Міністерство освіти і науки України
Державний університет «Житомирська політехніка»
Факультет інформаційно-комп'ютерних технологій
Кафедра комп’ютерної інженерії та кібербезпеки

ПОЯСНЮВАЛЬНА ЗАПИСКА

до випускної кваліфікаційної роботи бакалавра

на тему: «Проєкт клієнт-серверної програмної системи моніторингу подій безпеки та стану інфраструктури Active Directory»

Виконав: студент 4-го курсу групи КІ-22-1
спеціальності 123 «Комп’ютерна інженерія»
_____ Олександр ШЕПЕТЬКО

Керівник: доцент кафедри КІ та КБ, к.пед.н., доцент
_____ Олена ГОЛОВНЯ

Рецензент: завідувач кафедри КІ та КБ, к.т.н., доцент
_____ Андрій ЄФІМЕНКО

Житомир – 2026

Державний університет «Житомирська політехніка»
Факультет інформаційно-комп'ютерних технологій
Кафедра комп'ютерної інженерії та кібербезпеки
Спеціальність 123 «Комп'ютерна інженерія»
Освітня програма «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач
кафедри комп'ютерної
інженерії та кібербезпеки
Андрій Єфіменко
2 лютого 2026

З А В Д А Н Н Я

На випускну кваліфікаційну роботу бакалавра

Студент: Шепетько Олександр Петрович

Тема роботи: «Проєкт клієнт-серверної програмної системи моніторингу подій безпеки та стану інфраструктури Active Directory»

затверджена Наказом університету від 2 лютого 2026 р. №57/с

Термін здачі студентом закінченої роботи 23 червня 2026 р.

Вихідні дані роботи (зазначається предмет і об'єкт дослідження):

- процес безперервного моніторингу апаратних ресурсів та контролю подій інформаційної безпеки у гетерогенних корпоративних інфраструктурах на базі Microsoft Active Directory;
- архітектура та алгоритми оптимізації кросплатформних систем збору телеметрії, механізми делегованої фільтрації системних журналів XPath, технології віддаленого виклику процедур (gRPC), а також засоби агрегації та візуалізації даних (PostgreSQL, Grafana).

Обладнання та програмне забезпечення:

1. Oracle VirtualBox 7.0.22
2. Visual Studio Code
3. Windows Server 2022
4. Windows 10
5. Ubuntu 24.04 LTS

Консультанти з випускної кваліфікаційної роботи із зазначенням розділів, що їх стосуються

Розділ	Консультант	Підпис, дата	
		завдання видав	завдання прийняв
1	Олександр ПІРОГ	30.03.2026	30.03.2026
2	Олександр ПІРОГ	9.05.2026	9.05.2026
3	Олександр ПІРОГ	16.06.2026	16.06.2026

Студент _____ Олександр ШЕПЕТЬКО

Керівник _____ Олена ГОЛОВНЯ

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назви етапів випускної кваліфікаційної роботи	Термін виконання	Примітка
1	Вибір теми випускної кваліфікаційної роботи та аналіз проблематики систем моніторингу	02.02.2026	Виконано
2	Розробка та затвердження структури пояснювальної записки	09.02.2026	Виконано
3	Аналіз існуючих рішень для моніторингу та обґрунтування архітектури клієнт-сервер	16.02.2026	Виконано
4	Дослідження протоколу gRPC та механізмів оптимізації збору телеметрії	23.02.2026	Виконано
5	Аналіз методів обробки станів та проєктування стейтфул-алгоритмів	02.03.2026	Виконано
6	Проєктування тестового віртуального середовища та схеми мережевої взаємодії	09.03.2026	Виконано
7	Розгортання середовища віртуалізації та встановлення кросплатформних ОС	16.03.2026	Виконано
8	Налаштування Active Directory Domain Services та підготовка інфраструктури	23.03.2026	Виконано
9	Програмна реалізація клієнтських агентів, алгоритму ковзного вікна та моніторингу служб	06.04.2026	Виконано
10	Розробка підсистеми аудиту безпеки та делегованої фільтрації подій (XPath)	13.04.2026	Виконано
11	Програмна реалізація центрального сервера та налаштування веб-сервера Kestrel (HTTP/2)	27.04.2026	Виконано
12	Налаштування бази даних PostgreSQL та інтеграція міграцій Entity Framework Core	11.05.2026	Виконано
13	Розгортання платформи Grafana, налаштування дашбордів та інтеграція зі Slack	25.05.2026	Виконано
14	Проведення комплексного тестування (імітація навантаження, падіння служб, інциденти)	08.06.2026	Виконано
15	Аналіз результатів тестування та профілювання апаратного навантаження агента	15.06.2026	Виконано
16	Оформлення висновків та підготовка роботи до захисту	23.06.2026	Виконано

Студент _____ Олександр ШЕПЕТЬКО

Керівник _____ Олена ГОЛОВНЯ

АНОТАЦІЯ

Пояснювальна записка до кваліфікаційної роботи бакалавра з теми «Проєкт клієнт-серверної програмної системи моніторингу подій безпеки та стану інфраструктури Active Directory» містить відомості про розробку та тестування оптимізованої системи збору телеметрії для середовища Microsoft Active Directory. Обсяг пояснювальної записки становить 107 сторінок, які містять 46 рисунків, 1 таблицю та 3 додатки. Список використаних джерел включає 21 найменування.

Мета роботи: проєктування та розробка високопродуктивної, захищеної клієнт-серверної системи для централізованого збору апаратної телеметрії та моніторингу подій безпеки Active Directory в корпоративних мережах.У ході виконання роботи було проаналізовано існуючі архітектурні підходи до побудови корпоративних систем моніторингу, проблематику надмірного споживання апаратних ресурсів клієнтськими агентами та методи оптимізації збору телеметрії. Проведено дослідження протоколу gRPC, технологій об'єктно-реляційного відображення, а також розроблено стейтфул-алгоритми для інтелектуального аналізу станів системи. Окрему увагу приділено питанням криптографічного захисту каналів зв'язку та безпечного зберігання конфігураційних секретів.

У практичній частині роботи було розгорнуто тестове віртуальне середовище, що включає контролер домену Active Directory та кросплатформні клієнтські вузли. Програмно реалізовано клієнтські агенти та центральний сервер у вигляді автоматичних фонових служб. Налаштовано базу даних PostgreSQL за допомогою інструментарію міграцій Entity Framework Core, інтегровано платформу візуалізації Grafana та налаштовано систему автоматичних сповіщень через вебхуки у корпоративний месенджер. Проведено комплексне тестування алгоритмів фільтрації подій безпеки XPath,

профілювання навантаження та перевірку відмовостійкості розробленої архітектури.

Ключові слова: МОНІТОРИНГ, ACTIVE DIRECTORY, C#, .NET, GRPC, POSTGRESQL, GRAFANA, HTTP/2, ІНФОРМАЦІЙНА БЕЗПЕКА, ТЕЛЕМЕТРІЯ, ХРАТН.

ANNOTATION

Explanatory Note to the Bachelor's Qualification Thesis on the topic "Design of a Client-Server Software System for Monitoring Security Events and Active Directory Infrastructure Status" contains information on the development and testing of an optimized telemetry collection system for the Microsoft Active Directory environment. The volume of the explanatory note is 107 pages, including 46 figures, 1 tables, and 3 appendices. The list of references includes 21 sources.

The aim of the thesis: design and development of a high-performance, secure client-server system for centralized hardware telemetry collection and Active Directory security event monitoring in corporate networks.

In the course of the work, existing architectural approaches to building corporate monitoring systems, the issues of excessive hardware resource consumption by client agents, and methods for optimizing telemetry collection were analyzed. A study of the gRPC protocol and object-relational mapping technologies was conducted, and stateful algorithms for intelligent system state analysis were developed. Special attention was paid to the cryptographic protection of communication channels and the secure storage of configuration secrets.

In the practical part of the work, a test virtual environment was deployed, including an Active Directory domain controller and cross-platform client nodes. Client agents and a central server were programmatically implemented as automatic background services. The PostgreSQL database was configured using Entity Framework Core migration tools, the Grafana visualization platform was integrated, and an automated alert system via webhooks to a corporate messenger was set up. Comprehensive testing of XPath security event filtering algorithms, workload profiling, and fault tolerance verification of the developed architecture was performed.

Keywords: MONITORING, ACTIVE DIRECTORY, C#, .NET, GRPC, POSTGRESQL, GRAFANA, HTTP/2, INFORMATION SECURITY, TELEMETRY, XPATH.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AD – Active Directory

API – Application Programming Interface

CPU – Central Processing Unit

DHCP – Dynamic Host Configuration Protocol

DNS – Domain Name System

EF Core – Entity Framework Core

gRPC – gRPC Remote Procedure Calls

HTTP/2 – Hypertext Transfer Protocol Version 2

JSON – JavaScript Object Notation

KDC – Key Distribution Center

NTDS – NT Directory Services

OS – Operating System

RAM – Random Access Memory

RPC – Remote Procedure Call

SQL – Structured Query Language

TLS – Transport Layer Security

WMI – Windows Management Instrumentation

XML – eXtensible Markup Language

XPath – XML Path Language

Зміст

ВСТУП 10

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ТА АНАЛІЗ ТЕХНОЛОГІЙ 13

1.1. Теоретичні основи та класифікація систем моніторингу 13

1.2. Порівняльний аналіз методів збору та передачі даних 17

1.3. Огляд сучасних технологічних стеків для розробки систем моніторингу. 23

Висновки до першого розділу 31

РОЗДІЛ 2. ПРОЄКТУВАННЯ СИСТЕМИ МОНІТОРИНГУ 32

2.1. Проєктування клієнт-серверної топології та структури додатків 32

2.2. Проєктування моделі даних та контрактів взаємодії 37

2.3. Алгоритмічне забезпечення обробки телеметрії та інцидентів 40

2.4. Проєктування підсистеми безпеки та захисту інфраструктури 44

Висновки до розділу 2 47

РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ МОНІТОРИНГУ 48

3.1. Розгортання тестового стенда та конфігурація програмних компонентів 48

3.2. Програмна реалізація підсистеми збору телеметрії та мережевої взаємодії 50

3.3. Налаштування бази даних та панелей візуалізації Grafana 63

3.4. Імітація інцидентів безпеки та перевірка системи сповіщень 66

Висновки до розділу 3 72

ВИСНОВКИ 74

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 76

ДОДАТКИ 78

ВСТУП

Актуальність теми. У сучасну епоху тотальної цифровізації бізнес-процесів корпоративні комп'ютерні мережі перетворилися на головну артерію будь-якого підприємства. Від стабільності та безпеки їхньої роботи безпосередньо залежить не лише ефективність співробітників, а й фінансове благополуччя компаній. Зі стрімким зростанням обсягів даних та ускладненням IT-інфраструктур виникає гостра необхідність у безперервному контролі за станом серверів та робочих станцій. Водночас кіберзагрози стають дедалі складнішими: зловмисники вже рідко атакують мережу "в лоб", натомість вони намагаються скомпрометувати облікові записи користувачів, закріпитися в системі та непомітно переміщуватись усередині периметра.

Основою управління ідентифікацією та доступом у більшості сучасних корпоративних мереж залишається служба каталогів Active Directory. Вона є головною ціллю для будь-якої кібератаки, адже отримання контролю над контролером домену означає повну компрометацію всієї інфраструктури підприємства. Саме тому безперервний моніторинг подій безпеки (успішні та невдалі спроби входу, зміни групових політик, створення нових користувачів, блокування акаунтів тощо) є критично важливою задачею для системних адміністраторів та фахівців з інформаційної безпеки.

Однак на практиці більшість підприємств стикаються з серйозною проблемою. Традиційні системи моніторингу часто працюють за принципом постійного опитування (polling), використовуючи застарілі та важкі протоколи на кшталт WMI або стандартні REST API запити. Це призводить до надмірного споживання системних ресурсів на кінцевих пристроях. Крім того, класичні підходи генерують величезний обсяг мережевого трафіку та створюють ефект "втоми від алертів", коли адміністратор отримує сотні некритичних сповіщень щогодини і може пропустити дійсно важливий інцидент.

Вирішенням цієї проблеми є розробка оптимізованої, легкої та захищеної клієнт-серверної системи моніторингу, яка базується на сучасних технологіях обміну даними. Використання протоколу gRPC поверх HTTP/2 разом із

бінарною серіалізацією Protobuf дозволяє кардинально знизити навантаження на мережу порівняно з класичним JSON/REST [2][3]. Застосування подієво-орієнтованої архітектури для читання журналів операційної системи гарантує, що ресурси комп'ютера витрачаються лише в момент фактичного настання події. Крім того, впровадження алгоритмів згладжування даних дозволяє фільтрувати короткочасні стрибки навантаження апаратного забезпечення та надсилати лише валідовані сповіщення. Саме потреба підприємств у такому швидкому, захищеному та оптимізованому інструменті для моніторингу Active Directory та апаратної телеметрії зумовлює високу актуальність даної кваліфікаційної роботи.

Мета і завдання дослідження. *Метою* кваліфікаційної роботи є проєктування та розробка високопродуктивної, захищеної клієнт-серверної системи для централізованого збору апаратної телеметрії та моніторингу подій безпеки Active Directory в корпоративних мережах.

Для досягнення поставленої мети було сформульовано та вирішено такі основні завдання:

- провести аналіз проблематики моніторингу корпоративних мереж та дослідити основні вектори загроз для інфраструктури Active Directory;
- виконати порівняльний аналіз сучасних архітектурних підходів (REST API, gRPC) та методів збору телеметрії;
- спроєктувати клієнт-серверну топологію системи, структуру бази даних та алгоритми обробки даних;
- розробити архітектуру безпеки, що включає криптографічний захист каналів зв'язку TLS, сертифікатну автентифікацію через групові політики та ізоляцію секретних даних;
- реалізувати програмний комплекс, що складається з легкої клієнтської служби агента та центрального сервера обробки даних на базі платформи .NET 8;

- провести розгортання бази даних PostgreSQL, інтеграцію з системою візуалізації Grafana та налаштування автоматичних сповіщень через вебхуки;
- виконати тестування розробленої системи та оцінити її ефективність.

Об’єкт дослідження — процеси безперервного моніторингу, збору та обробки телеметричних даних і подій інформаційної безпеки в корпоративних комп'ютерних мережах.

Предмет дослідження — методи, алгоритми та програмні засоби побудови захищених клієнт-серверних систем моніторингу з використанням протоколу gRPC, технологій подієво-орієнтованого аналізу та криптографічного захисту.

Методи дослідження. Під час виконання роботи використовувався комплекс методів: системний аналіз — для формування вимог до програмного продукту та вивчення принципів роботи Active Directory; методи об'єктно-орієнтованого проєктування — для розробки архітектури додатків на мові C#; методи криптографії — для забезпечення конфіденційності та цілісності даних під час їх передачі мережею; експериментальний метод — для тестування швидкодії агента, перевірки алгоритмів фільтрації та загальної надійності системи.

Практичне значення одержаних результатів. Результатом роботи є повністю функціональна система моніторингу рівня підприємства, яка готова до впровадження в реальні IT-інфраструктури. Створений полегшений клієнтський агент здатний працювати як у Windows, так і в Linux-середовищах, збираючи дані з мінімальним впливом на ресурси комп'ютерів. Розроблена серверна частина забезпечує надійне збереження даних, а реалізовані механізми безпеки реалізують захист від перехоплення трафіку всередині мережі. Ця система може бути використана системними адміністраторами для проактивного виявлення збоїв обладнання та спроб несанкціонованого доступу.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ТА АНАЛІЗ ТЕХНОЛОГІЙ

1.1. Теоретичні основи та класифікація систем моніторингу

Розробка програмної системи моніторингу починається з прийняття рішення щодо способу збору метрик та системних подій з цільових машин. У сучасній практиці побудови IT-інфраструктур виділяють два базові підходи: безагентний та агентний. Кожна з них має свої чітко виражені переваги, обмеження та специфіку застосування, що безпосередньо впливає на загальну безпеку, масштабованість та надійність усього комплексу спостереження. З огляду на те, що цільовим об’єктом у нашому випадку виступає інфраструктура Active Directory, вибір архітектури має обов'язково враховувати не лише стандартні показники продуктивності обладнання, але й необхідність безперервного аналізу журналів подій безпеки.

Безагентний підхід передбачає, що збір телеметрії здійснюється виключно централізованим сервером без необхідності інсталяції додаткового програмного забезпечення на кінцеві вузли. У рамках цієї моделі сервер моніторингу бере на себе активну роль і періодично опитує цільові машини, використовуючи вбудовані мережеві протоколи та стандартні служби операційної системи. Для середовищ на базі ОС Windows найчастіше використовується WMI, тоді як для мережевого обладнання чи інших систем може застосовуватися SNMP. Головною і найбільш очевидною перевагою такого підходу є відносна простота розгортання на початкових етапах. Адміністраторам не потрібно витрачати час на розповсюдження, постійне оновлення чи перевірку сумісності сторонніх програм на десятках або сотнях серверів, оскільки всі процеси збору налаштовуються централізовано.

Однак під час спроб масштабування безагентної моделі у складних корпоративних мережах можуть виникати труднощі. Насамперед це стосується питань мережевої безпеки та налаштування міжмережевих екранів. Використання технологій віддаленого доступу до системних ресурсів зазвичай вимагає відкриття широкого діапазону динамічних портів. Це суперечить принципам мінімальних привілеїв та ускладнює проходження аудитів

інформаційної безпеки. Крім того, сервер моніторингу змушений зберігати облікові дані з високими правами доступу для підключення до кожної машини у домені, що створює єдину точку відмови та робить його вкрай привабливою ціллю для зловмисників. Додатково варто враховувати питання мережевого навантаження. Оскільки сервер змушений регулярно ініціювати запити до великої кількості вузлів, обсяг службового трафіку зростає. А в разі виникнення тимчасових проблем зі зв’язком усі метрики та події за період відключення зазвичай втрачаються безповоротно, оскільки ця модель здебільшого не передбачає механізмів локального кешування даних.

На противагу цьому, агентна архітектура базується на розгортанні спеціалізованого програмного забезпечення безпосередньо на кожній цільовій машині, яку необхідно контролювати. Такий агент працює як локальна фонова служба і бере на себе завдання зі збору, первинної обробки та подальшої відправки телеметрії на центральний сервер. Хоча подібна модель вимагає певних зусиль для встановлення, вона надає значно ширші можливості для глибокого аналізу стану операційної системи. Оскільки процес збору відбувається локально, агент може безпосередньо взаємодіяти з лічильниками продуктивності та системними журналами без мережевих затримок і з мінімальним споживанням ресурсів.

З точки зору ефективності обробки даних агентний підхід демонструє високу гнучкість. Замість того, щоб постійно передавати величезні масиви "сирих" даних мережею, агент має змогу фільтрувати та агрегувати інформацію на місці. Наприклад, він може аналізувати тисячі рутинних записів у журналах безпеки Windows, але відправляти на сервер лише ті, що свідчать про потенційні інциденти: запуск підозрілих процесів, очищення журналів або масове блокування облікових записів. Такий підхід суттєво зменшує загальний обсяг трафіку та знижує обчислювальне навантаження на центральну базу даних. Іншою вагомою перевагою є вищий рівень стійкості до збоїв. Якщо з'єднання з центральним сервером тимчасово переривається, агент зберігає здатність продовжувати збір інформації, накопичуючи її у власній оперативній пам'яті.

Після відновлення зв'язку ці дані передаються на сервер, що дозволяє зберегти загальну картину інциденту без розривів у часі.

Окрім цього, агентна модель набагато краще інтегрується в середовища із суворими правилами брандмауерів. У більшості сучасних реалізацій саме агенти ініціюють вихідне з'єднання із сервером по єдиному захищеному порту. Це означає, що системним адміністраторам не потрібно відкривати вхідні порти на критично важливих вузлах, що значно ускладнює завдання потенційним атакувальникам при спробах сканування внутрішньої мережі. Розглядаючи завдання моніторингу саме інфраструктури Active Directory, важливо розуміти, що контролери домену генерують великий обсяг специфічних подій. Спроби віддаленого читання таких журналів у реальному часі створюють навантаження на мережу і самі сервери. Встановлення ж легковагового агента, який локально контролюватиме стан критичних служб домену та швидко реагуватиме на аномалії, є більш раціональним та надійним інженерним підходом.

Після визначення загального архітектурного підходу на рівні збору даних важливим є вибір механізму їх доставки від кінцевих вузлів до центрального сервера. У розподілених системах моніторингу склалися дві основні парадигми транспортування інформації: модель Pull та Push. Об'єктивне порівняння цих підходів вимагає аналізу їхнього впливу на мережеву топологію, рівень безпеки та розподіл обчислювального навантаження між компонентами системи.

Модель Pull базується на тому, що центральний сервер моніторингу виступає єдиним ініціатором обміну даними. У цій схемі сервер містить конфігураційний реєстр усіх підключених вузлів і за визначеним розкладом звертається до них для отримання нових порцій телеметрії. Агенти або вбудовані служби на кінцевих машинах працюють у пасивному режимі, очікуючи вхідного запиту для повернення зібраної інформації. Сильною стороною такого підходу є повний і централізований контроль над мережевим трафіком. Сервер самостійно визначає, коли і в якому обсязі збирати дані, що дозволяє уникнути його перевантаження. Також ця архітектура значно спрощує моніторинг доступності самих машин: якщо сервер не може встановити з'єднання під час планового

опитування, система одразу фіксує проблему зі зв'язком. Проте цей метод має суттєві обмеження щодо безпеки та оперативності. Затримка між виникненням інциденту та його фіксацією залежить від частоти опитувань. Крім того, вимога тримати відкритими вхідні мережеві порти на кожній цільовій машині знижує загальний рівень захищеності інфраструктури, що є небажаним для контролерів домену.

Модель Push використовує протилежний принцип, передаючи ініціативу мережевої взаємодії кінцевим вузлам. Центральний сервер у цьому випадку виконує роль приймача, який постійно прослуховує визначений порт та обробляє вхідні потоки даних. Агенти самостійно агрегують інформацію і встановлюють з'єднання із сервером для її передачі. З точки зору безпеки цей підхід є більш доцільним, оскільки він не вимагає відкриття вхідних портів на клієнтських машинах, обмежуючись лише дозволом на вихідний трафік. Також Push-архітектура дозволяє реалізовувати подієво-орієнтований підхід, коли критичні сповіщення про інциденти відправляються на сервер одразу після їх виникнення на локальному вузлі, мінімізуючи час реакції системи.

Разом з тим, варто розглянути інженерні недоліки архітектури Push, оскільки вона створює специфічні виклики для розробників. Найбільшою проблемою є відсутність прямого контролю сервера над інтенсивністю вхідного потоку даних. Якщо в корпоративній мережі відбувається масова подія, наприклад, розгортання оновлень операційної системи або зміна глобальних політик, сотні агентів можуть одночасно ініціювати відправку метрик. Це генерує раптові пікові навантаження на серверний інтерфейс та базу даних, що вимагає впровадження надійних механізмів мультиплексування, асинхронної обробки та використання черг повідомлень. Ще одним недоліком є складність визначення поточного статусу агента. У моделі Push відсутність даних від вузла може свідчити як про стабільну роботу системи без нових подій, так і про апаратний збій чи втрату мережевого з'єднання агентом. Для розв'язання цієї проблеми доводиться додатково проектувати механізми регулярної відправки службових сигналів активності.

Незважаючи на вказані складнощі в реалізації серверної частини, застосування моделі Push забезпечує вагому архітектурну перевагу, яка полягає в усуненні необхідності постійного сканування мережі центральним сервером. Замість того, щоб витрачати значні обчислювальні ресурси сервера та пропускну здатність на безперервне циклічне опитування кожного зареєстрованого вузла, відповідальність за ініціацію передачі даних переноситься на кінцеві машини. Такий децентралізований підхід суттєво знижує фонове навантаження на сервер і зменшує обсяг марного службового трафіку, оскільки підключення відбуваються лише тоді, коли агент дійсно має підготовлений пакет показників або алертів для відправки. Це робить систему більш масштабованою та ефективною в умовах моніторингу великої кількості кінцевих точок.

1.2. Порівняльний аналіз методів збору та передачі даних

Ефективність системи моніторингу значною мірою залежить від вибору протоколів, що використовуються для транспортування зібраних метрик та подій від кінцевих вузлів до центрального сервера. Історично склалося декілька стандартних підходів до передачі телеметрії, які досі широко застосовуються в корпоративних середовищах. Однак стрімкий розвиток розподілених та мікросервісних архітектур висуває нові вимоги до продуктивності, безпеки та надійності каналів зв'язку, що змушує переглядати доцільність використання традиційних методів на користь більш сучасних рішень.

Упродовж багатьох років базовим стандартом для моніторингу мережевого обладнання та серверних операційних систем залишається протокол **SNMP (Simple Network Management Protocol)**. Його головна перевага полягає у кросплатформній підтримці: майже будь-який маршрутизатор, комутатор або сервер "з коробки" здатен віддавати метрики через SNMP. Протокол використовує ієрархічну базу даних управління (MIB), яка дозволяє стандартизовано звертатися до різних параметрів пристрою. Проте використання SNMP у сучасних високонавантажених системах супроводжується низкою критичних обмежень. По-перше, ранні версії протоколу (SNMPv1 та SNMPv2c) передають дані, включно з ідентифікаторами спільнот (community strings), у

відкритому вигляді, що робить їх вразливими до перехоплення. Хоча версія SNMPv3 вирішує проблеми шифрування та автентифікації, її налаштування та адміністрування у великих інфраструктурах є складним. Крім того, SNMP базується на транспортному протоколі UDP, що не гарантує доставку пакетів. В умовах нестабільного мережевого з'єднання або високого навантаження частина критично важливої телеметрії може бути втрачена без можливості відновлення.

Для інфраструктур, побудованих на базі операційних систем сімейства Windows, глибоко інтегрованим методом збору метрик є **WMI (Windows Management Instrumentation)**. Це потужний інтерфейс, який надає адміністраторам та системам моніторингу доступ до вичерпної інформації про стан операційної системи, апаратного забезпечення та встановленого програмного забезпечення. За допомогою WMI-запитів (WQL) можна отримати детальні дані про споживання ресурсів, стан служб та записи в журналах подій. Незважаючи на багатство можливостей, архітектура WMI має суттєві недоліки при спробах її масштабування. Процес виконання WMI-запитів є ресурсомістким і створює помітне навантаження на центральний процесор цільової машини, особливо при частому опитуванні. Більше того, мережева взаємодія WMI базується на технології DCOM (Distributed Component Object Model), яка використовує динамічне призначення портів. Це значно ускладнює налаштування міжмережевих екранів, вимагаючи відкриття широкого діапазону RPC-портів (від 49152 до 65535 у сучасних ОС), що може бути недоречним з точки зору корпоративної мережевої безпеки, зокрема для контролерів домену.

Ще одним класичним протоколом, призначеним переважно для централізованого збору журналів подій, є Syslog. Він набув статусу індустріального стандарту в UNIX-подібних системах і широко підтримується мережевим обладнанням різних виробників. Syslog забезпечує простий механізм відправлення текстових повідомлень на центральний сервер (Syslog daemon), де вони можуть зберігатися або передаватися для подальшого аналізу. Низьке споживання ресурсів та простота реалізації є головними перевагами цього протоколу. Однак базова специфікація Syslog (RFC 3164) не передбачає

механізмів гарантованої доставки чи шифрування даних. Хоча сучасні реалізації підтримують передачу по TCP з використанням TLS (RFC 5424), сам формат повідомлень Syslog залишається слабко структурованим. Це суттєво ускладнює їх автоматизований машинний аналіз, вимагаючи написання складних регулярних виразів для вилучення корисних полів на стороні сервера, що сповільнює обробку великих потоків даних у реальному часі.

Підсумовуючи огляд традиційних протоколів, можна зробити висновок, що вони залишаються хорошими інструментами для локального моніторингу або управління ізольованими сегментами мережі. Однак їхні архітектурні обмеження — відсутність вбудованої гарантії доставки (SNMP/UDP, базовий Syslog), високе споживання ресурсів і проблеми з маршрутизацією через брандмауери (WMI), а також слабка структурованість даних (Syslog) — роблять їх неоптимальними кандидатами для побудови надійного, безпечного та продуктивного каналу передачі телеметрії в сучасних системах, які вимагають гарантованого та швидкого обміну подіями безпеки.

З розвитком розподілених обчислень та мікросервісних архітектур на зміну традиційним мережевим протоколам прийшли сучасні підходи до міжсервісної взаємодії. Тривалий час стандартом де-факто для побудови таких систем залишався **REST**, який переважно використовує протокол HTTP першої версії. Його популярність зумовлена високою універсальністю, зрозумілою семантикою методів та простотою інтеграції з більшістю існуючих платформ. Передача даних у REST API найчастіше відбувається у текстовому форматі JSON, що робить повідомлення легко читабельними для людини та спрощує процес розробки. Проте в умовах високонавантажених систем моніторингу цей підхід виявляє помітні архітектурні недоліки. Текстова природа JSON призводить до збільшення обсягу корисного навантаження, оскільки кожне повідомлення містить повторювані текстові ключі та структурну розмітку. Крім того, процеси серіалізації та десеріалізації масивів текстових даних вимагають відчутних витрат процесорного часу на обох кінцях мережевого з'єднання. Додатковим обмежувальним фактором особливості роботи HTTP/1.1, який для

обробки великої кількості одночасних запитів вимагає відкриття паралельних TCP-з'єднань або страждає від проблеми блокування початку черги, коли один "важкий" запит затримує виконання наступних.

Для розв'язання цих інженерних завдань було розроблено технологію **gRPC**, яка пропонує оптимізований підхід до віддаленого виклику процедур. Замість текстових форматів обміну даними gRPC за замовчуванням використовує бінарний протокол Protobuf. Ця технологія базується на попередньому описі структури даних у вигляді чітких контрактів, які компілюються у машинний код для цільової мови програмування. Завдяки цьому серіалізоване бінарне повідомлення містить переважно самі значення даних без метаінформації та текстових ключів, що суттєво зменшує його фізичний розмір [20]. Процес кодування та розкодування таких бінарних структур (рис. 1.1.) відбувається значно швидше за аналіз JSON. Це дозволяє суттєво вивільнити обчислювальні ресурси агента та сервера, що є важливою вимогою для систем моніторингу, де фонова служба має споживати мінімум ресурсів операційної системи і не заважати основним процесам сервера. Порівняння двох методів наведено в таблиці 1.1.

Таблиця 1.1. Порівняння JSON та Protocol Buffers.

Характеристика	**JSON (класичний REST API)**	**Protocol Buffers (gRPC)**
Формат представлення	Текстовий, зрозумілий для людини	Бінарний (оптимізований для машин)
Розмір корисного навантаження	Великий (через передачу назв ключів, пробілів та лапок у кожному пакеті)	Мінімальний (передаються лише значення та числові індекси полів)
Швидкість серіалізації/десеріалізації	Низька (вимагає тривалого парсингу рядків та виділення пам'яті під текст)	Надзвичайно висока (швидке перетворення структур у потік байтів)

Продовження таблиці 1.1. Порівняння JSON та Protocol Buffers.

Характеристика	**JSON (класичний REST API)**	**Protocol Buffers (gRPC)**
Схема даних (Контракт)	Відсутня або зовнішня (через інструменти на кшталт Swagger/OpenAPI)	Вбудована та обов'язкова (суворо описується у .proto файлах)
Споживання ресурсів процесора	Високе (через постійний аналіз текстових символів під час обробки запиту)	Мінімальне (пряме відображення послідовності байтів у пам'ять об'єктів C#)
Типізація даних	Динамічна (існує ризик невідповідності типів на клієнті та сервері)	Сувора (типи жорстко задані на етапі компіляції програмного коду)

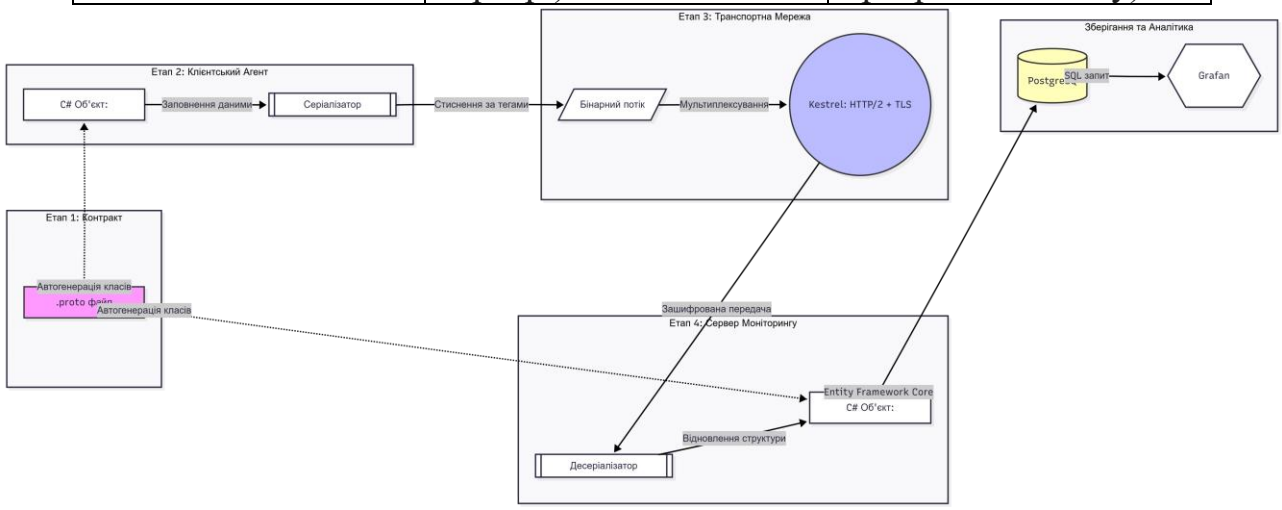


Рис. 1.1. Схема процесу серіалізації даних за допомогою Protocol Buffers та їх передача через мережу.

Однією з ключових технічних переваг gRPC є використання протоколу HTTP/2 як транспортного рівня. На відміну від попередньої версії, HTTP/2 підтримує повноцінне мультиплексування [21]. Це означає, що клієнт і сервер можуть паралельно обмінюватися безліччю незалежних запитів та відповідей через одне довготривале TCP-з'єднання. Протокол також забезпечує стиснення заголовків та бінарне форматування кадрів на рівні транспорту. Для системи моніторингу, яка працює за моделлю Push і має постійно відправляти невеликі пакети з апаратними метриками або сповіщеннями безпеки, така архітектура є найбільш раціональною. Агенту не потрібно витрачати час і ресурси на постійне

встановлення нових з'єднань або повторні процедури узгодження шифрування для кожної відправки даних, що мінімізує мережеві затримки та стабілізує потік телеметрії.

Порівнюючи обидва підходи, можна стверджувати, що REST API залишається прийнятним вибором для публічних інтерфейсів або систем, де пріоритетом є максимальна сумісність із різнорідними клієнтами та веб-браузерами. Однак для внутрішньої інфраструктурної комунікації між системними агентами збору даних та центральним сервером використання gRPC є більш обґрунтованим технічним рішенням. Поєднання бінарної серіалізації, строгої типізації контрактів та мультиплексування на рівні HTTP/2 забезпечує високу пропускну здатність та ефективне використання системних ресурсів. Це надає серверу можливість стабільно обробляти інтенсивні та часті потоки подій від значної кількості контролерів домену та локальних робочих станцій без деградації продуктивності мережевого стеку.

В умовах корпоративної інфраструктури, де центральним компонентом управління є каталог Active Directory, інформація, що передається від кінцевих вузлів до системи моніторингу, характеризується високим рівнем чутливості. Системні метрики, а особливо записи журналів безпеки, містять вичерпні дані про активність користувачів, спроби автентифікації, запуск специфічних процесів та загальну топологію мережі. Перехоплення такого масиву телеметрії зловмисником на етапі передачі дозволяє йому здійснити глибоку розвідку інфраструктури без безпосереднього сканування портів чи активної взаємодії з контролерами домену. З огляду на це, проєктування каналу зв'язку вимагає обов'язкового впровадження криптографічних механізмів, які здатні забезпечити як таємність самої інформації, так і гарантію того, що вона не була навмисно чи випадково модифікована під час транспортування [18].

Основним індустріальним стандартом для вирішення цього завдання є використання протоколу Transport Layer Security [19]. Застосування цього протоколу поверх традиційного мережевого з'єднання формує захищений тунель, у якому всі дані підлягають симетричному шифруванню. Це

ускладнюєчитання перехопленого трафіку сторонніми особами. Окрім забезпечення конфіденційності, протокол підтримує цілісність пакетів за допомогою кодів автентифікації повідомлень. Будь-яка спроба змінити вміст метрики чи підмінити ідентифікатор події в транзиті призведе до порушення криптографічного підпису, після чого сервер моніторингу просто відхилить такий пакет. Важливо зазначити, що сучасні технологічні стеки для міжсервісної взаємодії проєктуються з урахуванням тісної інтеграції шифрування, роблячи його невід'ємною частиною базової архітектури, а не додатковою надбудовою.

Для встановлення захищеного з'єднання ключовим етапом є перевірка справжності самого сервера моніторингу, що дозволяє запобігти атакам типу Man-in-the-Middle. У таких сценаріях зловмисник може спробувати перенаправити трафік агентів на власний підставний вузол для непомітного збору їхньої телеметрії. Захист від цієї загрози найчастіше реалізується через інфраструктуру відкритих ключів та використання цифрових сертифікатів стандарту X.509 [13]. Під час ініціалізації з'єднання сервер пред'являє свій сертифікат, а клієнтський додаток має підтвердити його легітимність. На практиці це може реалізовуватися шляхом перевірки ланцюжка довіри через внутрішній корпоративний центр сертифікації або шляхом перевірки конкретного криптографічного відбитка сертифіката. Перевірка за відбитком є досить дієвим методом для закритих інфраструктур, оскільки вона гарантує, що агент передаватиме чутливі дані лише тому серверу, який володіє визначеним закритим ключем, виключаючи можливість підміни вузла.

1.3. Огляд сучасних технологічних стеків для розробки систем моніторингу.

Розробка розподіленої системи моніторингу вимагає ретельного підходу до вибору інструментів. Оскільки компоненти такої системи мають безперервно функціонувати у фоновому режимі, збирати дані з глибоких рівнів операційної системи та підтримувати стабільний мережевий зв'язок, обраний технологічний стек безпосередньо визначає надійність, масштабованість та кросплатформність готового рішення.

Вибір мови програмування для розробки агентів збору даних та центрального сервера зводиться до пошуку оптимального балансу між продуктивністю, безпекою управління пам'яттю та швидкістю реалізації логіки. Історично для створення системних фонових служб найчастіше використовувалася мова C++. Вона забезпечує прямий доступ до апаратних ресурсів та системних викликів, що дозволяє створювати швидкі програми. Проте відсутність автоматичного збирання сміття вимагає від розробника ручного управління виділенням та звільненням пам'яті. У контексті довготривалих фонових процесів будь-яка помилка в роботі з вказівниками може призвести до поступового витоку пам'яті або критичного збою сегментації, що ускладнює розробку. Сучасною альтернативою у ніші системного програмування є мова Rust, яка розв'язує проблему безпеки пам'яті на етапі компіляції завдяки строгій моделі запозичення та володіння. Rust гарантує відсутність витоків пам'яті та станів гонитви потоків без використання збирача сміття, зберігаючи при цьому продуктивність на рівні C++. Однак архітектурна складність цієї мови та її стрімка крива навчання суттєво збільшують час розробки, особливо при реалізації високорівневої мережевої взаємодії та інтеграції з базами даних.

У сфері розробки хмарної інфраструктури значної популярності набула мова Go. Вона пропонує вбудований збирач сміття та надзвичайно ефективну модель багатопотоковості, що робить її вдалим вибором для обробки великої кількості одночасних мережевих з'єднань. Тим не менш, при створенні агентів для моніторингу гетерогенних середовищ мова Go виявляє певні обмеження в частині глибокої інтеграції зі специфічними підсистемами Windows. Робота з лічильниками продуктивності або журналами подій безпеки часто вимагає використання додаткових обгорток навколо нативного коду, що ускладнює підтримку проєкту.

На тлі розглянутих альтернатив платформа .NET та мова програмування C# пропонують найбільш збалансований набір характеристик для розробки подібних систем [1]. Використання керованого середовища виконання з

автоматичним збиранням сміття практично усуває ризики витоків пам'яті під час штатної роботи програми. Для забезпечення високої пропускної здатності сервера C# надає розвинену парадигму асинхронного програмування та бібліотеку паралельних задач. Це дозволяє ефективно обробляти тисячі одночасних вхідних підключень через gRPC, не блокуючи основні потоки виконання операційної системи, що гарантує стабільну роботу центрального компонента навіть під час пікових навантажень або масового надходження сповіщень про інциденти [17].

Важливою архітектурною перевагою сучасної екосистеми .NET є шаблон Worker Services, спеціально розроблений для створення довготривалих фонових процесів. Цей механізм інкапсулює логіку життєвого циклу служби, маршрутизації журналювання та впровадження залежностей. Що ще більш критично для сучасних інфраструктур — поточні версії .NET є повністю кросплатформними [15]. Це означає, що єдина кодова база агента може бути скомпільована та розгорнута як нативна служба Windows, так і як демон у середовищі Linux. Такий рівень універсалізації значно скорочує витрати на розробку та подальший супровід клієнтського програмного забезпечення, забезпечуючи при цьому нативну швидкість роботи на обох платформах.

Після вирішення завдань зі збору та безпечного транспортування телеметрії постає проблема її ефективного та надійного збереження на стороні центрального сервера. Дані інфраструктурного моніторингу зазвичай поділяються на дві категорії, кожна з яких висуває свої вимоги до системи управління базами даних. Перша категорія — це часові ряди, тобто безперервний потік числових показників навантаження на процесор, споживання оперативної пам'яті чи обсягу вільного дискового простору. Друга категорія — це дискретні структуровані події, такі як записи з журналів безпеки Active Directory, сповіщення про зупинку критичних служб або спроби несанкціонованого доступу.

Для роботи з першою категорією даних індустрія часто пропонує використовувати спеціалізовані Time-Series бази даних, яскравим

представником яких є InfluxDB. Її внутрішня архітектура оптимізована для максимально швидкого запису великих масивів числових значень із часовими мітками. Крім того, подібні системи мають вбудовані механізми автоматичного агрегування та видалення застарілих даних, що значно спрощує управління дисковим простором. Проте, незважаючи на високу продуктивність при роботі з метриками, їхнє використання для зберігання складних подій кібербезпеки є недоцільним. Time-Series бази зазвичай не підтримують повноцінних реляційних зв'язків та транзакцій, що ускладнює побудову комплексних аналітичних запитів, необхідних для розслідування багатоетапних інцидентів у доменній мережі.

Альтернативним підходом є застосування документо-орієнтованих NoSQL рішень, таких як MongoDB. Вони приваблюють розробників своєю гнучкістю та відсутністю жорсткої схеми даних. Це дозволяє легко зберігати гетерогенні журнали подій (наприклад, у форматі BSON), де кожен запис може мати унікальний набір полів залежно від типу операційної системи чи версії агента. З іншого боку, така гнучкість приховує ризики для цілісності інформації. У системах моніторингу безпеки критично важливо гарантувати, що кожна зафіксована подія має чітко визначений і незмінний набір обов'язкових атрибутів (ідентифікатор вузла, точний час, рівень критичності). Реляційні системи справляються з цим контролем значно надійніше на рівні архітектури самих таблиць.

Враховуючи гібридний характер навантаження, де система моніторингу має одночасно обробляти інтенсивний потік числової телеметрії та гарантовано зберігати критичні сповіщення безпеки, найбільш обґрунтованим інженерним рішенням виступає об'єктно-реляційна система управління базами даних PostgreSQL. Ця система забезпечує повну відповідність принципам транзакційної цілісності. У контексті моніторингу Active Directory це означає, що кожне сповіщення про зміну політик або зупинку інфраструктурної служби буде надійно збережене на жорсткому диску без ризику втрати чи пошкодження інформації під час паралельних мережевих записів. Висока продуктивність при

одночасній роботі великої кількості клієнтських агентів досягається завдяки використанню вбудованого механізму багатоверсійного управління конкурентним доступом. Цей архітектурний підхід дозволяє серверу моніторингу безперервно записувати нові масиви метрик, не блокуючи при цьому операції читання, які можуть виконуватися аналітиками або зовнішніми системами для побудови звітів [6].

Окрім класичної реляційної моделі, яка добре підходить для строго структурованих таблиць сповіщень та базових показників продуктивності, сучасні версії PostgreSQL пропонують потужний інструментарій для роботи зі слабко структурованою інформацією. Зокрема, підтримка спеціалізованого формату об'єктів формату JSON дозволяє ефективно зберігати та обробляти складні структури журналів операційної системи Windows. База даних здатна індексувати вкладені поля таких об'єктів, що забезпечує швидкий повнотекстовий пошук та фільтрацію специфічних атрибутів подій без необхідності нормалізації всієї бази. Для управління зростаючими обсягами історичної телеметрії адміністратори можуть використовувати механізми декларативного секціонування таблиць. Такий підхід дозволяє розбивати єдину велику таблицю метрик на менші логічні сегменти за часовими інтервалами, що суттєво пришвидшує виконання аналітичних запитів та спрощує процеси архівування або видалення застарілої інформації.

Для забезпечення високорівневої взаємодії між серверною частиною на платформі .NET та базою даних PostgreSQL існує об'єктно-реляційне відображення за допомогою фреймворку Entity Framework Core [4]. Цей інструмент дозволяє розробнику оперувати поняттями мови C# замість прямого написання сирих SQL-запитів, що значно прискорює розробку та знижує ймовірність помилок при типізації даних. У контексті збереження метрик та алертів, фреймворк автоматично транслює об'єкти моделей у відповідні рядки таблиць PostgreSQL за допомогою оптимізованого провайдера Npgsql [5]. Важливою перевагою такого підходу є підтримка механізму міграцій, який дозволяє гнучко керувати схемою бази даних та автоматично розгортати

необхідні структури таблиць безпосередньо під час запуску фонової служби сервера.

Додатковим фактором на користь вибору PostgreSQL є глибока екосистема драйверів та інструментів інтеграції. Високопродуктивна взаємодія між розробленим сервером на базі платформи .NET та базою даних реалізується через спеціалізовані оптимізовані провайдери, які забезпечують асинхронне виконання запитів з малими накладними витратами оперативної пам'яті. Важливою архітектурною перевагою використання цієї бази даних як центрального сховища є її повна сумісність із провідними системами візуалізації даних. Завдяки наявності стандартизованих конекторів та оптимізації запитів, зібрані метрики та події безпеки можуть бути легко інтегровані з відкритою аналітичною платформою Grafana. Вона дозволяє будувати динамічні інформаційні панелі в режимі реального часу, візуалізуючи масиви даних з PostgreSQL. Основним завданням цього інструменту є перетворення сирих масивів телеметрії, що зберігаються в центральній базі даних, на зручні для аналізу графіки, діаграми та комплексні дашборди. Завдяки наявності офіційного конектора для PostgreSQL, платформа дозволяє виконувати прямі запити до сховища без необхідності розробки додаткових проміжних інтерфейсів [7]. Це дає змогу системним адміністраторам формувати єдині інформаційні панелі, на яких паралельно відображається динаміка споживання апаратних ресурсів машинами та хронологія виникнення інцидентів безпеки в середовищі Active Directory. Гнучкість налаштування панелей та підтримка автоматичного оновлення показників забезпечують можливість ефективного спостереження за станом інфраструктури.

Заключним, але критично важливим компонентом архітектури будь-якої системи інфраструктурного моніторингу є механізм доставки сповіщень про інциденти відповідальному персоналу. Збір метрик та збереження подій у базі даних втрачають свою практичну цінність, якщо система не здатна оперативно привернути увагу адміністратора до аномалії. Тривалий час індустріальним стандартом та базовим методом оповіщення для більшості програмних продуктів

залишалася електронна пошта. Проте в реаліях сучасних високодинамічних корпоративних середовищ цей підхід продемонстрував свою недостатню ефективність для завдань оперативного реагування.

Основна проблема класичних поштових сповіщень полягає у явищі інформаційного перевантаження. У типовому робочому процесі системного адміністратора або фахівця з інформаційної безпеки поштова скринька заповнена десятками рутинних листів, автоматичних щоденних звітів, запитів від користувачів та організаційних розсилок. У такому щільному потоці неструктурованої інформації критично важливе сповіщення про зупинку служби каталогу або спробу несанкціонованої зміни політик аудиту може годинами залишатися непоміченим. Більше того, поштові клієнти часто автоматично групують подібні листи або відправляють їх у другорядні папки, формуючи у персоналу звичку ігнорувати цей канал зв'язку. Додатково варто зазначити і технічні обмеження: протоколи маршрутизації електронної пошти за своєю природою не гарантують миттєвої доставки повідомлення, що вносить затримки у процес локалізації кіберзагроз.

З огляду на описані недоліки, сучасним еталоном побудови модулів оповіщення є використання подієво-орієнтованої архітектури на базі технології вебхуків [14]. Цей механізм дозволяє центральному серверу моніторингу виконувати роль активного відправника і генерувати миттєві HTTP-запити до зовнішніх сервісів рівно в той момент, коли в базі даних фіксується порушення заданих правил або критичний інцидент. Такий підхід відкриває можливості для безшовної інтеграції з корпоративними месенджерами та платформами для командної роботи, серед яких одним із найпоширеніших рішень є Slack.

Переваги маршрутизації алертів через вебхуки у корпоративні месенджери мають фундаментальний характер. Насамперед це забезпечує гарантовану та миттєву доставку push-сповіщень безпосередньо на робочі станції та мобільні пристрої інженерів. Замість того, щоб змішувати критичні події з рутинною поштою, архітектура месенджерів дозволяє створювати ізольовані тематичні канали для конкретних типів інцидентів. Наприклад, події щодо падіння служб

спрямовуються команді інфраструктури, а попередження про підозрілі входи в Active Directory — у закритий канал служби безпеки. Найважливішим організаційним наслідком такої інтеграції є трансформація процесу реагування: як тільки сповіщення надходить у канал, команда отримує єдиний контекст проблеми і може розпочати координацію дій, обговорення та спільне розслідування інциденту в тому ж самому комунікаційному вікні. Це скорочує час між виявленням аномалії та початком її усунення.

Висновки до першого розділу

У першому розділі кваліфікаційної роботи було проведено аналіз проблематики моніторингу телеметрії та подій безпеки у сучасних корпоративних мережах. Визначено, що традиційні підходи до збору інформації, засновані на механізмах постійного опитування та використанні важких протоколів, є неефективними і створюють значне навантаження на кінцеві комп'ютери. Доведено необхідність переходу до подієво-орієнтованої архітектури для аналізу журналів Active Directory та використання низькорівневих системних викликів для збору апаратної телеметрії.

На основі порівняльного аналізу протоколів мережевої взаємодії було обґрунтовано вибір фреймворку gRPC та технології бінарної серіалізації Protocol Buffers. Це рішення дозволяє оптимізувати пропускну здатність мережі та забезпечити надшвидку передачу даних. Обраний технологічний стек, що включає платформу .NET 8, мову програмування C#, систему управління базами даних PostgreSQL та інструмент об'єктно-реляційного відображення Entity Framework Core, повністю відповідає вимогам до створення високонавантажених кросплатформних сервісів. Інтеграція з платформою візуалізації Grafana та корпоративним месенджером Slack забезпечує повноцінний цикл відслідковування та реагування на інциденти. Отримані теоретичні результати формують надійний фундамент для переходу до наступного етапу — практичного проєктування архітектури та інфраструктури безпеки системи моніторингу.

РОЗДІЛ 2. ПРОЄКТУВАННЯ СИСТЕМИ МОНІТОРИНГУ

2.1. Проєктування клієнт-серверної топології та структури додатків

Перехід від аналітичного етапу до безпосередньої програмної реалізації вимагає побудови ізольованого тестового середовища, яке б максимально точно імітувало інфраструктуру реального підприємства. Спроєктована тестова мережа базується на зіркоподібній фізичній топології, де всі комп'ютери об'єднані єдиним мережевим комутатором. Логічна архітектура стенда (рис. 2.1) включає три незалежні операційні системи, кожна з яких виконує строго визначену роль у процесі генерації, збору та обробки телеметричних даних. Головною особливістю спроєктованої топології є максимальна централізація серверних компонентів на одному високопродуктивному вузлі, що дозволяє спростити процес адміністрування та мінімізувати затримки під час обміну даними між базою даних та інструментами візуалізації.

Центром логічної архітектури виступає комп'ютер під управлінням операційної системи Windows Server. У межах даного проєкту цей вузол акумулює в собі як базові інфраструктурні, так і аналітичні функції. Насамперед він виконує роль головного контролера домену, забезпечуючи роботу служби каталогів Active Directory та супутніх компонентів авторизації. Одночасно з цим на даному сервері розгорнуто весь бекенд розробленої системи моніторингу. Тут функціонує додаток *MonitoringServer*, який прослуховує мережевий порт за допомогою веб-сервера Kestrel та приймає вхідні з'єднання від усіх агентів. На цьому ж фізичному накопичувачі розміщено реляційну базу даних PostgreSQL, куди миттєво записується вся отримана телеметрія. Для забезпечення візуального контролю на сервері розгорнуто аналітичну платформу Grafana, яка локально підключається до бази даних. Крім того, на цьому вузлі працює примірник клієнтського агента, який налаштований на максимальний рівень збору інформації: він контролює стан власних апаратних ресурсів сервера та безперервно аналізує локальні журнали безпеки домену.

Другим компонентом тестової мережі є типова робоча станція під управлінням десктопної операційної системи Windows 10. Цей комп'ютер

введено у домен, що дозволяє застосовувати до нього глобальні групові політики, зокрема для автоматичного поширення сертифікатів довіри. Роль цього вузла полягає в імітації активності звичайного користувача корпоративної мережі. На робочій станції встановлено фонову службу клієнтського агента, яка конфігурується на читання базових показників навантаження процесора, обсягу оперативної пам'яті та дискового простору. Також агент відслідковує локальні події безпеки, наприклад, спроби запуску заборонених процесів, і передає ці інциденти на центральний Windows Server.

Третім вузлом інфраструктури є комп'ютер під управлінням відкритої операційної системи Ubuntu Linux. Його наявність у топології переслідує єдину мету — практичну демонстрацію та тестування кросплатформних можливостей розробленого агента. На відміну від інших пристроїв у мережі, цей сервер не генерує подій безпеки у форматі Event Log і не підпорядковується контролеру домену. На ньому розгорнуто виключно скомпільований для середовища Linux клієнтський додаток моніторингу. Під час запуску агент ідентифікує тип операційної системи та автоматично активує модулі зчитування системних файлів ядра для отримання актуальної телеметрії, після чого відправляє зібрані пакети на центральний сервер. Така конфігурація стенда дозволяє комплексно перевірити здатність системи одночасно обробляти потоки даних від комп'ютерів з абсолютно різною архітектурою.

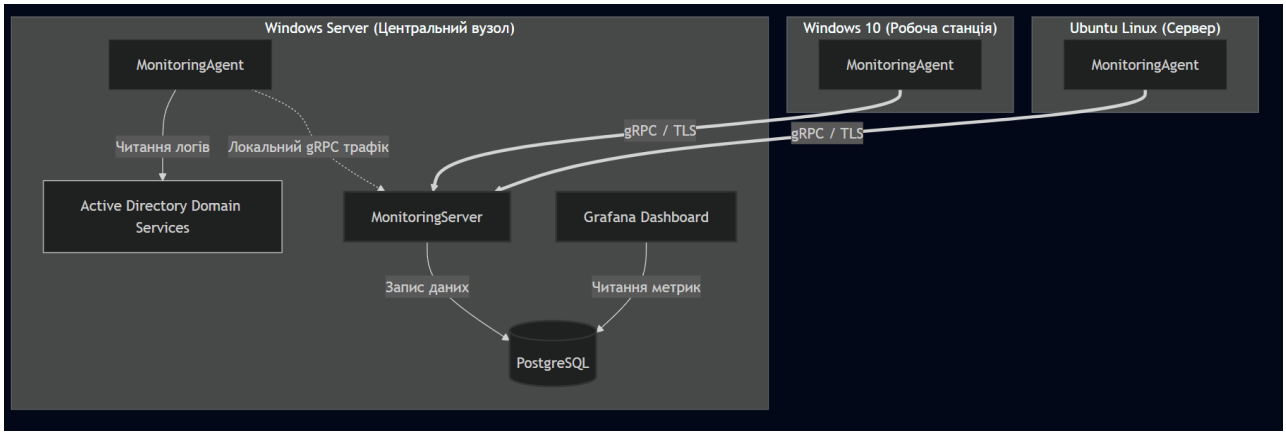


Рис. 2.1. Логічна топологія та розподіл компонентів тестового стенда.

Програмна архітектура клієнтського додатку проєктувалася з урахуванням сучасних вимог до розробки системних сервісів, де ключовим принципом є

чіткий розподіл відповідальності між компонентами. Замість створення монолітного алгоритму, який би послідовно збирав різні типи телеметрії, агент побудований як контейнер для запуску незалежних фонових задач. Фундаментом додатку виступає універсальний хост, який ініціалізується за допомогою стандартних механізмів платформи. Під час запуску процесу конфігуратор хоста зчитує локальні налаштування, реєструє сервіс для роботи як служби операційної системи та налаштовує систему впровадження залежностей. Залежно від ролі кінцевого комп'ютера, яка визначається параметром конфігурації, хост динамічно додає в конвеєр виконання відповідні класи-воркери. Усі ці класи успадковуються від базового абстрактного класу BackgroundService, що дозволяє їм функціонувати паралельно та асинхронно у єдиному адресному просторі процесу.

Першим і найголовнішим компонентом клієнтського агента є клас збору телеметрії *MetricsWorker*. Призначення цього модуля полягає у циклічному зчитуванні апаратних метрик комп'ютера та їхній відправці на центральний сервер. У методі виконання цього класу реалізовано безкінечний цикл, який зупиняється лише за сигналом токена скасування при завершенні роботи служби. Кожну ітерацію циклу воркер викликає набір спеціалізованих методів для отримання поточного завантаження центрального процесора, обсягу доступної оперативної пам'яті, вільного дискового простору та інтенсивності мережевого трафіку. Зібрані показники формуються у структурований запит і передаються через попередньо відкритий канал зв'язку. Важливою архітектурною особливістю цього модуля є те, що мережевий канал створюється лише один раз під час ініціалізації класу, що кардинально зменшує витрати обчислювальних ресурсів на постійне встановлення нових з'єднань під час кожної відправки даних.

Другим критично важливим модулем агента є клас *EventWorker*, який кардинально відрізняється за логікою своєї роботи від попереднього компонента. Якщо модуль телеметрії працює за таймером, то цей клас реалізує виключно подієво-орієнтовану модель. Його архітектурна роль полягає у моніторингу

локального журналу безпеки операційної системи. Під час свого запуску цей воркер формує жорсткий запит для фільтрації подій, створює об'єкт спостерігача і підписується на системні переривання про запис нових логів. Метод обробки подій активується миттєво при появі інциденту, перевіряючи ідентифікатор запису.

Для комп'ютерів, які виконують роль контролерів домену, архітектурою передбачено активацію третього, специфічного модуля *AddsWorker*. Завданням цього класу є безперервний контроль за життєздатністю критичних інфраструктурних служб каталогу. На відміну від стандартного збору телеметрії, цей воркер реалізує концепцію стейтфул-моніторингу, тобто моніторингу зі збереженням стану. Всередині класу ініціалізовано спеціальну хеш-таблицю, яка виконує роль оперативної пам'яті для фіксації інцидентів. Під час циклічної перевірки статусу служб воркер не просто фіксує їхнє падіння, а перевіряє, чи було вже відправлено сповіщення про цей збій раніше. Сповіщення генерується і відправляється лише у момент зміни стану служби з активного на неактивний, після чого проблемна служба додається до хеш-таблиці. Коли алгоритм фіксує відновлення роботи служби, вона автоматично видаляється з пам'яті інцидентів.

Така багатомодульна архітектура клієнтського агента забезпечує відмовостійкість розробленого програмного рішення. Якщо під час читання специфічних системних журналів виникне помилка доступу, або один із мережевих запитів буде відхилено сервером, відповідний воркер зможе безпечно обробити виняток, не впливаючи на роботу інших паралельних процесів. Агент продовжить збирати апаратну телеметрію навіть у разі критичних збоїв у підсистемі журналювання операційної системи, гарантуючи часткову видимість стану вузла за будь-яких умов.

Архітектура центрального серверного вузла проєктувалася з фокусом на високу пропускну здатність та здатність одночасно обробляти значну кількість паралельних з'єднань від клієнтських агентів без деградації продуктивності. Фундаментом серверної частини виступає кросплатформний веб-сервер Kestrel, який ініціалізується в межах хостингової моделі сучасного додатку [10].

Програмний код сервера явно перевизначає стандартні налаштування цього мережевого компонента, вказуючи прослуховування заданого порту виключно за протоколом передачі гіпертексту другої версії. Така жорстка конфігурація транспортного рівня є абсолютною вимогою для коректного функціонування фреймворку віддаленого виклику процедур. Для забезпечення автономності розгортання в корпоративному середовищі, архітектура передбачає реєстрацію серверного додатку як повноцінної системної служби операційної системи Windows. Це гарантує автоматичний запуск процесу збору телеметрії відразу після перезавантаження комп'ютера без необхідності ручного втручання чи авторизації системного адміністратора.

Наступною важливою складовою серверної архітектури є використання вбудованого механізму впровадження залежностей. Під час фази конфігурування додаток реєструє всі необхідні для своєї роботи внутрішні сервіси та параметри. Ключовим елементом цього етапу є підключення контексту бази даних, який налаштовується на використання драйвера реляційної системи PostgreSQL. Замість жорсткого кодування параметрів підключення у самому коді, сервер динамічно зчитує їх із зовнішніх конфігураційних джерел, забезпечуючи високий рівень гнучкості. Окремою архітектурною перевагою є імплементований механізм автоматичного застосування міграцій. Під час кожного запуску серверний додаток створює ізольовану область видимості та перевіряє актуальність структури таблиць у базі даних, автоматично оновлюючи її до останньої версії. Цей підхід повністю усуває проблему розсинхронізації між програмними моделями та фізичною структурою бази при розгортанні оновлень системи.

Безпосередня обробка вхідного мережевого трафіку реалізована через спеціалізований клас сервісу, який успадковує автоматично згенерований базовий клас контрактів взаємодії. Першою точкою входу є метод приймання телеметрії. Коли сервер отримує бінарний пакет від клієнтського агента, інфраструктура фреймворку автоматично десеріалізує його у строго типізований об'єкт запиту. Завданням серверного методу є витягнення цих числових

показників навантаження, створення екземпляра сутності бази даних та асинхронне збереження нового запису до відповідної таблиці через інструментарій об'єктно-реляційного відображення.

Другою ключовою точкою входу є метод обробки сповіщень (alerts), який реалізує дещо складнішу логіку. Отримавши від агента інформацію про критичний інцидент безпеки або зупинку служби, сервер насамперед фіксує його у базі даних для збереження довгострокової історичної звітності. Після успішного підтвердження транзакції запису активується підсистема зовнішніх комунікацій. Сервер перевіряє наявність валідного посилання цільового вебхука у своїх конфігураційних параметрах. У разі його наявності формується структуроване текстове повідомлення, яке містить назву проблемного комп'ютера та деталі інциденту. Це повідомлення серіалізується і за допомогою асинхронного клієнта відправляється HTTP-запитом до стороннього месенджера Slack. Завдяки такій архітектурі центральний вузол успішно виконує роль не лише пасивного сховища даних, але й проактивного інтеграційного центру, що оперативно реагує на загрози інфраструктурі підприємства.

2.2. Проєктування моделі даних та контрактів взаємодії

У процесі створення розподілених клієнт-серверних систем одним із найважливіших етапів є проєктування єдиної моделі даних та механізмів їх передачі мережею. Відсутність жорстких правил обміну інформацією між компонентами часто призводить до помилок типізації, неможливості коректної десеріалізації пакетів та загальної нестабільності програмного комплексу. Для вирішення цієї проблеми у розроблюваній системі було застосовано підхід проєктування на основі контрактів. Фундаментом мережевої взаємодії виступає спеціальний конфігураційний файл формату Protocol Buffers третьої версії. Цей файл є джерелом інформації про структуру мережевих повідомлень та доступні серверні методи для всіх клієнтських агентів, незалежно від їхньої операційної системи чи апаратної платформи. Під час компіляції проєкту компілятор фреймворку автоматично генерує на основі цього файлу базові класи як для

серверної, так і для клієнтської частини, що повністю виключає людський фактор при формуванні мережевих запитів.

Спроєктований контракт містить визначення єдиного глобального сервісу системного моніторингу (SystemMonitor), який надає агентам дві основні віддалені процедури. Перша процедура призначена для регулярного відправлення телеметрії, а друга спеціалізується на передачі асинхронних сповіщень. Для кожної з цих процедур спроєктовано власні структури повідомлень (рис. 2.2.). Запит на відправку метрик (MetricsRequest) містить рядковий ідентифікатор комп'ютера та набір числових показників подвійної точності. Використання двох знаків після коми для передачі відсотка завантаження процесора, обсягу доступної оперативної пам'яті, вільного дискового простору, часу безперервної роботи та швидкості мережевого трафіку дозволяє зберегти високу точність показників для подальшого аналізу. Бінарна серіалізація цих числових значень гарантує, що мережевий пакет матиме мінімально можливий розмір, економлячи пропускну здатність корпоративної мережі.

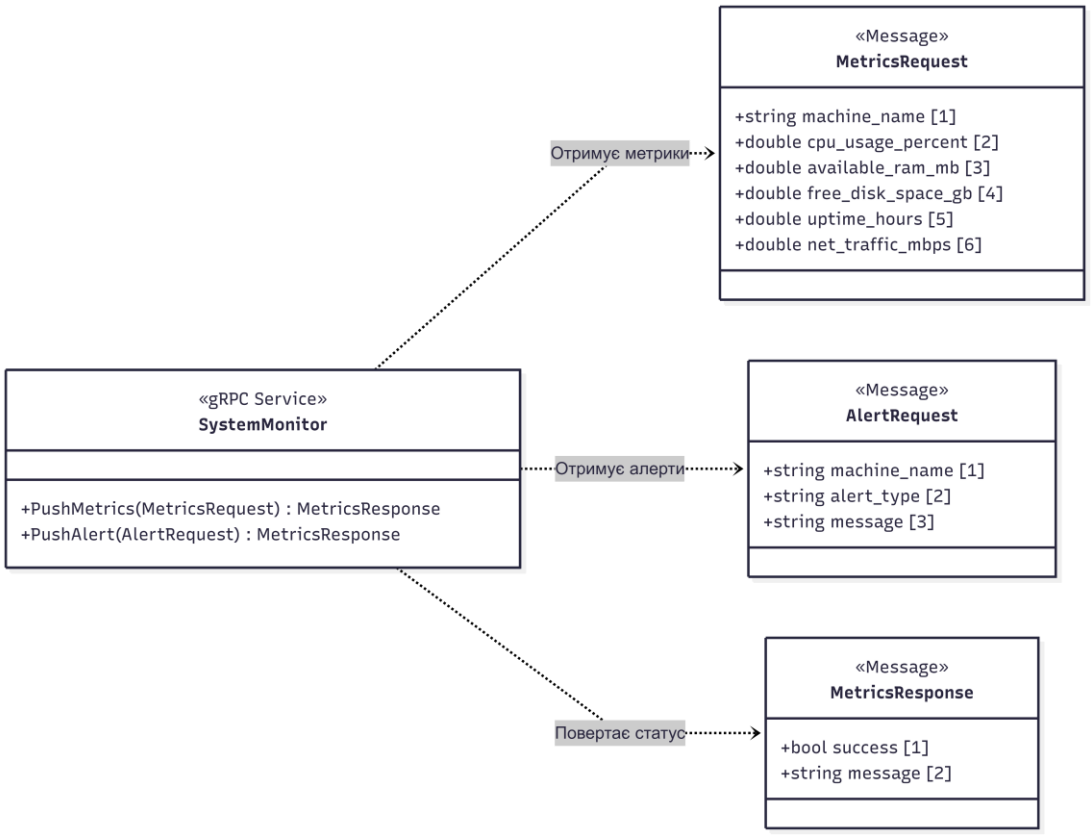


Рис. 2.2. Структура мережевих повідомлень.

Окремої уваги заслуговує структура повідомлення для передачі сповіщень про інциденти (AlertRequest). Цей запит є лаконічним і оперує переважно рядковими типами даних. Окрім обов'язкової назви проблемного вузла, пакет містить категорію інциденту та розгорнуте текстове повідомлення з деталями помилки. Наприклад, у це поле записується інформація про те, яка саме служба зупинилася, або назва забороненого процесу, який був зафіксований агентом. Для обох віддалених процедур передбачено єдиний формат серверної відповіді, що містить логічний прапорець успішності операції та текстове повідомлення від сервера. Це дозволяє клієнтському агенту розуміти, чи були дані коректно збережені, і у разі помилки ініціювати алгоритм повторної відправки пакета через визначений проміжок часу.

Після успішного отримання та десеріалізації мережевого пакету центральним сервером виникає необхідність його надійного збереження для подальшої обробки та візуалізації. Логічна структура бази даних PostgreSQL спроєктована у повній відповідності до сутностей програмного коду за допомогою технології об'єктно-реляційного відображення. База даних складається з двох основних незалежних таблиць, які відповідають за збереження телеметрії та реєстрацію інцидентів відповідно. Таблиця для збереження метрик містить колонки, які повністю дублюють структуру мережевого запиту, але додатково включають автоматично згенерований унікальний ідентифікатор та позначку світового часу в момент отримання пакета сервером. Наявність поля часу є критично важливою, оскільки саме за цим полем аналітична платформа будує хронологічні графіки навантаження.

Таблиця реєстрації сповіщень спроєктована за аналогічним принципом, але орієнтована на збереження текстової інформації про інциденти безпеки та апаратні збої. Кожен запис у цій таблиці містить час настання події, назву комп'ютера, категорію інциденту та його детальний опис. Такий підхід до проєктування бази даних створює надійний історичний архів стану всієї корпоративної мережі. У разі виникнення серйозного інциденту безпеки системний адміністратор має можливість звернутися до цієї таблиці і провести

ретроспективний аудит, проаналізувавши хронологію подій та виявивши початкову точку компрометації системи або апаратного збою обладнання.

2.3. Алгоритмічне забезпечення обробки телеметрії та інцидентів

Класичні системи моніторингу часто використовують дуже прямолінійний підхід до аналізу апаратного навантаження. Якщо налаштувати сервер на відправку сповіщення щоразу, коли завантаженість процесора перевищує дев'яносто відсотків, системний адміністратор почне отримувати сотні хибних спрацьовувань щодня. Будь-який запуск важкої програми, встановлення оновлень операційної системи або побудова складного звіту в базі даних викликає короткочасний сплеск навантаження. Це є абсолютно нормальною поведінкою комп'ютера. Щоб навчити розроблену систему відрізняти такі безпечні сплески активності від реального "зависання" або нестачі обчислювальних потужностей, було імплементовано алгоритм евристичного аналізу на базі математичної моделі ковзного вікна.

Програмна реалізація цього алгоритму всередині клієнтського агента спирається на використання класичної структури даних типу черга. Оскільки агент збирає показники процесора кожні десять секунд, зберігати всю історію за день не має сенсу, адже це призведе до безконтрольного споживання оперативної пам'яті самим моніторингом. Тому алгоритм створює обмежену чергу, розмір якої динамічно вираховується залежно від заданого інтервалу опитування. Для забезпечення оптимальної точності було визначено, що репрезентативною вибіркою є історія навантаження за останні дві хвилини. При десятисекундному інтервалі алгоритм резервує в пам'яті місце рівно для дванадцяти останніх показників процесора.

Логіка роботи цього ковзного вікна базується на принципі "першим прийшов — першим вийшов". Щоразу, коли агент зчитує новий відсоток навантаження, він додає це значення в кінець своєї черги. Якщо черга вже заповнена до свого максимуму, алгоритм автоматично видаляє найстаріший запис. Таким чином, система завжди має актуальний зріз стану комп'ютера за останні сто двадцять секунд. Лише після повного заповнення цієї вибірки

алгоритм починає свою аналітичну роботу. Він вираховує середнє арифметичне значення всіх елементів у пам'яті. Якщо це середнє значення за дві хвилини стабільно тримається на рівні дев'яноста відсотків або вище, алгоритм робить висновок, що це не випадковий стрибок, а критичне перевантаження сервера, яке вимагає негайного втручання.

Однак проста фіксація перевантаження створює іншу інженерну проблему. Якщо процесор продовжує "висіти" на ста відсотках, агент почне генерувати нове сповіщення під час кожної наступної десятисекундної перевірки. Щоб уникнути такого шторму повідомлень, в алгоритм було вбудовано спеціальний механізм охолодження. У пам'яті агента зберігається точний час відправки останнього сповіщення. Коли алгоритм фіксує перевантаження, він спочатку порівнює поточний час із часом останнього відправленого сповіщення. Якщо з моменту попереднього сповіщення пройшло менше п'яти хвилин, система тихо ігнорує інцидент. Тривожний сигнал буде відправлено знову лише в тому випадку, якщо після повноцінної п'ятихвилинної паузи сервер все ще залишатиметься в стані критичного навантаження. Такий підхід робить моніторинг більш збалансованим і підвищує високу цінність кожного отриманого сповіщення.

Контролер домену є серцем корпоративної мережі, і його працездатність залежить від кількох критично важливих служб. До таких системних процесів належать ядро каталогу NTDS, центр розподілу ключів Kdc, служба локатора Netlogon та локальний DNS-сервер. Зупинка будь-якої з цих служб означає зупинку авторизації користувачів усій компанії. Відповідно, агент моніторингу повинен вміти швидко фіксувати такі збої. Проте використання класичного підходу без збереження стану тут є неприйнятним. Якщо агент просто перевірятиме статус служби кожні тридцять секунд і відправлятиме тривогу при виявленні проблеми, то у випадку реальної аварії системний адміністратор отримає понад сто однакових повідомлень за одну годину. Це призведе до ігнорування сповіщень.

Для вирішення цієї проблеми розроблено алгоритм моніторингу зі збереженням стану, який наділяє клієнтського агента власною оперативною пам'яттю інцидентів. Програмною основою цього механізму виступає структура даних HashSet, яка є оптимізованою хеш-таблицею. Вибір саме цієї структури замість звичайних масивів чи списків зумовлений її математичними властивостями. Хеш-таблиця дозволяє перевіряти наявність елемента всередині себе практично миттєво, незалежно від кількості збережених записів. Це робить алгоритм ефективним і гарантує відсутність додаткового навантаження на процесор сервера під час перевірок.

Логіка роботи алгоритму побудована на відстеженні переходів між станами. Кожні тридцять секунд фоновий процес звертається до диспетчера служб операційної системи і запитує поточний статус чотирьох цільових компонентів. Якщо служба відповідає, що вона не працює, алгоритм не поспішає бити на сполох. Спочатку він перевіряє свою хеш-таблицю проблемних служб. Якщо назви цієї служби там немає, алгоритм розуміє, що це абсолютно новий інцидент, про який центральний сервер ще не знає. У цей момент генерується і відправляється критичне сповіщення про падіння служби, після чого її назва записується у хеш-таблицю як підтверджений збій.

Під час наступного циклу перевірки, якщо адміністратор ще не встиг підняти службу, алгоритм знову побачить, що вона не працює. Але цього разу, зазирнувши у хеш-таблицю, він знайде там її назву. Це слугує для алгоритму сигналом зупинки: система розуміє, що сповіщення вже було доставлено раніше, тому повторний мережевий запит до месенджера не формується. Тиша зберігатиметься доти, поки проблема не буде вирішена. Щойно системний адміністратор запустить службу і її статус зміниться на активний, алгоритм помітить це відновлення. Він знову перевірить свою хеш-таблицю, знайде там запис про нещодавню аварію цієї служби і назавжди видалить його. Таким чином, внутрішній стан агента очищується, і він знову стає готовим до миттєвої реакції на наступні можливі збої. Додатково весь цей процес захищений блоком обробки помилок. Навіть якщо цільова служба взагалі відсутня в системі або

агент не має прав на перевірку її стану, програма просто проігнорує цей компонент і продовжить стабільно контролювати інші служби.

Останнім, але найскладнішим алгоритмічним вузлом агента є підсистема моніторингу подій безпеки. У середовищі Windows система журналювання генерує величезні обсяги даних, де кожна дія користувача чи служби залишає свій слід. Спроба зчитувати абсолютно всі записи, а потім програмно фільтрувати їх засобами мови C#, призвела б до швидкого вичерпання ресурсів процесора та оперативної пам'яті. Щоб уникнути цього, було застосовано алгоритм делегованої фільтрації, який переносить основне навантаження з прикладного рівня на рівень ядра операційної системи. В основі цього підходу лежить використання мови запитів XPath, яка дозволяє надіслати до системи журналювання чіткі інструкції про те, які саме події нас цікавлять.

Процес ініціалізації цього алгоритму починається з формування запиту, де перераховуються конкретні ідентифікатори подій, такі як успішні чи невдалі спроби входу, зміни у глобальних групах безпеки або очищення журналів аудиту. Цей запит передається об'єкту спостерігача, який реєструє спеціальний обробник у підсистемі Windows Event Log[9]. Важливо розуміти, що після реєстрації агент фактично переходить у стан очікування і не споживає обчислювальних ресурсів до того моменту, поки система не зафіксує подію, що відповідає критеріям XPath-запиту. Тільки тоді операційна система ініціює виклик нашого обробника, передаючи йому готову структуру даних. Це суттєво відрізняється від методів опитування, де програма змушена постійно стукати в систему з питанням: "Чи з'явилося щось нове?".

Отримавши подію, алгоритм переходить до етапу класифікації та розбору її вмісту. Для переважної більшості подій безпеки достатньо зчитати ідентифікатор події та відформатувати опис, який вже підготовлений системою для читання людиною. Однак існують винятки, де простого зчитування опису недостатньо. Яскравим прикладом є подія створення нового процесу, яка містить технічну інформацію у форматі XML. Для таких специфічних випадків алгоритм активує додатковий розбір даних. Наприклад, під час фіксації запуску програми,

агент парсить XML-структуру, видобуваючи назву виконуваного файлу. Якщо в цьому полі виявляється ім'я процесу, який було додано до списку заборонених, алгоритм негайно змінює пріоритет обробки події. Замість звичайного логування він ініціює термінову відправку сповіщення з позначкою про виявлення потенційно небезпечної активності.

Завдяки такій багатоступеневій схемі обробки, агент не витрачає час на аналіз тисяч системних процесів чи звичайних операцій, фокусуючись виключно на тому, що заздалегідь визначено як критично важливе. Використання нативних засобів Windows для фільтрації дозволяє системі моніторингу залишатися непомітною для кінцевого користувача, навіть на комп'ютерах з обмеженими апаратними потужностями. Цей алгоритмічний підхід забезпечує баланс між високою швидкістю реакції на загрози безпеці та мінімальним впливом на загальну продуктивність операційної системи.

2.4. Проєктування підсистеми безпеки та захисту інфраструктури

Передача системної телеметрії та інцидентів безпеки відкритими каналами зв'язку є абсолютно неприпустимою для сучасного корпоративного середовища. Якщо агент моніторингу відправлятиме зібрані дані у вигляді звичайного незашифрованого тексту, зловмисник зможе перехопити цей трафік. У такому сценарії система моніторингу перетвориться на інструмент розвідки для хакера, адже він отримає детальну інформацію про структуру Active Directory, імена критичних служб та навіть назви запущених процесів на комп'ютерах. З огляду на ці загрози, архітектурою комплексу передбачено обов'язкове використання криптографічного шифрування транспортного рівня для всього мережевого обміну між агентами та центральним сервером.

Налаштування безпечного з'єднання вимагає наявності цифрового сертифіката на серверному вузлі. Найпростішим підходом у розробці є зберігання файлу сертифіката разом із його приватним ключем безпосередньо в каталозі з програмою. Однак такий метод створює значні ризики компрометації ключа у разі несанкціонованого доступу сторонніх осіб до файлової системи сервера. Для уникнення цієї вразливості було прийнято рішення відмовитися від

файлового зберігання та інтегрувати розроблений сервер моніторингу з нативним локальним сховищем сертифікатів операційної системи Windows. Це ізольована та криптографічно захищена область пам'яті, де сама операційна система відповідає за безпечне зберігання приватних ключів та суворо контролює права доступу до них з боку різних програм.

Програмна реалізація цього захисного механізму відбувається на ранньому етапі конфігурації веб-сервера Kestrel під час його запуску. Додаток звертається до локального сховища комп'ютера у безпечному режимі лише для читання. Щоб знайти правильний сертифікат серед десятків інших системних ключів, алгоритм використовує пошук за унікальним відбитком. Цей відбиток являє собою хеш-суму сертифіката, яка гарантує стовідсоткову точність ідентифікації необхідного криптографічного документа. Серверний код передає цей відбиток системному інструментарію пошуку, який сканує захищене сховище і повертає знайдений сертифікат безпосередньо в оперативну пам'ять додатку.

Отриманий таким чином сертифікат застосовується для налаштування протоколу HTTPS на визначеному мережевому порту. Важливою архітектурною деталлю є те, що сервер обов’язково перевіряє успішність процесу пошуку. Якщо з певних причин системний адміністратор забув встановити сертифікат у сховище Windows або ввів неправильний відбиток під час налаштування, система блокує подальшу роботу. Замість того, щоб автоматично перемкнутися на незахищене з'єднання, додаток генерує критичну помилку та примусово завершує своє виконання. Такий безкомпромісний підхід гарантує, що жоден байт чутливої корпоративної інформації не покине межі комп'ютерів без належного рівня шифрування, що відповідає сучасним стандартам безпеки програмного забезпечення.

Будь-який програмний комплекс, який взаємодіє з базами даних або зовнішніми платформами, потребує використання автентифікаційних даних. У розробленій системі такими критичними даними виступають рядок підключення до реляційної бази даних PostgreSQL та унікальна адреса вебхука для відправки тривожних сповіщень у корпоративний месенджер. Найпоширенішою і водночас

найнебезпечнішою практикою серед розробників є збереження цих секретів у відкритому вигляді у стандартних конфігураційних файлах. Якщо зберегти реальний пароль від бази даних у звичайному текстовому файлі, будь-який співробітник або шкідливий процес, що отримає права на читання системного каталогу, автоматично здобуде повний контроль над усім історичним архівом моніторингу.

Для усунення цієї вразливості було впроваджено практику суворої ізоляції секретів на рівні операційної системи. У конфігураційних файлах розробленого сервера замість реальних паролів та адрес використовуються звичайні слова-заглушки. Самі ж справжні автентифікаційні дані перенесені у локальні змінні середовища операційної системи. Проте простого перенесення у загальносистемні змінні недостатньо для забезпечення максимальної безпеки, оскільки вони залишаються видимими для всіх користувачів комп'ютера. Тому було застосовано більш захищений підхід із використанням змінних рівня конкретного користувача.

Під час розгортання центрального серверного вузла в операційній системі створюється окремий сервісний обліковий запис, від імені якого і запускається процес моніторингу. Критичні секрети прописуються виключно в оточення цього конкретного сервісного користувача. Під час свого старту програмний комплекс зчитує конфігураційний файл із заглушками, але вбудований механізм платформи розробки автоматично перехоплює ці ключі та підміняє їх реальними значеннями зі змінних середовища поточного облікового запису. Завдяки цьому досягається ізоляція: навіть якщо інший системний адміністратор увійде на цей же сервер під своїм логіном, він фізично не зможе побачити або скопіювати пароль від бази даних чи токен месенджера. Цей підхід гарантує високий рівень захисту інфраструктури та повністю відповідає галузевим стандартам безпечного управління конфігураціями.

Висновки до розділу 2

У другому розділі дипломної роботи було здійснено детальне проєктування архітектури системи моніторингу та її внутрішніх алгоритмів. Сформовано клієнт-серверну топологію тестового стенда, яка імітує гетерогенну корпоративну мережу з центральним контролером домену та комп'ютерами під управлінням різних операційних систем. Розроблено модульну структуру клієнтського агента, що дозволяє незалежно та паралельно виконувати завдання зі збору телеметрії та відстеження подій безпеки без блокування робочих потоків.

Завдяки проєктуванню єдиних контрактів взаємодії та жорсткої структури реляційної бази даних забезпечено надійний механізм передачі та збереження зібраної інформації. Запропоновано та обґрунтовано оптимізовані алгоритми обробки даних. Зокрема імплементовано алгоритм ковзного вікна для аналізу навантаження процесора, механізм збереження стану для моніторингу критичних інфраструктурних служб та алгоритм делегованої фільтрації журналів безпеки. Це дозволяє системі ефективно виявляти реальні інциденти без створення надмірного інформаційного шуму для персоналу. Спроєктована підсистема безпеки, що включає криптографічний захист каналів зв'язку за допомогою локальних системних сертифікатів та сувору ізоляцію конфігураційних секретів на рівні користувача операційної системи, забезпечує захист телеметричних даних від несанкціонованого доступу та перехоплення.

РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ МОНІТОРИНГУ

3.1. Розгортання тестового стенда та конфігурація програмних компонентів

Практичний етап роботи розпочався зі створення ізольованого лабораторного середовища на базі технологій віртуалізації. Для розгортання тестового стенда було використано гіпервізор Oracle VirtualBox, у якому створено три віртуальні машини (рис. 3.1) з розподілом апаратних ресурсів, відповідно до потреб моделі. Для головного сервера під управлінням Windows Server було виділено чотири гігабайти оперативної пам'яті та три процесорних ядра, що є необхідним для стабільної роботи служб каталогу та компонентів моніторингу. Клієнтська робоча станція Windows 10 та віртуальна машина з Ubuntu Linux отримали по два гігабайти оперативної пам'яті.

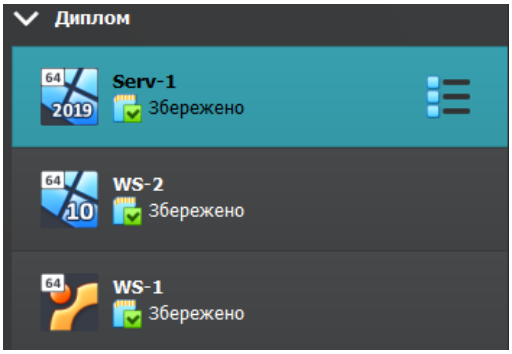


Рис. 3.1. Розгорнуті віртуальні машини.

Наступним кроком стало підняття інфраструктури домену на базовому сервері. Через диспетчер серверів було встановлено ролі Active Directory Domain Services разом із інтегрованим DNS-сервером. Після завершення інсталяції файлів сервер було підвищено до рівня контролера домену в новому лісі (рис. 3.2). Під час конфігурації було створено доменну зону та налаштовано статичну IP-адресу сервера. Оскільки коректна робота DNS є критичною для служби каталогу, у налаштуваннях мережевого адаптера сервера як головний DNS-сервер було вказано його власну локальну адресу.

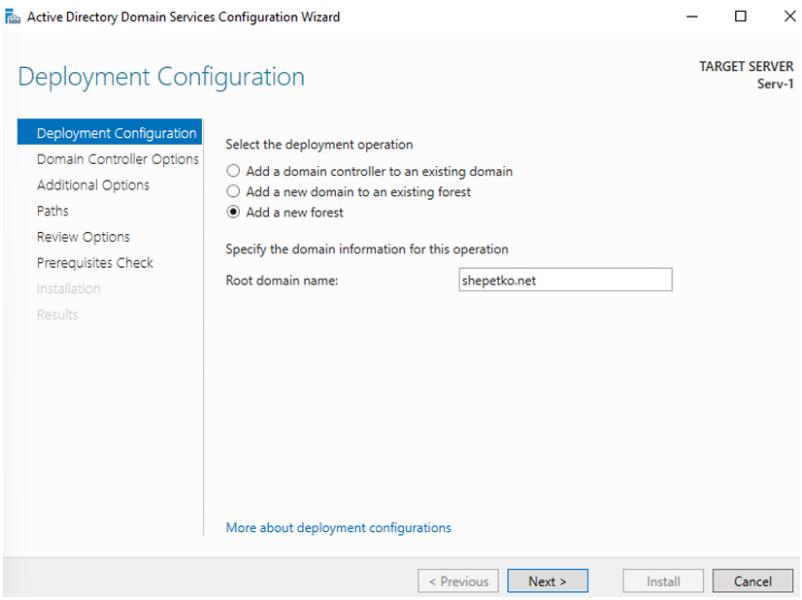


Рис. 3.2. Створення нового домену у новому лісі.

Після успішного перезавантаження контролера домену було проведено підключення інших вузлів мережі до спільного логічного простору. На робочій станції Windows 10 у налаштуваннях мережевого підключення IPv4 як єдиний DNS-сервер було прописано IP-адресу контролера домену (рис. 3.3). Це дозволило клієнтській машині успішно розпізнати ім'я створеного домену в мережі. Через властивості системи було запущено процедуру введення комп'ютера в домен, після чого система запитала реквізити адміністратора і успішно завершила процес приєднання.

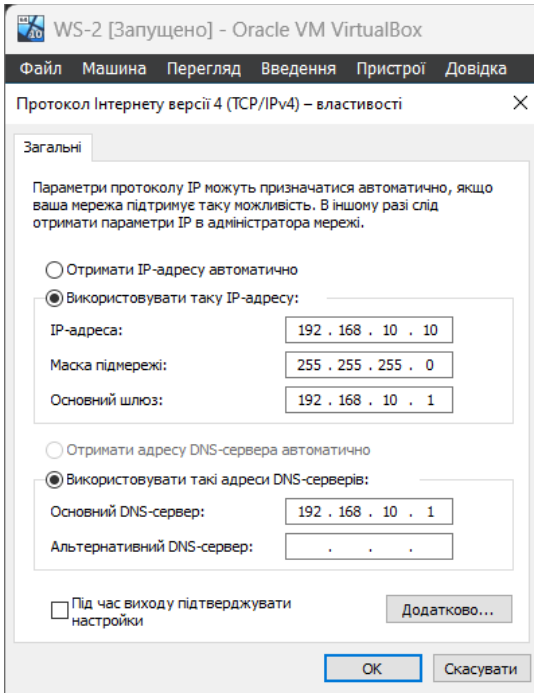


Рис. 3.3. Налаштування адресації на робочій станції..

Для сервера Ubuntu Linux, який за своєю архітектурою не вводився в домен Active Directory, налаштування обмежилося конфігурацією статичного IP в одній підмережі з головним сервером. Через утиліту netplan було прописано параметри мережевої карти та вказано адресу контролера домену як DNS-сервера, щоб Linux-версія клієнтського агента могла резолвити доменне ім'я Serv-1 під час встановлення gRPC-з'єднання з сервером обробки даних (рис. 3.4).

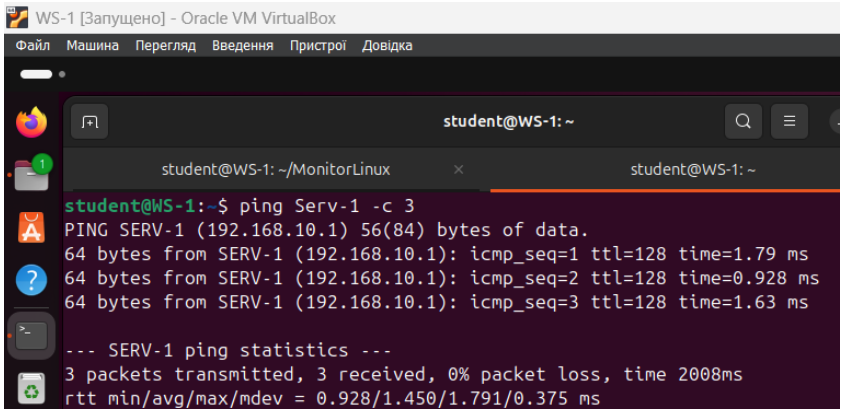


Рис. 3.4. Перевірка зв’язку.

3.2. Програмна реалізація підсистеми збору телеметрії та мережевої взаємодії

Першочерговим завданням при розробці клієнтської частини системи став запуск надійної мережевої транспортної системи. Оскільки збір метрик відбувається безперервно, критично важливо було відмовитися від постійного створення нових з'єднань, які навантажують мережу та сервер. Нижче наведено фрагмент коду ініціалізації gRPC-каналу та налаштування сокетів у методі виконання служби моніторингу телеметрії.

Лістинг MetricWorker.cs:

```
        using var channel = GrpcChannel.ForAddress(_options.ServerAddress, new
GrpcChannelOptions
        {
            HttpHandler = new SocketsHttpHandler
            {
                EnableMultipleHttp2Connections = true,
                KeepAlivePingDelay             = TimeSpan.FromSeconds(60),
                KeepAlivePingTimeout           = TimeSpan.FromSeconds(30),
            }
        });

        var client = new SystemMonitor.SystemMonitorClient(channel);
```

Логіка цього блоку полягає в налаштуванні поведінки клієнта на транспортному рівні TCP. Створення каналу відбувається один раз при старті воркера, після чого цей канал повторно використовується для всіх наступних ітерацій відправки даних. Вбудований обробник сокетів налаштовується на роботу виключно в режимі другого протоколу HTTP. Прапорець дозволу кількох з'єднань дозволяє клієнту динамічно відкривати додаткові потоки в межах одного TCP-каналу, якщо обсяг даних або частота повідомлень різко зростають. Особливу роль відіграють параметри підтримки життєздатності з'єднання. Налаштування затримки пінгів змушує клієнта кожну хвилину надсилати порожні пакети серверу, якщо реального трафіку немає. Це обманює мережеві екрани та маршрутизатори, не дозволяючи їм примусово закривати з'єднання через уявну неактивність. Якщо сервер не відповідає на такий пінг протягом тридцяти секунд, клієнт вважає з'єднання обірваним і починає процедуру його безпечного відновлення, що гарантує високу стабільність зв'язку між машинами стенда.

Другим серйозним викликом була реалізація кросплатформності, зокрема збору даних про оперативну пам'ять без використання ресурсомістких системних запитів операційної системи. Нижче представлено архітектурне рішення для низькорівневого зчитування доступної пам'яті у системах Windows та Linux.

Лістинг MetricWorker.cs:

```
[System.Runtime.InteropServices.DllImport("kernel32.dll", SetLastError = true)]
    private static extern bool GlobalMemoryStatusEx(ref MEMORYSTATUSEX lpBuffer);

    [System.Runtime.InteropServices.StructLayout(System.Runtime.InteropServices.Lay
outKind.Sequential)]
    private struct MEMORYSTATUSEX
    {
        public uint  dwLength;
        public uint  dwMemoryLoad;
        public ulong ullTotalPhys;
        public ulong ullAvailPhys;
        public ulong ullTotalPageFile;
        public ulong ullAvailPageFile;
        public ulong ullTotalVirtual;
        public ulong ullAvailVirtual;
        public ulong ullAvailExtendedVirtual;
    }
```

```
 private static double GetWindowsAvailableRamMb()
    {
        try
        {
            var status = new MEMORYSTATUSEX { dwLength =
(uint)System.Runtime.InteropServices.Marshal.SizeOf<MEMORYSTATUSEX>() };
            if (GlobalMemoryStatusEx(ref status))
                return Math.Round(status.ullAvailPhys / 1024.0 / 1024.0, 0);
        }
        catch { /* fall through */ }
        return -1;
    }

    private static double GetLinuxAvailableRamMb()
    {
        try
        {
            var memAvailLine = File.ReadLines("/proc/meminfo")
                .FirstOrDefault(l => l.StartsWith("MemAvailable:",
StringComparison.Ordinal));
            if (memAvailLine is null) return -1;

            var kb = long.Parse(memAvailLine.Split(':',
StringSplitOptions.TrimEntries)[1]
                                            .Replace(" kB", "",
StringComparison.Ordinal)
                                            .Trim());
            return Math.Round(kb / 1024.0, 0);
        }
        catch
        {
            return -1;
        }
    }
```

Логіка представлених методів демонструє різні підходи до взаємодії з ядрами різних операційних систем, об'єднані єдиним інтерфейсом виклику. Для середовища Windows застосовано механізм імпорту нативних динамічних бібліотек. Замість важких запитів, програма через атрибут імпорту звертається безпосередньо до базової системної бібліотеки ядра. Спроєктована структура опису пам'яті суворо повторює розташування байтів у пам'яті самої операційної системи. Метод передає посилання на цю структуру системній функції, яка миттєво заповнює її актуальними даними про стан заліза, минаючи всі проміжні рівні абстракції додатків. Витягнуте значення доступної фізичної пам'яті у байтах математично переводиться у мегабайти та округлюється.

З іншого боку, коли цей же агент запускається на віртуальній машині під управлінням Linux, архітектура Windows стає недоступною. Вбудована логіка верхнього рівня перемикає виконання на метод роботи з файловою системою Linux. В операційних системах цього сімейства вся інформація про апаратне забезпечення у вигляді віртуальних текстових файлів ядра. Алгоритм відкриває на читання системний файл стану пам'яті і за допомогою методів моніторингу знаходить перший рядок, який починається зі специфічного маркера доступної пам'яті. Далі рядок очищується від тексту та одиниць вимірювання, а чисте числове значення в кілобайтах конвертується в мегабайти. Обидва методи захищені порожніми блоками перехоплення винятків, що гарантує стабільність: якщо операційна система заблокує доступ до файлу або функції, сервіс не впаде, а просто поверне маркер помилки, дозволяючи іншим модулям продовжувати роботу.

Для вирішення проблеми хибних спрацьовувань під час короткочасних сплесків навантаження на процесор було реалізовано алгоритм ковзного вікна спільно з механізмом затримки сповіщень. Нижче наведено код асинхронного методу перевірки перевантаження сервера.

Лістинг MetricWorker.cs:

```
    private async Task CheckServerOverloadAsync(double currentCpu, double
currentAvailableRamMb, SystemMonitor.SystemMonitorClient client, CancellationToken
ct)
    {
        if (!_options.IsDomainController) return;

        bool isRamStarved = currentAvailableRamMb > 0 && currentAvailableRamMb <
1024.0;

        int maxSamples = 120 / Math.Max(1, _options.PushIntervalSeconds);
        _cpuHistory.Enqueue(currentCpu);

        if (_cpuHistory.Count > maxSamples)
        {
            _cpuHistory.Dequeue();
        }

        if (_cpuHistory.Count == maxSamples)
        {
            double avgCpu = _cpuHistory.Average();
```

```
if (avgCpu >= 90.0)
            {
                if (DateTime.UtcNow - _lastOverloadAlertTime > _alertCooldown)
                {
                    _logger.LogWarning("Triggering CPU Overload Alert! Avg 2m CPU:
{Cpu}%", avgCpu);

                    await client.PushAlertAsync(new AlertRequest
                    {
                        MachineName = Environment.MachineName,
                        AlertType   = "ServerOverload",
                        Message     = $"Процесор перевантажений. Середнє за 2
хвилини: {avgCpu:F1}%."
                    }, cancellationToken: ct);

                    _lastOverloadAlertTime = DateTime.UtcNow;
                }
            }
        }
    }
}
```

Представлена логіка відповідає за інтелектуальний аналіз стану процесора виключно на контролерах домену. Метод динамічно вираховує максимальний розмір черги на основі інтервалу відправки даних, щоб сформувати вибірку рівно за дві хвилини. Нове значення завантаження процесора додається в кінець черги, а якщо ліміт перевищено, найстаріший запис автоматично видаляється. Після повного заповнення масиву програма обчислює середнє арифметичне значення. У разі перевищення критичної позначки у дев'яносто відсотків алгоритм перевіряє час відправки останнього тривожного сигналу. Якщо з моменту попереднього сповіщення минуло більше п'яти хвилин, сервер формує новий мережевий запит до месенджера і фіксує поточний час як нову точку відліку для таймера охолодження. Це надійно захищає адміністратора від інформаційного перевантаження під час тривалих системних збоїв.

Об'єднання всіх вищеописаних механізмів в єдиний робочий процес відбувається всередині головного циклу фонової служби. Нижче наведено фрагмент коду, який відповідає за безперервний збір та відправку телеметрії з урахуванням вимог відмовостійкості.

Лістинг MetricWorker.cs:

```
        while (!stoppingToken.IsCancellationRequested)

{
            try
            {
                var request = BuildMetricsRequest();

                _logger.LogInformation(
                    "Pushing metrics → CPU: {Cpu:F1}%| RAM: {Ram:F0} MB | Disk:
{Disk:F2} GB | Up: {Up:F1}h | Net: {Net:F2}MB",
                    request.CpuUsagePercent, request.AvailableRamMb,
                    request.FreeDiskSpaceGb, request.UptimeHours,
request.NetTrafficMbps);

                await CheckServerOverloadAsync(request.CpuUsagePercent,
request.AvailableRamMb, client, stoppingToken);

                var response = await client.PushMetricsAsync(
                    request,
                    cancellationToken: stoppingToken);

                if (response.Success)
                    _logger.LogInformation("Server ACK: {Message}",
response.Message);
                else
                    _logger.LogWarning("Server returned failure: {Message}",
response.Message);
            }
            catch (Exception ex) when (!stoppingToken.IsCancellationRequested)
            {
                _logger.LogError(ex,
                    "Failed to push metrics. Will retry in {Interval}s.",
                    _options.PushIntervalSeconds);
            }

            await Task.Delay(
                TimeSpan.FromSeconds(_options.PushIntervalSeconds),
                stoppingToken);
        }
    }
```

Цей архітектурний блок є серцем клієнтського агента. Нескінченний цикл функціонує доти, поки операційна система не надішле сигнал зупинки. На кожній ітерації програма спочатку формує структурований пакет актуальних метрик, після чого проводить локальний аналіз на наявність перевантажень. Далі ініціюється асинхронний віддалений виклик процедури на центральному сервері. Найважливішою частиною цього блоку є обробка виняткових ситуацій.

Якщо центральний сервер раптово втратить живлення або станеться обрив мережевого кабелю, відправка даних завершиться критичною помилкою. Завдяки блоку перехоплення винятків клієнтський додаток не припинить свою роботу. Він залогує факт відсутності зв'язку, перейде в режим очікування на заданий конфігурацією інтервал часу і згодом здійснить нову спробу відправки. Така поведінка гарантує високу живучість агента в умовах нестабільної корпоративної мережі.

Наступним важливим компонентом клієнтського агента є модуль відстеження подій інформаційної безпеки. На відміну від збору метрик, який відбувається за розкладом, цей модуль працює за подієво-орієнтованою моделлю. Для мінімізації споживання апаратних ресурсів було реалізовано механізм делегованої фільтрації журналів на рівні операційної системи. Нижче наведено фрагмент коду ініціалізації системного спостерігача.

Лістинг EventWorker.cs:

```
        string queryStr = "*[System[(EventID=4740 or EventID=1102 or EventID=4720
or EventID=4726 or EventID=4728 or EventID=4756 or EventID=4719 or EventID=4771 or
EventID=4777 or EventID=4688)]]";

        var query = new EventLogQuery("Security", PathType.LogName, queryStr);
        using var watcher = new EventLogWatcher(query);
```

Замість того, щоб зчитувати кожен новий запис із журналу безпеки та перевіряти його програмно, агент формує жорсткий текстовий запит мовою XPath. Цей запит містить вичерпний перелік ідентифікаторів подій, які становлять інтерес для моніторингу інфраструктури Active Directory. До цього переліку входять події блокування облікових записів, очищення журналів аудиту, створення нових користувачів, зміна складу глобальних груп та помилки автентифікації протоколу Kerberos. Сформований запит передається спеціальному об'єкту спостерігача, який реєструється безпосередньо у підсистемі Windows Event Log. Завдяки цьому агент переходить у сплячий режим і не витрачає процесорний час, поки операційна система сама не викличе його обробник при появі цільової події.

Коли операційна система фіксує порушення політик безпеки або іншу критичну дію, спрацьовує асинхронний обробник переривань. Нижче наведено код логіки аналізу отриманого системного запису та формування сповіщення.

Лістинг EventWorker.cs:

```
watcher.EventRecordWritten += async (sender, e) =>
        {
            if (e.EventRecord == null) return;
            var record = e.EventRecord;
            var eventId = record.Id;

            if (eventId == 4688)
            {
                var xml = record.ToXml();
                if (xml.Contains("notepad.exe",
StringComparison.OrdinalIgnoreCase))
                {
                    await client.PushAlertAsync(new AlertRequest
                    {
                        MachineName = Environment.MachineName,
                        AlertType = "ForbiddenProcess",
                        Message = "Запущено заборонений процес: notepad.exe"
                    });
                }
                return;
            }

            string alertType = $"SecurityEvent_{eventId}";
            string desc = record.FormatDescription() ?? "Опис події недоступний";
            string shortDesc = desc.Length > 500 ? desc[..500] + "..." : desc;

            string message = eventId switch
            {
               //Форматування
            };

            await client.PushAlertAsync(new AlertRequest
            {
                MachineName = Environment.MachineName,
                AlertType = alertType,
                Message = message
            });
        };
```

Цей блок коду відповідає за розбір отриманого інциденту. Більшість стандартних подій безпеки обробляються шляхом зчитування готового текстового опису, його скорочення до допустимих розмірів та відправки на центральний сервер із відповідним ідентифікатором типу сповіщення. Проте

архітектура модуля передбачає можливість більш складного евристичного аналізу. Яскравим прикладом є перехоплення події створення нового процесу під номером 4688. У цьому випадку стандартного текстового опису недостатньо, тому агент зчитує сиру структуру запису у форматі розширюваної мови розмітки та шукає там ім'я виконуваного файлу. Для проведення стендових випробувань як індикатор забороненого програмного забезпечення було обрано стандартний текстовий редактор. Якщо система фіксує його запуск, алгоритм негайно формує спеціальне сповіщення про виявлення небажаного процесу. У реальних умовах експлуатації цей механізм дозволяє миттєво реагувати на спроби запуску скриптів шифрувальників або утиліт для несанкціонованого доступу до пам'яті.

Третім спеціалізованим компонентом клієнтського додатка є модуль безперервного контролю життєздатності критичних служб каталогу Active Directory. Цей воркер активується виключно на машинах, які виконують роль контролера домену, та реалізує стейтфул-підхід для запобігання надлишковому дублюванню сповіщень. Нижче наведено повний програмний код логіки циклічного моніторингу інфраструктурних служб.

Лістинг addsWorker.cs:

```
string[] criticalServices = { "NTDS", "Kdc", "Netlogon", "DNS" };

        while (!stoppingToken.IsCancellationRequested)
        {
            foreach (var svcName in criticalServices)
            {
                try
                {
                    using var sc = new ServiceController(svcName);

                    if (sc.Status != ServiceControllerStatus.Running)
                    {
                        if (!_downServices.Contains(svcName))
                        {
                            await client.PushAlertAsync(new AlertRequest {
                                MachineName = Environment.MachineName,
                                AlertType = "ServiceDown",
                                Message = $"АХТУНГ! Служба {svcName} впала. Статус:
{sc.Status}"
                            }, cancellationToken: stoppingToken);

                            _downServices.Add(svcName);
```

```
                    }
                }
                else
                {
                    if (_downServices.Contains(svcName))

{

                        _downServices.Remove(svcName);
                    }
                }
            }
            catch
            {
                // Служби нема або немає доступу
            }
        }

        await Task.Delay(TimeSpan.FromSeconds(30), stoppingToken);
    }
```

Логіка роботи цього модуля базується на ітераційному обході масиву назв системних процесів за допомогою вбудованого інструментарію операційної системи Windows. Кожні тридцять секунд фонова задача створює об'єкт контролера для кожної цільової служби, такої як ядро каталогу, центр розподілу ключів Kerberos, служба локатора та сервер доменних імен. Якщо під час опитування виявляється, що статус служби відрізняється від штатного режиму роботи, алгоритм звертається до внутрішньої хеш-таблиці інцидентів. Перевірка наявності назви служби у цій колекції дозволяє дізнатися, чи фіксувалося це падіння раніше. Якщо запис відсутній, програма ініціює відправку критичного gRPC-пакета на центральний сервер і негайно додає маркер служби до своєї пам'яті.

Протягом усіх наступних циклів перевірки, поки служба залишається неактивною, алгоритм буде бачити її наявність у хеш-таблиці та блокуватиме повторну генерацію мережевих запитів, мінімізуючи навантаження на канал зв'язку. Щойно системна служба відновлює свою роботу і переходить у стан успішного виконання, логічна гілка перемикається на зворотну дію. Алгоритм видаляє назву служби з колекції інцидентів, повертаючи систему в початковий режим готовності до фіксації нових збоїв. Увесь конвеєр обробки поміщений у блок перехоплення винятків, що дозволяє безпечно ігнорувати ситуації, коли

певна служба заблокована системою безпеки або взагалі не встановлена на поточному вузлі мережі.

Перехід до серверної частини комплексу вимагає забезпечення безперебійної обробки вхідних з'єднань від усіх розгорнутих агентів. Серцевиною центрального вузла виступає файл конфігурації хоста, де визначаються правила роботи мережевого рівня та підключення до бази даних. Нижче наведено програмний код ініціалізації сервера та налаштування його захисту.

Лістинг Program.cs (Server):

```
builder.Host.UseWindowsService();

builder.Services.AddDbContext<AppDbContext>(dbOptions =>
    dbOptions.UseNpgsql(builder.Configuration.GetConnectionString("PostgresConnecti
on")));

builder.Services.Configure<MonitoringServer.Models.SlackOptions>(builder.Configurat
ion.GetSection("SlackOptions"));

builder.Services.AddGrpc(grpcOptions =>
{
    grpcOptions.EnableDetailedErrors = builder.Environment.IsDevelopment();
});


builder.WebHost.ConfigureKestrel(kestrel =>
{
    kestrel.ListenAnyIP(5050, listenOptions =>
    {
        listenOptions.Protocols =
Microsoft.AspNetCore.Server.Kestrel.Core.HttpProtocols.Http2;

        using var store = new X509Store(StoreName.My, StoreLocation.LocalMachine);
        store.Open(OpenFlags.ReadOnly);

        var certs = store.Certificates.Find(X509FindType.FindByThumbprint,
"F808DE70065B6AD8E35C35A758A2D46F5D0ABD4A", false);

        if (certs.Count > 0)
        {
            listenOptions.UseHttps(certs[0]);
        }
        else
        {
            throw new Exception("ПОМИЛКА: Сертифікат не знайдено за цим
відбитком!");
```

```
        }
    });
});

var app = builder.Build();
using (var scope = app.Services.CreateScope())
{
 var context = scope.ServiceProvider.GetRequiredService<AppDbContext>();
    context.Database.Migrate();
}
```

У наведеному коді насамперед відбувається інтеграція з підсистемою керування службами операційної системи. Це дозволяє серверу моніторингу запускатися у фоновому режимі автоматично без необхідності ручного втручання адміністратора. Далі через механізм впровадження залежностей підключається контекст бази даних PostgreSQL. Окремої уваги заслуговує підхід до розгортання структури бази даних. Замість ручного створення таблиць додаток під час кожного запуску створює ізольовану область видимості та автоматично застосовує всі необхідні міграції коду в базу даних через інструментарій об'єктно-реляційного відображення. Це повністю знімає проблему розсинхронізації версій програми та структури сховища.

Мережева взаємодія забезпечується кросплатформним веб-сервером Kestrel. Програмний код жорстко конфігурує сервер на прослуховування порту з обов'язковим використанням HTTP/2. Використання саме цієї версії протоколу є фундаментальною вимогою для роботи фреймворку віддаленого виклику процедур. Найважливішим аспектом налаштування мережевого рівня є забезпечення криптографічного захисту каналу зв'язку. Щоб уникнути зберігання приватних ключів у звичайних текстових файлах поруч із програмою додаток звертається до захищеного локального сховища сертифікатів комп'ютера (рис. 3.5).

Код відкриває сховище у режимі виключно для читання та здійснює пошук необхідного криптографічного документа за його унікальним відбитком. Якщо сертифікат успішно знайдено він негайно застосовується для шифрування трафіку. У випадку відсутності ключа з вказаним відбитком система безпеки блокує запуск сервера і генерує критичну помилку. Такий підхід повністю

виключає ймовірність передачі телеметрії корпоративної мережі відкритим текстом.

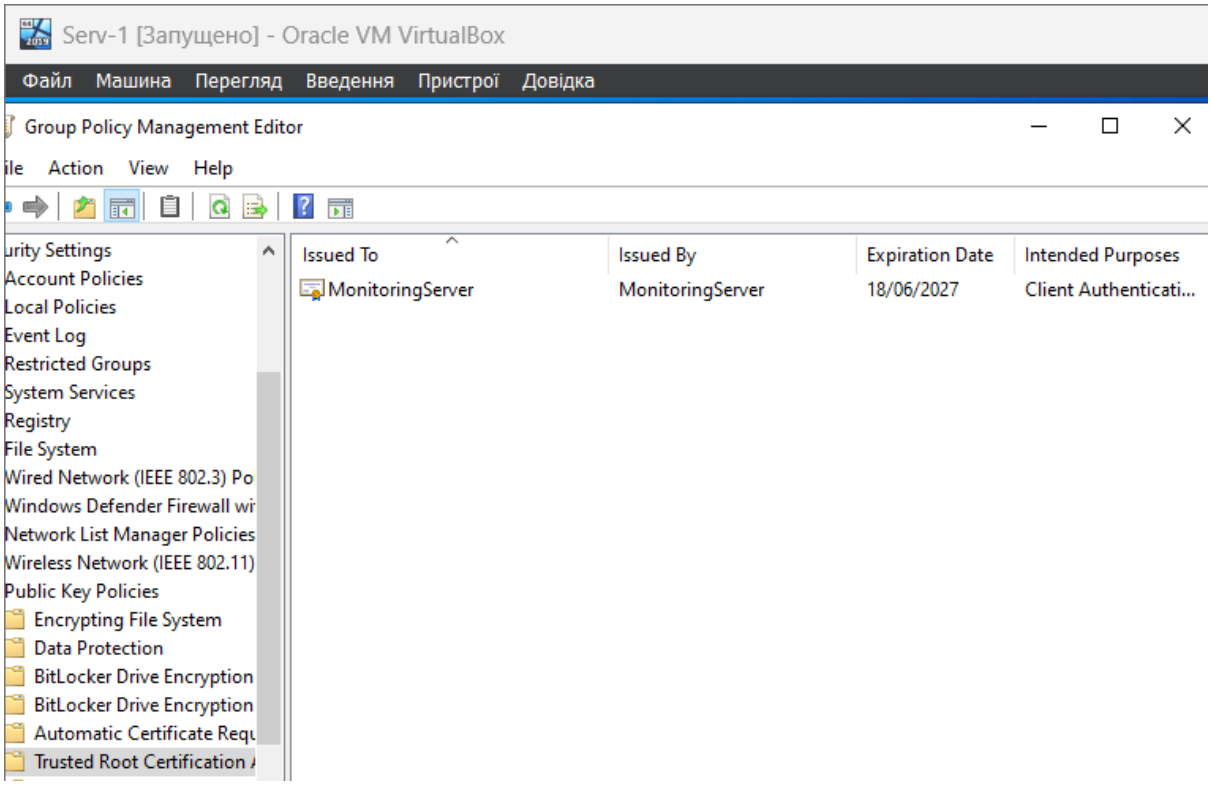


Рис. 3.5. Доданий сертифікат через групові політики.

Для роботи з чутливими даними, а саме для автентифікації в базі даних PostgreSQL та доступу до вебхука Slack, в операційній системі було створено окремого сервісного користувача (рис. 3.6). Цьому обліковому запису через групові політики безпеки було надано необхідні дозволи для збору телеметрії та логів інформаційної безпеки. На рівні профілю цього користувача в операційній системі налаштовано змінні середовища, які зберігають чутливі дані (рис. 3.7). У конфігураційному файлі додатка appsettings.json натомість введено текстові заглушки. Завдяки стандартному механізму побудови конфігурації в платформі .NET, змінні середовища завантажуються після зчитування файлів налаштувань, що дозволяє автоматично перевизначити значення заглушок у пам'яті актуальними даними [1]. Такий підхід є надійним методом ізоляції конфіденційної інформації та захисту від її компрометації.

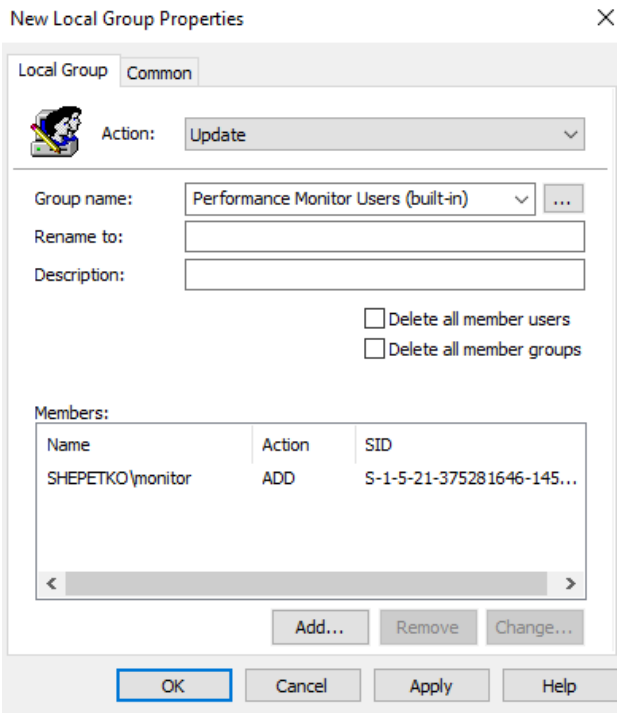


Рис. 3.6. Додавання користувача процесу до локальних груп через групові політики.

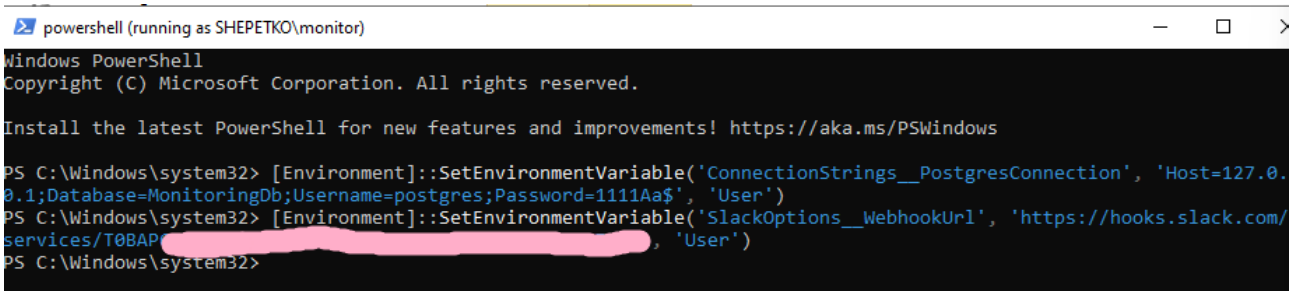


Рис. 3.7. Створення змінних середовища для користувача процесу.

3.3. Налаштування бази даних та панелей візуалізації Grafana

Після успішного розгортання програмного коду сервера необхідно було створити фізичну структуру для зберігання телеметрії. Замість ручного написання SQL-скриптів було застосовано інструментарій Entity Framework Core. У консолі середовища розробки було виконано команду створення первинної міграції, яка проаналізувала класи сутностей та згенерувала код для створення таблиць.

Під час першого тестового запуску сервер моніторингу автоматично підключився до екземпляра PostgreSQL та застосував цю міграцію. В результаті було створено базу даних із двома основними таблицями: Metrics (рис. 3.8) та Alerts (рис. 3.9). Перевірка через графічний клієнт підтвердила правильність

створення схеми: дробові показники навантаження процесора отримали тип double precision, а час реєстрації збережено як timestamp with time zone. Сервер успішно прийняв перший тестовий пакет від клієнтського агента і зафіксував його у сховищі.

Data Output Messages Notifications

Showing rows: 5001 to 5476 Page No: 6 of 6

	Id [PK] integer	MachineName text	Timestamp timestamp with time zone	CpuUsage double prec	RamMb double prec	DiskGb double prec	NetTrafficMbps double precision	UptimeHours double precision
5001	5120	WS-1	2026-06-21 02:58:52.628661+...	0	1819	22.44	0	1.747960277777777
5002	5121	SERV-1	2026-06-21 02:58:54.360756+...	14.2	1236	26.87	0	3.926505833333333
5003	5122	WS-2	2026-06-21 02:58:55.409462+...	0	1934	47.52	0	0.1878775
5004	5123	WS-1	2026-06-21 02:59:04.283203+...	0	1819	22.44	0	1.750885833333333
5005	5124	SERV-1	2026-06-21 02:59:06.541457+...	14.3	1247	26.87	0	3.929570277777777
5006	5125	WS-2	2026-06-21 02:59:07.605926+...	0	1935	47.52	0	0.190946111111111
5007	5126	WS-1	2026-06-21 02:59:15.830436+...	0	1819	22.44	0	1.753813333333333

Рис. 3.8. Структура таблиці Metrics.

	Id [PK] integer	MachineName text	Timestamp timestamp with time zone	AlertType text	Message text
21	85	SERV-1	2026-06-18 21:32:53.187379+...	SecurityEvent_4719	Audit Policy Changed: System audit policy was changed. Subject:
22	86	SERV-1	2026-06-18 21:35:55.317157+...	SecurityEvent_4726	User Deleted: A user account was deleted. Subject: Security ID:
23	87	SERV-1	2026-06-19 00:11:01.755696+...	SecurityEvent_4771	Kerberos Pre-auth Failed: Kerberos pre-authentication failed. Accoun
24	88	SERV-1	2026-06-19 00:11:06.440957+...	SecurityEvent_4771	Kerberos Pre-auth Failed: Kerberos pre-authentication failed. Accoun
25	89	SERV-1	2026-06-20 20:07:56.916731+...	ServerOverload	АХТУНГ! На сервері критично мало RAM. Залишилось: 959 MB.

Рис. 3.9. Структура таблиці Alerts.

Наступним практичним кроком стало налаштування інструментів візуалізації. На центральному сервері було встановлено платформу Grafana. В інтерфейсі управління було додано нове джерело даних типу PostgreSQL, вказано адресу сервера та автентифікаційні дані сервісного облікового запису. Після успішного підключення розпочалося проєктування інформаційних панелей.

Для кожної машини тестового стенда було сформовано власний дашборд (рис. 3.10). За допомогою SQL-макросів Grafana було налаштовано витягування часових рядів із таблиці Metrics. На панель виведено лінійні графіки динаміки завантаження процесора, датчики вільного дискового простору та гістограми інтенсивності мережевого трафіку. Додатково створено табличний віджет, який безперервно опитує таблицю Alerts і виводить на екран усі зафіксовані агентами інциденти, забезпечуючи системного адміністратора єдиною точкою контролю за станом корпоративної мережі.

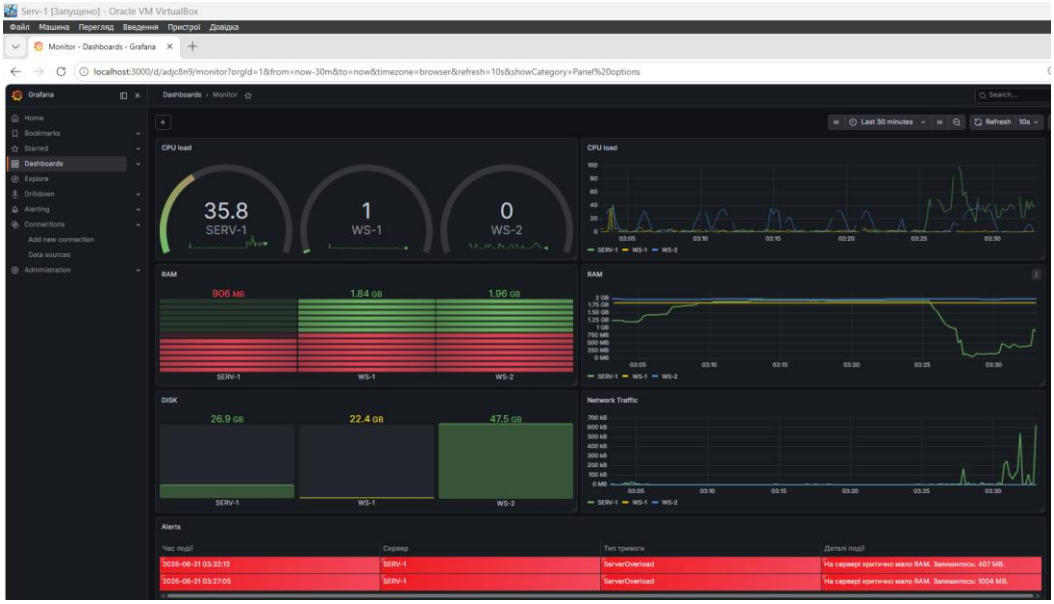


Рис. 3.10. Інформаційний дашборд візуалізації телеметрії та панель інцидентів в інтерфейсі Grafana.

Для забезпечення безперервної роботи системи моніторингу в корпоративному середовищі звичайного ручного запуску виконуваного файлу недостатньо. Тому фінальним етапом практичного розгортання як клієнтських агентів, так і центрального серверного вузла стала їхня інтеграція в операційну систему у вигляді повноцінних фонових служб.

Програмна підтримка цього механізму була закладена ще на етапі написання коду шляхом додавання відповідного розширення до конфігуратора хоста. За допомогою системної команди створення нової служби в реєстрі Windows було зареєстровано процес агента моніторингу та налаштовано його тип запуску як автоматичний (рис. 3.11). Це гарантує, що програма стартує разом з ядром операційної системи ще до появи екрана вітання. Найвідповідальнішим кроком на цьому етапі стала прив'язка створеної служби до спеціального ізольованого сервісного облікового запису. Саме завдяки запуску процесу від імені цього користувача система безпечно підтягує локальні змінні середовища з реальними паролями бази даних та токенами авторизації месенджера. Проте для цього потрібно через групові політики дати окремий дозвіл на запуск процесів від імені користувача (рис. 3.12) [8]. Для серверу було налаштовано відкладений запуск для того, щоб система встигла коректно завантажитись до початку збору метрик і стану домену (рис. 3.13).

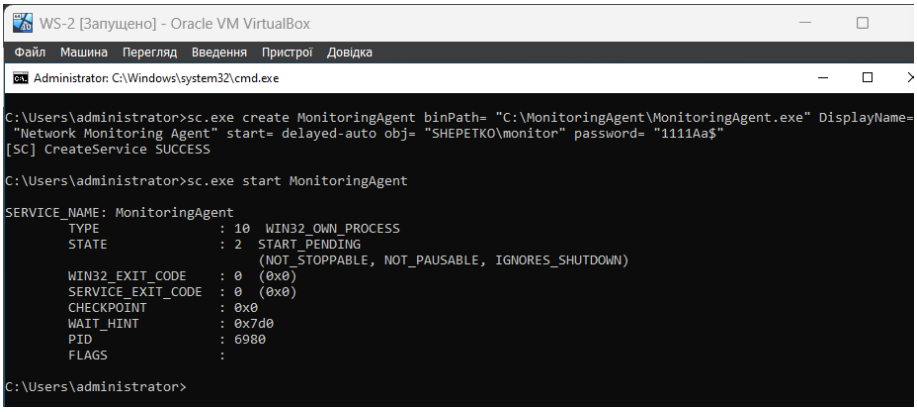


Рис. 3.11. Створення та запуск програми як служби.

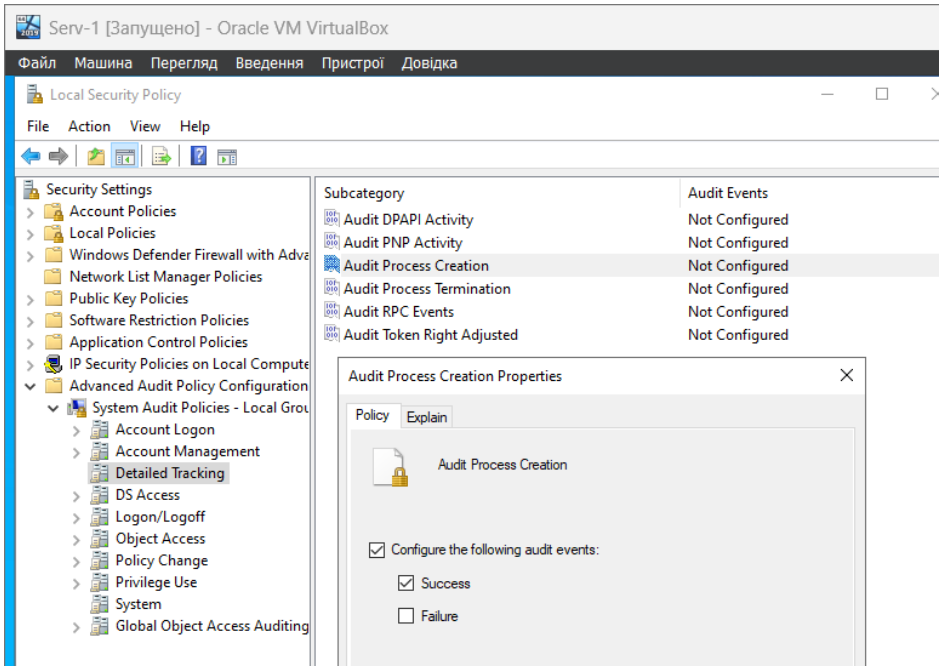


Рис. 3.12. Надання дозволу на запуск процесу від імені користувача.

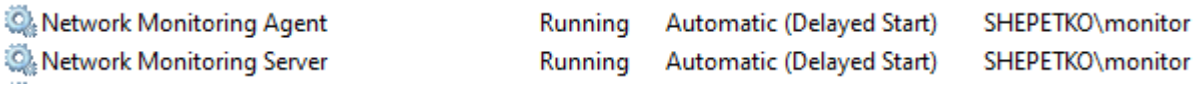


Рис. 3.13. Перевірка створення служб.

3.4. Імітація інцидентів безпеки та перевірка системи сповіщень

Фінальним етапом практичного випробування розробленого комплексу стала перевірка реакції системи на виникнення локальних загроз інформаційній безпеці. Метою тестування була практична демонстрація роботи алгоритму делегованої фільтрації подій та перевірка всього ланцюга доставки тривожного сповіщення від кінцевого вузла до системного адміністратора. Як імітацію запуску шкідливого програмного забезпечення було ініційовано відкриття стандартного текстового редактора, який заздалегідь був внесений до списку заборонених процесів у програмному коді клієнтського агента.

Відразу після запуску виконуваного файлу користувачем підсистема аудиту операційної системи згенерувала системну подію створення нового процесу. Завдяки налаштованому фільтру ця подія була миттєво перехоплена фоновим спостерігачем агента моніторингу. Алгоритм успішно розібрав структуру отриманого запису у форматі розширюваної мови розмітки, виявив назву забороненого виконуваного файлу та сформував критичний мережевий пакет. На центральному сервері цей інцидент був миттєво прийнятий та оброблений. У консолі серверного додатку з'явилося відповідне форматоване повідомлення червоного кольору із зазначенням імені скомпрометованої машини та типу виявленої загрози. Одночасно з цим запис про порушення політик безпеки був автоматично збережений до реляційної бази даних (рис. 3.15).

Alerts

Час події	Сервер	Тип тривоги	Деталі події
2026-06-21 03:52:20.634	SERV-1	ForbiddenProcess	Запущено заборонений процес: notepad.exe
2026-06-21 03:52:18.797	SERV-1	ServerOverload	На сервері критично мало RAM. Залишилось: 966 MB.

Рис. 3.14. Виведення тривожного сповіщення про запуск забороненого процесу.

Паралельно із фіксацією інциденту в базі даних центральний сервер ініціював виконання підпрограми зовнішнього сповіщення. Використовуючи ізольований токен авторизації зі змінних середовища, сервер сформував мережевий запит до вказаної адреси вебхука. У тексті сповіщення чітко відображалася назва комп'ютера порушника та безпосередній опис виявленої проблеми (рис. 3.15). Цей експеримент повністю довів працездатність спроєктованої архітектури та високу швидкість інформування персоналу про критичні події в інфраструктурі підприємства.

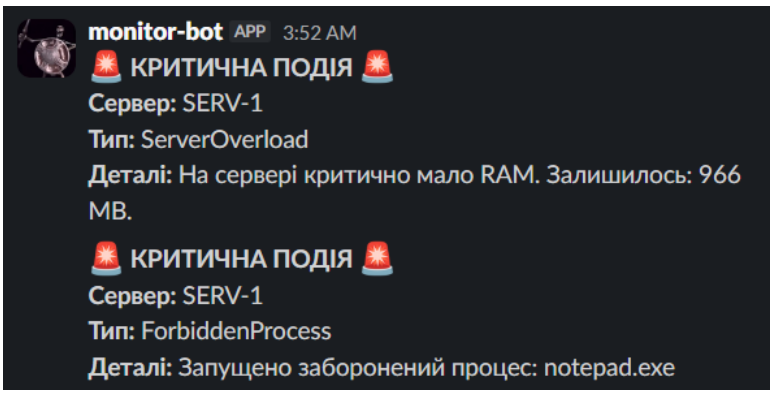


Рис. 3.15. Сповіщення в Slack.

Для підтвердження працездатності розробленого комплексу було проведено масовану імітацію збоїв інфраструктури та порушень політик безпеки. Експеримент складався з двох взаємопов'язаних етапів. На першому етапі було ініційовано почергову зупинку критичних служб контролера домену: ядра каталогу, центру розподілу ключів Kerberos, служби локатора та сервера доменних імен. Агент моніторингу успішно зафіксував кожне падіння, згенерував відповідні сповіщення та задіяв алгоритм збереження стану для запобігання дублюванню сигналів під час їхнього одночасного простою (рис. 3.16). Після перезапуску служб система автоматично очистила кеш аварій та повернулася до штатного режиму моніторингу.

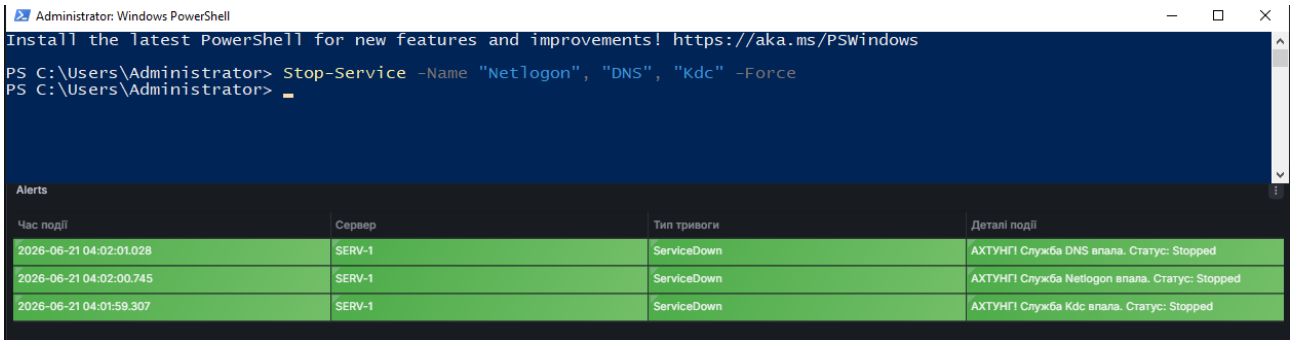


Рис. 3.16. Зупинення сервісів та перевірка роботи.

На другому етапі було згенеровано повний спектр подій інформаційної безпеки, закладених у логіку делегованої фільтрації. Шляхом стандартних маніпуляцій в консолі PowerShell було ініційовано події створення нового облікового запису, додавання його до глобальної групи безпеки та подальшого видалення (рис 3.17). Додатково було імітовано спробу підбору пароля, що призвело до фіксації помилок попередньої автентифікації Kerberos та автоматичного блокування облікового запису тестового користувача (рис.3.18). Фінальним етапом перевірки стало примусове очищення журналу аудиту безпеки (рис. 3.19). Усі без винятку згенеровані інциденти були миттєво перехоплені агентом, класифіковані за відповідними ідентифікаторами та успішно доставлені на центральний сервер. Перевірка панелей аналітики Grafana та каналу корпоративного месенджера підтвердила стовідсоткову реєстрацію тестових подій без втрати пакетів чи затримок на мережевому рівні.

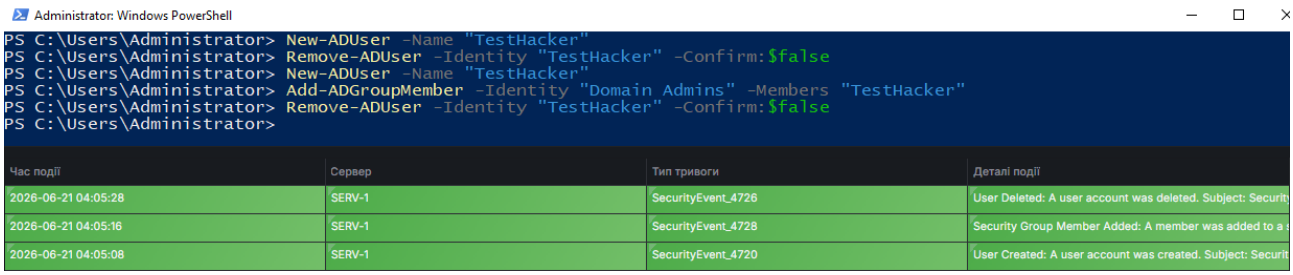


Рис. 3.17. Створення користувача, додавання до групи, його видалення та перевірка роботи сервісу.

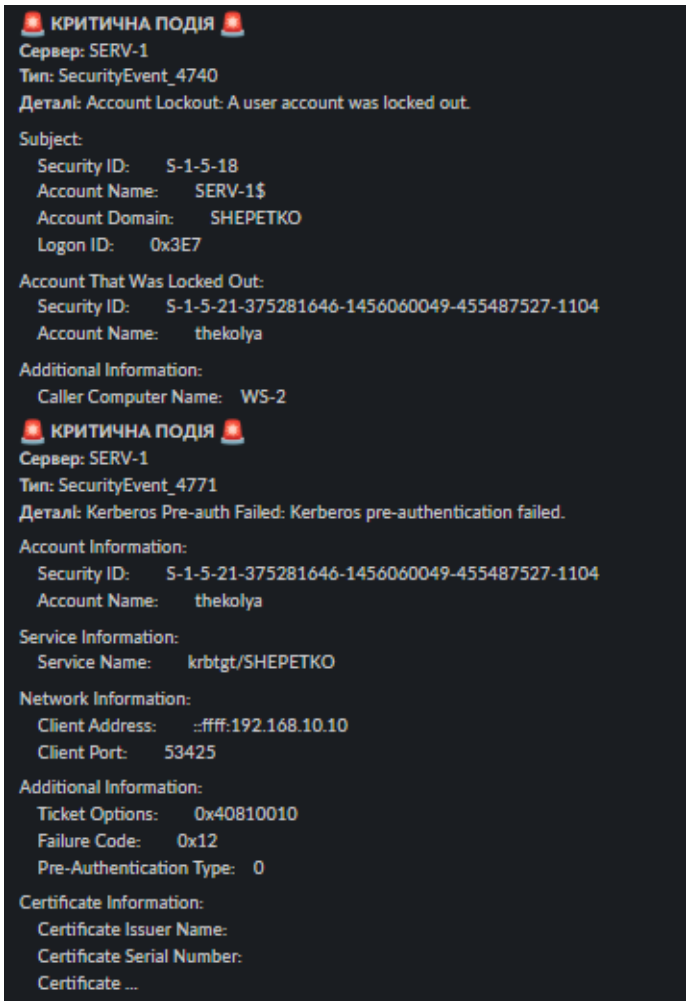


Рис. 3.18. Перевірка невдалої авторизації та блокування облікового запису.

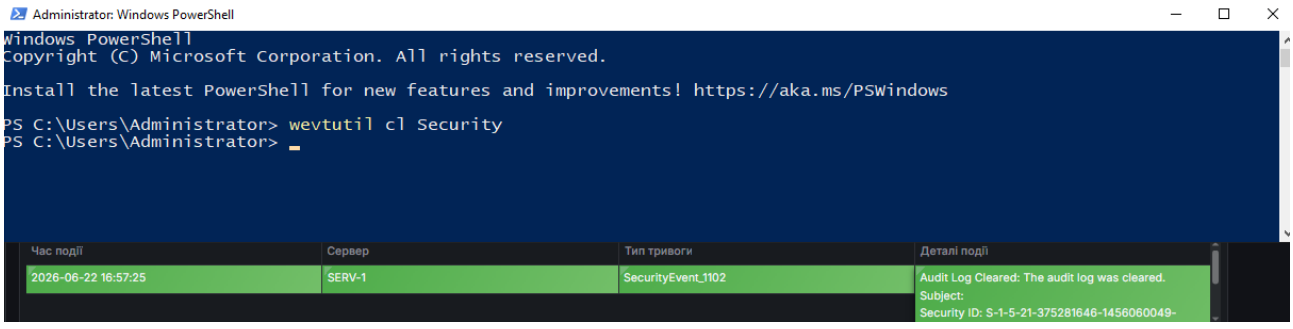


Рис. 3.19. Очищення журналу аудиту та перевірка роботи сервісу.

Для перевірки інтелектуального алгоритму аналізу відсотку завантаженості процесора було проведено тест на стійкість до хибних спрацьовувань. Було тимчасово обмежено потужності сервера у налаштуваннях VirtualBox та створено штучне навантаження. Клієнтський агент зафіксував ці показники та заніс їх до черги, проте середня арифметична величина ковзного вікна за дві хвилини не досягла 90%. Система успішно розпізнала цей сплеск як безпечний і не відправила тривожний сигнал.

На другому етапі навантаження було запущено на дві хвилини. Наприкінці другої хвилини безперервного стрес-тесту черга ковзного вікна повністю заповнилася максимальними показниками, і її середнє значення досягло дев'яноста п’яти відсотків. Алгоритм миттєво згенерував критичне сповіщення та надіслав пакет на сервер, який ретранслював повідомлення у месенджер. Протягом наступної хвилини тест продовжувався, але завдяки увімкненому механізму охолодження таймауту на п'ять хвилин, агент не створив жодного повторного сповіщення, повністю підтвердивши закладену логіку захисту від спаму (рис. 3.20).

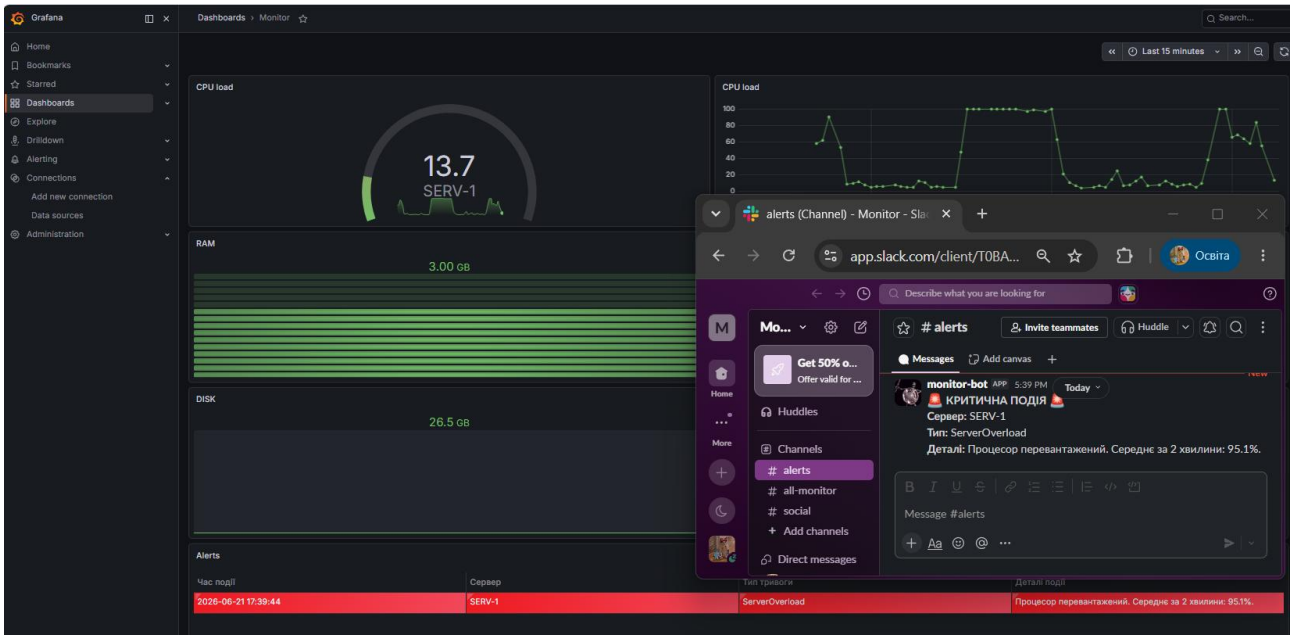


Рис. 3.20. Перевірка роботи ковзного вікна.

Критичним показником системи моніторингу є її власне споживання апаратних ресурсів. У попередніх розділах було закладено низку архітектурних рішень для мінімізації впливу агента на цільові комп'ютери. Для практичного

підтвердження цих теоретичних викладок було проведено перевірку фонового процесу агента під час його безперервної штатної роботи. Вимірювання проводилися за допомогою вбудованого системного монітора ресурсів на контролері домену під стандартним навантаженням.

Результати тестування підтвердили ефективність обраного інженерного підходу. Завдяки відмові від важких системних запитів та переходу на прямі виклики функцій ядра через імпорт нативних бібліотек, споживання оперативної пам'яті процесом стабілізувалося на рівні близько 20 мегабайт. Протягом тривалого часу роботи (30 хвилин) не було зафіксовано жодних витоків пам'яті (рис. 3.21).



Рис. 3.21. Навантаження на систему в час простою та безпосередньо обміну даними.

Додатково було проведено тестування програми в умовах високого навантаженням. В конфігурації агенту було зменшено частоту опитування з 10 секунд до 1 секунди і запущено 20 процесів на кінцевому пристрої. Такий тест показує приблизне навантаження при роботі в штатному режимі до 200 пристроїв. В результаті тестування використання оперативної пам’яті стабілізувалось на рівні близько 50 мб, а використання процесора близько 0.7%, що відповідає критеріям ефективності (рис. 3.22).

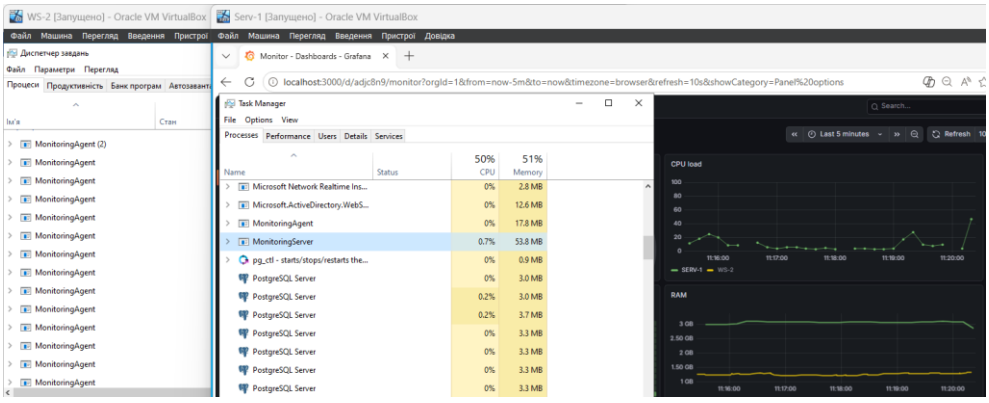


Рис. 3.22. Навантаження на систему під час високого навантаження.

Висновки до розділу 3

У третьому розділі дипломної роботи було успішно виконано весь комплекс практичних робіт із розгортання лабораторного тестового стенда, програмної реалізації компонентів та комплексного випробування створеної системи моніторингу. На базі сучасних технологій віртуалізації побудовано ізольовану мережу, у межах якої розгорнуто контролер домену Active Directory з повним набором інфраструктурних служб та успішно інтегровано кросплатформні клієнтські вузли. Усі програмні модулі серверної та клієнтської частин скомпільовано у вигляді незалежних виконуваних файлів і впроваджено в операційні системи як автоматичні фонові служби. Високий рівень безпеки розгортання забезпечено створенням окремого сервісного облікового запису з обмеженими правами та налаштуванням локальних змінних середовища користувача для ізоляції чутливих даних. Завдяки стандартному механізму побудови конфігурації у платформі .NET було реалізовано динамічний перезапис текстових заглушок актуальними секретами в оперативній пам'яті, що повністю усунуло ризик компрометації паролів бази даних PostgreSQL та токенів авторизації месенджера.

Мережева взаємодія між компонентами стенда реалізована за допомогою високопродуктивного фреймворку віддаленого виклику процедур gRPC, а налаштування веб-сервера Kestrel дозволило впровадити надійний криптографічний захист каналів зв'язку через завантаження цифрових сертифікатів із системного сховища за їх унікальними відбитками. Автоматизація розгортання сховища даних забезпечена інструментарієм міграцій Entity Framework Core, який під час старту сервера самостійно генерує та оновлює структуру таблиць у реляційній базі даних. Для забезпечення безперервного контролю розгорнуто аналітичну платформу Grafana, де інформаційні панелі візуалізації телеметрії та таблиця для відображення зафіксованих інцидентів безпеки.

Проведене комплексне випробування системи повністю підтвердило працездатність та високу ефективність усіх закладених алгоритмів.

Експериментальна перевірка алгоритму ковзного вікна довела його здатність успішно ігнорувати короткочасні сплески та виявляти реальні тривалі перевантаження процесора із застосуванням таймауту охолодження для захисту від спаму повідомленнями. Випробування підсистеми контролю доменних служб підтвердили точність роботи стейтфул-алгоритму на базі хеш-таблиць HashSet, який миттєво фіксує падіння критичних процесів та надійно блокує дублювання сповіщень під час аварій. Тестування підсистеми безпеки на базі запитів мовою XPath продемонструвало швидку фіксацію спроб запуску забороненого програмного забезпечення, створення або видалення користувачів, помилок автентифікації Kerberos та примусового очищення журналів аудиту. Результати профілювання продуктивності засвідчили мінімальний апаратний слід клієнтського агента, споживання оперативної пам'яті якого стабілізувалося на позначці двадцяти мегабайт при практично нульовому навантаженні на процесор у стані спокою, що доводить готовність комплексу до впровадження у реальну інфраструктуру підприємства.

ВИСНОВКИ

У кваліфікаційній роботі було вирішено актуальне практичне завдання, яке полягає у проєктуванні та розробці комплексної кросплатформної системи моніторингу апаратних ресурсів та подій інформаційної безпеки в корпоративній інфраструктурі на базі каталогу Active Directory. Метою роботи було проєктування та розробка високопродуктивної, захищеної клієнт-серверної системи для централізованого збору апаратної телеметрії та моніторингу подій безпеки Active Directory в корпоративних мережах. За підсумками виконання проєкту цю мету було повністю досягнуто.

На етапі системного аналізу було проаналізовано теоретичні основи та обґрунтовано вибір технологій для побудови системи моніторингу Active Directory. На основі порівняння архітектурних підходів визначено, що агентна Push-модель є найбільш доцільною, оскільки вона усуває потребу в постійному скануванні мережі та відкритті вхідних портів на критичних вузлах. Для забезпечення високої пропускної здатності та безпеки обрано бінарний протокол gRPC поверх HTTP/2 із шифруванням TLS на базі сертифікатів X.509. Також обґрунтовано використання платформи .NET для створення кросплатформних фонових служб, СУБД PostgreSQL із фреймворком Entity Framework Core для надійного збереження метрик і логів, а також платформи Grafana та вебхуків Slack для візуалізації та миттєвого оповіщення про інциденти.

На проєктування було закладено фундаментальні рішення, що забезпечили високу ефективність розробленої системи. Обґрунтовано відмову від важких застарілих механізмів збору даних на користь низькорівневої взаємодії з ядрами операційних систем через імпорт нативних системних бібліотек та пряме читання віртуальних файлів стану. Для забезпечення надійного обміну даними між клієнтськими вузлами та центральним сервером інтегровано сучасний фреймворк віддаленого виклику процедур, який використовує переваги мультиплексування з'єднань та швидкої бінарної серіалізації. Крім того, спроєктовано власні алгоритми обробки телеметрії, зокрема механізм ковзного вікна для згладжування хибних сплесків навантаження процесора та стейтфул-

алгоритм на базі внутрішніх колекцій для відстеження стану критичних інфраструктурних служб.

Програмна реалізація комплексу довела життєздатність та надійність обраних підходів. Клієнтські агенти та серверний додаток розроблено як незалежні фонові процеси, що гарантує їхню стабільну роботу в корпоративному середовищі. Високий пріоритет було надано питанням інформаційної безпеки: реалізовано ізоляцію конфіденційних конфігураційних даних за допомогою сервісних облікових записів та динамічного перезапису змінних середовища, впроваджено обов'язковий криптографічний захист усього мережевого трафіку цифровими сертифікатами та налаштовано делеговану фільтрацію подій аудиту оптимізованими текстовими запитами. Для масштабованого збереження метрик розгорнуто надійну реляційну базу даних із автоматизованим управлінням схемою таблиць.

Комплексне тестування на розгорнутому віртуальному стенді повністю підтвердило закладені інженерні гіпотези. Розроблена система на практиці довела свою здатність миттєво фіксувати спроби запуску несанкціонованого програмного забезпечення, маніпуляції з обліковими записами, помилки автентифікації та падіння служб каталогу, автоматично сповіщаючи системного адміністратора через інтеграцію з корпоративним месенджером. Результати профілювання продуктивності засвідчили мінімальний вплив агентів на цільові операційні системи, що дозволяє вільно розгортати комплекс навіть на застарілому серверному обладнанні. Інтеграція спеціалізованої платформи візуалізації забезпечила системного адміністратора зручним інструментом централізованого контролю. Загалом розроблений програмний продукт є повністю функціональним, відмовостійким та готовим до впровадження у реальні корпоративні мережі.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Worker Services in .NET [Електронний ресурс] - Режим доступу: https://learn.microsoft.com/en-us/dotnet/core/extensions/workers
2. Introduction to gRPC [Електронний ресурс] - Режим доступу: https://grpc.io/docs/what-is-grpc/introduction/
3. Hypertext Transfer Protocol Version 2 (HTTP/2) [Електронний ресурс] - Режим доступу: https://datatracker.ietf.org/doc/html/rfc7540
4. Entity Framework Core Overview [Електронний ресурс] - Режим доступу: https://learn.microsoft.com/en-us/ef/core/
5. Npgsql Entity Framework Core Provider [Електронний ресурс] - Режим доступу: https://www.npgsql.org/efcore/
6. PostgreSQL 16 Official Documentation [Електронний ресурс] - Режим доступу: https://www.postgresql.org/docs/16/index.html
7. Grafana Documentation [Електронний ресурс] - Режим доступу: https://grafana.com/docs/grafana/latest/
8. Advanced Security Audit Policy Settings [Електронний ресурс] - Режим доступу: https://learn.microsoft.com/en-us/windows/security/threat-protection/auditing/advanced-security-audit-policy-settings
9. Consuming Events (Windows Event Log) [Електронний ресурс] - Режим доступу: https://learn.microsoft.com/en-us/windows/win32/wes/consuming-events
10. Kestrel web server in ASP.NET Core [Електронний ресурс] - Режим доступу: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel
11. Platform Invoke (P/Invoke) [Електронний ресурс] - Режим доступу: https://learn.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke
12. GlobalMemoryStatusEx function (sysinfoapi.h) [Електронний ресурс] - Режим доступу: https://learn.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-globalmemorystatusex

13. X509Store Class and Certificate Storage [Електронний ресурс] - Режим доступу: https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.x509certificates.x509store
14. Sending messages using Incoming Webhooks [Електронний ресурс] - Режим доступу: https://api.slack.com/messaging/webhooks
15. Price M. J. C# 12 and .NET 8 – Modern Cross-Platform Development Fundamentals / Mark J. Price. – Birmingham: Packt Publishing, 2023. – 828 p.
16. Indrasiri K. gRPC: Up and Running / Kasun Indrasiri, Danesh Kuruppu. – Sebastopol: O'Reilly Media, 2020. – 202 p.
17. Obe R. O. PostgreSQL: Up and Running, 3rd Edition / Regina O. Obe, Leo S. Hsu. – Sebastopol: O'Reilly Media, 2017. – 328 p.
18. Best Practices for Securing Active Directory [Електронний ресурс] - Режим доступу: https://learn.microsoft.com/en-us/windows-server/identity/ad-ds/plan/security-best-practices/best-practices-for-securing-active-directory
19. The Transport Layer Security (TLS) Protocol Version 1.3 (RFC 8446) [Електронний ресурс] - Режим доступу: https://datatracker.ietf.org/doc/html/rfc8446
20. Protocol Buffers Language Guide [Електронний ресурс] - Режим доступу: https://learn.microsoft.com/en-us/aspnet/core/grpc/protobuf
21. Belshe M. Hypertext Transfer Protocol Version 2 (HTTP/2) (RFC 7540) [Електронний ресурс] - Режим доступу: https://datatracker.ietf.org/doc/html/rfc7540

ДОДАТКИ

Програмний код компонента MonitoringServer

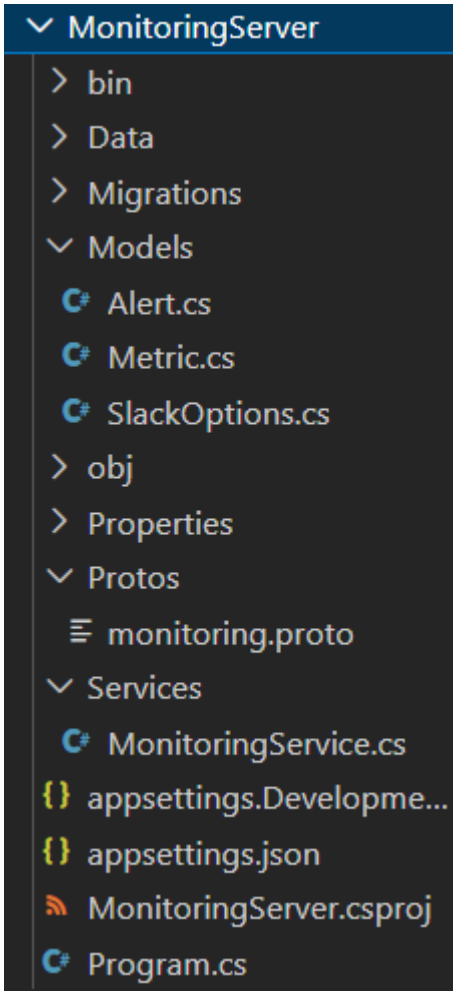


Рис. 4.1. Структура MonitoringServer.

Лістинг Program.cs

```
using MonitoringServer.Services;
using Microsoft.EntityFrameworkCore;
using MonitoringServer.Data;
using System.Security.Cryptography.X509Certificates;

var options = new WebApplicationOptions
{
    Args = args,
    ContentRootPath = AppContext.BaseDirectory
};
var builder = WebApplication.CreateBuilder(options);

builder.Host.UseWindowsService();

builder.Services.AddDbContext<AppDbContext>(dbOptions =>
    dbOptions.UseNpgsql(builder.Configuration.GetConnectionString("PostgresConnecti
on")));
```

```
builder.Services.Configure<MonitoringServer.Models.SlackOptions>(builder.Configurat
ion.GetSection("SlackOptions"));

builder.Services.AddGrpc(grpcOptions =>
{
    grpcOptions.EnableDetailedErrors = builder.Environment.IsDevelopment();
});


builder.WebHost.ConfigureKestrel(kestrel =>
{
    kestrel.ListenAnyIP(5050, listenOptions =>
    {
        listenOptions.Protocols =
Microsoft.AspNetCore.Server.Kestrel.Core.HttpProtocols.Http2;

        using var store = new X509Store(StoreName.My, StoreLocation.LocalMachine);
        store.Open(OpenFlags.ReadOnly);

        var certs = store.Certificates.Find(X509FindType.FindByThumbprint,
"F808DE70065B6AD8E35C35A758A2D46F5D0ABD4A", false);

        if (certs.Count > 0)
        {
            listenOptions.UseHttps(certs[0]);
        }
        else
        {
            throw new Exception("ПОМИЛКА: Сертифікат не знайдено за цим
відбитком!");
        }
    });
});

var app = builder.Build();

app.MapGrpcService<MonitoringService>();

app.MapGet("/", () =>
    "MonitoringServer is running. Connect via a gRPC client.");

Console.ForegroundColor = ConsoleColor.Magenta;
Console.WriteLine("═══════════════════════════════════════════════════");
Console.WriteLine("        MonitoringServer  |  Listening on :5050   ");
Console.WriteLine("═══════════════════════════════════════════════════");
Console.ResetColor();

using (var scope = app.Services.CreateScope())
{
    var context = scope.ServiceProvider.GetRequiredService<AppDbContext>();
    context.Database.Migrate();
```

```
}

app.Run();
```

Лістинг appsettings.json

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "PostgresConnection": "Hampter"
  },
  "SlackOptions": {
    "WebhookUrl": "Hampter"
  }
}
```

Лістинг MonitoringService.cs:

```
using Grpc.Core;
using MonitoringSystem.Contracts;
using MonitoringServer.Data;
using MonitoringServer.Models;
using MonitoringSystem;
using Microsoft.Extensions.Options;
using System.Text;
using System.Text.Json;

namespace MonitoringServer.Services;

public sealed class MonitoringService : SystemMonitor.SystemMonitorBase
{
    private readonly ILogger<MonitoringService> _logger;
    private readonly AppDbContext _dbContext;
    private readonly SlackOptions _slackOptions;

    private static readonly HttpClient _httpClient = new HttpClient();

    public MonitoringService(
        ILogger<MonitoringService> logger,
        AppDbContext dbContext,
        IOptions<SlackOptions> slackOptions)
    {
        _logger = logger;
        _dbContext = dbContext;
        _slackOptions = slackOptions.Value;
    }
```

```
 public override async Task<MetricsResponse> PushMetrics(
        MetricsRequest request,
        ServerCallContext context)
    {
        var timestamp = DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss");
        var separator = new string('─', 52);

        Console.ForegroundColor = ConsoleColor.Cyan;
        Console.WriteLine($"\n┌{separator}┐");
        Console.ResetColor();

        Console.Write("│ ");
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.Write($"  Metrics Received  [{timestamp}]");
        Console.ResetColor();
        Console.WriteLine("          │");

        Console.ForegroundColor = ConsoleColor.Cyan;
        Console.WriteLine($"├{separator}┤");
        Console.ResetColor();

        WriteMetricRow("  Machine",   request.MachineName,                padRight:
38);
        WriteMetricRow("  CPU Usage", $"{request.CpuUsagePercent:F1}
%",        padRight: 38);
        WriteMetricRow("  Free RAM",  $"{request.AvailableRamMb:F0}
MB",       padRight: 38);
        WriteMetricRow("  Free Disk", $"{request.FreeDiskSpaceGb:F2}
GB",      padRight: 38);
        WriteMetricRow("  Uptime",    $"{request.UptimeHours:F1}
hours",       padRight: 38);
        WriteMetricRow("  Network",   $"{request.NetTrafficMbps:F2}
MB",       padRight: 38);

        Console.ForegroundColor = ConsoleColor.Cyan;
        Console.WriteLine($"└{separator}┘");
        Console.ResetColor();

        _logger.LogInformation(
            "Metrics from {Machine} | CPU: {Cpu:F1}% | RAM: {Ram:F0} MB | Disk:
{Disk:F2} GB",
            request.MachineName, request.CpuUsagePercent, request.AvailableRamMb,
request.FreeDiskSpaceGb);

        var metricRecord = new Metric
        {
            MachineName = request.MachineName,
            Timestamp = DateTime.UtcNow,
            CpuUsage = request.CpuUsagePercent,
            RamMb = request.AvailableRamMb,
            DiskGb = request.FreeDiskSpaceGb,
```

```
 UptimeHours = request.UptimeHours,
            NetTrafficMbps = request.NetTrafficMbps
        };

        _dbContext.Metrics.Add(metricRecord);
        await _dbContext.SaveChangesAsync();

        return new MetricsResponse
        {
            Success = true,
            Message = $"Metrics from '{request.MachineName}' saved to DB at
{timestamp}."
        };
    }

    private static void WriteMetricRow(string label, string value, int padRight)
    {
        Console.Write("| ");
        Console.ForegroundColor = ConsoleColor.Green;
        Console.Write($"{label,-14}");
        Console.ResetColor();
        Console.Write(": ");
        Console.ForegroundColor = ConsoleColor.White;
        Console.Write(value.PadRight(padRight));
        Console.ResetColor();
        Console.WriteLine(" |");
    }

    public override async Task<MetricsResponse> PushAlert(
        AlertRequest request,
        ServerCallContext context)
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine($"\n[АЛЕРТ!] Машина: {request.MachineName}");
        Console.WriteLine($"Тип: {request.AlertType}");
        Console.WriteLine($"Повідомлення: {request.Message}");
        Console.ResetColor();

        var alertRecord = new Alert
        {
            MachineName = request.MachineName,
            Timestamp = DateTime.UtcNow,
            AlertType = request.AlertType,
            Message = request.Message
        };

        _dbContext.Alerts.Add(alertRecord);
        await _dbContext.SaveChangesAsync();

        Console.ForegroundColor = ConsoleColor.Yellow;
```

```
        Console.WriteLine($"[DEBUG] Поточний URL для Slack:
'{_slackOptions.WebhookUrl}'");
        Console.ResetColor();


        if (!string.IsNullOrEmpty(_slackOptions.WebhookUrl) &&
            _slackOptions.WebhookUrl != "Hampter")
        {
            try
            {
                var payload = new
                {
                    text = $"🚨 *КРИТИЧНА ПОДІЯ* 🚨\n*Сервер:*
{request.MachineName}\n*Тип:* {request.AlertType}\n*Деталі:* {request.Message}"
                };

                var content = new StringContent(JsonSerializer.Serialize(payload),
Encoding.UTF8, "application/json");

                await _httpClient.PostAsync(_slackOptions.WebhookUrl, content);
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "Failed to send alert to Slack.");
            }
        }

        return new MetricsResponse
        {
            Success = true,
            Message = "Алерт збережено в БД і відправлено в месенджер!"
        };
    }
}
```

Лістинг Alert.cs:

```
namespace MonitoringServer.Models;

public class Alert
{
    public int Id { get; set; }
    public string MachineName { get; set; } = string.Empty;
    public DateTime Timestamp { get; set; }
    public string AlertType { get; set; } = string.Empty;
    public string Message { get; set; } = string.Empty;
}
```

Лістинг Metric.cs:

```
namespace MonitoringServer.Models;

public class Metric
{
```

```
  public int Id { get; set; }
    public string MachineName { get; set; } = string.Empty;
    public DateTime Timestamp { get; set; }
    public double CpuUsage { get; set; }
    public double RamMb { get; set; }
    public double DiskGb { get; set; }
    public double UptimeHours { get; set; }
    public double NetTrafficMbps { get; set; }
}
```

Лістинг SlackOptions.cs:

```
namespace MonitoringServer.Models;

public sealed class SlackOptions
{
    public string WebhookUrl { get; set; } = string.Empty;
}
```

Лістинг monitoring.proto:

```
syntax = "proto3";

option csharp_namespace = "MonitoringSystem.Contracts";

package monitoring;

service SystemMonitor {
  rpc PushMetrics (MetricsRequest) returns (MetricsResponse);
  rpc PushAlert (AlertRequest) returns (MetricsResponse);
}

message MetricsRequest {
  string machine_name        = 1;
  double cpu_usage_percent   = 2;
  double available_ram_mb    = 3;
  double free_disk_space_gb  = 4;
  double uptime_hours        = 5;
  double net_traffic_mbps    = 6;
}

message AlertRequest {
  string machine_name = 1;
  string alert_type   = 2;
  string message      = 3;
}

message MetricsResponse {
  bool   success = 1;
  string message = 2;
}
```

Додаток Б

Програмний код компонента MonitoringAgent

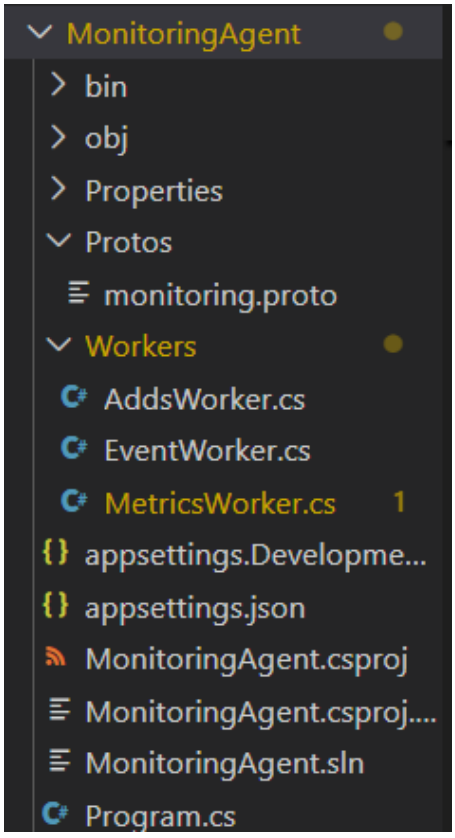


Рис. 4.2. Структура проєкту програми-клієнта.

Лістинг Program.cs:

```
using MonitoringAgent.Workers;

var host = Host.CreateDefaultBuilder(args)
    .UseWindowsService(options =>
    {
        options.ServiceName = "Monitoring Agent";
    })
    .ConfigureServices((context, services) =>
    {
        var optionsSection = context.Configuration.GetSection("AgentOptions");
        services.Configure<AgentOptions>(optionsSection);

        var agentOptions = optionsSection.Get<AgentOptions>();

        services.AddHostedService<MetricsWorker>();
        services.AddHostedService<EventWorker>();

        if (agentOptions != null && agentOptions.IsDomainController)
        {
            services.AddHostedService<AddsWorker>();
        }
```

```
 })
    .Build();

await host.RunAsync();
```

Лістинг appsettings.json:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AgentOptions": {
    "ServerAddress": "https://Serv-1:5050",
    "PushIntervalSeconds": 10,
    "MonitoredDriveLetter": "C:",
    "IsDomainController": true
  }
}
```

Лістинг MetricsWorker.cs:

```
using System.Diagnostics;
using System.Runtime.InteropServices;
using Grpc.Net.Client;
using Microsoft.Extensions.Options;
using MonitoringSystem.Contracts;
using System.Management;

namespace MonitoringAgent.Workers;
public sealed class MetricsWorker : BackgroundService
{
    private readonly ILogger<MetricsWorker> _logger;
    private readonly AgentOptions           _options;

    private readonly PerformanceCounter? _cpuCounter;
    private long _lastNetworkBytes = -1;
    private long _lastNetworkTimeTicks = 0;
    private readonly Queue<double> _cpuHistory = new();

    private DateTime _lastOverloadAlertTime = DateTime.MinValue;
    private readonly TimeSpan _alertCooldown = TimeSpan.FromMinutes(5);

    public MetricsWorker(
        ILogger<MetricsWorker> logger,
        IOptions<AgentOptions> options)
    {
        _logger  = logger;
        _options = options.Value;

        if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
```

```
    {
            _cpuCounter = new PerformanceCounter(
                categoryName:  "Processor",
                counterName:   "% Processor Time",
                instanceName:  "_Total",
                readOnly:      true);

            _cpuCounter.NextValue();
        }
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        _logger.LogInformation(
            "MetricsWorker started. Pushing to {ServerAddress} every
{IntervalSec}s.",
            _options.ServerAddress,
            _options.PushIntervalSeconds);

        using var channel = GrpcChannel.ForAddress(_options.ServerAddress, new
GrpcChannelOptions
        {
            HttpHandler = new SocketsHttpHandler
            {
                EnableMultipleHttp2Connections = true,
                KeepAlivePingDelay             = TimeSpan.FromSeconds(60),
                KeepAlivePingTimeout           = TimeSpan.FromSeconds(30),
            }
        });

        var client = new SystemMonitor.SystemMonitorClient(channel);

        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                var request = BuildMetricsRequest();

                _logger.LogInformation(
                    "Pushing metrics → CPU: {Cpu:F1}%| RAM: {Ram:F0} MB | Disk:
{Disk:F2} GB | Up: {Up:F1}h | Net: {Net:F2}MB",
                    request.CpuUsagePercent, request.AvailableRamMb,
                    request.FreeDiskSpaceGb, request.UptimeHours,
request.NetTrafficMbps);

                await CheckServerOverloadAsync(request.CpuUsagePercent,
request.AvailableRamMb, client, stoppingToken);

                var response = await client.PushMetricsAsync(
```

```
 request,
                    cancellationToken: stoppingToken);

                if (response.Success)
                    _logger.LogInformation("Server ACK: {Message}",
response.Message);
                else
                    _logger.LogWarning("Server returned failure: {Message}",
response.Message);
            }
            catch (Exception ex) when (!stoppingToken.IsCancellationRequested)
            {
                _logger.LogError(ex,
                    "Failed to push metrics. Will retry in {Interval}s.",
                    _options.PushIntervalSeconds);
            }

            await Task.Delay(
                TimeSpan.FromSeconds(_options.PushIntervalSeconds),
                stoppingToken);
        }
    }

    // ── Metric Collection ──────────────────────────────────────────────

    private MetricsRequest BuildMetricsRequest() => new()
    {
        MachineName       = Environment.MachineName,
        CpuUsagePercent   = GetCpuUsage(),
        AvailableRamMb    = GetAvailableRamMb(),
        FreeDiskSpaceGb   = GetFreeDiskSpaceGb(_options.MonitoredDriveLetter),
        UptimeHours       =
TimeSpan.FromMilliseconds(Environment.TickCount64).TotalHours,
        NetTrafficMbps    = GetNetworkTraffic(),
    };

    private double GetCpuUsage()
    {
        if (_cpuCounter is not null)
        {
            Thread.Sleep(1000);
            return Math.Round(_cpuCounter.NextValue(), 1);
        }

        return GetLinuxCpuUsage();
    }

    private static double GetLinuxCpuUsage()
    {
        try
```

```
{
            static long[] ReadProcStat()
            {
                var line = File.ReadLines("/proc/stat").First();
                return line.Split(' ', StringSplitOptions.RemoveEmptyEntries)
                           .Skip(1)
                           .Select(long.Parse)
                           .ToArray();
            }

            var s1    = ReadProcStat();
            Thread.Sleep(500);
            var s2    = ReadProcStat();

            long idle1  = s1[3], idle2  = s2[3];
            long total1 = s1.Sum(), total2 = s2.Sum();

            double totalDelta = total2 - total1;
            double idleDelta  = idle2  - idle1;

            if (totalDelta == 0) return 0;
            return Math.Round((1.0 - idleDelta / totalDelta) * 100.0, 1);
        }
        catch
        {
            return -1;
        }
    }

    private static double GetAvailableRamMb()
    {
        if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
            return GetWindowsAvailableRamMb();

        return GetLinuxAvailableRamMb();
    }

private double GetNetworkTraffic()
    {
        try
        {
            var interfaces =
System.Net.NetworkInformation.NetworkInterface.GetAllNetworkInterfaces()
                .Where(i => i.OperationalStatus ==
System.Net.NetworkInformation.OperationalStatus.Up &&
                            i.NetworkInterfaceType !=
System.Net.NetworkInformation.NetworkInterfaceType.Loopback);

            long currentTotalBytes = interfaces.Sum(i =>
                i.GetIPStatistics().BytesReceived + i.GetIPStatistics().BytesSent);
```

```
            long currentTimeTicks = Environment.TickCount64;

            if (_lastNetworkBytes == -1)
            {
                _lastNetworkBytes = currentTotalBytes;
                _lastNetworkTimeTicks = currentTimeTicks;
                return 0;
            }

            long deltaBytes = currentTotalBytes - _lastNetworkBytes;
            double deltaSeconds = (currentTimeTicks - _lastNetworkTimeTicks) /
1000.0;

            _lastNetworkBytes = currentTotalBytes;
            _lastNetworkTimeTicks = currentTimeTicks;

            if (deltaSeconds <= 0 || deltaBytes < 0) return 0;

            double mbps = (deltaBytes * 8.0) / 1_000_000.0 / deltaSeconds;

            return Math.Round(mbps, 2);
        }
        catch
        {
            return -1;
        }
    }

    [System.Runtime.InteropServices.DllImport("kernel32.dll", SetLastError = true)]
    private static extern bool GlobalMemoryStatusEx(ref MEMORYSTATUSEX lpBuffer);

    [System.Runtime.InteropServices.StructLayout(System.Runtime.InteropServices.Lay
outKind.Sequential)]
    private struct MEMORYSTATUSEX
    {
        public uint  dwLength;
        public uint  dwMemoryLoad;
        public ulong ullTotalPhys;
        public ulong ullAvailPhys;
        public ulong ullTotalPageFile;
        public ulong ullAvailPageFile;
        public ulong ullTotalVirtual;
        public ulong ullAvailVirtual;
        public ulong ullAvailExtendedVirtual;
    }

    private static double GetWindowsAvailableRamMb()
    {
        try
```

```
    {
            var status = new MEMORYSTATUSEX { dwLength =
(uint)System.Runtime.InteropServices.Marshal.SizeOf<MEMORYSTATUSEX>() };
            if (GlobalMemoryStatusEx(ref status))
                return Math.Round(status.ullAvailPhys / 1024.0 / 1024.0, 0);
        }
        catch { /* fall through */ }
        return -1;
    }

    private static double GetLinuxAvailableRamMb()
    {
        try
        {
            var memAvailLine = File.ReadLines("/proc/meminfo")
                .FirstOrDefault(l => l.StartsWith("MemAvailable:",
StringComparison.Ordinal));
            if (memAvailLine is null) return -1;

            var kb = long.Parse(memAvailLine.Split(':',
StringSplitOptions.TrimEntries)[1]
                                            .Replace(" kB", "",
StringComparison.Ordinal)
                                            .Trim());
            return Math.Round(kb / 1024.0, 0);
        }
        catch
        {
            return -1;
        }
    }

    private static double GetFreeDiskSpaceGb(string driveLetter)
    {
        try
        {
            var targetDrive = DriveInfo.GetDrives()
                .FirstOrDefault(d =>
                {
                    if (!d.IsReady) return false;

                    try
                    {
                        if (d.TotalSize <= 0) return false;
                    }
                    catch
                    {
                        return false;
                    }
```

```
  if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
                    {
                        return d.Name.StartsWith(driveLetter,
StringComparison.OrdinalIgnoreCase);
                    }
                    else
                    {
                        return d.Name.Equals(driveLetter, StringComparison.Ordinal)
||
                               d.RootDirectory.FullName.Equals(driveLetter,
StringComparison.Ordinal);
                    }
                });

            if (targetDrive is null) return -1;

            return Math.Round(targetDrive.AvailableFreeSpace / 1024.0 / 1024.0 /
1024.0, 2);
        }
        catch
        {
            return -1;
        }
    }

    public override void Dispose()
    {
        _cpuCounter?.Dispose();
        base.Dispose();
    }
    private async Task CheckServerOverloadAsync(double currentCpu, double
currentAvailableRamMb, SystemMonitor.SystemMonitorClient client, CancellationToken
ct)
    {
        if (!_options.IsDomainController) return;

        bool isRamStarved = currentAvailableRamMb > 0 && currentAvailableRamMb <
1024.0;

        if (isRamStarved)
        {
            if (DateTime.UtcNow - _lastOverloadAlertTime > _alertCooldown)
            {
                _logger.LogWarning("Triggering RAM Overload Alert! Free RAM: {Ram}
MB", currentAvailableRamMb);

                await client.PushAlertAsync(new AlertRequest
                {
                    MachineName = Environment.MachineName,
                    AlertType   = "ServerOverload",
```

```
Message     = $"На сервері критично мало RAM. Залишилось:
{currentAvailableRamMb:F0} MB."
                }, cancellationToken: ct);

                _lastOverloadAlertTime = DateTime.UtcNow;
                return;
            }
        }

        int maxSamples = 120 / Math.Max(1, _options.PushIntervalSeconds);
        _cpuHistory.Enqueue(currentCpu);

        if (_cpuHistory.Count > maxSamples)
        {
            _cpuHistory.Dequeue();
        }

        if (_cpuHistory.Count == maxSamples)
        {
            double avgCpu = _cpuHistory.Average();

            if (avgCpu >= 90.0)
            {
                if (DateTime.UtcNow - _lastOverloadAlertTime > _alertCooldown)
                {
                    _logger.LogWarning("Triggering CPU Overload Alert! Avg 2m CPU:
{Cpu}%", avgCpu);

                    await client.PushAlertAsync(new AlertRequest
                    {
                        MachineName = Environment.MachineName,
                        AlertType   = "ServerOverload",
                        Message     = $"Процесор перевантажений. Середнє за 2
хвилини: {avgCpu:F1}%."
                    }, cancellationToken: ct);

                    _lastOverloadAlertTime = DateTime.UtcNow;
                }
            }
        }
    }
}

public sealed class AgentOptions
{
    public string ServerAddress       { get; set; } = "https://localhost:5050";
    public int    PushIntervalSeconds { get; set; } = 10;

    public string MonitoredDriveLetter { get; set; } =
```

```
 RuntimeInformation.IsOSPlatform(OSPlatform.Windows) ? "C:" : "/";

    public bool IsDomainController { get; set; } = false;
}
```

Лістинг EventWorker.cs:

```
using System.Diagnostics.Eventing.Reader;
using Grpc.Net.Client;
using Microsoft.Extensions.Options;
using MonitoringSystem.Contracts;

namespace MonitoringAgent.Workers;

public sealed class EventWorker : BackgroundService
{
    private readonly AgentOptions _options;

    public EventWorker(IOptions<AgentOptions> options)
    {
        _options = options.Value;
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        if (!OperatingSystem.IsWindows()) return;

        using var channel = GrpcChannel.ForAddress(_options.ServerAddress);
        var client = new SystemMonitor.SystemMonitorClient(channel);

        string queryStr = "*[System[(EventID=4740 or EventID=1102 or EventID=4720
or EventID=4726 or EventID=4728 or EventID=4756 or EventID=4719 or EventID=4771 or
EventID=4777 or EventID=4688)]]";

        var query = new EventLogQuery("Security", PathType.LogName, queryStr);
        using var watcher = new EventLogWatcher(query);

        watcher.EventRecordWritten += async (sender, e) =>
        {
            if (e.EventRecord == null) return;
            var record = e.EventRecord;
            var eventId = record.Id;

            if (eventId == 4688)
            {
                var xml = record.ToXml();
                if (xml.Contains("notepad.exe",
StringComparison.OrdinalIgnoreCase))
                {
                    await client.PushAlertAsync(new AlertRequest
                    {
                        MachineName = Environment.MachineName,
```

```
AlertType = "ForbiddenProcess",
                        Message = "Запущено заборонений процес: notepad.exe"
                    });
                }
                return;
            }

            string alertType = $"SecurityEvent_{eventId}";
            string desc = record.FormatDescription() ?? "Опис події недоступний";
            string shortDesc = desc.Length > 500 ? desc[..500] + "..." : desc;

            string message = eventId switch
            {
                4740 => $"Account Lockout: {shortDesc}",
                1102 => $"Audit Log Cleared: {shortDesc}",
                4720 => $"User Created: {shortDesc}",
                4726 => $"User Deleted: {shortDesc}",
                4728 => $"Security Group Member Added: {shortDesc}",
                4756 => $"Global Group Member Added: {shortDesc}",
                4719 => $"Audit Policy Changed: {shortDesc}",
                4771 => $"Kerberos Pre-auth Failed: {shortDesc}",
                4777 => $"DC Validation Failed: {shortDesc}",
                _ => $"Security Event {eventId}: {shortDesc}"
            };

            await client.PushAlertAsync(new AlertRequest
            {
                MachineName = Environment.MachineName,
                AlertType = alertType,
                Message = message
            });
        };

        try
        {
            watcher.Enabled = true;

            while (!stoppingToken.IsCancellationRequested)
            {
                await Task.Delay(10000, stoppingToken);
            }
        }
        catch (UnauthorizedAccessException)
        {
            Console.WriteLine("[ПОМИЛКА] Нема прав адміна для читання журналу
Security!");
        }
        catch (EventLogNotFoundException)
        {
```

```
      Console.WriteLine("[ПОМИЛКА] Журнал Security не знайдено на цій машині.");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"[ПОМИЛКА] {ex.Message}");
        }
        finally
        {
            watcher.Enabled = false;
        }
    }
}
```

Лістинг AddsWorker.cs:

```
using System.ServiceProcess;
using Grpc.Net.Client;
using Microsoft.Extensions.Options;
using MonitoringSystem.Contracts;

namespace MonitoringAgent.Workers;

public sealed class AddsWorker : BackgroundService
{
    private readonly AgentOptions _options;
    private readonly HashSet<string> _downServices = new();

    public AddsWorker(IOptions<AgentOptions> options)
    {
        _options = options.Value;
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        if (!OperatingSystem.IsWindows() || !_options.IsDomainController)
            return;

        using var channel = GrpcChannel.ForAddress(_options.ServerAddress);
        var client = new SystemMonitor.SystemMonitorClient(channel);

        string[] criticalServices = { "NTDS", "Kdc", "Netlogon", "DNS" };

        while (!stoppingToken.IsCancellationRequested)
        {
            foreach (var svcName in criticalServices)
            {
                try
                {
                    using var sc = new ServiceController(svcName);

                    if (sc.Status != ServiceControllerStatus.Running)
```

```
{
                    if (!_downServices.Contains(svcName))
                    {
                        await client.PushAlertAsync(new AlertRequest {
                            MachineName = Environment.MachineName,
                            AlertType = "ServiceDown",
                            Message = $"АХТУНГ! Служба {svcName} впала. Статус:
{sc.Status}"
                        }, cancellationToken: stoppingToken);

                        _downServices.Add(svcName);
                    }
                }
                else
                {
                    if (_downServices.Contains(svcName))
                    {
                        _downServices.Remove(svcName);
                    }
                }
            }
            catch
            {
                // Служби нема або немає доступу
            }
        }

        await Task.Delay(TimeSpan.FromSeconds(30), stoppingToken);
    }
  }
}
```

Додаток В

Додаткові скриншоти налаштувань та тестування проєкту

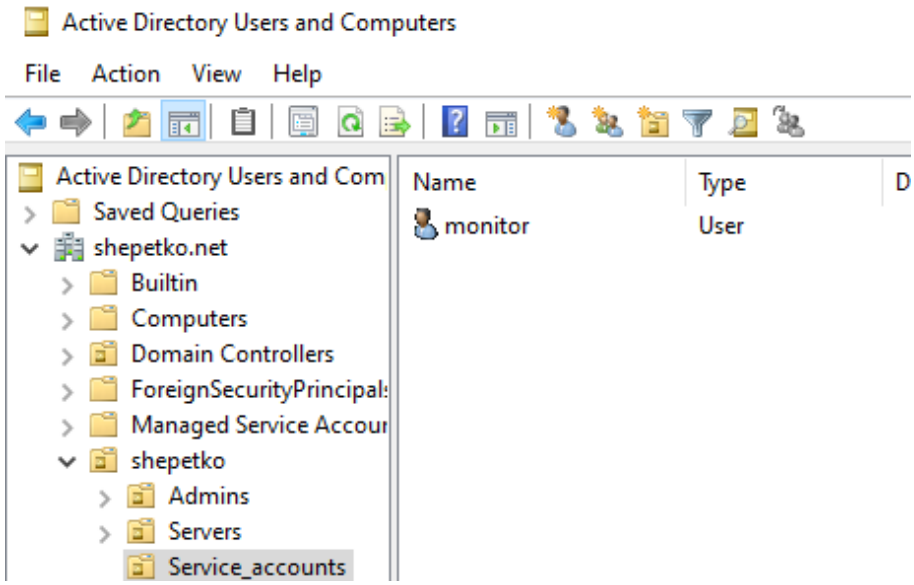


Рис. 4.3. Створення сервісного облікового запису.

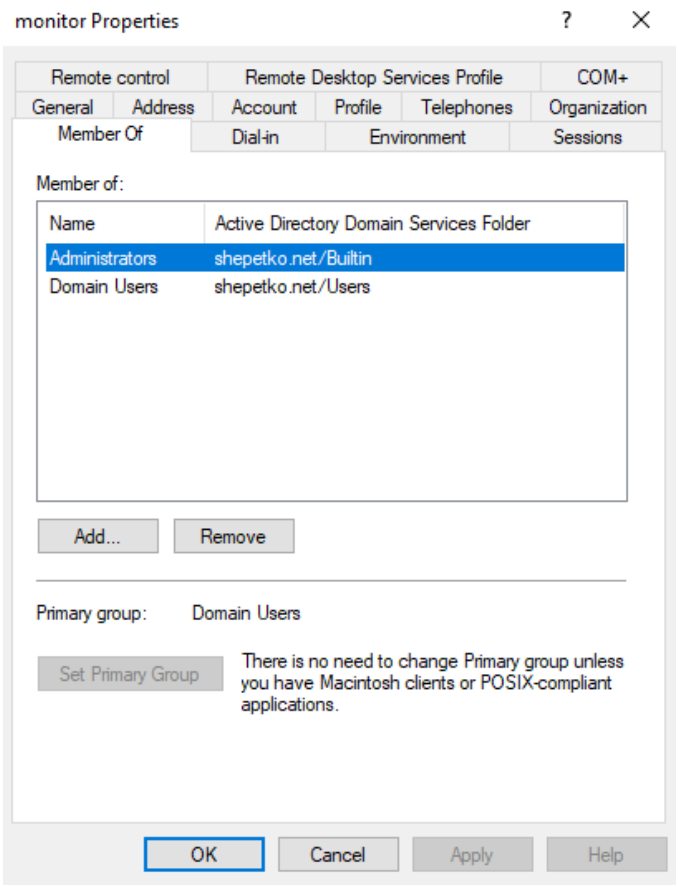


Рис. 4.4. Налаштування членства у групах для сервісного акаунта.

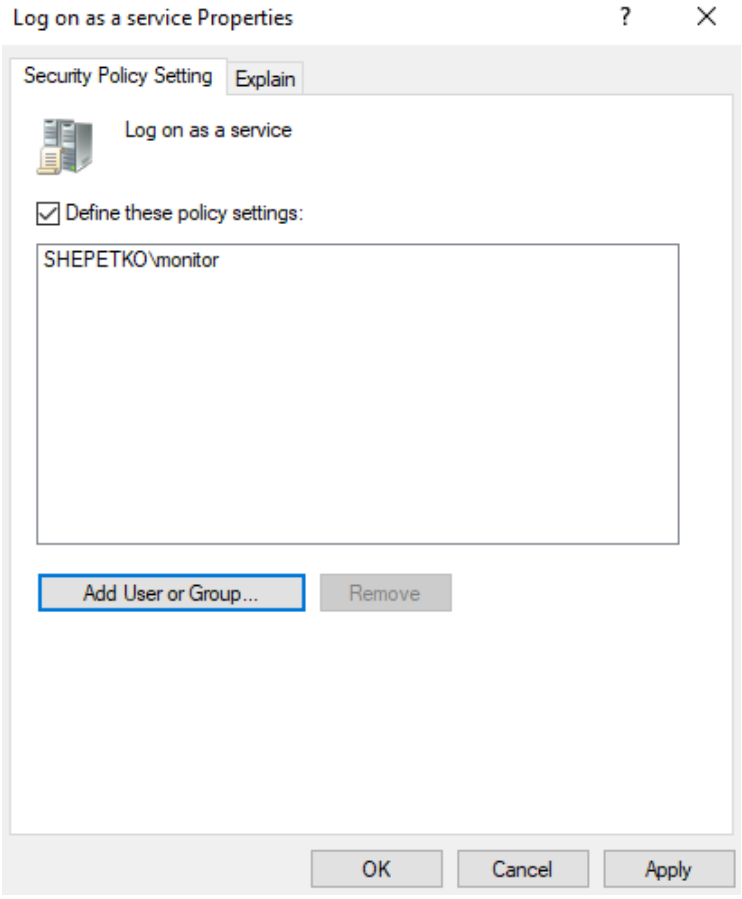


Рис. 4.5. Надання прав входу як служби.

```
C:\Users\Administrator>sc.exe create MonitoringServer binPath= "C:\MonitoringServer\MonitoringServer.exe" DisplayNa
me= "Network Monitoring Server" start= auto obj= "SHEPETKO\monitor" password= "1111Aa$"
[SC] CreateService SUCCESS
```

Рис. 4.6. Створення служби MonitoringServer.

```
C:\Users\Administrator>sc.exe create MonitoringAgent binPath= "C:\MonitoringAgent\MonitoringAgent.exe" DisplayName=
 "Network Monitoring Agent" start= auto obj= "SHEPETKO\monitor" password= "1111Aa$"
[SC] CreateService SUCCESS
```

Рис. 4.7. Створення служби MonitoringAgent.

```
C:\Users\Administrator>sc.exe start MonitoringServer

SERVICE_NAME: MonitoringServer
        TYPE               : 10  WIN32_OWN_PROCESS
        STATE              : 2  START_PENDING
                               (NOT_STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE    : 0  (0x0)
        SERVICE_EXIT_CODE  : 0  (0x0)
        CHECKPOINT         : 0x0
        WAIT_HINT          : 0x7d0
        PID                : 1048
        FLAGS              :

C:\Users\Administrator>sc.exe start MonitoringAgent

SERVICE_NAME: MonitoringAgent
        TYPE               : 10  WIN32_OWN_PROCESS
        STATE              : 2  START_PENDING
                               (NOT_STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE    : 0  (0x0)
        SERVICE_EXIT_CODE  : 0  (0x0)
        CHECKPOINT         : 0x0
        WAIT_HINT          : 0x7d0
        PID                : 4032
        FLAGS              :
```

Рис. 4.8. Запуск створених служб у командному рядку.

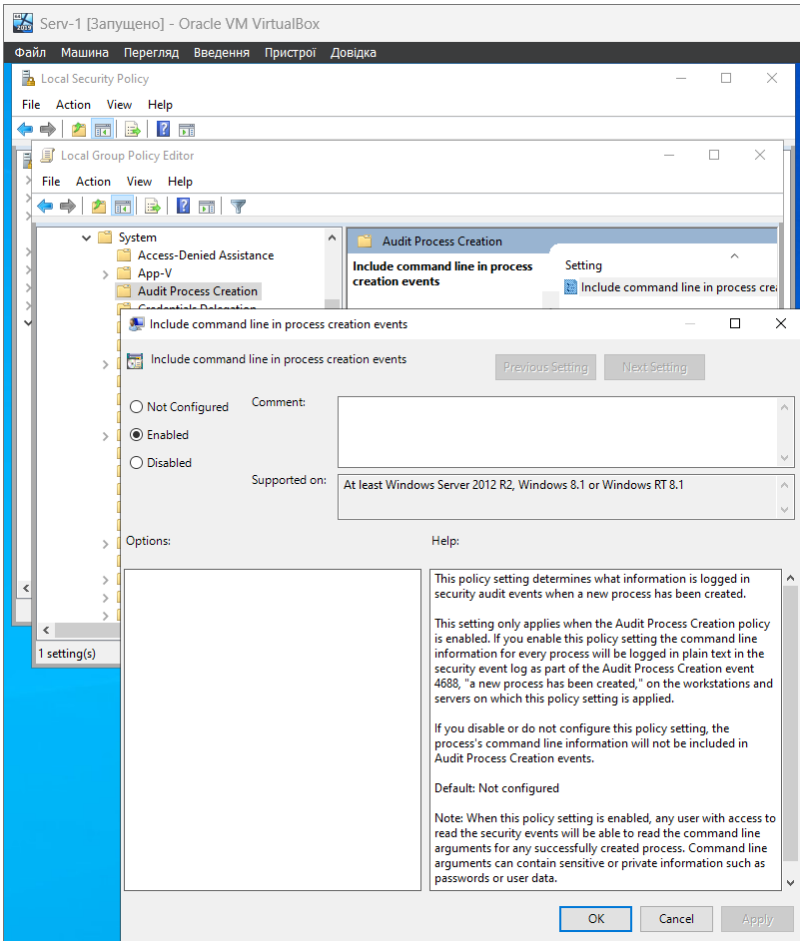


Рис. 4.9. Увімкнення аудиту командного рядка під час створення процесів.

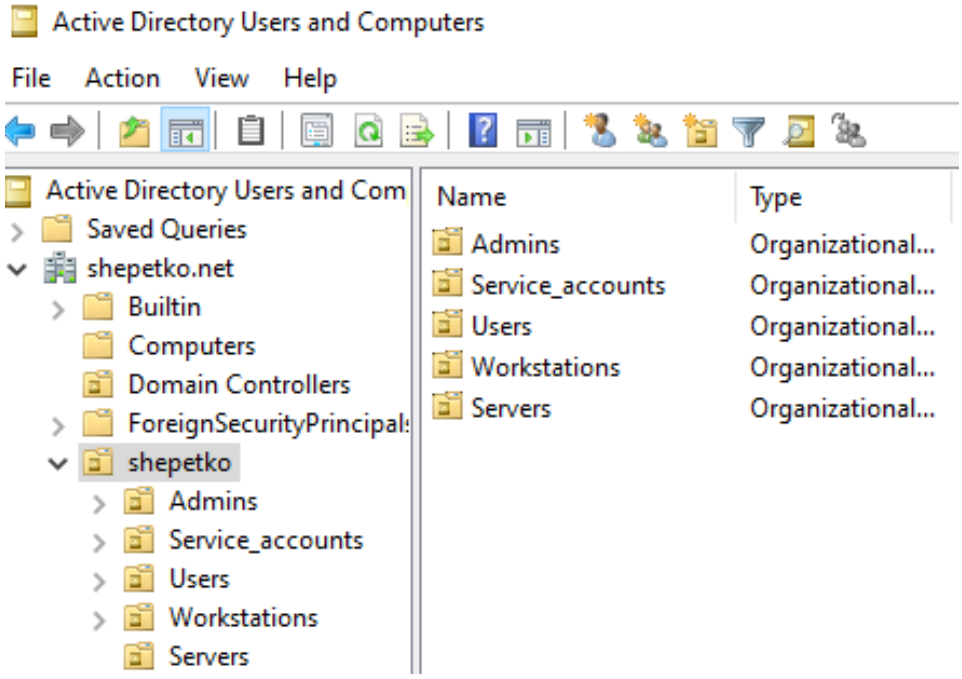


Рис. 4.10. Організаційна структура Active Directory.

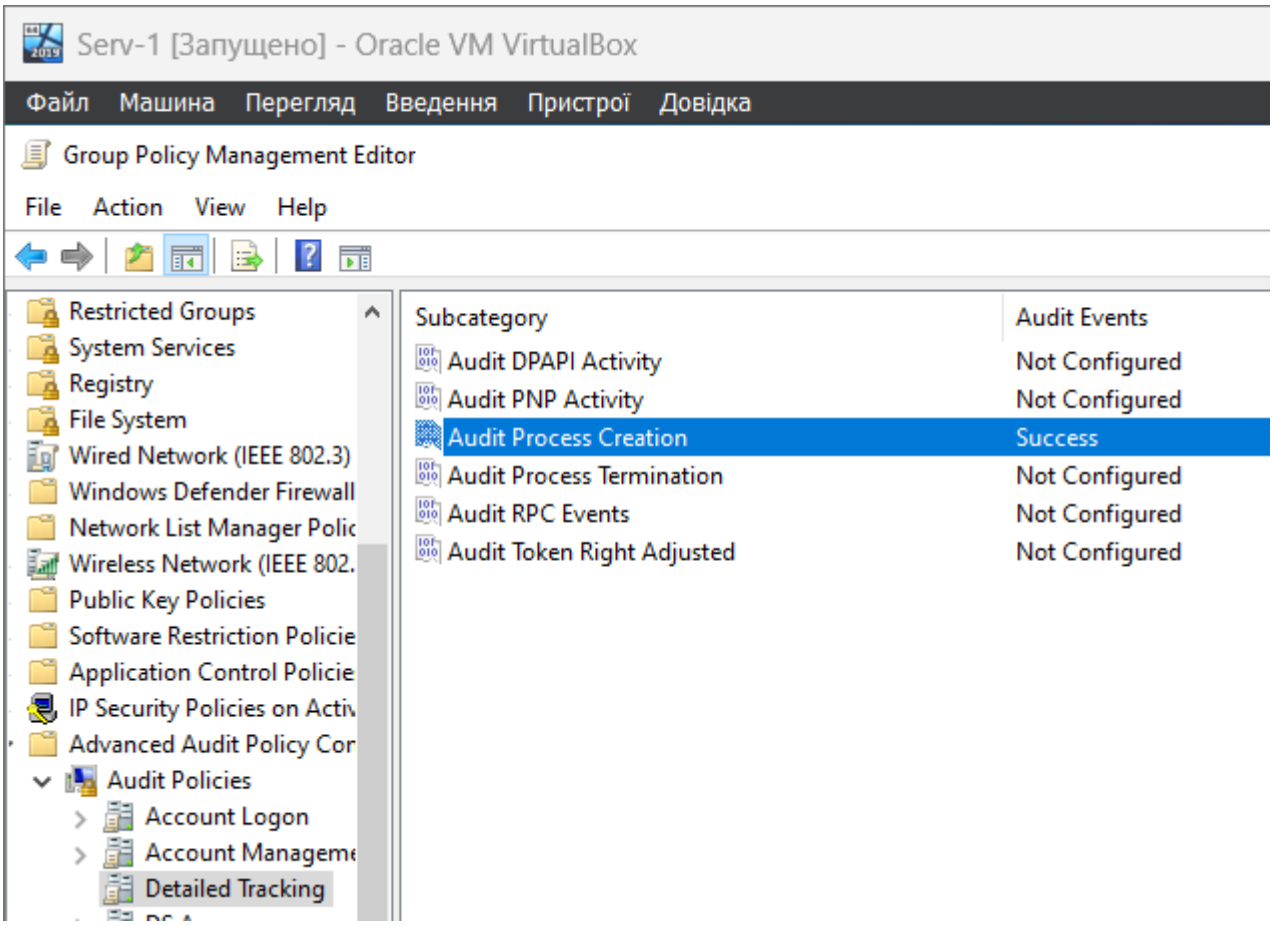


Рис. 4.11. Налаштування політики аудиту створення процесів

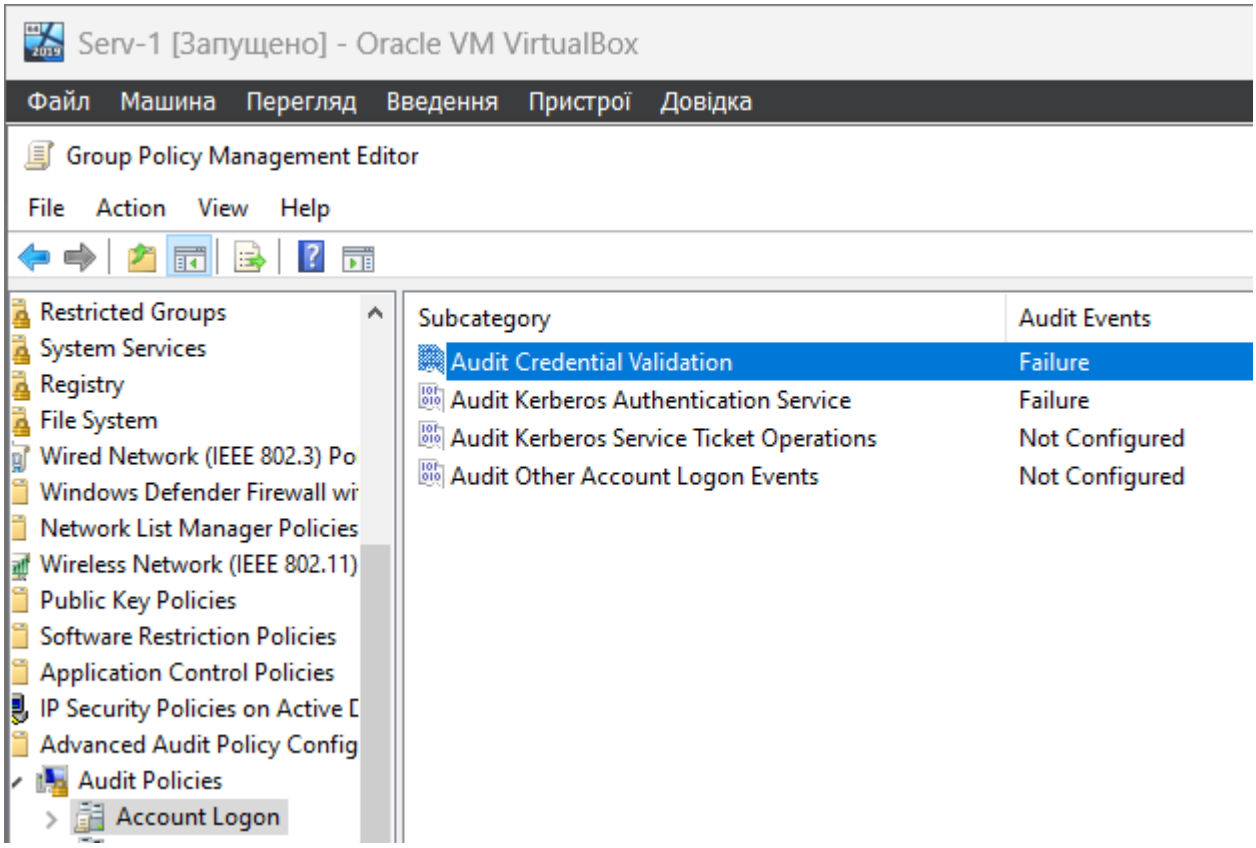


Рис. 4.12. Налаштування політики аудиту перевірки облікових даних.

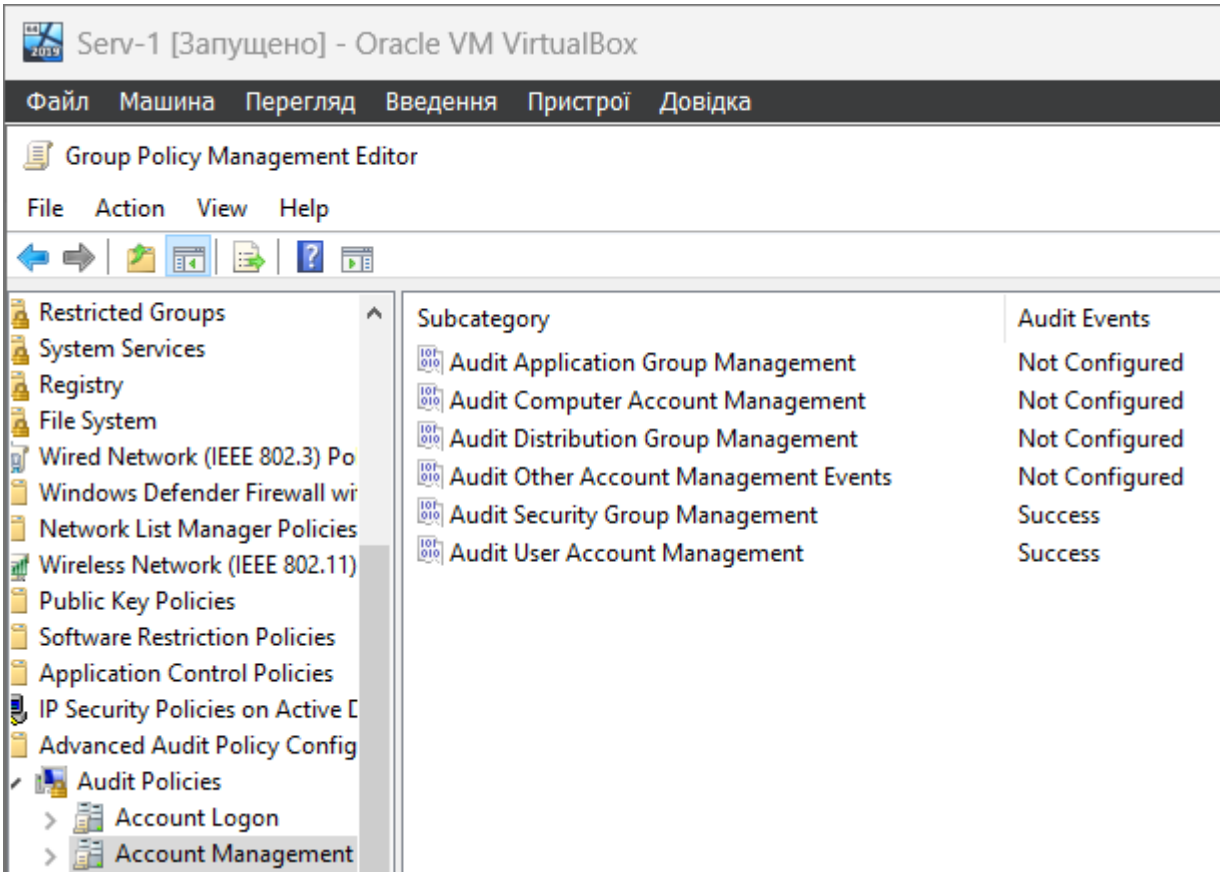


Рис. 4.13. Налаштування політики аудиту керування обліковими записами.

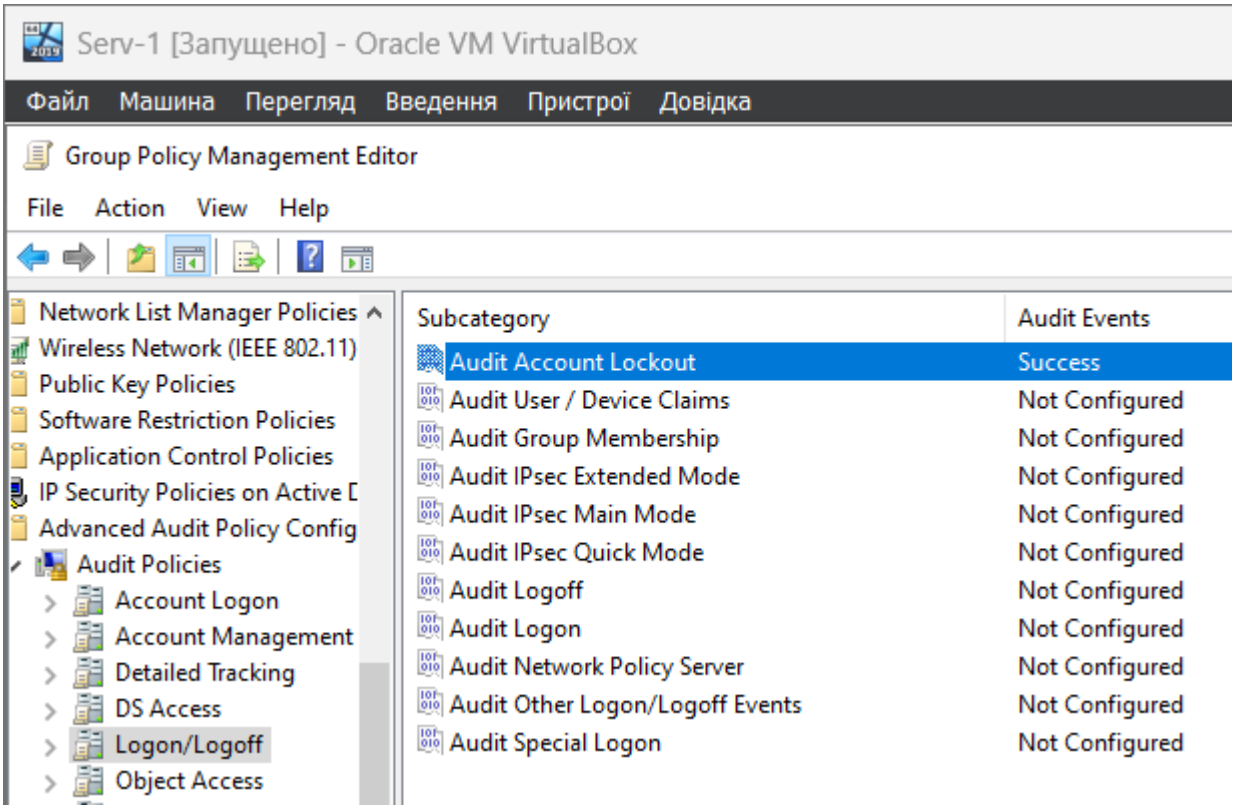


Рис. 4.14. Налаштування політики аудиту блокування облікових записів.

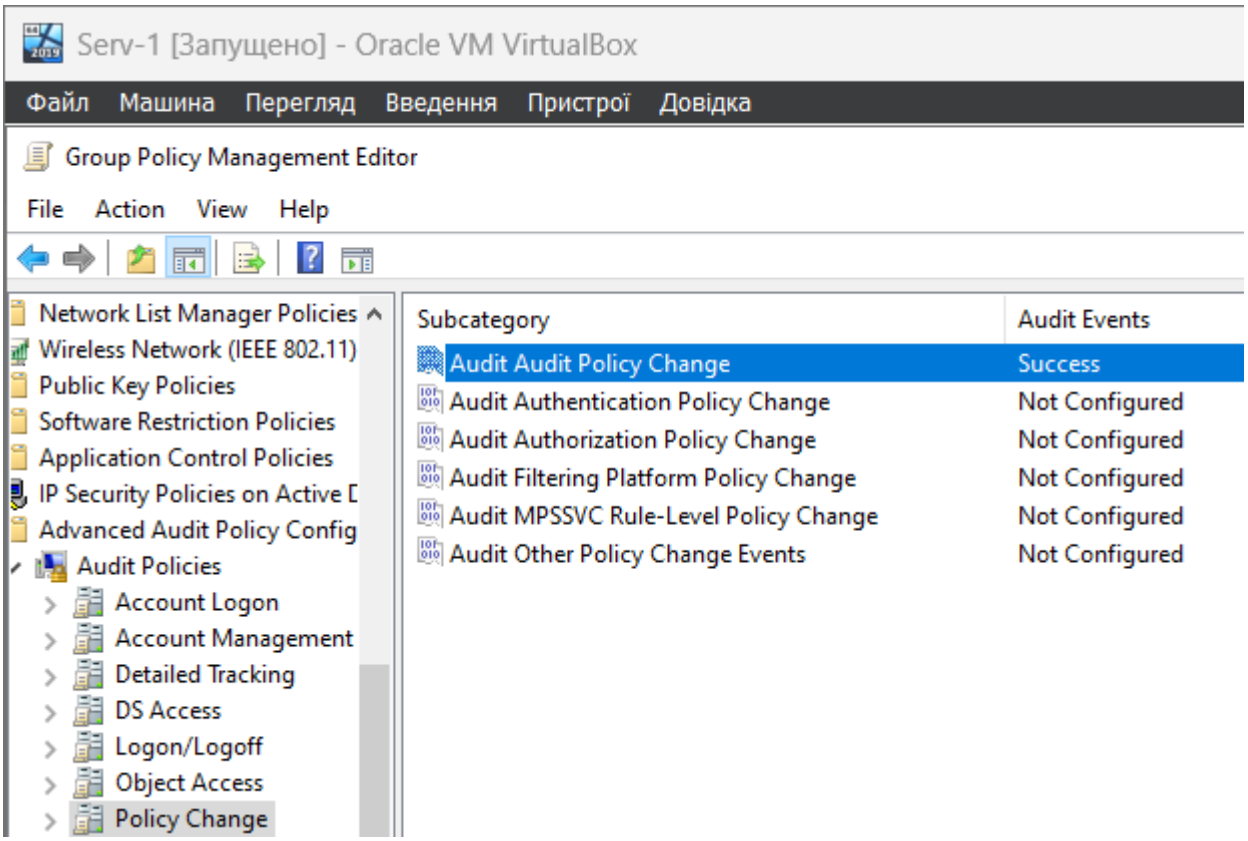


Рис. 4.15. Налаштування політики аудиту змін політики.

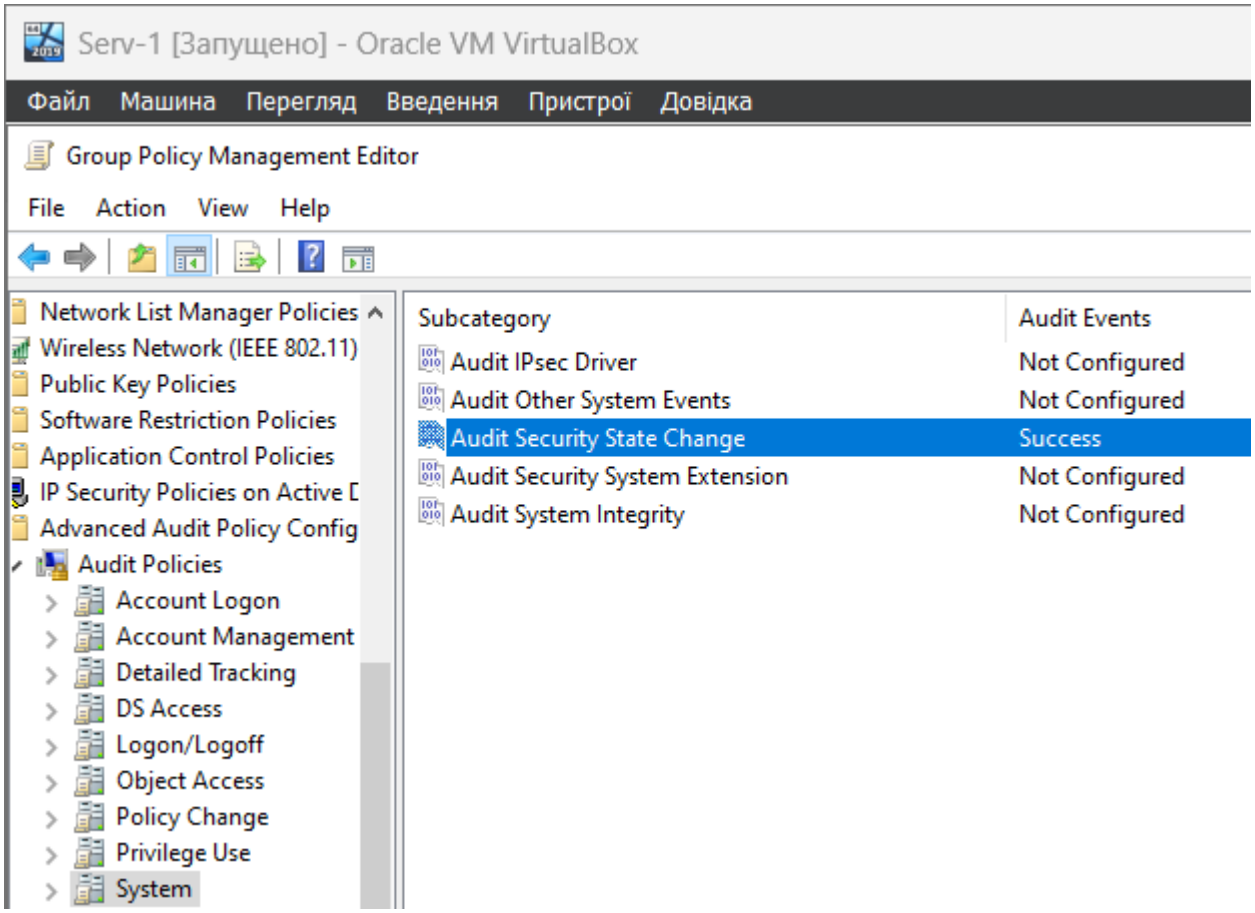


Рис. 4.16. Налаштування політики аудиту зміни стану безпеки.

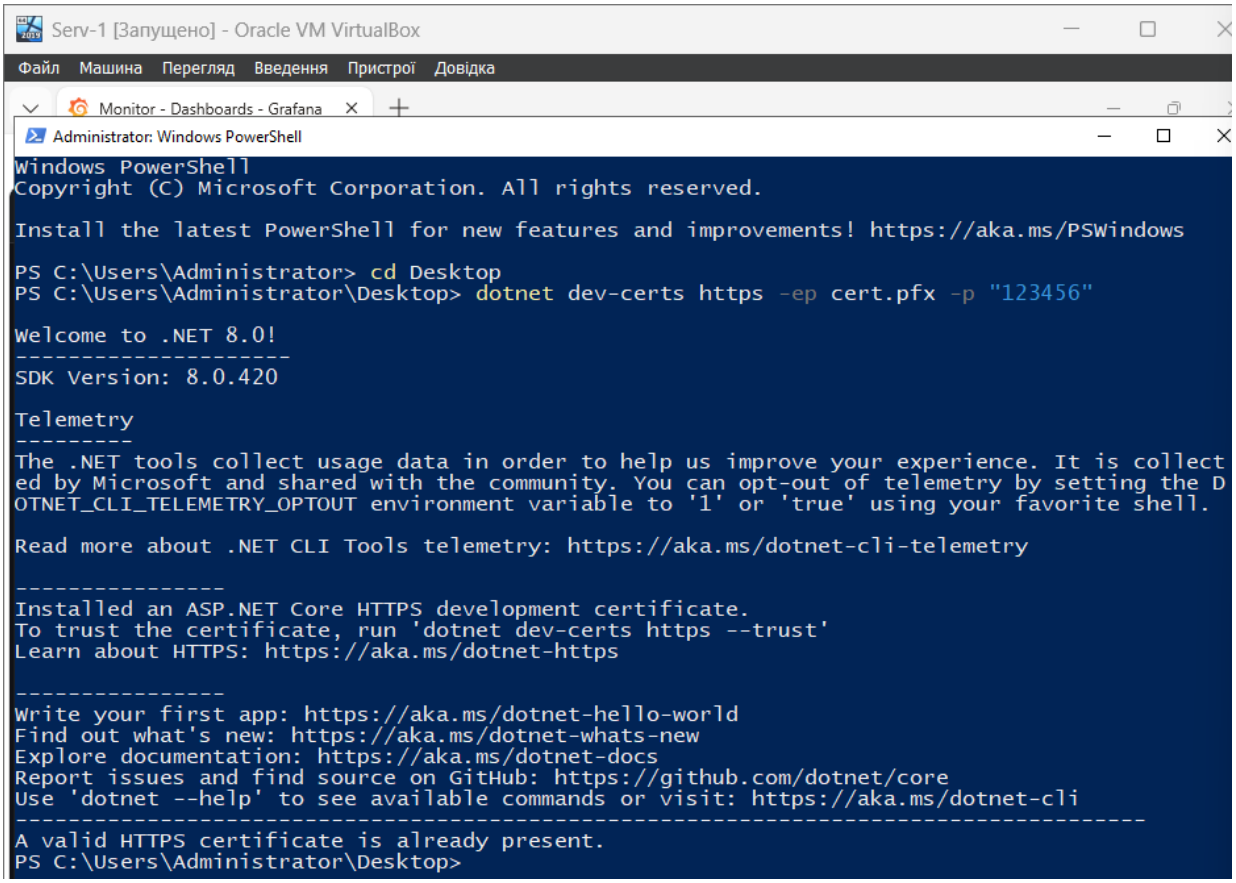


Рис. 4.17. Генерація SSL-сертифіката за допомогою утиліти dotnet.

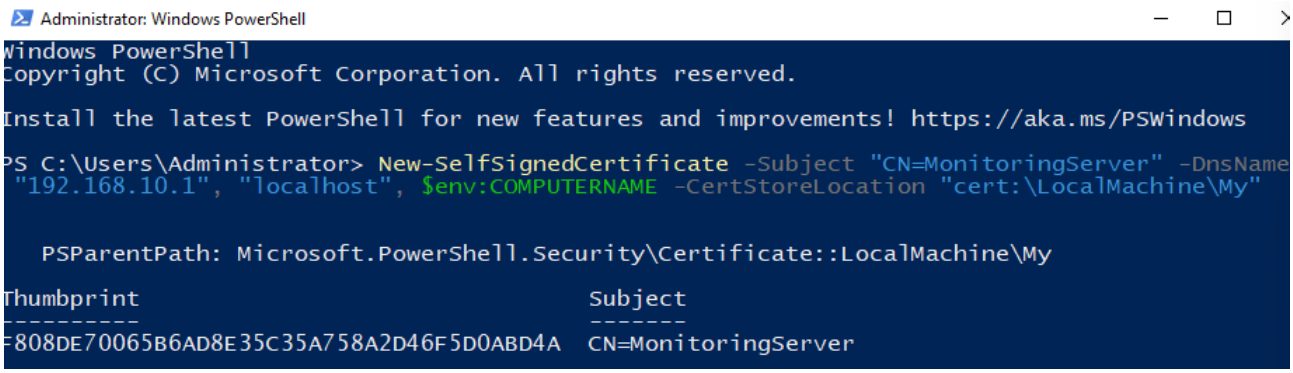


Рис. 4.18. Створення самопідписаного сертифіката через PowerShell.

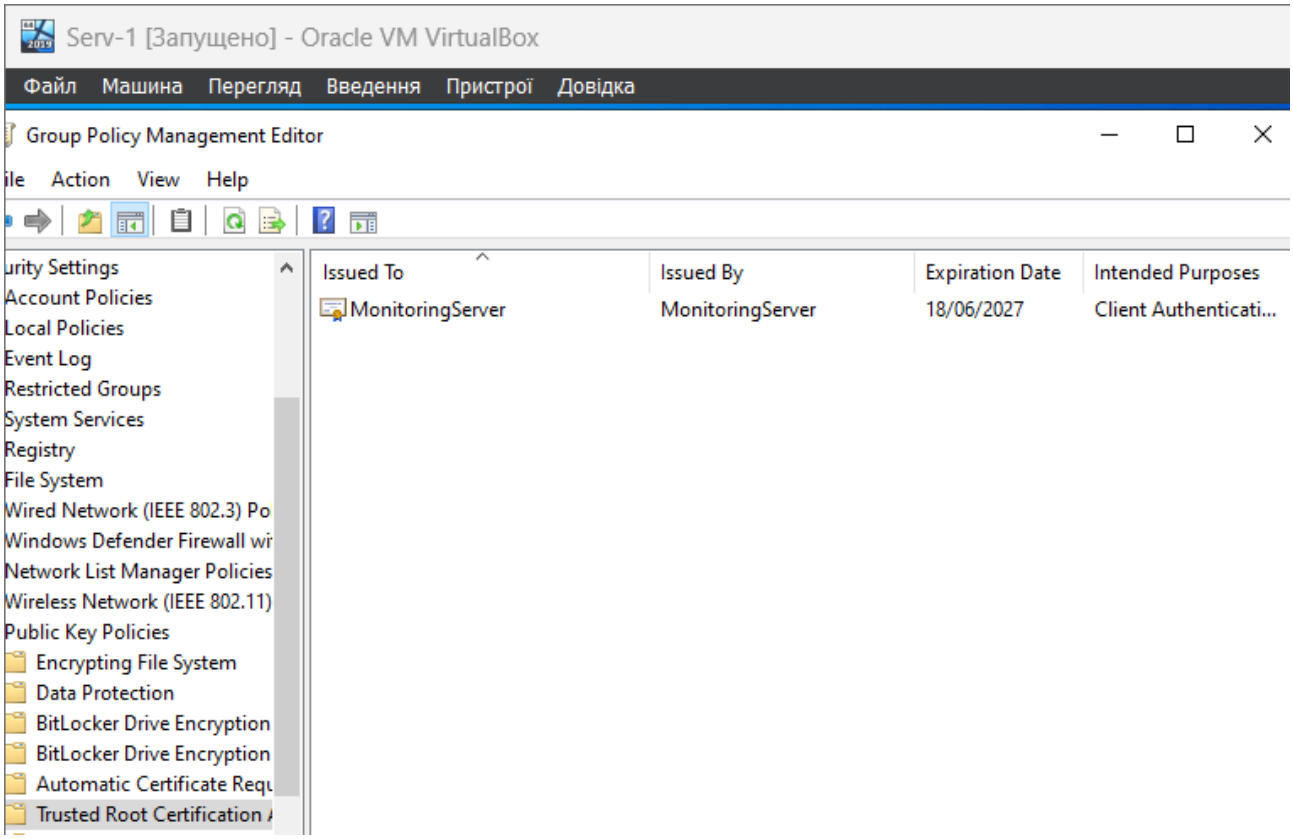


Рис. 4.19. Перевірка наявності сертифіката у списку довірених.

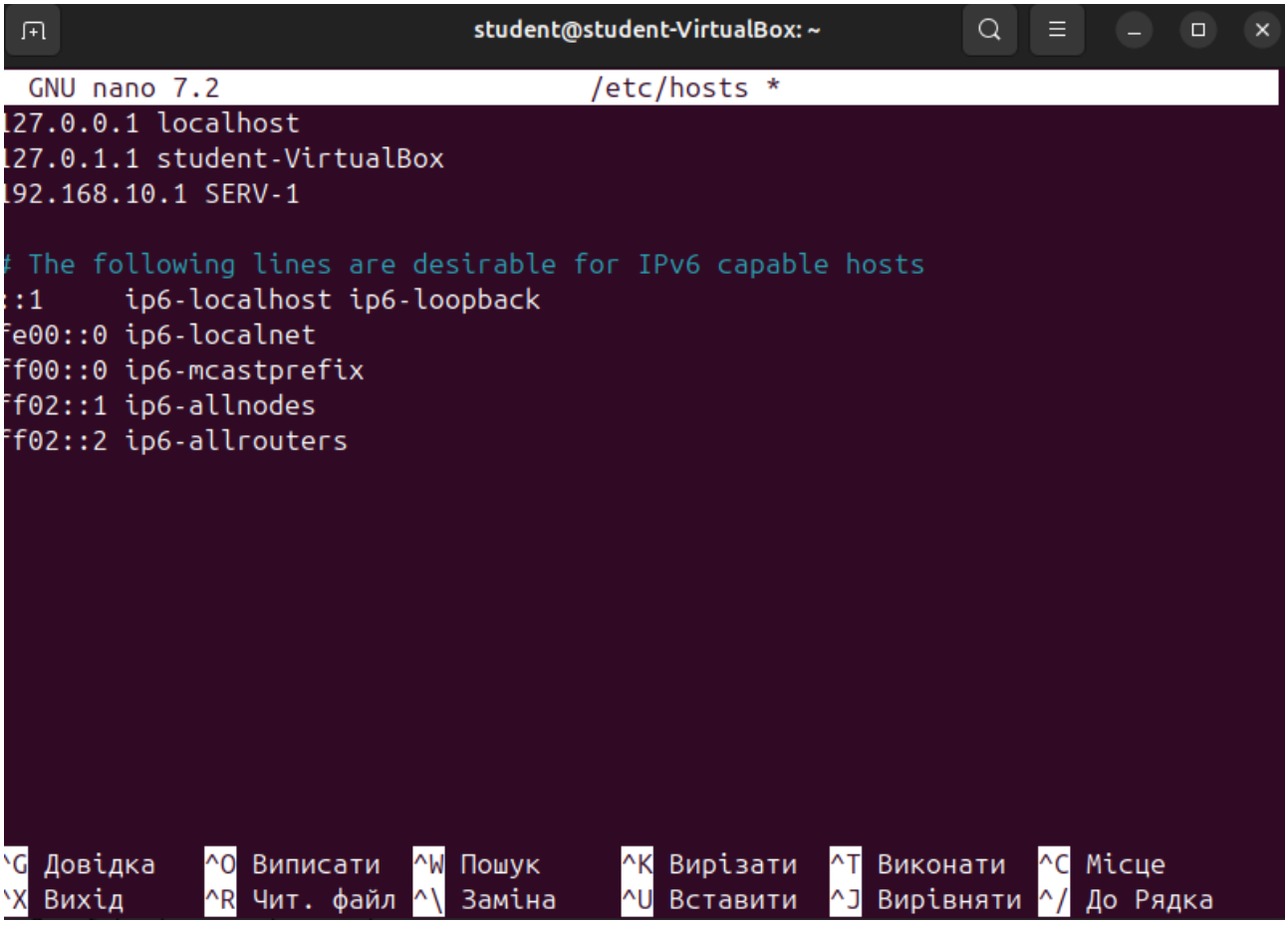


Рис. 4.20. Налаштування DNS-імені сервера у файлі hosts на клієнті.

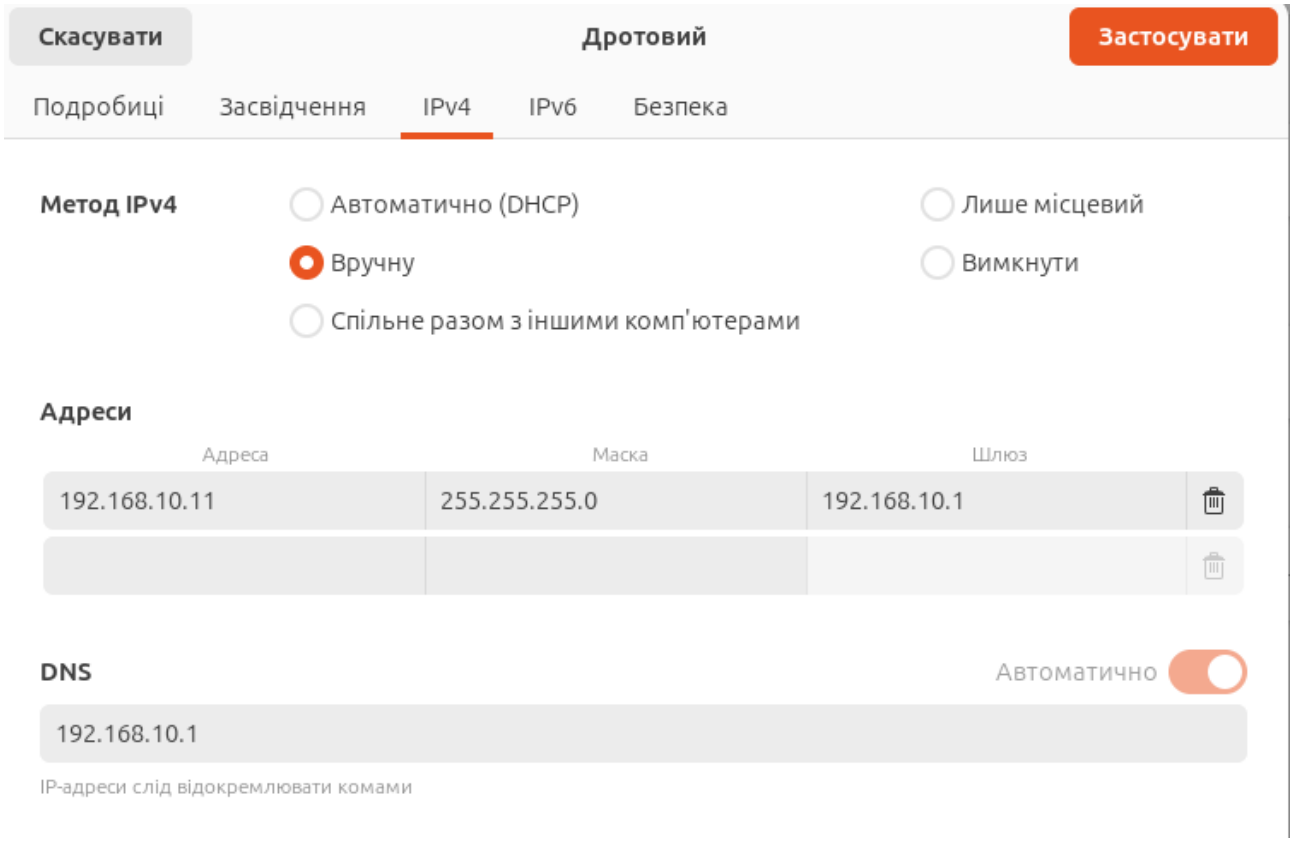


Рис. 4.21. Налаштування статичної IP-адреси Ubuntu.

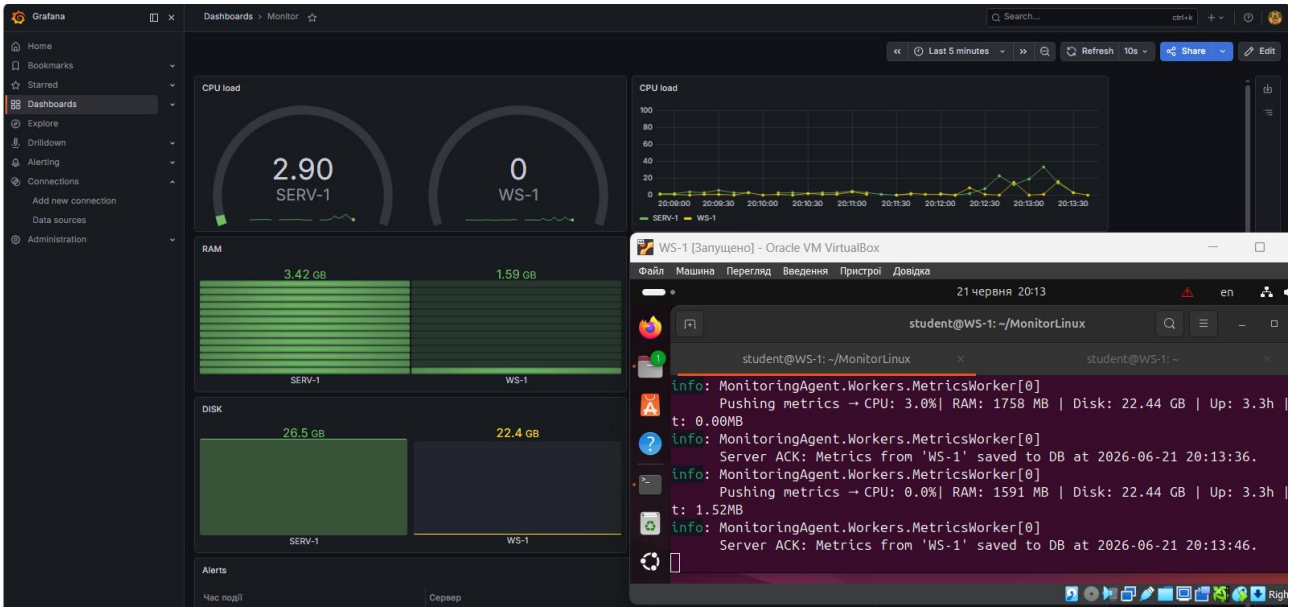


Рис. 4.22. Перевірка роботи клієнта на Ubuntu.