, Siriwardena P. Microservices security in action. Manning Publications Co. LLC, 2020.

5. Gilman E., Barth D. Zero trust networks: building secure systems in untrusted networks. O'Reilly Media, 2017. 240 p.

6. Richer J., Sanso A. OAuth 2 in action. Manning Publications, 2017. 360 p.

7. Programming Techie. Spring boot microservice project full course in 6 hours 🔥🔥🔥, 2022. YouTube. URL: https://www.youtube.com/watch?v=mPPhcU7oWDU (date of access: 12.04.2023).

8. OWASP foundation, the open source foundation for application security | OWASP foundation. OWASP Foundation, the Open Source Foundation for Application Security | OWASP Foundation. URL: https://owasp.org/ (date of access: 11.05.2023)

9. Luksa M. Kubernetes in action. Manning Publications, 2018. 624 p.

10. Wittig A., Wittig M. Amazon web services in action. Manning Publications, 2018. 528 p.

11. Davis R. Art of network penetration testing: taking over any company in the world. Manning Publications Company, 2020. 280 p.

12. Agile application security: enabling security in a continuous delivery pipeline / L. Bell et al. O'Reilly Media, 2017. 386 p.

13. The ZAP homepage. OWASP ZAP. URL: https://www.zaproxy.org/ (date of access: 16.05.2023).

14. SonarQube 10.0. SonarQube 10.0. URL: https://docs.sonarqube.org/latest/ (date of access: 17.05.2023).