Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

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

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

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

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

освітній ступень *магістр*

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

на тему: «Метод захисту від атаки сканування мережевих портів на основі аналізу трафіку»

Виконавець: студент ІІ курсу, групи КБм-21

________________ **Антон ШПИЛЬОВИЙ**
(підпис) (ім’я, ПРІЗВИЩЕ)

	Ім’я, ПРІЗВИЩЕ	Підпис
Керівник	Олександр ЛАПТЄВ	
Нормоконтроль	Іван БІЛОКОНЬ	

Київ 2025

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

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

ЗАВДАННЯ
на виконання кваліфікаційної роботи

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

освітній ступень *магістр*

Здобувача(ки) КБм-21 Шпильовому Антону Миколайовичу
(група) (прізвище ім'я по-батькові)

Тема кваліфікаційної роботи Метод захисту від атаки сканування мережевих портів на основі аналізу трафіку

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

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

2. МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень Процес захисту від атаки сканування мережевих портів на основі аналізу трафіку.

Предмет досліджень Методи захисту від атаки сканування мережевих портів на основі аналізу трафіку.

Мета Розробка методу та видача рекомендацій щодо захисту від атаки сканування мережевих портів на основі аналізу трафіку.

Вихідні дані для Методи сканування портів, методи захисту від сканування портів.

проведення роботи ____

3. ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна розроблений метод захисту від атаки сканування мережевих портів на основі аналізу трафіку

Практична цінність метод захисту від атак сканування мережевих портів, що дозволяє підвищити рівень інформаційної безпеки в мережах.

4. ЕТАПИ ВИКОНАННЯ РОБОТИ

Найменування етапів робіт	Строки виконання робіт (початок-кінець)
Уточнення постановки задачі	25.10.2024 – 28.12.2024
Аналіз літературних джерел	29.12.2024 – 02.02.2025
Ознайомлення з методами та техніками сканування мережевих портів	03.02.2025 – 21.02.2025
Аналітичний огляд методів захисту від атаки сканування мережевих портів	22.02.2025 – 26.02.2025
Дослідження методів захисту від атаки сканування мережевих портів на основі аналізу трафіку	27.02.2025 – 05.03.2025
Аналіз рекомендацій стосовно розробки програмного забезпечення	06.03.2025 – 10.03.2025
Розробка алгоритму ідентифікації сканування мережевих портів	11.03.2025 – 17.03.2025
Розробка методу захисту від атаки сканування мережевих портів на основі аналізу трафіку	18.03.2025 – 27.03.2025
Розробка методичних рекомендацій щодо захисту від атаки сканування мережевих портів на основі аналізу трафіку	28.03.2025 – 10.04.2025
Апробація роботи на науково-методичному семінарі	11.04.2025 – 12.04.2025
Оформлення пояснювальної записки згідно методичних рекомендацій	13.04.2025 – 15.05.2025
Подача пакету документів на розгляд ЕК	15.05.2025 – 19.05.2025

Завдання видав ____________ Олександр ЛАПТЄВ
(підпис) (Ім'я, ПРІЗВИЩЕ)

Завдання прийняв
до виконання ____________ Антон ШПИЛЬОВИЙ
(підпис) (Ім'я, ПРІЗВИЩЕ)

Дата видачі завдання: 25.10.2024 р.

Термін подання кваліфікаційної роботи до ЕК 19.05.2025 р.

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Метод захисту від атаки сканування мережевих портів на основі аналізу трафіку»: 80 сторінок, 14 рисунків, 3 таблиці та 69 літературних джерел.

Об'єкт дослідження – процес захисту від атаки сканування мережевих портів на основі аналізу трафіку.

Мета роботи – розробка методу та видача рекомендацій щодо захисту від атаки сканування мережевих портів на основі аналізу трафіку.

Методи дослідження – спостереження, порівняння, аналіз, формалізація та експеримент.

У роботі досліджено сучасні методи та техніки сканування мережевих портів. Проведено аналіз наявних методів та засобів захисту, здійснено їх порівняння. Запропоновано метод захисту від атаки сканування мережевих портів на основі аналізу трафіку.

Наукова новизна: розроблений метод захисту від атаки сканування мережевих портів на основі аналізу трафіку.

Актуальність теми: Сканування мережевих портів є основним та найпоширенішим кроком будь-якої кібератаки, оскільки воно дозволяє кіберзлочинцям досить швидко виявити відкриті служби та підготувати подальші етапи вторгнення. Традиційні мережеві екрани та сигнатурні системи виявлення рідко здатні вчасно виявити повільні чи фрагментовані сканування, особливо за умов низького інтенсивного трафіку. Інтеграція глибокого аналізу пакетів на рівні сирих сокетів та автоматичним динамічним блокуванням через файрвол забезпечує суттєве підвищення точності детекції й оперативності реакції. Поєднання цих методів створює багаторівневий захист, що значно зменшує ризики несанкціонованого доступу та підвищує загальний рівень мережевої безпеки.

Ключові слова: кібербезпека, аналіз трафіку, сканування портів, IDS/IPS, сканери портів, файрволи, захист мережі, мережеві атаки, виявлення загроз.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

TCP – Transmission Control Protocol
UDP – User Datagram Protocol
(D)DoS – (Distributed) Denial-of-Service
ICMP – Internet Control Message Protocol
IoT – Internet-of-Things
ACL – Access Control List
DPI – Deep Packet Inspection
VPN – Virtual Private Network
IDS – Intrusion Detection System
IPS – Intrusion Prevention System
CSV – Comma-Separated Values
TTL – Time To Live
NSE – Nmap Scripting Engine
CLI – Command Line Interface
DNS – Domain Name System
HTTP(S) – HyperText Transfer Protocol (Secure)
CVE – Common Vulnerabilities and Exposures
AI – Artificial Intelligence
XSS – Cross-Site Scripting
IPFIX – IP Flow Information Export
ML – Machine Learning
SMTP – Simple Mail Transfer Protocol
FTP – File Transfer Protocol
VLAN – Virtual Local Area Network
DMZ – Demilitarized Zone
PCAP – Packet Capture
SDN – Software-Defined Networking
FPR – False Positive Rate
TPR – True Positive Rate
XML – Extensible Markup Language
SSH – Secure Shell
ПЗ – Програмне забезпечення
CMDB – Configuration Management Database
DVR – Digital Video Recorder

RDP – Remote Desktop Protocol
ОС – Операційна система
NGFW – Next-Generation Firewall
LSTM – Long Short-Term Memory
CI/CD – Continuous Integration / Continuous Deployment
API – Application Programming Interface
ARP – Address Resolution Protocol
OSPF – Open Shortest Path First
CPU – Central Processing Unit
PID – Process Identifier

ЗМІСТ

ВСТУП 9
РОЗДІЛ 1 АНАЛІЗ ІСНУЮЧИХ ТИПІВ МЕРЕЖЕВИХ КІБЕРАТАК 11
1.1 Основні категорії мережевих атак 11
1.2 Сканування портів: визначення, цілі та ризики 13
1.3 Методи сканування портів 14
1.4 Програмне забезпечення для сканування портів 16
1.5 Техніки сканування портів 19
1.6 Вплив сканування портів на безпеку мережі 20
1.7 Приклади реальних атак, що включають сканування портів 21
Висновки за розділом 1 23
РОЗДІЛ 2 ВИЗНАЧЕННЯ СУЧАСНИХ МЕТОДІВ ТА ЗАСОБІВ ПРОТИДІЇ МЕРЕЖЕВИМ КІБЕРАТАКАМ 25
2.1 Файрволи та їх обмеження в контексті сканування портів 25
2.2 Системи виявлення та запобігання вторгнень 26
2.3 Методи аналізу мережевого трафіку для виявлення атак 28
2.4 Методика Honeypot 30
2.5 Аналіз аномалій із використанням статистичних та машинно-навчальних підходів 32
2.6 Автоматизоване блокування та динамічна фільтрація на основі результатів аналізу 34
2.7 Оцінка ефективності різних методів захисту 35
2.8 Методичні рекомендації щодо захисту від сканування 37
Висновки за розділом 2 38
РОЗДІЛ 3 РЕКОМЕНДАЦІЇ ЩОДО РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 40
3.1 Вибір мови програмування та платформи 40
3.2 Захоплення та препроцесінг пакетів 44

3.3 Механізм реагування на атаку 47
3.4 Забезпечення потокобезпечності алгоритму 48
3.5 Логування роботи програмного забезпечення 50
3.6 Рекомендації з оптимізації та розширюваності 51
Висновки за розділом 3 52
РОЗДІЛ 4 РОЗРОБКА ПРИКЛАДНОГО ЗАСОБУ ДЛЯ ЗАХИСТУ ВІД СКАНУВАННЯ ПОРТІВ 54
4.1 Алгоритм ідентифікації сканування 54
4.2 Підготовка середовища 57
4.3 Написання коду та компіляція 58
4.4 Налаштування конфігураційного файлу 61
4.5 Організація як системної служби 62
4.6 Тестування розробленого засобу 66
4.7 Рекомендації щодо удосконалення засобу 67
4.8 Переваги та недоліки 69
4.9 Документація для користувача 70
Висновки за розділом 4 71
ВИСНОВКИ 72
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 74
ДОДАТКИ 81
ДОДАТОК А 81
ДОДАТОК Б 82
ДОДАТОК В 103

ВСТУП

Сканування мережевих портів є першим кроком для зловмисників, які шукають спосіб проникнути в систему. Відкриття кожного порту дає зрозуміти, які сервіси слухають мережеві запити й які версії ПЗ на них запущені. Сьогодні атаки не обмежуються ручним увімкненням telnet чи netcat – автоматизовані сканери можуть вистрілити тисячами пакетів за секунду і перетворити мережеву активність на хаос.

Мережа Інтернет зростає дуже швидко, а разом із нею зростають і ризики. На початку 1990-х дослідники та хакери експериментували з інструментами для виявлення відкритих портів на віддалених хостах. Перші утиліти працювали так: пробивали всі порти поспіль, перевіряючи відповіді. Зі зростанням складності мереж виникла потреба у точнішому скануванні, адже масове сканування створювало надмірне навантаження на інфраструктуру.

З 1997 року, коли Гордон Ліон опублікував Nmap, почалася ера гнучких скриптів, налаштованих під різні мережеві стеки [1]. З'явилися техніки SYN-сканування, UDP-сканування, пінг-сканування. Інструменти розвивалися: вбудували маніпуляцію TTL, IP-фрагментацію, маскування вмісту [2]. Перші атаки на основі сканування стали складовою частиною комплексних DDoS-кампаній та проникнень у мережі великих компаній.

У 1990-х роках, пробне сканування з портів з 1 до 1024 вважалося нормою. Після появи файрволів та IDS/IPS класичні підходи довелося модифікувати. У 2000-х роках з'явилися stealth-методи: сканування з багатьма інтервалами, таймаутами, псевдосесіями TCP-handshake. При цьому фіксувалися лише аномалії у часі відповіді.

Далі почали експериментувати з поведінковим аналізом трафіку. Системи, такі як Bro, навчилися реагувати не на окремий пакет, а на нетипову послідовність запитів. У 2010-х роках паралельно з масовими сканерами з'явилися класи детекторів, що залучали машинне навчання для класифікації шуму та сигналу.

Сьогодні підхід “сканування та аналіз” розвинено до рівня платформ для SIEM, де аналітика й кореляція подій відбувається в реальному часі. Але навантаження на мережу та обчислювальні потоки залишається проблемою. Необхідно оптимізувати методи, щоб виявляти сканування без зайвого захаращення логів і без затримки легітимного трафіку.

У сучасних умовах захисту інфраструктури важливо зосередитися саме на методах протидії скануванню портів, тому актуальність роботи полягає в тому, що:

- підготовчі стадії атак часто ґрунтуються на скануванні портів;
- надмірне споживання ресурсів при класичних методах призводить до деградації мережі;
- існуючі системи виявлення реагують із затримкою або дають забагато хибних спрацьовувань;
- для малих і середніх підприємств рішення з SIEM є надто дорогими;
- оптимізовані алгоритми аналізу знижують навантаження на сервери логування.

Для досягнення зазначеної мети дипломної роботи поставлено наступні завдання:

- аналіз методів захисту від атаки сканування мережевих портів на основі аналізу трафіку;
- розробка методу захисту від атаки сканування мережевих портів на основі аналізу трафіку;
- розробка методичних рекомендацій щодо захисту від атаки сканування мережевих портів на основі аналізу трафіку.

Результати дослідження можуть бути використані для покращення безпеки корпоративних мереж, центрів обробки даних, державних установ та інших критичних інфраструктур. Використання аналізу трафіку дозволить забезпечити раннє виявлення атак. Це зменшить ризик витоку даних та підвищить стійкість мережевих систем до несанкціонованих дій.

РОЗДІЛ 1
АНАЛІЗ ІСНУЮЧИХ ТИПІВ МЕРЕЖЕВИХ КІБЕРАТАК

1.1 Основні категорії мережевих атак

Сканування портів відкриває завісу над інфраструктурою мережі: воно дозволяє виявити, які служби працюють на вузлах, які версії застосунків слухають запити і де ховаються потенційні дірки у захисті. На цьому етапі зловмисник формує повну картину мережевого ландшафту, комбінуючи прості TCP-з'єднання, напіввідкриті SYN-пакети, перевірки UDP-портів та хитрі варіанти типу FIN, XMAS чи NULL. Такий підхід дає змогу вивчати мережеву поведінку й підбирати стратегію подальших атак.

Коли сканування перетворюється на активний збір інформації, воно вже не обмежується тихим підслуховуванням. Використовуючи повне встановлення TCP-з'єднання, аналіз UDP-відповідей або методи напіввідкритого сканування, атакуючий отримує деталі конфігурації сервісів і їхніх контрольних механізмів. Ідентифікація версій програмного забезпечення допомагає знайти сумісні експлойти, а приховані варіанти пакетів дозволяють обходити базові файрволи.

Інколи інтенсивність сканування сама по собі стає зброєю. Швидкі потоки запитів, особливо розподілені серед ботнетів, здатні перевантажити маршрутизатори й сервіси, викликаючи відмову в обслуговуванні. Найвідоміший приклад – SYN-flood атака, коли масив напіввідкритих з'єднань вичерпує ресурси системи і блокує легітимні запити.

Після того як список відкритих портів складено, він слугує відправною точкою для більш складних вторгнень. Незахищені канали типу SSH чи Telnet стають майданчиком для brute-force перебору паролів, а вразливі служби легко взламуються за допомогою готових експлойтів. Таким чином, початкове сканування формує маршрут для ексфільтрації даних чи ескалації прав.

Щоб обійти складнішу систему фільтрації, зловмисники нарощують арсенал: фрагментують пакети, прокидають скани через VPN або анонімні проксі, а іноді використовують idle-сканування, делегуючи перевірку сторонньому хосту. Ці техніки дозволяють залишатися непоміченими для IDS/IPS і обходити ACL без прямого контакту з ціллю.

На основі способу сканування, його цілей атаки, можна поділити на кілька категорій:

1. Розвідка та збір даних. Цей етап передбачає перевірку відкритих портів і аналіз відповіді хостів. Інструменти сканування дебажать мережу, фіксують активні сервіси та версії ПЗ. Результат – карта точок входу та потенційних вразливостей.

2. Атаки на доступність через інтенсивне сканування. Коли частота запитів зростає до рівня DoS, маршрутизатори й сервери вичерпують ресурси. Botnet-сканування або SYN-flood із масивом напіввідкритих з’єднань блокують легітимний трафік і викликають відмову в обслуговуванні.

3. Підготовка до подальших вторгнень. Після складання переліку відкритих портів зловмисник планує експлойти, brute-force-атаки або ексфільтрацію даних. Сканування задає маршрут руху по системі й точки ескалації прав.

4. Обхід фільтрів та файрволів. Зловмисники адаптують сканування під налаштування захисту: фрагментують пакети, прокидають запити через VPN або проксі, застосовують idle-scan. Мета – дізнатися правила ACL і непомітно пробитися за їх межі.

Усе це підкреслює значущість захисту на рівні моніторингу мережевого трафіку: гнучкі правила файрвола, налаштовані сигнатури в IDS/IPS та обмеження доступу до критичних портів мінімізують ризики на початковому етапі. Без адекватної протидії скануванню інші механізми безпеки втрачають ефективність, бо будь-яка складніша атака починається саме з перевірки портів. Своєчасна аналітика мережевих потоків дозволяє виявити нетипові запити ще до ескалації загрози. Кореляція подій у SIEM і оперативні оповіщення значно скорочують час реакції та ускладнюють життя зловмисникам.

1.2 Сканування портів: визначення, цілі та ризики

Сканування портів – це процес послідовного або розподіленого відправлення мережевих запитів до різних портів на цільовому хості з метою визначити їхній статус: відкритий, закритий або фільтрований. Інструменти, такі як Nmap, Masscan чи кастомні скрипти, сканують мережу, відправляють TCP або UDP пакети, парсять відповіді і формують детальну карту доступних сервісів та їхніх версій.

Сканування портів може проводитися як у легальних, так і в злочинних цілях.

Легальні цілі (етичне сканування):

- аудит безпеки та пен-тестинг;
- інвентаризація активних хостів і сервісів;
- перевірка коректності налаштувань файрвола;
- аналіз продуктивності мережі;
- підтримка внутрішніх політик.

Зловмисні цілі (неетичне сканування):

- збір даних про відкриті сервіси та їхні версії;
- пошук вразливостей для подальших експлойтів;
- планування DoS/DDoS-кампаній;
- автоматизовані brute-force атаки на автентифікацію;
- обхід політик і прихована розвідка.

Сканування портів може становити загрозу для безпеки мережі, особливо якщо воно виконується зловмисниками або автоматизованими ботами.

Основні ризики:

- Визначення мережевої конфігурації – отримання переліку відкритих портів і активних сервісів на хості.
- Підготовка до експлуатації вразливостей – вибір і завантаження відповідних експлойтів під знайдені сервіси.
- Перевантаження ресурсів – інтенсивне сканування, що створює зайве навантаження на мережеві пристрої та сервери.

- Ухилення від виявлення – застосування прихованих технік сканування для обходу IDS/IPS і брандмауерів.
- Соціальна інженерія – використання зібраних даних для маніпуляцій з адміністраторами чи кінцевими користувачами.

Для безпечного та контрольованого сканування рекомендується використовувати внутрішні тестові середовища або узгоджувати дії з адмінами мережі. При цьому важливо постійно моніторити частоту запитів, логувати відповіді й налаштовувати сигнатури IDS/IPS для коректного розпізнавання легітимного тестування.

Таким чином, сканування портів – інструмент із двома гранями: воно незамінне для оцінки безпеки, але вимагає грамотного підходу до швидкості, методів і правових аспектів. Надалі в роботі я перегляну механізми детекції сканування на основі аналізу трафіку, щоб мінімізувати ризики і виявляти розвідку на ранніх етапах.

1.3 Методи сканування портів

Сканування портів відкриває картину мережевої поверхні: воно показує, які сервіси слухають запити, які порти доступні, а також які засоби захисту встановлені [3]. Залежно від стратегії сканування атакуючий може зібрати базову або детальну інформацію про топологію та активні служби цільової системи.

Ідея проста – перебрати всі потенційні канали зв’язку і позначити ті, що відповідають на запит. У ролі каналу виступає абстрактний порт – логічний інтерфейс у TCP/UDP-стеку, що спрощує встановлення з’єднань і обмін даними. Через порт додаток спілкується із мережею, але водночас відкриті порти дають змогу аналізувати інфраструктуру без прямого доступу.

Для сканування надсилають пакети на номери від 1 до 65535. За тим, які порти відповідають (або мовчать), визначають, які служби запущені й готові прийняти з’єднання. Цей простий підхід дозволяє виявити вразливі точки, спланувати експлойти та підготувати подальші етапи атаки [4].

Методи сканування можна розділити на активні та пасивні:

- Активне сканування – передбачає безпосереднє надсилання пакетів до цільової системи для визначення її стану.
- Пасивне сканування – відбувається шляхом аналізу вже наявного трафіку без активної взаємодії з мережею.

Найпоширенішими є наступні методи сканування портів:

1. TCP SYN – надсилається пакет TCP SYN на цільовий порт. Якщо отримано SYN-ACK, порт відкритий, якщо RST – закритий. Даний метод швидкий та ефективний. Його важко виявити, оскільки не завершується тристороннє з’єднання. До недоліків слід віднести, що дане сканування може бути заблоковане файрволами або системами IDS/IPS.
2. TCP Connect – встановлюється повноцінне TCP-з'єднання з цільовим портом. Використовується у випадках, коли SYN-сканування заборонене та не потребує спеціальних привілеїв. До недоліків слід віднести, що це сканування генерує багато логів на цільовій системі, що може призвести до швидкого виявлення. Також вимагає більше ресурсів.
3. UDP Scan – надсилаються UDP-пакети на цільові порти. Відсутність відповіді вказує на відкритий порт, ICMP Destination Unreachable – на закритий. Дозволяє виявити працюючі UDP-сервіси (DNS, SNMP, DHCP тощо). Повільніше за TCP-сканування, оскільки UDP не генерує автоматичних відповідей. Багато файрволів блокують аномальний UDP-трафік.
4. FIN Scan – надсилається TCP FIN-пакет без встановлення з’єднання. Відсутність відповіді означає, що порт відкритий. Допомагає обійти слабкі файрволи та IDS. Нажаль, не працює на Windows.
5. XMAS Scan – надсилаються пакети з усіма можливими TCP-прапорами (FIN, PSH, URG), що викликає нестандартну поведінку сервера. Може обійти деякі системи захисту. Не працює на Windows та легко виявляється IDS.

6. NULL Scan – надсилається TCP-пакет без прапорців. Відсутність відповіді означає відкритий порт. Може допомогти обійти слабкі мережеві фільтри. Не працює на Windows.

7. Idle Scan – використовується ботнет мережа для сканування цілі, щоб приховати справжнє джерело атаки. Дозволяє анонімно сканувати систему без прямого контакту. Складний у реалізації, потребує ретельного налаштування.

Підбір методу сканування – це баланс між швидкістю, точністю та ризиком бути виявленим. Вище були розглянуті ключові підходи до перевірки доступності мережевих портів - від базового TCP CONNECT та SYN-сканування до спеціалізованих методів FIN, NULL та XMAS, а також UDP-сканів. Кожен із цих методів по-своєму відображає компроміси між швидкістю, надійністю виявлення відповіді та помітністю для систем захисту. Мікс кількох методів дає змогу отримати максимально повну картину мережевої поверхні атаки, та заодно – підвищити стійкість системи виявлення до різноманітних тактик сканування.

1.4 Програмне забезпечення для сканування портів

Програмне забезпечення для сканування портів – незамінний інструмент у арсеналі як пентестера, так і сисадміна. Від простих утиліт, що перевіряють з’єднання по одному порту, до розподілених фреймворків, що можуть охоплювати тисячі адрес за секунди – вибір залежить від завдання та доступних ресурсів. Найпоширеніший і найбільш різносторонній серед них – Nmap. Він поєднує скриптовий движок NSE, fingerprinting ОС, визначення версій сервісів і різноманітні timing templates. З його допомогою можна автоматизувати завдання аудиту, накочувати патчі для виявлення слабких місць, а також інтегрувати результати сканування в конвеєри за допомогою XML або JSON експорту.

Якщо Nmap – універсал, то Masscan – це утиліта заточена під швидкість. Реалізований на асинхронних сокетах, він дозволяє проганяти весь інтернет за лічені хвилини. Утиліта не глибинна: вона не робить OS fingerprint та не запускає скрипти, але це й не її задача. Masscan – про масштаб. Використовуючи параметри rate і

source-port spoofing, можна протестувати великі мережеві діапазони й відразу ж передати сирі результати в парсер або інший сканер для детальнішого аналізу.

Для нетривіальних сценаріїв із відстеженням змін у часі підійде ZMap – інструмент із відкритим кодом, орієнтований на високошвидкісні обстеження одному або кількох портів. Його основна фішка – pipeline сканування із записом міток часу, що дозволяє будувати тренди доступності сервісів. Зібрані дані безболісно інтегруються у BI-системи або SIEM [5].

Коли хочеться більш гнучких рішень, на допомогу приходить hping3 і netcat. Перший – чудово підходить для формування будь-яких TCP/UDP/ICMP-пакетів із кастомними заголовками. Він служить для відпрацювання конкретних кейсів, коли треба змоделювати нетривіальну атаку чи обійти шаблонні сигнатури IDS. Netcat, у свою чергу, швидко аналізує відповіді, відкриває шелли та проксі-тунелі, допомагаючи перевірити, чи коректно працює порт після умовної фільтрації [6]. Також варто згадати про Unicornscan. Даний інструмент поєднує багатопотокову обробку з розподіленим підходом: він здатний координувати кілька агентів, зберігати результати в централізованій базі даних та надавати кореляційний аналіз банерів і версій сервісів для багатьох хостів одночасно.

Визначаючи вибір ПЗ для сканування, завжди варто врахувати кілька факторів: чи потрібна максимальна швидкість, наскільки важлива прихованысть, чи потрібні скрипти для конкретного протоколу, чи необхідна інтеграція з SIEM чи CMDB. Часом достатньо одного Nmap із набором кастомних NSE-скриптів, а іноді краще поєднати Masscan для ширшого охоплення й Nmap для глибокого занурення.

У підсумку, сканери варто розглядати як модульну систему: базова логіка – швидкий інвентор портів, а інструменти та плагіни – це вже надбудова для fingerprinting, bypass-методів і вбудованої автоматизації [7]. Також варто зважати на оверхед інструментів і простоту розгортання в CI/CD-пайплайні. Активне ком’юніті та регулярні апдейти плагінів спрощують підтримку й накочування нових скриптів. Такий підхід дозволяє гнучко реагувати на зміни інфраструктури, дебажити політики

безпеки й оперативно виявляти вразливості в мережевому захисті. Порівняння описаних утиліт наведено в таблиці 1.1.

Таблиця 1.1

Порівняння утиліт для сканування портів

Інструмент	Методи сканування	Протоколи	Швидкість	Особливості	Типове застосування
Nmap	TCP (усі види), UDP, Idle, NSE	TCP, UDP, ICMP	Середня	Багатофункціональний рушій скриптів, грабінг банерів, розпізнавання ОС	Аудит безпеки, дослідження хостів
Masscan	TCP SYN	TCP	Дуже висока	Асинхронний рушій, конфігурується через CLI, вивід JSON/CSV	Швидке покриття великих діапазонів адрес
ZMap	TCP SYN	TCP	Дуже висока	Інтеграція з базами даних, збирання результатів у потік	Масштабна інвентаризація Інтернету
Hping3	TCP (усі види), UDP, ICMP, RAW	TCP, UDP, ICMP, RAW	Вище середньої	Ручне формування пакетів, тестування DoS, скрипти	Тестування захищеності, обходи файрволів
Unicornscan	TCP SYN-ACK, UDP, асинхронні потоки	TCP, UDP	Висока	Розподілений скан із централізованою БД, кореляція банерів	Кореляційний аудит багатьох хостів
Netcat	TCP CONNECT, UDP	TCP, UDP	Нижче середньої	Проксування, тунелювання, прості скани	Прості перевірки портів

1.5 Техніки сканування портів

У рамках будь-якого інструментального середовища важливо застосовувати

техніки сканування, які підвищують прихованість та адаптивність процесу. Повільні скани з великими інтервалами між запитами значно знижують ймовірність виявлення системами IDS/IPS, які зазвичай аналізують частотність і шаблони запитів. Метод випадкового порядку портів розмиває характерні послідовності та ускладнює збирання точних показників цільового середовища. Фрагментація пакетів дозволяє обійти прості фільтри, що не підтримують повторну збірку IP-фрагментів, а техніки зі вставкою приманок замітають сліди сканування між кількома псевдо-IP, роблячи його джерело менш однозначним. Для глибшого маскування використовують зміни поля TTL, завдяки яким пакети можуть здаватися локальними або належати іншому сегменту мережі [8].

Крім класичних мережевих підходів, останнім часом популярні варіанти сканування через анонімні мережі Tor чи проксі, що дозволяють приховати оригінальну IP-адресу, а також тунелювання скан-запитів через широко поширені протоколи HTTP або DNS, що забезпечує приховування оригінального трафіку в обрамленні легітимних запитів. Комбінація таких технологій і ретельний підбір інструментів дають змогу фахівцям зі захисту мереж проводити як швидку розвідку великих просторів, так і детальне багаторівневе дослідження окремих хостів, зберігаючи при цьому максимально можливу невидимість власної активності.

Повільні скани розтягують інтервали між запитами на години чи добу, щоб IDS/IPS не фіксували сплески нових з’єднань. Завдяки цьому можна сканувати мережу практично непомітно.

Випадковий порядок портів розмиває характерні послідовності – кожен номер запитується в довільній послідовності, що ускладнює збір статистики та кореляцію запитів у логах.

Фрагментація пакетів розбиває TCP/IP-заголовки на шматки, які багато файрволів та IDS/IPS не збирають докупи, тож підозрілий трафік пролітає повз стандартні правила ACL [9].

Сканування із закиданням приманок розділяє процес сканування між кількома приманковими IP і справжньою адресою – логи моніторів збільшуються, і джерело атаки губиться серед псевдо-хостів.

Маніпуляція полем TTL дає змогу пакету виглядати як локальний або з іншого сегмента мережі – поміж усієї маси запитів реальне походження сховається під іншим маршрутом.

Сканування через ланцюжок проксі чи Tor маскує початковий IP: ціль бачить лише фінальний хоп, а первинний вузол лишається поза зоною видимості базових ACL.

Тунелювання через HTTP чи DNS упаковує скан-запити в стандартні запити до веб-серверів чи резолверів—легітимний трафік закриває справжню активність і збиває з пантелику будь-які сигнатури.

Кожна з цих технік спрямована на підвищення стелс-ефекту сканування й ускладнення його автоматичного виявлення, тому ефективний захист передбачає аналіз не тільки інтенсивності трафіку, а й його часових та просторових патернів.

1.6 Вплив сканування портів на безпеку мережі

Сканування портів часто відкриває ворота до небажаного навантаження й непередбачених ризиків. Навіть легітимний аудит може дебажити інфраструктуру на піку її можливостей, а агресивні скани часом виступають у ролі передатаку перед справжньою атакою [10].

Основні наслідки портсканування для мережі:

- навантаження на маршрутизатори й комутатори, де масив SYN- чи UDP-запитів заповнює черги обробки, призводить до скидання пакетів і затримок легітимного трафіку;
- вражаючий обсяг логів у файрволах та IDS/IPS, коли тисячі записів створюють інформаційний шум і перевантажують SIEM;
- порталізація вразливостей через банер-грабінг – отримані версії ПЗ перевіряються за базами CVE, що пришвидшує підбір експлойтів і ескалацію прав;
- витік внутрішньої топології: навіть low-rate скан може надати карту підмереж, TTL-аналізом і пасивним моніторингом вираховують локальні сервери поза DMZ;

- організаційні наслідки – втрата довіри користувачів, додаткові процедури аудиту, навчання персоналу, що створює суттєве навантаження на IT-відділ;
- ризик раннього запуску вторинних атак: після сканування зловмисник може одразу накотити brute-force чи SQL-ін'єкцію по знайдених відкритих портах.

Без належного контролю сканування портів здатне деградувати продуктивність, забити логи зайвими даними і відкривати нові вектори для проникнення. Щоб цьому запобігти, варто моніторити розподіл запитів, налаштувати rate-ліміти на файрволах, підганяти правила у IDS/IPS і накотити патч із розумними порогами спрацювань. Такий підхід мінімізує ризики й дозволяє виявляти розвідку на самому старті.

1.7 Приклади реальних атак, що включають сканування портів

Сканування портів стало невід'ємною складовою підготовчих рухів у сучасних кібератаках: перш ніж накотити основний удар, зловмисник запускає серію запитів, щоб зібрати карту мережі та зрозуміти, де засіли вразливі точки.

SQL Slammer 2003 року розгортався зі швидкістю, яка перевищувала всі очікування. Черв'як миттєво надсилав UDP-пакети на порт 1434, перебираючи цілі з випадковими IP-адресами. Як тільки Slammer натрапляв на сервер із незашитим багом у Microsoft SQL Server 2000, одразу ж розгортав свій код у пам'яті, без дискових операцій. За лічені хвилини він "завалив" десятки тисяч SQL-серверів, зумівши створити глобальний затор у маршрутизації Інтернету і викликати колапс у поштових та банківських системах. Реакція захисників полягала у негайному патчингу MS03-039 і введенні суворих ACL, але шкода вже була зроблена: експериментальний скрипт перетворився на один із найшвидших мережевих черв'яків свого часу.

Через п'ятнадцять років з'явився Mirai, який виніс на рівень загальносвітової інфраструктури поняття "IoT-ботнет". Mirai вивчав мережу, пробиваючи порти 22, 23, 2323, 7547 і 5555 у пошуках камер, роутерів, DVR та інших розумних гаджетів із

типовими логінами. Як тільки скрипт знаходив вразливий пристрій, він автоматично входив під стандартними паролями і залітав у зомбі-мережу. Пік атаки на Dyn у жовтні 2016-го показав, що навіть Twitter чи Netflix можуть піти в офлайн через DDoS із десятків тисяч точок, накочених упродовж кількох хвилин масовим скануванням і услід за ним потужним трафіком.

WannaCry 2017 року додав до репертуару не лише шифрування файлів, а й алгоритм саморозмноження через EternalBlue. Після ін'єкції в жертву шпигунський код починав шукати порт 445 на сусідніх машинах у локальній мережі та в інтернеті. Розвідка тривала миттєво: скан портів та перевірка на SMBv1 призводили до автоматичного захоплення нових вузлів. Жертви отримували таблицю файлів, зашифрованих AES і RSA, а екран наповнювався закликом заплатити зловмисникам у біткоїнах. Лише активація "kill switch" зупинила ланцюг заражень, але сотні тисяч організацій відчули наслідки втрачених даних.

Поява Shodan 2013 року перевернула уявлення про пасивний пошук. Замість запускати власні сканери, можна просто відкрити індекс і побачити всі відкриті порти 21, 22, 80, 3389, 27017 тощо. Атакуючі побудували на цьому потужні скрипти: пошук MongoDB та Elasticsearch із default-конфігурацією відкритих баз, RDP з налаштуванням без паролю. За лічені секунди вони знаходили тисячі вразливих серверів і дискредитовані системи відеоспостереження, а далі автоматично ламали їх.

BlueKeep 2019 року показав ще один сценарій із RDP-портом 3389. Хакери масово сканували інтернет у пошуках Windows-систем із відкритим RDP і використовували вразливість CVE-2019-0708. Після входу вони могли запустити криптомайнінг у фоні або розгорнути масштабний ботнет із десктопів. На щастя, Microsoft випустив патч для Windows XP, але всі неодноразово бачили, як пізнє виправлення коштує мільйонних втрат.

Кожен із цих випадків підкреслює: без виявлення та блокування сканування на самому старті немає значення, наскільки міцні файрволи чи наскільки часто відбувається оновлення ОС. Захист має ідентифікувати початкові ознаки розвідки,

фіксувати незвичні портові патерни і реагувати до того, як зловмисник накотить основну атаку.

Висновки за розділом 1

У ході дослідження теми сканування портів та методів захисту від відповідних атак було розглянуто ключові аспекти, що мають критичне значення для забезпечення кібербезпеки мережевих систем.

Розділ дав цілісне бачення, чому сканування портів – не дрібний інструмент, а стартова лінія будь-якої серйозної кампанії як з боку легальних аудиторів, так і з боку зловмисників. Ми почали з категорій атак, побачили, де пасивні методи перетікають у активну розвідку, а потім змістили фокус на сам процес сканування: як він будується, яких цілей допомагає досягти та які ризики несе навіть у руках білих капелюхів.

Далі були розглянуті методи: від простого TCP-CONNECT до тонких стелс-сканів, що маскуються під інший трафік. Важливо, що кожна методика – це компроміс між швидкістю, точністю і прихованістю. Коли треба охопити тисячі адрес за хвилини, накочують Masscan чи ZMap. Коли потрібен детальний fingerprinting з таймінг-патернами – беруть Nmap із NSE-скриптами. А для точкових експериментів із TTL-маніпуляціями чи нетривіальними прапорцями урізноманітнюють Hping3 та Unicornscan. Netcat при цьому лишається універсальним “швейцарським ножем” для швидкого дебагу.

Також проаналізовані техніки прихованого сканування: розділення запитів між фейковими адресами, фрагментація пакетів, рандомізація порядку номерів портів, тунелювання через HTTP/DNS і навіть Tor-запити. Ці методи дозволяють залишатися непоміченими для базових IDS/IPS, але водночас дають додаткові точки вразливості для адміністраторів, які повинні підганяти правила під кожний патерн.

Окремий блок присвячений впливу сканування на мережу: надмірний трафік може вбити черги обробки, логи швидко забиваються тисячами записів, а аналітики потрапляють у хаос і пропускають справжні інциденти. До цього додається

порталізація вразливостей через банер-грабінг, крадіжка топології за допомогою low-rate сканів і психологічний тиск на IT-команди.

На завершення, приклади з Slammer, Mirai, WannaCry, Shodan-атаки та BlueKeep підкреслили: якщо не спіймати розвідку на самому початку, далі ескалація відбувається в автоматичному режимі. Один порт, одна неконтрольована відповідь і хвиля заражень іде далі, накочуючи критичні служби.

Щоб не дати скануванню перерости у більший інцидент, потрібно поєднати технічні заходи (rate-лімітинг, кешування, аналіз нетипових патернів) з організаційними (регулярні перевірки правил, автоматичні тригери на аномальні запити, навчання персоналу). Саме ця комбінована стратегія дає можливість вчасно зреагувати, відбити першу хвилю розвідки й зберегти інфраструктуру непохитною перед більш агресивними етапами атаки.

РОЗДІЛ 2
ВИЗНАЧЕННЯ СУЧАСНИХ МЕТОДІВ ТА ЗАСОБІВ ПРОТИДІЇ МЕРЕЖЕВИМ КІБЕРАТАКАМ

2.1 Файрволи та їх обмеження в контексті сканування портів

Файрволи – перший рубіж захисту: вони фільтрують трафік за IP, портами та протоколами, інспектують пакети на предмет відповідності правилам [11]. Однак у контексті портсканування їхні можливості виявляють слабкі місця. Базові ACL пропускають або блокують пакети на рівні 5-го та 7-го шарів, але не завжди в змозі диференціювати легітимні спроби з’єднання від розвідки. Через це повільні скани з довгими інтервалами проходять як звичайний трафік.

Stateful-файрволи тримають таблиці з’єднань, перевіряють TCP-handshake і дотримуються станів SYN→SYN-ACK→ACK, але великі об’єми напіввідкритих з’єднань здатні вичерпати їхню пам’ять. Коли атакуючий запускає масовий SYN-flood або розкидає запити з різних IP, файрвол бачить купу окремих дозволених пакетів і не пов’язує їх як сканування портів [12].

У багатьох мережах екран стоїть лише на периметрі [13]. Внутрішній сегмент без стейтфул-фільтрації або із простими ACL залишається без захисту, і скомпрометований IoT-пристрій може безперешкодно сканувати сервера зсередини.

Базові ACL оцінюють лише заголовки IP/TCP/UDP, але не аналізують вміст пакетів. Тому скан-запити, замасковані як легітимний HTTP чи HTTPS, пролітають повз фільтри. Next-Generation Firewall із DPI та контекстною інспекцією виправляють це, але вартість і складність розгорнення часто непідйомні для малих організацій.

Фрагментація IP-пакетів дозволяє розбити запит на шматки без повного заголовка, і багато екранних пристроїв не збирають їх докупи. Тож сканер надсилає по шматочку, обходячи сигнатурні правила [14].

Idle Scan через зомбі-хост маскує початковий IP: файрвол реагує лише на трафік від проміжного вузла, не пов’язуючи його з атакуючим. Пересилання таких запитів через сторонню адресу робить їх непримітними для локальних файрволів, оскільки вони бачать лише пасивний трафік від “зомбі” і не пов’язують його із сканером. Це підвищує прихованість та ускладнює кореляцію [15].

Логи стають ще одним тягарем: тисячі записів від сканів засмічують SIEM, автоматичні правила хибно спрацьовують, а аналітики втрачають фокус. Часто через це детальне логування знижують або взагалі відключають.

Щоб не дати сканам перерости у атаку, треба встановити патч з комплексом заходів: rate-лімітинг, tarpitting для сповільнення підозрілих з’єднань, поведінковий аналіз патернів трафіку та інтеграція з IDS/IPS. Лише так можна аналізувати мережу на рівні аномалій і відбити розвідку ще до активної фази атаки.

Нарешті, активне сканування рідко проявляється одразу як порушення: кожен окремий SYN або FIN пакет технічно може бути допустимим запитом від користувача, що не встиг завершити сесію. Тому лише комплексні системи виявлення аномалій, які спостерігають за частотою появи нових незавершених з’єднань, можуть точно визначити сканування – а звичайний файрвол для цього не підходить.

2.2 Системи виявлення та запобігання вторгнень

Системи виявлення та запобігання вторгнень – це набір інструментів і механізмів, що аналізують мережевий або локальний трафік із метою знайти підозрілі патерни й запустити реакцію ще до того, як атака встигне завдати шкоди [16]. IDS працює пасивно: збирає пакети, порівнює їх із відомими сигнатурами або відхиленнями від базового профілю трафіку, і, коли знаходить аномалію, генерує оповіщення для адміністратора. IPS здатна діяти автоматично – обривати сеанс, скидати пакети або міняти правила файрвола, щоб одразу зупинити атаку. Дані системи – це наступний шар захисту після файрвола. Вони аналізують мережевий трафік не лише за IP та портами, а також перевіряють патерни та контекст сесій, щоб різнити легітимні запити від шкідливих.

За підходом до виявлення розрізняють:

- Сигнатурні рішення, що порівнюють пакети з базою відомих шаблонів атак (Snort, Suricata) [17,18,19].
- Аномальні системи, які формують профіль нормальної активності й реагують на відхилення від нього (Zeek, Prelude) [20].
- Гібридні платформи, що комбінують сигнатури та поведінковий аналіз для зниження числа хибних спрацьовувань [21].

За способом розгортання існують:

- Мережеві IDS/IPS (NIDS/NIPS), які моніторять трафік на центральних точках мережі, між DMZ та внутрішньою зоною.
- Хостові IDS/IPS (HIDS/HIPS), що встановлюються на окремих серверах і аналізують логи, цілісність файлів та системні події.
- Inline-системи (IPS), які працюють по напрямку трафіку, здатні автоматично блокувати підозрілі з’єднання.
- Пасивні сенсорні платформи (IDS), які лише спостерігають і генерують попередження, не впливаючи на потік.

Ключові функції для детекції сканування:

- відстеження стану з’єднань , щоб фіксувати незавершені сеанси як підозрілі;
- аналіз частоти нових сесій з пороговими значеннями на одиницю часу;

- кореляція запитів по різних IP і портах у межах одного вікна спостереження;
- виявлення нерегулярних послідовностей портів;
- контроль аномалій TTL – фіксація підозрілих змін часу життя пакета;
- виявлення фрагментованих пакетів, які можуть обходити файрволи;
- запити до «honeypot»-портів і кореляція відповідей;
- інтеграція з SIEM для накопичення та аналізу логу звернень до портів;
- можливість автоматичного блокування або обмеження швидкості сканування.

Ключові завдання IDS/IPS – швидко виявити портскан або підпис frag-scan, корелювати події з різних сегментів, знизити кількість хибних спрацювань і надати адміністратору чіткі індикатори на кшталт незавершених TCP-сеансів чи серій випадкових SYN-запитів [22]. Успішне розгортання означає не просто купу алертів, а робочий механізм, що запобігає розвідці атак ще на етапі сканування та зберігає цілісність мережі.

2.3 Методи аналізу мережевого трафіку для виявлення атак

Методи аналізу мережевого трафіку допомагають виявляти атаки ще до того, як вони перетворяться на серйозні інциденти [23]. Ключові підходи можна умовно поділити на такі варіанти:

1. Сигнатурний аналіз. Спирається на базу відомих сигнатур атак – шаблонів вмісту пакетів, рядків у заголовках чи послідовностей прапорів. Коли система бачить збіг із правилом (наприклад, характерний SYN-flood чи ряд FIN-пакетів із певними опціями), вона генерує попередження або блокує запит. Перевага в швидкості та точності на старих атаках, недолік – нездатність вловити zero-day чи модифіковані payload-и без оновлення сигнатур [24].
2. Аномальний аналіз. Спочатку створюється профіль нормального трафіку – інтенсивність запитів, розподіл портів, середній розмір пакетів, часові інтервали між сесіями. Все, що виходить за межі цих кордонів, розглядається як потенційна

загроза. Цей метод підходить для виявлення сканів із повільними інтервалами, рандомізованих запитів і нових технік. Вимагає навчання на чистих даних і ретельного тюнінгу, щоб не генерувати купу хибних спрацьовувань під час пікових навантажень [25].

3. Протокольно-станові (stateful) аналіз. Система відстежує життєвий цикл TCP чи UDP сеансу: бачить SYN, ACK прапори або відповідні завершальні пакети. Незавершені хендшейки чи надмір однотипних запитів сигналізують про сканування чи DoS-автоматизовані атаки. Завдяки збереженню стану з'єднання аналізатор розуміє контекст, але через обмеженість пам'яті й таблиць з'єднань може здатись жертвою DoS при великій інтенсивності.

4. Потоковий аналіз (flow-based). Замість аналізу кожного пакета беруть до уваги характеристики потоку (NetFlow, IPFIX, sFlow): джерело/призначення, порти, обсяг байтів, тривалість. Порівняння потоків із шаблонами сканування дає змогу фіксувати характер з низки коротких сеансів до різних портів чи адрес. Ефективно в крупних мережах із централізованим збором метрик і мінімальним впливом на продуктивність [26].

5. Глибока інспекція пакетів (DPI). Аналізують не лише заголовки, а й корисне навантаження: HTTP-запити, DNS-повідомлення, TLS-handshake. DPI дозволяє виявити тунелювання скан-запитів через легітимні протоколи, маскування під веб-трафік і приховані команди [27]. Потребує потужного апаратного забезпечення або спеціалізованих модулів у NGFW, але критично важливий у випадках складних атак.

6. Статистичні та евристичні моделі. Підраховують частоту появи нових сесій, співвідношення успішних і невдалих спроб, ентропію полів. Наприклад, величезний сплеск нових розмов із різними портами за короткий час – ознака масового сканування. Евристика підвищує точність і здатна навчатися новим патернам без жорсткого кодування правил.

7. Кореляція подій та інтеграція з Threat Intelligence. Збирають дані з IDS, файрволів, honeypot-ів, логів хостів і сторонніх баз IOC (Indicators of Compromise). Поєднання сповіщень із різних джерел дає змогу підтвердити сканування як перший

крок атак і швидко визначити, які саме IP чи сервіси під прицілом. Автоматизація кореляції пришвидшує реагування та скорочує час на розслідування.

8. Машинне навчання та штучний інтелект. Сучасні системи застосовують класифікатори для розпізнавання нетипових патернів у високорозмірних даних. Наприклад, нейромережа може аналізувати мультивимірні характеристики потоків і відокремлювати сканування портів від пікового легітимного трафіку. Незамінно для великих дата-центрів і хмарних середовищ, де традиційні методи не справляються з обсягом.

Поєднання цих методів дозволяє побудувати багаторівневий захист: сигнатури ловлять відомі скан-шаблони, аномальні модулі — нові і приховані варіанти, потоки фільтрують масштаб, DPI підсвічує тунельовані атаки, а ML-движок адаптується до змін у поведінці мережі. Такий підхід дає змогу ідентифікувати сканування ще на підготовчому етапі й заблокувати його до ескалації.

2.4 Методика Honeypot

Honeypot – це пастка для зловмисників: спеціальне середовище, яке імітує справжні сервери або сервіси, але не містить цінних даних. Мета полягає в тому, щоб відволікти увагу атакуючих, зафіксувати їхню поведінку та підставні підходи до сканування портів [28].

Для початку розгортають низькоінтерактивний honeypot – простий емульований сервіс із обмеженим набором команд. Він швидко монтується на периметрі мережі і реагує на SYN, TCP чи UDP запити, записуючи час звернення, IP-адресу та параметри пакетів. Такий рівень дозволяє збирати базові метрики сканерів портів без ризику реального вторгнення [29].

Далі додають високоінтерактивний honeypot. Це повноцінна віртуальна машина або контейнер із працюючими демонами – SSH, HTTP, FTP та інші. Туди спрямовують підозрілий трафік із низькоінтерактивного рівня. Місія така: відстежити подальші дії атакуючого: банер-грабінг, спроби входу, використання

експлойтів. Кожна команда, кожен запит логуються детально: вміст payload, таймінги, структура сесії.

Важливий етап – розташування honeypot-систем у різних сегментах: на зовнішньому периметрі, у внутрішній зоні, навіть в окремій VLAN. Так простіше зрозуміти, чи атака розпочалася з публічного інтернету, чи це внутрішній вузол-агент із скомпрометованого IoT [30].

Зібрані логи передають у SIEM, де корелюють із подіями файрвола та IDS. Непрямі індикатори – велика кількість синхронних спроб з’єднання, нестандартні прапорці TCP, фрагментація пакетів або рандомізовані порти – дозволяють виявити сканування на початковому етапі [31].

Honeypot залучають до мережі через розкриття публічних IP-адрес або через DNS-записи. Це приваблює сканери та шкідливі боти. Як тільки зловмисник починає якусь активність – наприклад, сканування чи спробу експлуатації – адміністратори отримують детальну інформацію про тактики, техніки та процедури зловмисника.

Загалом, дані системи мають ряд переваг, серед яких:

- Точне відсіювання хибних спрацювань.
- Детальний аналіз пакетів зловмисника.
- Мінімум навантаження на продуктивні сервіси.
- Адаптивне формування сигнатур.
- Виявлення невідомих методик.
- Можливість тестувати реакцію безпеки.
- Розгортання даних систем відносно просте.

Незважаючи на очевидні переваги проактивного збору розвідданих, необхідно враховувати ризики: неправильна ізоляція може перетворити Honeypot на плацдарм для подальших атак, а занадто прості пастки можуть бути легко розпізнані досвідченими зловмисниками. Тому успішна реалізація Honeypot-системи передбачає гібридний підхід із поєднанням регулярного оновлення підроблених сервісів під актуальні вразливості та автоматизоване реагування на аномалії з подальшою кореляцією подій у загальній системі безпеки [32].

На рисунку 1.1 зображена схема з багаторівневою архітектурою захисту мережі з Honeypot-системами: зловмисник із Інтернету спочатку натрапляє на зовнішній Honeypot та міжмережевий екран, який фільтрує зловмисний трафік і перенаправляє підозрілі запити на пастки. Попереду екрану розташований зовнішній Honeypot, що імітує вразлиий сервіс, а позаду, у DMZ – реальні поштові, DNS та веб-сервери, доповнені другорядною Honeypot-системою. У внутрішній мережі за DMZ розміщено критичні корпоративні та серверні ресурси, до яких доступ можливий лише через перевірений трафік, що пройшов фільтрацію і не відхилений засобами виявлення та ізоляції підозрілих потоків.

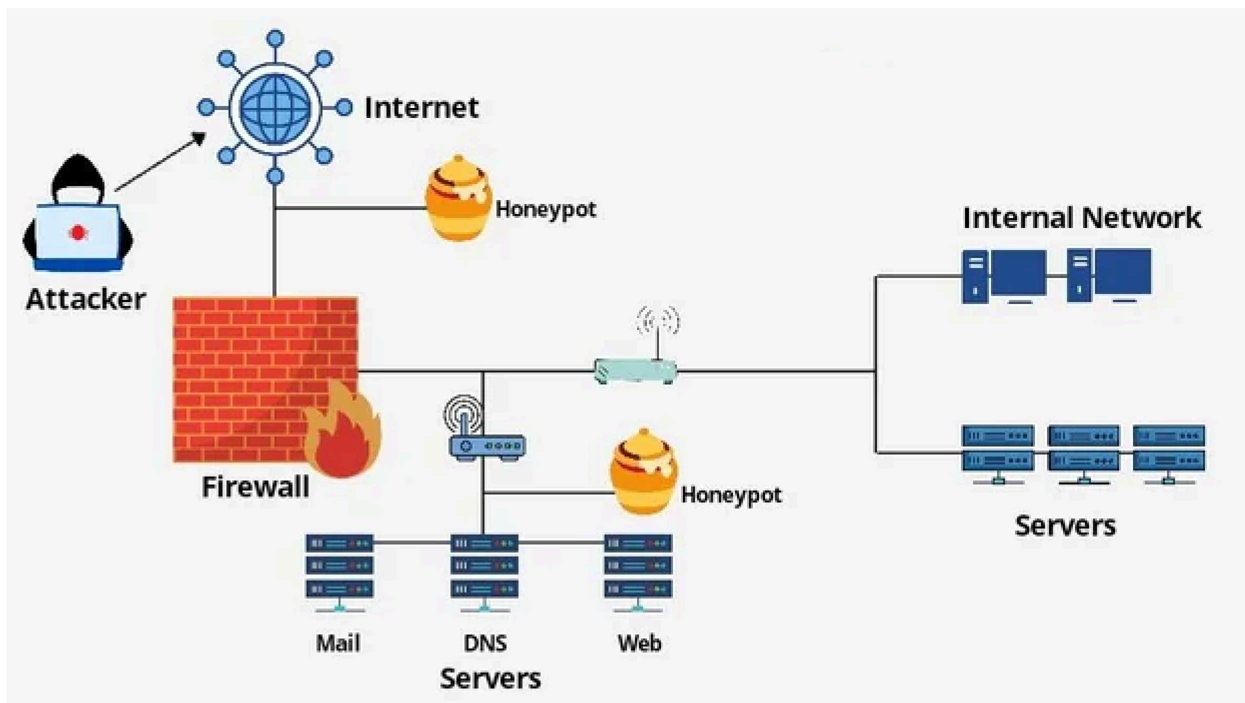

Рисунок 1.1 – схема архітектури мережі

Також варто пам’ятати про ризики – підтримка honeypot потребує окремого моніторингу, а надмірна взаємодія з ним може видати його природу. Тому оптимальне поєднання пасток із традиційними IDS/IPS і тарпіттінгом створює гнучкий та проактивний захист, який реагує ще до запуску основної фази атаки.

2.5 Аналіз аномалій із використанням статистичних та машинно-навчальних підходів

Аномалії в мережевому трафіку видають підозрілі зразки, які не вкладаються в профіль звичних перевантажень. Щоб їх ловити, застосовують два базові підходи – статистичний і машинно-навчальний [33].

У статистичному аналізі вибирають ключові метрики: кількість нових TCP-сеансів за хвилину, розподіл портів за популярністю, середню тривалість сесій, зміну ентропії полів IP та TTL. Для кожного показника накочують історичні дані, розраховують середнє, дисперсію, границі довірчого інтервалу. Коли поточне значення виходить за встановлені пороги, система сигналізує про аномалію. Цей підхід добре працює з повільними скануваннями – slow scan не накочується миттєво,

тому поріг частоти нових з'єднань легко перевищити в типовому режимі [34].

Для детекції розподілених сканерів портів аналізують співвідношення нових сесій на різні порти від одного джерела чи групи IP. Якщо у вікні спостереження 70% запитів припадає на випадкові порти без завершених рукостискань – це явний зловмисник. Реалізують sliding window: кожні N секунд оцінюють статистики й калібрують границі з урахуванням довготривалої бази.

Машинне навчання підходить для складних випадків та великих дата-сетів. У безнаглядному режимі запускають моделі кластеризації або ізоляційний ліс, які відокремлюють нормальний трафік від скан-трафіку як аутлайєрів. Ізоляційний ліс особливо корисний, бо автоматично розставляє пріоритети ознакам із найбільшим вкладом в унікальність об'єкта [35].

Однокласові алгоритми навчають модель тільки на легітимному трафіку, а потім кожний новий потік оцінюється за відстанню від гіперплощини. Критична відстань – сигнал про сканування. Водночас доводиться редагувати різні параметри, щоб не мати сотень хибних спрацювань під час пікових навантажень.

Глибинні мережі навчають кодувати й відтворювати нормальний трафік. Якщо відтворення поточного пакета чи потоку дає велику помилку, це вказує на аномалію. LSTM-архітектура враховує часову послідовність запитів, тому вловлює як раптові сплески, так і довгі серії випадкових портів [36].

Усі ML-підходи покладаються на даний стек показників:

- швидкість запитів і пікові значення;
- коефіцієнт невдалих рукостискань;
- число унікальних портів за вікно;
- середня довжина пакетів і розподіл TTL;
- співвідношення TCP/UDP/ICMP;
- час між пакетами.

Пайплайн виглядає так: натренувати модель на позитивному трафіку, перенести на продакшн, збирати дані онлайн, потім періодично переїхати на свіжі зразки і корегувати пороги. CI/CD для моделей автоматизує цей цикл, щоб нові варіанти сканів не проходили непоміченими [37].

Поєднання статистики з ML – золотий стандарт. Спершу ідентифікуються грубі сплески, далі ML обробляє це, відсіюючи хибнопозитиви і виявляючи нетривіальні варіанти сканів [38]. Такий двошаровий захист дає змогу помітити розвідку ще на підготовчому етапі й блокувати її на рівні аналітики, перш ніж вона переросте в реальну атаку.

2.6 Автоматизоване блокування та динамічна фільтрація на основі результатів аналізу

Після виявлення аномалії або підозрілого потоку настає перехід від пасивного моніторингу до активної протидії. Автоматичне блокування та динамічна фільтрація стає критично важливою. Замість ручного втручання система сама оновлює правила файрвола чи IDS/IPS, скорочуючи час від детекції до реакції до мілісекунд [39].

Перший компонент – автоматичне додавання IP-адрес до чорного списку. Коли аналізатор фіксує сплеск SYN-запитів або серію випадкових портів від однієї адреси, вона через API відправляє команду до мережевого екрана (iptables, nftables та ішні) та блокує це джерело на певний час. Такий підхід називають rate-based blocking.

Другий елемент – адаптивний rate-лімітинг. На основі профілю трафіку система змінює порогові значення дозволеної швидкості нових сесій. Під час піку аномалії правила автоматично підвищують затримку або знижують пропускну спроможність для підозрілих портів, не торкаючись звичайних користувачів.

Також варто згадати про тарпіттінг і грейхолінг:

- Тарпіттінг – замість миттєвого відкидання підозрілих з’єднань система штучно затримує відповіді на TCP-запити від заданого IP. Це змушує сканер чекати таймаутів, уповільнює темп розвідки та виводить на світ його стратегію без навантаження бойових сервісів [40].
- Грейхолінг – техніка часткового фільтрування: система дозволяє пройти лише невеликій долі запитів із підозрілої адреси, а решту відкидає. Сервіс залишається доступним для легітимних клієнтів, але сканер стикається з високим відсотком провалів і не може наростити швидкість атак [41].

Після того як локальна система помітила підозрілу активність, вона звертається до зовнішніх баз даних з шкідливими IP, доменами чи хешами. Якщо там фігурує той самий адрес, блокування триває довше або стає постійним.

Ключові функції для реалізації:

- API-інтерфейси між IDS/IPS, SIEM і файрволом для миттєвого оновлення ACL.
- Динамічні правила в SDN-середовищі або оркестратори для швидкого розгортання змін.
- Модуль rate-лімітінгу з можливістю плавного підвищення/зниження порогів згідно з аналітикою.
- Тарпіттінг через спеціальні політики TCP.
- Механізми автоматичного видалення IP з чорного списку після завершення атаки.

Автоматизація блокує початкову розвідку і не дає зловмиснику набрати обертів. Динамічна фільтрація адаптується до поточної ситуації в мережі, мінімізує хибні блоки і зберігає стабільність сервісів. Саме така стратегія скорочує вікно вразливості і дає змогу реагувати на сканери портів ще до етапу експлуатації вразливостей.

2.7 Оцінка ефективності різних методів захисту

При оцінці методів захисту від мережевих атак зазвичай порівнюють кілька ключових параметрів: точність виявлення, рівень хибних спрацювань, затримку обробки трафіку, масштабованість та витрати на впровадження й підтримку.

Статичні ACL на рівні файрвола працюють миттєво й не потребують особливих ресурсів, але пропускають stealth-скани та затримані запити. Stateful-екрани краще розрізняють початок сесії, проте при масовому SYN-флуді їхні таблиці швидко заповнюються, що може призвести до DoS самої системи захисту [42]. NGFW із DPI вловлюють тунелювання через HTTP/HTTPS, але їхня продуктивність падає під час шифрованого трафіку, а вартість апаратної платформи

досить висока.

Сигнатурні IDS/IPS дають високу точність на відомих атаках та мінімум хибних спрацювань, але вимагають постійного оновлення. Аномальні модулі вловлюють нетипові варіанти slow-scan або randomized scan, але без тонкого калібрування створюють шквал фальшивих попереджень під час пікових навантажень. Гібриди поєднують обидва підходи – дають баланс між виявленням і шумом, але складні в налаштуванні та підтримці [43].

Аналіз аномалій із використанням статистики відпрацьовує грубі сплески та нетипові патерни, але не розпізнає добре замасковані або адаптивні скани. ML-моделі виявляють нові атаки й мінімізують фальшиві спрацювання при коректному навчанні, але потребують чималих обчислювальних ресурсів і регулярного пере навчання на свіжих даних [44].

Глибока перевірка пакетів дозволяє отримувати майже стовідсоткову точність за рахунок аналізу контенту пакетів, але має високу затримку і потребує потужних серверів [45]. Flow-аналітика обробляє інформацію про сеанси з мінімальною затримкою і добре масштабується до десятків тисяч потоків, але не бачить вмісту пакетів і дає середню точність.

Honeypot-пастки мають майже нульовий рівень хибних спрацювань: будь-яка взаємодія з ним – ознака розвідки. Вони забезпечують глибоку аналітику тактик атакуючих, але самі по собі не захищають бойові сервери – мінімум відстрочують атаку, а потреба в окремому моніторингу та ізольованих середовищах додає складності й витрат.

У підсумку: жоден метод не закриває всі вектори. Лише поєднання вищеописаних методів дає збалансовану картину – швидке реагування, мінімум хибних спрацьовувань і збереження пропускної здатності [46]. Регулярний аналіз логів і накочування оновлень на основі зібраних даних дозволяють адаптувати захист під нові методики сканування без простоїв. У таблиці 1.2 порівняно п’ять підходів до виявлення та протидії мережевим загрозам за ключовими метриками: чутливість, рівень помилкових спрацьовувань, затримка обробки, масштабованість

та вартість впровадження.

Таблиця 1.2

Порівняння методів виявлення сканування

Метод	TPR	FPR	Затримка обробки	Масштабованість	Вартість впровадження
Signature-IDS	95-99 %	<1 %	1–5 мс	середня	низька
Anomaly-IDS / ML	85-95 %	5-10 %	5–15 мс	середня	висока
Deep Packet Inspection	≈100 %	< 1 %	10–50 мс	середня	висока
Flow-аналітика	80-90 %	2-5 %	< 1 мс	висока	середня
Honeypot	100 %	≈0 %	Залежить від реалізації	середня	середня

Ця таблиця демонструє, що жоден метод сам по собі не є універсальним: ефективна безпека досягається комбінуванням підходів – наприклад, поєднанням DPI для глибокого аналізу, flow-моніторингу для продуктивності, сигнатурного й аномального виявлення для точності, а також honeypot для додаткової розвідки й підтвердження.

2.8 Методичні рекомендації щодо захисту від сканування

Сканування портів – це базова розвідка, яка дає змогу атакуючому швидко зрозуміти, куди нанести основний удар. Захист треба будувати шарами: від скорочення самої поверхні атаки до активних механізмів виявлення [47]. Дотримання наведених нижче рекомендацій допоможе знизити ймовірність успішного сканування та не дати зловмиснику вивчати мережу:

1. Мінімізувати відкриті порти: вимкнути непотрібні сервіси та перевірити залишкові через netstat або ss.
2. Впровадити stateful-файрвол: жорсткі правила за IP та портами, rate-лімітинг для TCP-запитів.
3. Розгорнути IDS/IPS: сигнатурна та аномальна детекція з оновленням правил за результатами логів.
4. Маскування відкритих портів: ідея полягає в тому, щоб змусити зловмисника повірити, що порт просто не відкритий, а не фільтрується файрволом.
5. Збирати і аналізувати трафік: raw-сокети або libpcap, централізовані логи в SIEM, кореляція подій.
6. Використовувати honeypot-пастки: low і high interaction на критичних портах для збору тактик сканування.

У сумі виконання даних рекомендацій підвищить стійкість мережі та знизить ризик успішного вторгнення ще на етапі розвідки.

Висновки за розділом 2

Цей розділ показав, що захист від сканування портів та суміжних атак потребує багаторівневого підходу. Файрволи – перший рубіж, але stateless-фільтри пропускають приховані скани, а stateful-системи легко перевантажити масовим SYN-флудом або повільним скануванням [48].

Наступним кроком ідуть IDS/IPS, які ловлять сигнатурні атаки та аномалії в мережевих потоках. Сигнатурні детектори швидкі й точні щодо відомих загроз, а аномальні модулі вловлюють нові схеми, але потребують тонкої конфігурації та ресурсів для ML-моделей.

Методи аналізу трафіку доповнюють цей захист: поєднання сигнатур із потоковою аналітикою дає швидке виявлення, DPI допомагає виявити тунелювання, а кореляція подій із SIEM створює повну картину атаки.

Honeypot-пастки дозволяють вивчити тактики зловмисників без ризику для головних серверів: приманка з низьким рівнем взаємодії фіксує сам факт розвідки, а

приманка з високим рівнем взаємодії – повний ланцюжок експлуатації. Зібрані дані дають змогу налаштувати IDS/IPS та файрволи на основі реальних прикладів.

Статистичний та ML-аналіз аномалій концентруються на нетипових патернах: швидкість нових сесій, ентропія портів, співвідношення невдалих рукостискань [49].

Автоматизоване блокування й динамічна фільтрація скорочують час від виявлення до реагування: rate-лімітинґ, тарпіттінг і грейхолінг уповільнюють розвідку.

Оцінка ефективності методів демонструє: жоден інструмент не закриває всі вектори. Лише комбінація з файрвола, IDS/IPS, аналізу трафіку, honeypot-ів, статистики/ML та динамічних правил забезпечує комплексний захист, зберігаючи стабільність мережі й мінімізуючи хибні спрацьовування.

Нарешті, для довготривалої надійності необхідно регулярно оцінювати ефективність кожного інструмента за низкою показників – від рівня виявлення і кількості хибних тривог до затримок обробки трафіку й вартості експлуатації.

РОЗДІЛ 3
РЕКОМЕНДАЦІЇ ЩОДО РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Вибір мови програмування та платформи

При розробці системи детекції та блокування сканування портів критично важливо обрати таку мову програмування та операційну систему, які забезпечать максимальну продуктивність, гнучкість у роботі з мережевими інтерфейсами та стабільність у серверному середовищі. Я розглянув основні альтернативи та аргументи на користь поєднання C та Linux.

Низькорівнева робота з мережею є основним критерієм. Система має безпосередньо захоплювати IP-пакети нижче рівня TCP/IP-стека, використовувати необроблені сокети, аналізувати заголовки IP, TCP і UDP. Для цього необхідний прямий доступ до POSIX API, включно із викликами socket(AF_INET, SOCK_RAW), а не через обгортки високорівневих бібліотек.

Необхідна продуктивність та низькі затримки. Очікується обробка десятків тисяч пакетів за секунду з мінімальною затримкою до декількох мікросекунд. Інтерпретовані мови (Python, PHP) або середовища з віртуальною машиною (Java, .NET) накладають надто великі накладні витрати на виклики системних функцій і потребують додаткових оптимізацій, що ускладнює реалізацію. Така мова як Go здобула популярність завдяки вбудованій підтримці concurrency та стандартному пакету net. Go-інструменти типу gopacket спрощують роботу з пакетами, але для супер-тонкого DPI чи eBPF-фільтрів все ж зазвичай дописують C-проксі.

Рішення має бути надійним і портативним у серверних середовищах. Більшість промислових серверів та вбудованих пристроїв працюють під Linux або іншими UNIX-подібними ОС (FreeBSD, Solaris). Використання мови C у поєднанні зі стандартами POSIX гарантує максимум переносимості між дистрибутивами та мінімум зовнішніх залежностей [50,51].

Аналіз альтернатив:

- C++ дозволяє писати високорівневий код, але додає складність через менеджмент пам'яті, шаблони та STL, що накладають непередбачувані накладні витрати. Для чистої, детерміністичної обробки мережевого трафіку ці особливості є зайвими.
- Go / Rust - нові, системні мови з вбудованою підтримкою конкурентності й безпеки пам'яті. Але їх рантайм може збільшувати час реакції на пакети та ускладнює інтеграцію з низькорівневими мережевими API без сторонніх бібліотек.
- Інтерпретовані мови (Python, PHP) - простіші в розробці, але практично не підходять для роботи з необробленими сокетами на високих швидкостях без використання C-модулів.

Переваги C як мови реалізації:

- Мінімальні накладні витрати: компіляція в нативний машинний код дає абсолютний контроль над кожним байтом пам'яті та часом виконання.
- Прямий доступ до POSIX API: робота з socket(), recvfrom(), fcntl() без проміжних шарів, що гарантує стабільність і передбачувані затримки.
- Контроль пам'яті на рівні програми: відсутність непередбачуваного збирання сміття (garbage collection) спрощує розрахунок часу обробки та уникнення пауз у виконанні.
- Широке поширення: компілятори (GCC, Clang) встановлені на будь-якому сервері Linux, а код C легко супроводжувати і портувати.

Що дає вибір операційної системи на базі ядра Linux:

- Нативна підтримка raw-сокетів і Netfilter: Linux має зрілий стек для роботи з мережевими інтерфейсами, модулі ядра (iptables, nftables) та механізми CAP_NET_RAW, що дозволяють програмі безпечніше отримувати пакети та створювати правила файрвола [52].
- Сучасні API (epoll, timerfd, signalfd) забезпечують масштабовану обробку великої кількості одночасних подій, що може бути використано для подальшого оптимізованого перехоплення пакетів замість usleep().

- Велика екосистема: більшість серверів використовують дистрибутиви Linux, що полегшує розгортання демона через systemd-юніти, керування привілеями (systemd capabilities) та централізоване логування через journal [53].
- Спільнота та документація: велика кількість прикладів, документації та готових рішень на GitHub та у man-сторінках спрощують розробку та підтримку.

У результаті аналізу технічних й організаційних вимог обрана мова C та операційна система Linux як оптимальне рішення для реалізації високопродуктивного, надійного та легко підтримуваного детектору портсканування. Це поєднання забезпечує бездоганну продуктивність, прямий доступ до необхідних мережевих API й інтеграцію з існуючими інструментами безпеки в серверному середовищі.

Наявність інструментів оптимізації на Linux дозволяє впровадити lock-free алгоритми й використовувати eBPF для відфільтровування трафіку прямо в просторі ядра. Мінімальна залежність від зовнішніх бібліотек, можливість статичного лінкування та простий деплой у контейнерах підвищують портативність рішення та спрощують його підтримку у хмарних середовищах.

У таблиці 1.3 порівняно декілька мов програмування за основними критеріями.

Таблиця 1.3

Порівняння мов програмування

Критерій	C	C++	Go	Python
Низькорівневий доступ	Прямий доступ до raw-сокетів і POSIX API	Так само, плюс стандартні та сторонні обгортки	Обмежений, через net пакет без raw-сокетів	Через модуль socket, але без прямої роботи з raw-сокетами
Продуктивність	Максимальна, мінімальні затримки	Майже на рівні C	Дуже висока, ефективний рантайм	Нижча, інтерпретована мова

Продовження таблиці 1.3

Керування пам’яттю	Явне (malloc/free), повний контроль	Контроль вручну та смарт-вказівники (RAII)	Автоматичне через garbage collector	Автоматичне через reference counting
Паралелізм та конкурентність	POSIX threads, epoll/kqueue	std::thread, Boost.Thread, асинхронні шаблони	Вбудовані *Таблиця 1.3* та канали, легка модель конкурентності	GIL обмежує багатопоточність, multiprocessing для процесів
Швидкість розробки	Низька	Середня	Висока	Дуже висока
Екосистема та бібліотеки	libpcap, pthread, низькорівневі API	STL, Boost, Qt, Poco	Стандартна бібліотека (net, html, testing)	Широкі можливості для веба, ML, Data Science
Безпека виконання	Вразливий до пам’яті, потребує ручного контролю безпеки	Менше помилок завдяки RAII, але можливі переповнення	Відсутність управління пам’яттю вручну знижує помилки	Безпечний за замовчуванням, але ризики в логіці програми
Рекомендовані сценарії	Системні демони, драйвери, висока продуктивність	Складні серверні програми, ігри, високонавантажені сервіси	Мережеві сервіси, мікросервіси, інфраструктурні утиліти	Скрипти, швидкий прототипинг, веб-API, аналітика даних

3.2 Захоплення та препроцесінг пакетів

На етапі проектування слід передбачити захоплення мережевого трафіку безпосередньо на рівні IP-пакетів, обминаючи додаткові протокольні прошарки. Прийом даних має організовуватися у неблокуючому режимі через механізм подій, щоб система миттєво реагувала на надходження нових пакетів. Буфери прийому рекомендується виділити наперед у фіксованому обсязі та вирівняти, що знизить накладні витрати на копіювання й запобігатиме кеш-промахам. На етапі

препроцесінгу варто відокремити базове розбирання заголовків від подальшої логіки аналізу, відкидаючи некоректні або фрагментовані пакети одразу на вході. Такий підхід забезпечить найменші затримки між прийомом та детекцією аномалій і дозволить масштабувати обробку на високошвидкісних каналах.

Raw-сокети дають можливість слухати мережу на найнижчому рівні – можна отримати прямий доступ до Ethernet-фреймів без посередництва транспортного стека [54]. Їх ще називають сирі або необроблені сокети. Вони надають програмісту ширші можливості порівняно з рештою сокетного API. Необроблені сокети використовуються, коли нам потрібно отримати безпосередній доступ до мережевого рівня та рівня каналу передачі даних [55]. Наприклад, ми можемо використовувати необроблені сокети для реалізації мережевого протоколу нижче транспортного рівня, такого як ICMP, ARP або OSPF. Крім того, ми можемо використовувати їх для моніторингу мережі, перехоплення пакетів або злому, надсилаючи підроблені пакети [57].

Підсумовуючи, необроблені сокети мають такі відмінності від традиційних сокетів:

- Його можна використовувати для надсилання та отримання пакетів на мережевому рівні та канальному рівні.
- Його можуть відкрити лише користувачі root.
- Його можна відкрити лише для сокета типу SOCK_RAW.
- Програміст може створити заголовок пакета мережевого рівня або повинен вручну створити заголовок кадру каналу передачі даних.

Щоб не копати тонни зайвих даних, перед захопленням накочують фільтри на рівні ядра. Вони відсівають все, окрім потрібних протоколів або портів. Це економить ресурси і дозволяє зосередитися на скан-патернах ще до препроцесінгу.

Після отримання сирих байтів у користувацькому просторі йде препроцесінг. Спочатку додають точний таймстемп для кожного пакета, щоб корелювати події між різними хостами. Наступний етап – виділення ключових полів: IP-пари, номери портів, прапорці TCP і довжину payload. Кожен пакет отримує унікальний хеш, щоб виключити дублікати й мінімізувати зайве опрацювання.

Після цього структура даних передається в сигнатурний детектор або ML-движок для подальшого аналізу. У разі необхідності одразу можна зробити патч до фільтрів або правил інспекції, щоб миттєво реагувати на нові техніки сканування [58].

Такий підхід дає змогу отримати чисті, структуровані дані з мінімальним оверхедом і зберегти продуктивність системи навіть під час високошвидкісного захоплення.

Крім того, варто розділити обробку TCP і UDP пакетів на дві незалежні гілки логіки. TCP-пакети аналізуватимуться окремо, з урахуванням порядку флагів і стану з’єднання, а UDP-пакети - у своєму потоці, з огляду на характер протоколу. Це дозволить спростити розбір заголовків для кожного протоколу, уникнути перехресних перевірок і оптимізувати обробку з урахуванням особливостей TCP та UDP.

На рисунку 1.2 зображено порядок обробки вхідного мережевого трафіку з урахуванням рівнів моделі OSI. Спочатку кадр надходить через фізичний рівень, потім потрапляє на канальний рівень: тут він дублюється у бібліотеці libpcap для пасивного моніторингу, а основна лінія маршруту переходить до мережевого рівня. На рівні мережі пакет опрацьовується модулем iptables (netfilter) – якщо правило дозволяє, він передається до сирих сокетів та вбудованого мережевого стека ядра і піднімається на транспортний рівень, звідки доставляється у відповідний UDP або TCP сокет. Користувацька програма може підняти сирий сокет та інспектувати заголовки IP/TCP/UDP на найнижчих рівнях без участі високорівневих API.

Таким чином, можна поєднати простий файрвол із розділеним потоком сирих і звичайних сокетів, забезпечуючи як блокування небажаних з’єднань, так і глибоку перевірку підозрілого трафіку.

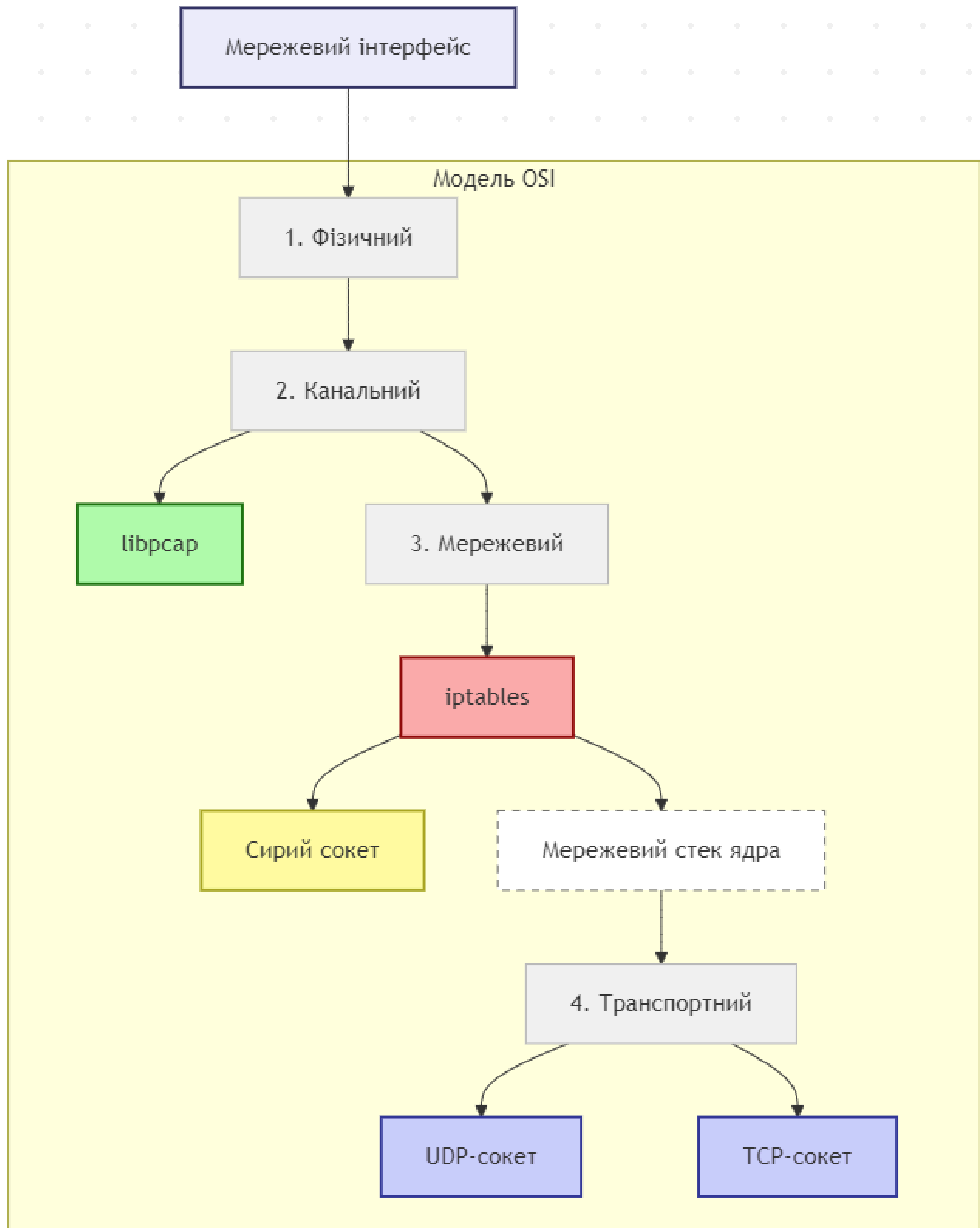

Рисунок 1.2 – схема декепсуляції пакетів

3.3 Механізм реагування на атаку

Механізм реагування на атаку починається з моменту детекції аномалії. Для оперативного блокування виявлених сканерів рекомендовано:

- Одразу після порушення правил поміщати IP-адресу в чорний список мережевого фільтра через відповідний набір команд (наприклад, через додавання елемента до ipset blacklist з таймаутом або вставку правила в nftables).
- Фіксувати в журналі подію блокування з міткою часу та причиною, щоб адміністратор міг простежити хронологію інцидентів.
- Делегувати зняття заборони в ізольований фоновий процес, який через конфігурований інтервал автоматично видаляє запис із чорного списку, не перериваючи основну обробку пакетів.
- Перевіряти наявність IP у чорному списку перед спробою блокування, щоб уникнути дублювання правил та надмірних викликів до системи фільтрації.
- Відокремити виконання команд файрволу від критичних шляхів детекції, передбачивши чергу або окремий робочий потік для взаємодії з ядром, щоб мінімізувати затримки в основному алгоритмі [59].
- Підтримувати конфігурування часу бану через зовнішній файл, аби в разі потреби змінити політику блокування без перекомпіляції.
- Вмикати перевірку результату виконання команд фільтра (код повернення або опитування чорного списку) та логувати можливі помилки для швидкого реагування на проблеми з мережевим інструментарієм.

Дотримання цих рекомендацій забезпечить миттєве реагування на підозрілий скан, ізольовану обробку затримок бану та своєчасне очищення правил без втрати продуктивності сервісу.

На рисунку 1.3 наведена схема, на якій проілюстровано, як працює міжмережевий екран Netfilter: пакети спочатку потрапляють у точку PREROUTING, де відбувається попередня обробка, далі ядро вирішує, чи призначені вони локальному хосту (ланцюг INPUT), чи мають бути переадресовані (FORWARD), а потім остаточно проходять через POSTROUTING перед виходом у мережу. Механізм

реагування на атаку полягає в тому, що одразу після виявлення аномалії, потрібно програмно додавати правило DROP у відповідні гачки Netfilter (зазвичай PREROUTING для вхідних чи FORWARD для транзитних пакетів). Це гарантує миттєве блокування подальших пакетів від зловмисного IP ще до їх передачі у мережевий стек або назад у мережу.

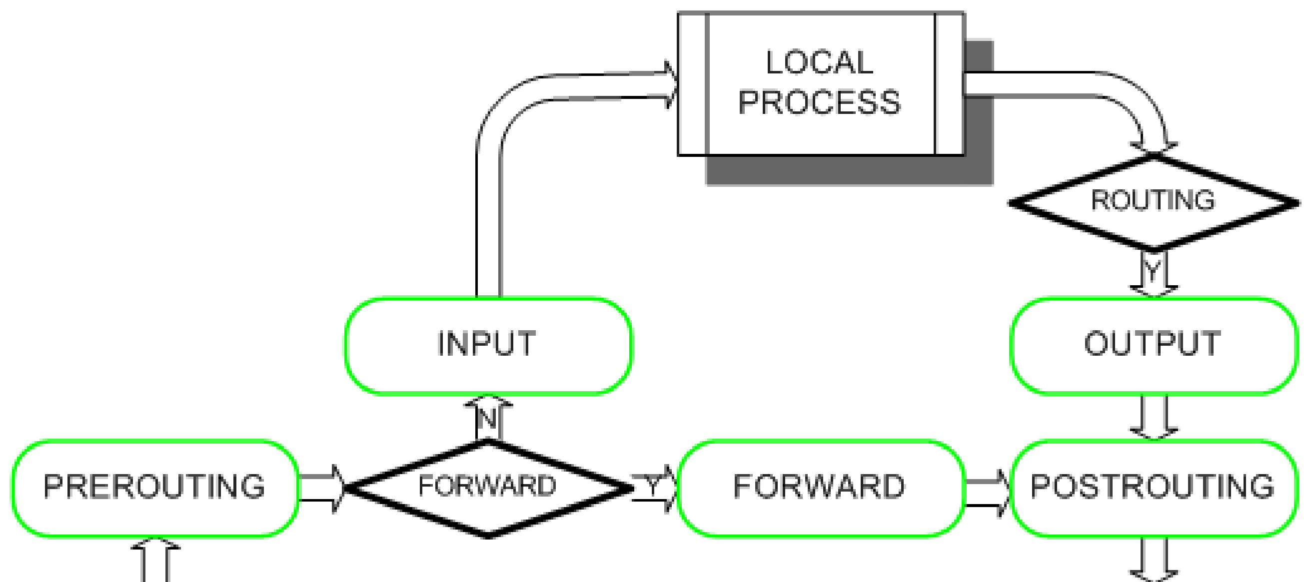

Рисунок 1.3 – принцип роботи Netfilter

3.4 Забезпечення потокобезпечності алгоритму

Потокобезпечність означає, що код у багатопоточному середовищі працює коректно без гонок даних і крашів. Іншими словами, кілька потоків можуть одночасно викликати один і той самий модуль або функцію водночас результати залишаться передбачуваними. Без потокобезпечності під навантаженням виникають невловимі баги – неправильні підрахунки, втрати пакетів чи deadlock-и. Забезпечивши коректні синхронізаційні механізми, можна гарантувати стабільну та передбачувану обробку трафіку на високих швидкостях.

Щоб гарантувати коректність обробки у багатопоточному середовищі, варто дотриматися певних рекомендацій. Для обробки високошвидкісного трафіку потрібна чітка стратегія уникнення затримок від блокувань. У багатопотоковій

обробці трафіку кілька ниток одночасно працюють із загальними даними – лічильниками пакетів, таблицями станів і буферами кадрів [60]. Без потокобезпечності з ними працювати майже нереально: гонки даних призводять до некоректних підрахунків, втрати пакетів або навіть краху сервісу під навантаженням.

В майбутньому сервісі захисту від атаки сканування мережевих портів на основі аналізу трафіку потрібно дотриматись таких правил:

- Відокремлений прийом і обробка. Ввесь трафік з мережі читає окремий сніффер-потік, який пише сирі пакети в lock-free кільцевий буфер. Цей буфер працює на атомарних індексах head/tail, тому ніяких mutex-ів не потрібно — сніффер та воркери не заважають одне одному і не блокують критичні ділянки.
- Критичні структури під захистом локів. Таблиці станів IP і лічильники запитів – спільні ресурси для всіх воркерів. Оновлення лічильника одного IP відбувається під захистом pthread_mutex, а читання стану – через pthread_rwlock, щоб десяток потоків міг паралельно перевіряти пороги без черги на запис.
- Бар’єри пам’яті. Щоб воркери одразу бачили завершені записи сніффера в буфері, потрібно використовувати бар’єри пам’яті. Це гарантує, що дані повністю записані перед тим, як інший потік їх прочитає, й ніяких половинчастих структур не виникає.
- Thread-local storage для тимчасових даних. Парсинг заголовків і формування структури пакета виконує кожний потік у власних буферах. Ніяких алокацій у кожному циклі і ніяких mutex-ів на пул пам’яті – кожен воркер має свій кеш для тимчасових об’єктів.
- Graceful shutdown. Для зупинки сервісу потрібно застосувати стоп-прапор. Кожний воркер перевіряє цей флаг перед читанням нового пакета; коли він встановлений, потік виходить з циклу та очищає свої ресурси без deadlock-ів.

Поєднання цих методів дає змогу обробляти сотні тисяч пакетів у секунду без зависань і простоїв, ефективно реагувати на пікові навантаження та гарантувати коректність під час паралельної детекції та блокування [61]. У результаті алгоритм залишається стабільним навіть під високим навантаженням та оперативно реагує на аномалії.

3.5 Логування роботи програмного забезпечення

Логування є критично важливим елементом будь-якого мережевого сервісу детекції й блокування атак. По-перше, воно забезпечує прозорість: записуючи ключові події – старт і зупинку служби, моменти блокування та розблокування IP-адрес, а також помилки взаємодії з мережевими інтерфейсами і файрволом – адміністратор отримує чіткий хронологічний ланцюжок дій програми. По-друге, логи полегшують відлагодження й усунення несправностей: у разі збоїв або некоректної роботи можна швидко виявити, на якому кроці сталася помилка, та зрозуміти її причину [62].

Крім того, логи є невід’ємною частиною процесу безпеки: вони слугують доказовою базою під час розслідування інцидентів, дозволяють корелювати події з іншими системами (IDS/IPS, SIEM) і відстежувати розвиток атаки в реальному часі. Регулярний аналіз журналів допомагає виявити приховані шаблони поведінки зловмисників, коригувати пороги детекції та своєчасно адаптувати захисні політики.

Нарешті, продумане логування сприяє управлінню ресурсами: за допомогою статистики обсягу логів і часу обробки записів можна оптимізувати налаштування буферів, ротацію логів та інтеграцію з центральними сховищами, мінімізуючи вплив на продуктивність у пікові моменти навантаження. Продумана політика логування – це не просто архів подій, а інструмент реагування: на основі аналізу минулих інцидентів можна підганяти пороги, відстежувати тихі скани та відпрацьовувати нові сценарії захисту.

Загалом, логування має такі переваги:

- миттєве відстеження інцидентів за timestamp і thread-ID для швидкого дебагу;
- кореляція подій між кількома вузлами і компонентами без зайвих зусиль;
- асинхронний запис у lock-free чергу без блокувань гарячих ділянок обробки;
- автоматична ротація й архівування старих логів для форензік аналізу;

- різні формати для легкого парсингу й інтеграції з SIEM;

3.6 Рекомендації з оптимізації та розширюваності

Щоб забезпечити високу продуктивність і легкість розвитку проекту в майбутньому, варто виділити кілька ключових напрямів оптимізації та розширення функціональності. Оптимізація критична, щоб зберегти низьку латентність і високу пропускну здатність під піками трафіку – без цього алгоритм зависає й втрачає пакети. Розширюваність потрібна, щоб швидко додавати нові методи детекції під вектор атак, не переписуючи весь сервіс [63]. Розділення компонентів і плагінна архітектура спрощують підтримку й дають змогу накотити патч тільки в потрібний модуль. У сукупності це забезпечує стабільний захист мережі й мінімізує витрати на подальший розвиток.

Насамперед, рекомендується замінити періодичні цикли з usleep на механізми асинхронного вводу-виводу (epoll на Linux або kqueue на BSD/macOS). Це дозволить відмовитися від зайвих очікувань у циклі захоплення пакетів і обробляти події лише тоді, коли вони дійсно надходять, що суттєво знижує навантаження на CPU при малому або середньому трафіку.

Ще одним важливим кроком є винесення взаємодії з підсистемою файрвола в окремий пул робочих процесів або асинхронну чергу. Виклики system() у головному потоці можуть призводити до затримок та непередбачуваних блокувань, тому краще передавати команди через pipe або використовувати NFQUEUE, де ядро передає пакети на обробку у користувацький простір без затримок блокуючих викликів.

Для розширюваності архітектури доцільно впровадити плагінну систему: виділити чіткий API для модулів аналізу і завантажувати їх динамічно через dlopen()/dlsym(). Це дозволить додавати нові методи детекції – машинне навчання, DPI, кореляцію з SIEM – без потреби рекомпіляції ПЗ.

Конфігурацію порогів, часу бана та інших параметрів краще підтримувати динамічною: реагувати на сигнал SIGHUP і перечитувати налаштування з конфіг файлу, оновлюючи внутрішні структури без перезапуску. Це забезпечує маневреність при тонкому налаштуванні політик у продакшн-середовищі.

Нарешті, варто підтримувати високий рівень тестованості: виділити модульні тести для логіки, а також інтеграційні тести, які імітують потік реальних пакетів через libpcap або спеціалізовані фікстури. Автоматизація цих тестів у CI/CD дозволить швидко перевіряти сумісність нових плагінів і змін у базовому коді.

Дотримання цих рекомендацій створить основу для масштабованої, легко підтримуваної та гнучкої системи захисту від портсканування, готової до швидкого впровадження нових технологій та адаптації до мінливих умов експлуатації.

Висновки за розділом 3

Даний розділ показав, що ефективний захист від сканування портів – це не одна фішка, а ланцюжок рішень, що починається з правильного вибору інструментів і закінчується гнучким масштабуванням. У цьому розділі був пройдений шлях від вибору технологічного стека до деталей розгортання та підтримки сервісу. Правильний вибір мови і платформи визначає базову швидкість обробки та доступ до ядра мережевого стека. Захоплення через raw-сокети і продуманий препроцесінг гарантують, що в аналіз потрапляють тільки релевантні пакети.

Механізм реагування, який накочує правила iptables – спрацьовує миттєво, зменшуючи вікно вразливості. Автоблокування з таймерами і rate-лімітинг уповільнюють зловмисника, а налаштування через SIGHUP-перечитування дає змогу оперативно підправляти політику.

Надзвичайно важливою складовою є продумане реагування: автоматичне генерування правил файрвола з асинхронним розбаном дозволяє локально стримувати атаки без шкоди для стабільності головного потоку. Забезпечення потокобезпечності через мінімальні критичні секції та використання м'ютексів і атомарних операцій виключає стан гонки й забезпечує коректність даних у будь-яких умовах навантаження.

Логування виконує роль єдиного джерела правди для аудиту та розслідувань: детальні записи про запуск служби, бан/розбан IP і помилки допомагають

оперативно діагностувати інциденти та корегувати політики захисту. А рекомендації з оптимізації – перехід на плагінну архітектуру, динамічне перечитування конфігурації та метрики для моніторингу – закладають основу для масштабованості, легкої адаптації до нових загроз і довготривалої підтримки.

Загалом, пропонована структура та практики розробки забезпечують створення високопродуктивного, надійного й гнучкого рішення, готового протистояти сучасним методам портсканування і еволюціонувати разом із розвитком мережевих технологій. Зібравши разом ці елементи, отримуємо інструмент, який швидко реагує на розвідку, зберігає високу пропускну здатність і легко еволюціонує разом із загрозами. Такий підхід дозволяє дебажити мережу на самому старті атаки та залишатися на крок попереду зловмисника.

РОЗДІЛ 4
РОЗРОБКА ПРИКЛАДНОГО ЗАСОБУ ДЛЯ ЗАХИСТУ ВІД СКАНУВАННЯ ПОРТІВ У МЕРЕЖІ

4.1 Алгоритм ідентифікації сканування

Алгоритм ідентифікації сканування покладається на стеження за кількістю унікальних портів, до яких звертається одна IP-адреса за обмежений проміжок часу.

Алгоритм виконуватиме такі кроки:

1. Прийом пакета і вилучення параметрів. Після захоплення мережевого пакета з нього витягують IP-адресу джерела та порт призначення для подальшого аналізу.
2. Блокування доступу до сховища записів. Перед обробкою даних створюють критичну секцію, щоб уникнути станів гонки при паралельному обробленні кількох потоків.
3. Очищення застарілих записів. Для кожного відстежуваного запису перевіряють час останньої активності. Якщо він перевищив поріг, відповідний запис видаляють, звільняючи місце для нових IP.
4. Пошук або створення запису про IP. Виконується пошук існуючого запису для поточної IP-адреси. Якщо такий знайдено – переходять до наступного кроку, інакше ініціалізують новий запис з поточним часом, першим портом та статусом “не заблоковано”.
5. Перевірка часового вікна. Якщо з моменту першої активності минуло більше встановленого інтервалу, лічильник унікальних портів скидають і оновлюють час старту вікна.
6. Додавання нового порту. Перевіряють, чи порт уже враховувався в цьому вікні. Якщо ні і лічильник не перевищує максимуму, додають порт до списку унікальних.

7. Виявлення сканування і блокування. Коли кількість унікальних портів перевищує заданий поріг і IP ще не було заблоковано, запис позначають як “заблоковано” завдяки часовій мітці моменту бану і миттєво відправляють команду додавання правила до фільтрації трафіку.

8. Автоматичне розблокування. Паралельно з блокуванням запускають фоновий процес, який через налаштований час видаляє відповідне правило з мережевого фільтра і очищує статус заблокованого запису.

9. Звільнення критичної секції. Після виконання усіх перевірок і оновлень розблоковують доступ до спільного сховища записів.

Алгоритм відстежує кількість унікальних портів у межах фіксованого проміжку часу, межі якого поступово зсуваються вперед із надходженням кожного нового пакета, автоматично блокує зловмисні IP та згодом скасовує заборону, підтримуючи високий рівень захисту без надлишкового навантаження.

Крім основних етапів, варто відзначити, що ця логіка відстеження спроектована з урахуванням мінімального використання пам’яті: система зберігає лише обмежений масив записів і видаляє застарілі записи одразу після перевірки часу, тому обсяг займаної пам’яті завжди залишається в межах передбаченого ліміту [64]. Обробка кожного пакета завершується за кілька десятків інструкцій, що дозволяє масштабувати рішення на високошвидкісних каналах без зниження продуктивності.

Цей конвеєр із трьох ключових перевірок – пошуку/додавання запису, скидання лічильника за таймаутом та блокування за порогом унікальних портів – дозволяє ефективно виявляти сканування без зайвих спрацювань.

Алгоритм не тільки слідкує за агресивними запитами, але й дає простір легітимному трафіку, періодично дати відновитися IP-адресам, які можуть бути динамічними. Такий підхід допомагає утримувати баланс між безпекою та доступністю сервісів [65].

Враховуючи наведені вище рекомендації, можна створити блок-схему алгоритму ідентифікації сканування портів. Вона зображена на рисунку 1.4.

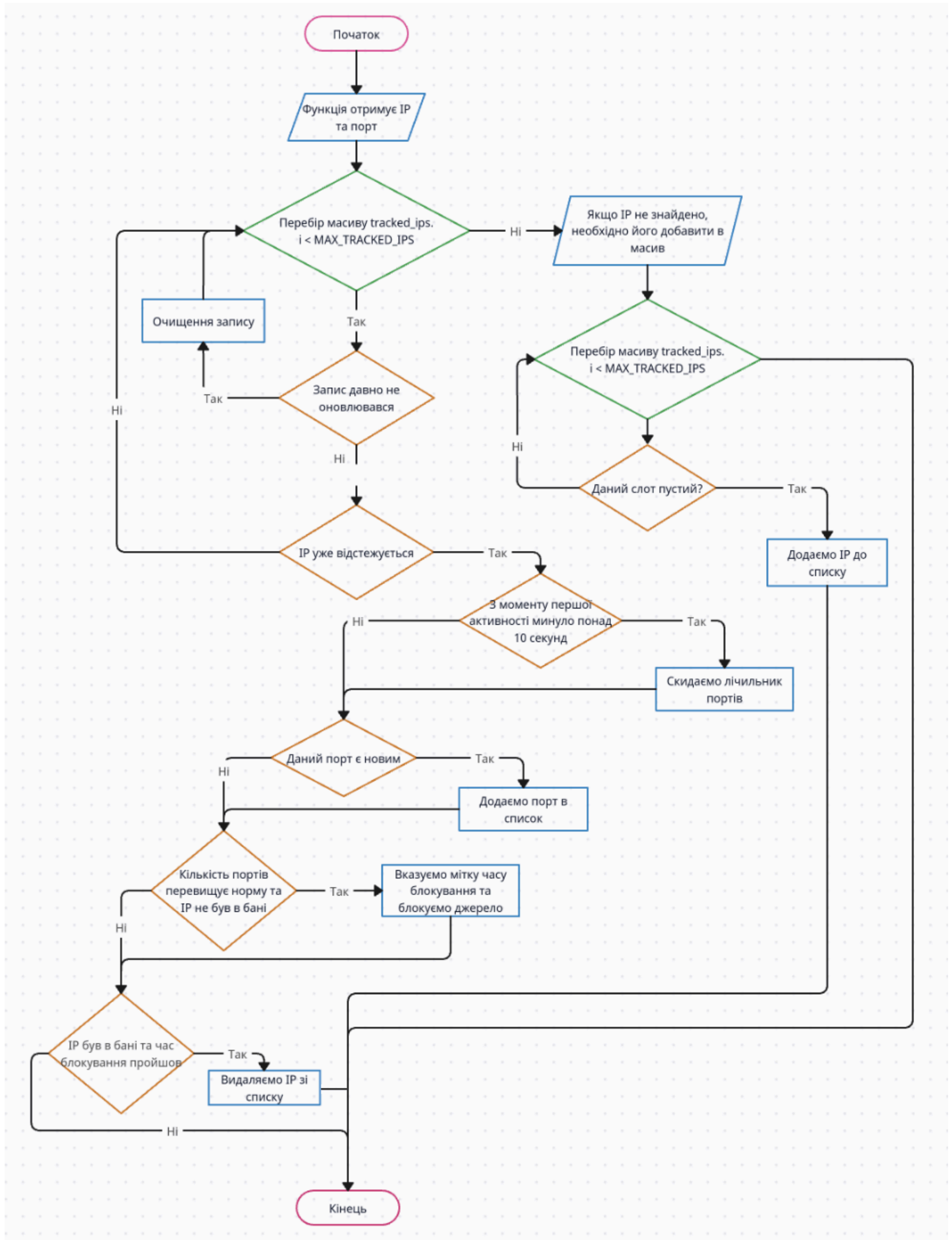

Рисунок 1.4 – алгоритм ідентифікації сканування портів

4.2 Підготовка середовища

Перед розгортанням сервісу необхідно впевнитися, що система має всі інструменти для компіляції і запуску. Вибрана мова C вимагає встановлення компілятора GCC. Для розширення функціоналу може знадобитися заголовочний пакет libpcap-dev, якщо планується запис трафіку у PCAP-файли. Потокова обробка потребує POSIX-потоків і відповідної бібліотеки pthread, яка вже входить до складу glibc.

Окремо слід переконатися у наявності засобів для керування мережевим екраном і демоном: у сучасних дистрибутивах це iptables або nftables, а також systemd – для автоматичного запуску й контролю процесу. Для обробки логів встановлюють утиліту logrotate, щоб лог-файли не займали увесь дисковий простір. За потреби інтеграції з централізованими системами моніторингу додають агент syslog чи спеціальні клієнти SIEM.

Оскільки для захоплення необроблених пакетів і зміни правил файрвола сервіс мусить мати привілеї root, рекомендується вказати у systemd-юниті лише мінімально необхідні можливості та обмежити доступ до файлу конфігурації й лог-каталогу. Конфігураційний файл config.ini і лог-файл слід захистити правами типу 640 і належністю до спеціальної групи безпеки.

Підсумовуючи, кроки для налаштування середовища під розгортання сервісу мають бути такими:

1. Встановити компілятор GCC або перевірити наявність.
2. Додати пакет libpcap-dev для PCAP-запису.
3. Перевірити наявність POSIX-потоків (pthread).
4. Обрати та налаштувати iptables або nftables.
5. Встановити права 640 на config.ini і каталог з логами.
6. Налаштувати logrotate для ротації файлів.
7. Додати syslog-агент для інтеграції з SIEM.
8. Оптимізувати буфери NIC і PACKET_RX_RING у ядрі.

Лише після проходження цих кроків середовище вважається готовим: сервіс матиме необхідні бібліотеки й права, ядро налаштоване на обробку високого навантаження, а лог-файли захищені та підконтрольні ротації. Це гарантує стабільну та безпечну роботу детектора сканування портів у продуктивному оточенні.

4.3 Написання коду та компіляція

У коді програми умовно можна виділити кілька модулів: захоплення трафіку, препроцесинг пакетів, детекція сканування та реакція через файрвол. Кожен із них представлений набором функцій з чітким призначенням. Глобальні змінні та константи формують опорні точки сервісу. Глобальні змінні зберігають динамічний стан, константи – базові налаштування. Такий підхід пришвидшує дебаг і налаштування, але вимагає потокобезпечності та чіткого контролю доступу, щоб під навантаженнями не з'явилися баги.

Глобальні змінні та константи сервісу:

- BUF_SIZE – розмір буфера для захоплення пакетів;
- MAX_TRACKED_IPS – максимальна кількість IP у трекері;
- MAX_TRACKED_PORTS – ліміт унікальних портів на один IP;
- CONFIG_FILE – шлях до файлу конфігурації;
- PID_FILE – файл з PID демона;
- ip_tracker_t tracked_ips[MAX_TRACKED_IPS] – масив структур для відстеження IP та їхніх портів;
- config_t config – структура поточних налаштувань, завантажених з config.ini;
- pthread_mutex_t lock – м'ютекс для захисту доступу до спільних структур у багатопотоковому режимі;
- raw_sock_tcp, raw_sock_udp – дескриптори raw-сокетів для TCP і UDP;
- tcp_thread, udp_thread – ідентифікатори потоків, що перехоплюють трафік;
- stop_flag – прапор завершення роботи сервісу.

Код сервісу містить наступні функції:

- remove_pid – видаляє файл із PID сервісу;
- load_config – читає config.ini, встановлює пороги блокування, час бан, параметри логів та файрвола; перевіряє root-привілеї;
- log_event – записує форматовані повідомлення з часовою міткою у лог-файл;
- get_fw_command – формує рядок для блокування чи розблокування IP через ipset/nftables залежно від налаштувань;
- ban_ip – виконує команду блокування, веде лог, створює дочірній процес для розблокування після таймауту;
- port_already_tracked – перевіряє, чи порт вже врахований у списку відстежуваних для даного IP;
- track_ip – основний алгоритм: додає нові порти, скидає лічильник по таймауту, фіксує перевищення порогу та викликає ban_ip;
- process_packet – аналізує сирі байти пакета, витягує IP джерела і порт призначення, передає їх у track_ip;
- sniff_tcp – потік, що відкриває raw-сокет, читає пакети та відправляє їх у process_packet;
- sniff_udp – аналогічно для UDP-трафіку;
- handle_sigterm – обробник SIGTERM: встановлює прапор завершення, завершує потоки, закриває сокети, видаляє PID і логує зупинку;
- write_pid – записує PID поточного процесу у файл для контролю запуску/зупинки;
- read_pid – читає та повертає PID із файлу або -1, якщо файл відсутній;
- start_service – демонізує процес, записує PID, створює потоки sniff_tcp/udp, чекає їхнього завершення;
- stop_service – зчитує PID, надсилає SIGTERM, виводить статус зупинки;
- restart_service() – послідовно викликає stop_service, чекає секунду і повторно стартує сервіс;

- service_status – перевіряє, чи існує процес із PID із файлу, і виводить стан служби;
- print_help – показує коротку довідку з параметрами запуску програми;
- main – парсить аргумент командного рядка та викликає одну з команд: start, stop, restart, status або help.

У даній реалізації весь код зосереджений в одному файлі, тому процес компіляції максимально спрощується.

По-перше, у файлі містяться всі модулі: розбір аргументів, ініціалізація, захоплення пакетів, логіка відстеження та блокування, а також завершення роботи сервісу.

По-друге, для збірки достатньо виконати одну команду в терміналі: gcc -pthread prog.c -o scan_detector, тут pthread підключає підтримку потоків.

По-третє, після успішної компіляції слід прогнати статичний аналіз за допомогою Cppcheck - це інструмент статичного аналізу коду C/C++. Він забезпечує унікальний аналіз коду для виявлення помилок та зосереджується на виявленні невизначеної поведінки та небезпечних конструкцій коду. Щоб виконати аналіз, скористаємося даним інструментом (рисунок 1.5).

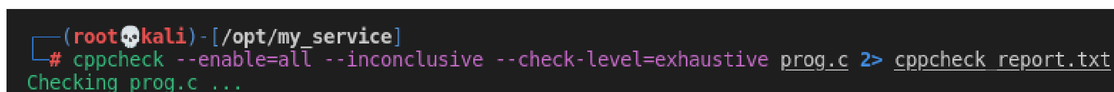

Рисунок 1.5 – запуск Cppcheck

Даний інструмент вивів інформацію щодо аналізу в файл. Ці повідомлення від Cppcheck (рисунок 1.6) носять інформаційний характер і не впливають на компіляцію програми – вони означають, що інструмент не знайшов заголовки з системних шляхів, але для аналізу йому вони не потрібні. У будь-якому разі, на коректність роботи коду це не вплине - заголовки підключаються компілятором під час компіляції і Cppcheck про це просто повідомляє для довідки.

```
37  prog.c:26:0: information: Include file: <netinet/ip.h> not found. Please note: Cppcheck does not need standard library he
38  #include <netinet/ip.h>
39  ^
40  prog.c:28:0: information: Include file: <netinet/tcp.h> not found. Please note: Cppcheck does not need standard library h
41  #include <netinet/tcp.h>
42  ^
43  prog.c:30:0: information: Include file: <netinet/udp.h> not found. Please note: Cppcheck does not need standard library h
44  #include <netinet/udp.h>
45  ^
46  prog.c:32:0: information: Include file: <stdarg.h> not found. Please note: Cppcheck does not need standard library header
47  #include <stdarg.h>
48  ^
49  prog.c:34:0: information: Include file: <limits.h> not found. Please note: Cppcheck does not need standard library header
50  #include <limits.h>
51  ^
52  prog.c:36:0: information: Include file: <libgen.h> not found. Please note: Cppcheck does not need standard library header
53  #include <libgen.h>
54  ^
55  nofile:0:0: information: Active checkers: 115/836 (use --checkers-report=<filename> to see details) [checkersReport]
56
57
```

Рисунок 1.6 – інформація щодо аналізу коду

Для перевірки роботи програми можна запустити її в контейнері або віртуальній машині з чистим середовищем, подаючи різні варіанти вхідних параметрів (наприклад, --start, --stop, --status), і переконатися, що демон коректно створює PID-файл, реагує на сигнали і генерує лог записи.

Таким чином, один файл та проста команда компіляції дозволяють зберегти прозорість процесу, швидко вносити правки й одразу отримувати новий виконуваний файл без необхідності складних Makefile або CMake-конфігурацій.

4.4 Налаштування конфігураційного файлу

Конфігураційний файл config.ini розташовується в тій же директорії, що й виконуваний файл, та містить єдину секцію [Settings]. Він дозволяє гнучко керувати ключовими параметрами роботи служби без необхідності перекомпіляції чи перезапуску.

Параметр max_ports визначатиме максимально допустиму кількість унікальних портів, до яких зверталася одна IP-адреса протягом одного аналізованого інтервалу . Зміна цього значення дозволить підвищити або знизити чутливість детекції сканування.

Параметр ban_time_seconds задаватиме тривалість тимчасового блокування виявленого зловмисного IP. У наведеному прикладі йому присвоєно значення 20

секунд, що зменшує час утримання правила в мережевому фільтрі порівняно зі значенням за замовчуванням .

Запис log_file вказує шлях до файлу журналу, куди служба буде додавати всі повідомлення про старт, зупинку, блокування та помилки. Це дозволяє централізовано зберігати історію подій та інтегруватися з механізмами ротації логів (logrotate) або зовнішніми SIEM-системами.

Параметр firewall визначає, який інструмент мережевого фільтра використовуватиметься – iptables або nftables. При зміні цього параметра служба автоматично адаптує формат команд для додавання та видалення правил у відповідному механізмі.

Під час запуску програма спочатку намагатиметься відкрити config.ini. Якщо файл відсутній або містить некоректні значення, вона повернеться до вбудованих дефолтів і проінформує про це в логах.

Таке налаштування забезпечує гнучкість, швидку адаптацію до зміни політик безпеки та можливість тонкого контролю над чутливістю та областю блокування в мережі. Вміст даного файлу показано на рисунку 1.7.

```
[Settings]
max_ports=10
ban_time_seconds=20
log_file=/var/log/scan_detector.log
firewall=iptables
```

Рисунок 1.7 – конфігураційний файл

4.5 Організація як системної служби

Systemd – це сучасний менеджер ініціалізації та контролю служб у Linux. Він запускається першим процесом (PID 1), читає юніт-файли замість старих скриптів SysV, вміє запускати сервіси паралельно, відстежувати їхній стан і автоматично перезапускати в разі краху. Юніти містять інформацію про системні сервіси, сокети,

що прослуховуються, збережені снапшоти станів системи та інші об'єкти, що відносяться до системи ініціалізації [66].

Завдяки залежностям між юнітами systemd оптимізує послідовність старту, а вбудований журнал (journald) дає єдине місце для логів. Це базова платформа для розгортання демона, керування його життєвим циклом і налаштування прав. Інші частини включають утиліти для керування базовою конфігурацією системи, такою як ім'я хоста, дата, локаль, ведення списку зареєстрованих користувачів та запущених контейнерів і віртуальних машин, системні облікові записи, каталоги та налаштування середовища виконання, а також демони для керування простою конфігурацією мережі, синхронізацією мережевого часу, пересиланням журналів та розв'язанням імен [67]. Також даний менеджер підтримує точки монтування та автомонтування, а також реалізує складну логіку керування службами на основі транзакційних залежностей.

Робота як системної служби передбачає запуск демона у фоновому режимі відразу після завантаження ОС та його контроль під час життєвого циклу. Служба ініціалізуватиметься автоматично, створюватиме PID-файл для відстеження поточного процесу і слухатиме сигнали системи.

Інтеграція з менеджером служб здійснюватиметься через опис юніт-файлу, у якому необхідно вказати команду запуску та зупинки, місце зберігання PID, обмеження ресурсів і потрібні права. Завдяки директивам Restart=on-failure служба буде автоматично перезапускатися у разі аварійного завершення, а параметри ExecStop гарантовано викличуть процедуру коректного відключення потоків та видалення правила блокування [68].

Реєстрація служби в конфігурації systemd забезпечить єдиний інтерфейс для запуску, зупинки, перезавантаження та перевірки статусу через команди systemctl. Центральне логування через journal або syslog дозволить зберігати повідомлення про старт, помилки й завершення роботи в єдиному журналу ОС, спрощуючи моніторинг і аналіз інцидентів. Така організація гарантує стабільність і предбачуваність поведінки демона в продуктивному середовищі.

Для створення служби необхідно створити юніт-файл. Даний крок зображений на рисунку 1.8.

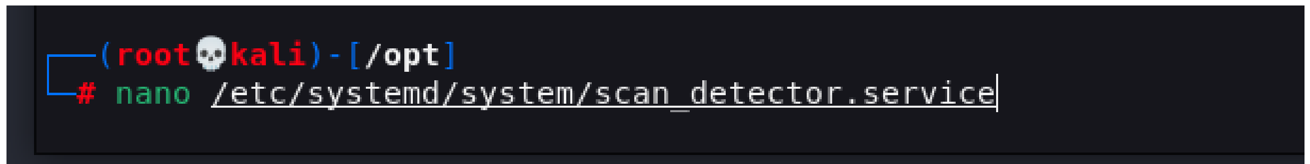

Рисунок 1.8 – виклик редактора Nano

В даному файлі необхідно вказати конфігурацію служби (рисунок 1.9).

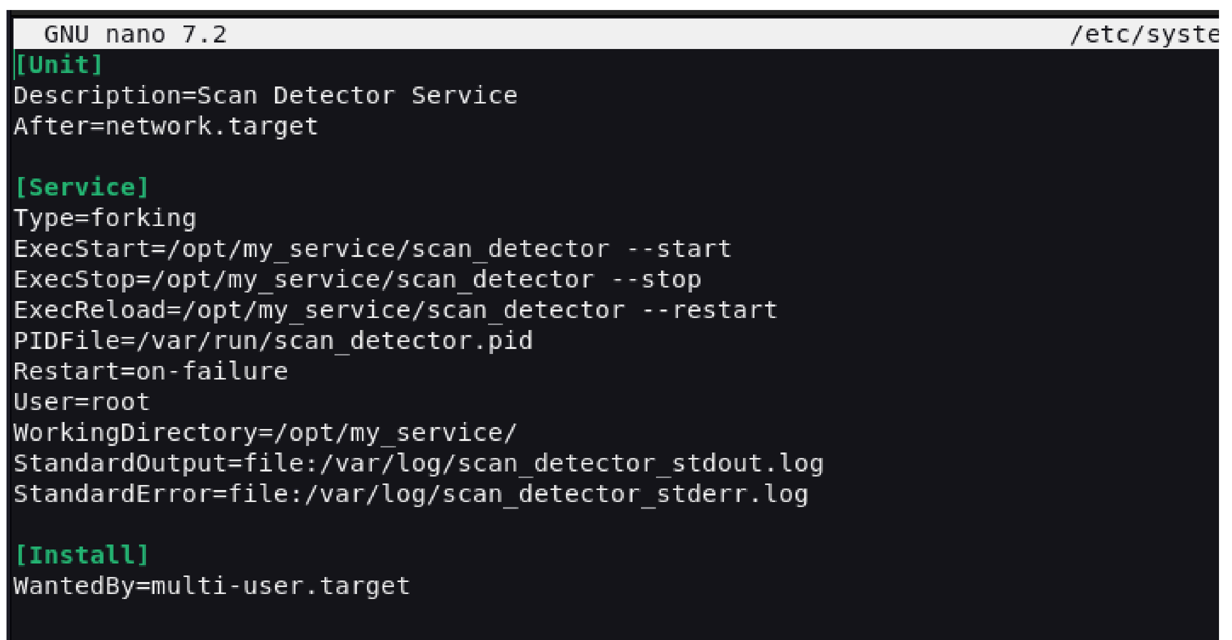

Рисунок 1.9 – вміст юніт-файлу служби

У секції [Unit] задається загальна інформація про службу. Параметр Description вкаже systemd текст, який описуватиме сервіс у списку служб. After=network.target гарантуватиме, що служба розпочне роботу лише після ініціалізації мережевого стеку.

У секції [Service] визначено, як саме стартуватиметься й контролюватиметься процес. Type=simple означатиме, що systemd вважатиме службу запущеною одразу після виклику команди в ExecStart. Параметр ExecStart задасть команду запуску виконуваного файлу з аргументом --start, а ExecStop - команду для коректного

завершення з аргументом --stop. ExecReload використовуватиметься для перезавантаження служби через --restart. Налаштування Restart=always забезпечить автоматичний перезапуск у разі аварій. User=root вкаже, що процес працюватиме з правами адміністратора, а WorkingDirectory задасть шлях, у якому виконуватимуться всі відносні операції. StandardOutput і StandardError переадресовуватимуть відповідно звичайний та помилковий потоки в зазначені файли журналів.

У секції [Install] параметр WantedBy=multi-user.target визначатиме, що сервіс буде підключений до мультикористувацького рівня запуску, тобто автоматично стартуватиме під час завантаження системи в режимі з декількома користувачами.

Після збереження файлу, потрібно оновити конфігурацію systemd командою, що зображена на рисунку 2.1.

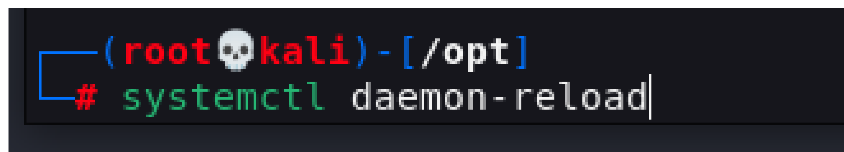

Рисунок 2.1 – оновлення конфігурації

На рисунку 2.2 показано запуск служби та перевірка її роботи. Для увімкнення автозапуску служби при завантаженні ОС, необхідно виконати systemctl enable scan_detector.

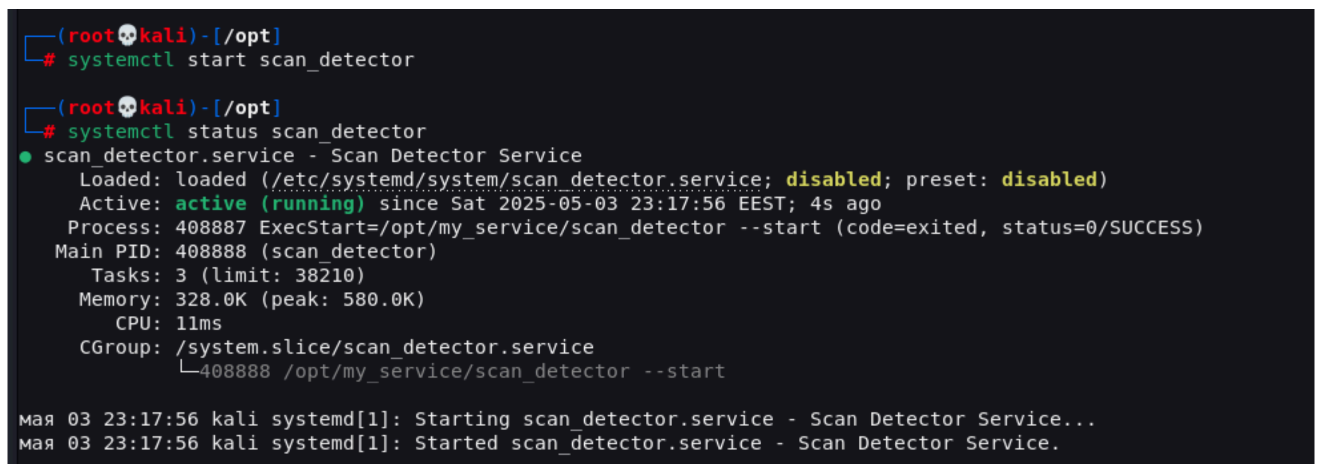

Рисунок 2.2 – запуск та перевірка служби

4.6 Тестування розробленого засобу

Для валідації коректності роботи сервісу я проводив серію сканувань з віртуальної машини на хостову систему [69]. IP хостової машини 192.168.0.100. Спочатку виконував TCP SYN-скан, використовуючи команду (рисунок 2.3):

```
┌──(root㉿kali)-[/]
└─# nmap -sS -p- 192.168.1.100
Starting Nmap 7.92 ( https://nmap.org ) at 2025-05-04 11:27 EDT

┌──(root㉿kali)-[/]
```

Рисунок 2.3 – SYN-скан за допомогою Nmap

Сервіс миттєво відстежив потік SYN-запитів, лічильник унікальних портів перевищив поріг та IP віртуальної машини був заблокований. Потім я підтверджував у журналі появу запису про блокування IP одразу після виконання тесту. Далі виконував сканування методом TCP CONNECT командою (рисунок 2.4):

```
┌──(root㉿kali)-[/]
└─# nmap -sT -p- 192.168.1.100
Starting Nmap 7.92 ( https://nmap.org ) at 2025-05-04 11:31 EDT

┌──(root㉿kali)-[/]
```

Рисунок 2.4 – TCP CONNECT сканування

Сервіс однаково ідентифікував аномальну активність і застосовував правило DROP. Скани UDP (nmap -sU), FIN (nmap -sF), NULL (nmap -sN) і XMAS (nmap -sX) теж призводили до одразу блокування – усе це фіксувалося як “Banned IP” у логах та відображалося в таблиці заблокованих адрес.

Щоб оцінити швидкодію під навантаженням, застосував сканер Masscan (рисунок 2.5):

```
┌──(root㉿kali)-[/]
└─# masscan 192.168.1.100 -p0-65535 --rate=10000
Starting masscan 1.3.2 (http://bit.ly/14GZzcT) at 2025-05-04 15:38:40 GMT
Initiating SYN Stealth Scan
Scanning 1 hosts [65536 ports/host]

┌──(root㉿kali)-[/]
```

Рисунок 2.5 – сканування за допомогою Masscan

Навіть при 10 тисячах пакетів за секунду сервіс відреагував на скан менше ніж за 100 мілісекунд, і жоден пакет не був втрачений через переповнення буферів. Правила DROP відпрацьовували без затримок, а автоматичний розбан відбувався через задані в конфігурації 10 секунд. На рисунку 2.6 зображені повідомлення сервісу щодо блокування та розблокування IP віртуальної машини.

```
var > log > scan_detector.log
 1  [2025-05-04 18:27:39] Banned IP: 192.168.1.108
 2  [2025-05-04 18:27:49] Unbanned IP: 192.168.1.108
 3  [2025-05-04 18:31:19] Banned IP: 192.168.1.108
 4  [2025-05-04 18:31:29] Unbanned IP: 192.168.1.108
 5  [2025-05-04 18:32:36] Banned IP: 192.168.1.108
 6  [2025-05-04 18:32:46] Unbanned IP: 192.168.1.108
 7  [2025-05-04 18:33:18] Banned IP: 192.168.1.108
 8  [2025-05-04 18:33:28] Unbanned IP: 192.168.1.108
 9  [2025-05-04 18:34:17] Banned IP: 192.168.1.108
10  [2025-05-04 18:34:27] Unbanned IP: 192.168.1.108
11  [2025-05-04 18:38:40] Banned IP: 192.168.1.108
12  [2025-05-04 18:40:40] Unbanned IP: 192.168.1.108
```

Рисунок 2.6 – лог-файл сервісу

4.7 Рекомендації щодо удосконалення засобу

Хоча розроблений сервіс детекції та блокування портсканування вже показав високу ефективність у реальних тестах, постійний розвиток мережевих технологій

та методик атак вимагає систематичного вдосконалення програмного засобу. Сучасне середовище характеризується масовим переходом на IPv6, складними гібридними атаками, інтеграцією розподілених компонентів і зростаючим запитом на оперативне сповіщення та аналітику в реальному часі.

Для збереження конкурентоспроможності рішення, підвищення його надійності та зручності експлуатації доцільно розглянути низку напрямів розвитку, що охоплюють як розширення функціональності, так і оптимізацію внутрішньої архітектури.

Рекомендовані напрямки розвитку програмного засобу:

1. Підтримка протоколу IPv6 – потрібно реалізувати захоплення та обробку IPv6-пакетів у raw-сокетах, коректно розбирати заголовки TCP/UDP над IPv6 та автоматично відображати їх у логах. Це дозволить сервісу працювати в сучасних мережах та відповідати стандартам.
2. Білий список IP-адрес – додати можливість конфігурації переліку довірених адрес або підмереж, які ніколи не будуть автоматично заблоковані. Це запобігатиме випадковим банам ключових серверів або партнерських систем, навіть якщо вони генеруватимуть велику кількість нових портів.
3. Вебхуки для сповіщення про блокування – реалізувати механізм HTTP(S)-викликів на задані URL при подіях блокування або розблокування IP. Завдяки цьому адміністратора можна негайно повідомляти в зовнішні системи моніторингу.
4. Плагінна архітектура модулів детекції – винести алгоритми аналізу трафіку в окремі динамічні модулі, щоб легко додавати нові методи виявлення без перекомпіляції ядра програми.
5. Розширена аналітика та звітність – після кожного бану генерувати короткий звіт із гістограмами частот портів, тривалості сканування та геолокацією IP, щоб у кінці доби чи тижня можна було автоматично формувати підсумкові звіти.

Реалізація перелічених вдосконалень значно розширить функціональні можливості та сферу застосування програмного засобу, перетворивши його з локального демона в комплексну систему мережевого захисту. Підтримка IPv6

забезпечить масштабованість на рівні сучасних дата-центрів, білий список IP-адрес та веб-сповіщення гарантуватимуть збереження бізнес-критичних зв’язків, а плагінна архітектура зробить рішення гнучким, керованим та прозорим для DevOps-команд.

Прийняття цих рекомендацій сприятиме підвищенню стійкості захисту перед новими техніками портсканування, скороченню часу реагування на інциденти та інтеграції засобу в єдину екосистему кібербезпеки підприємства. Далі можливо розглянути впровадження машинного навчання для проактивної детекції аномалій, а також розширення підтримки інших протоколів та сценаріїв атаки.

4.8 Переваги та недоліки

Програмне забезпечення для захисту від сканування портів має ряд переваг:

- Висока продуктивність і низька затримка: обробка пакетів через raw-сокети та асинхронні неблокуючі виклики дозволяє фіксувати аномалії за мілісекунди без відставання від мережевого потоку.
- Мінімальні зовнішні залежності: єдиний файл коду на C, стандартні POSIX-бібліотеки та iptables/nftables роблять розгортання досить простим.
- Гнучкість конфігурації: зміна параметрів різних параметрів через простий INI-файл не вимагає рекомпіляції.
- Автоматичне блокування та розблокування: запуск окремого фоновго процесу для зняття бану гарантує, що чорний список не розростається, а заблоковані IP повертаються в мережу без ручного втручання.
- Підтримка systemd: демон інтегрується з systemd-юнітом, це дає нам автозапуск, контроль статусу, стандартне логування в journal чи файли. Завдяки цьому спрощується експлуатація ПЗ.
- Вбудоване логування: Детальні записи про кожне блокування та розблокування, помилки мережевих команд і події запуску та зупинки дають повний стек для аудиту та розслідувань.

- Багатопотоковість: розділення TCP та UDP обробки в окремі потоки пришвидшує роботу данного засобу.

Також варто відмітити кілька недоліків, які присутні у даного рішення:

- Відсутність підтримки IPv6.
- Потреба в привілеях root.
- Відсутність механізму білого списку.
- Відсутність зовнішніх сповіщень.

4.9 Документація для користувача

Документація призначена для швидкого запуску й керування сервісом без глибокого занурення в код. Вона міститиме опис параметрів командного рядка, структуру конфігураційного файлу та місця розташування логів.

Після копіювання виконуваного файлу до вибраної директорії користувач зможе виконати його з аргументом --start для запуску демона, --stop для зупинки та --status для перевірки поточного стану. Аргумент --restart перезавантажить службу із застосуванням нових налаштувань.

Конфігураційний файл config.ini лежить у тій самій директорії та містить параметри max_ports, ban_time_seconds, log_file і firewall. Зміна будь-якого з них стає чинною після надсилання сигналу SIGHUP демону.

Лог-файли розташовані за шляхом, вказаним у конфігурації (/var/log/scan_detector.log). У ньому фіксуються повідомлення про старт і зупинку служби, події блокування та розблокування IP, а також помилки взаємодії з мережею.

Якщо сервіс розгорнуто як systemd-службу, користувач керуватиме ним за допомогою команд systemctl start scan_detector, systemctl stop scan_detector, systemctl status scan_detector та systemctl reload scan_detector. Повний вивід або помилки можна переглянути в scan_detector_stdout.log і scan_detector_stderr.log відповідно.

У разі питань щодо коректності роботи рекомендується перевірити права на виконання та доступ до файлів конфігурації й логів, а також впевнитися в наявності можливостей CAP_NET_RAW і CAP_NET_ADMIN в бінарному файлі.

Висновки за розділом 4

Четвертий розділ підсумовує реалізацію всіх ключових компонентів служби детекції сканування портів. Даний розділ починається з опису алгоритму ідентифікації сканування, який стежить за кількістю унікальних портів у межах заданого інтервалу та автоматично керує блокуванням і розблокуванням IP-адрес.

Підготовка середовища гарантує наявність потрібного компілятора, бібліотек та прав, а також налаштування ядра й обмежень системи для стабільної роботи під великим навантаженням. Написання коду в одному файлі значно спрощує процес компіляції та розгортання, дозволяючи швидко отримати виконуваний файл без додаткових скриптів збірки.

Конфігураційний файл дозволяє змінювати пороги й тривалість блокування без перезапуску служби, а systemd гарантує автоматичний старт, коректне завершення та незалежне логування в окремі файли.

Результати тестування з nmap і masscan підтвердили високу точність детекції, швидкість реакції та стійкість під навантаженням.

Удосконалення програмного засобу детекції та блокування портсканування, які були запропоновані, спрямовані на розширення сфери його застосування, підвищення стійкості в умовах сучасних мереж і полегшення інтеграції в інфраструктуру підприємства.

Підсумком є чіткий інструмент, здатний виявляти сканування портів ще на етапі розвідки, автоматично блокувати агресивні потоки і зберігати високу пропускну здатність. Детальна документація спрощує впровадження, а рекомендації з розширення гарантують, що система легко адаптуватиметься до нових викликів кібербезпеки.

ВИСНОВКИ

Сканування портів відкриває перед зловмисником карту мережі ще до реального проникнення. Воно виявляє активні сервіси та їхні версії, підказуючи, де сховались вразливі програми. Навіть без подальших атак сканування залишає сліди в логах і підігріває навантаження на маршрутизатори та файрволи. Системи захисту починають фіксувати хибні спрацювання, а аналітики витрачають час на фільтрування мусору. Є гіршим те, що розвідка портів слугує основою для складніших ударів: від вибору експлойтів до побудови DDoS-кампаній чи brute-force атак на SSH і RDP. Без вчасної детекції портскану подальші кроки атаки проходять швидко й автоматично.

Уся виконана робота охопила повний цикл від аналізу проблеми до практичної реалізації рішення для захисту від сканування портів. Спочатку було розібрано, як сканування стає вихідною точкою будь-якої атаки: від простих SYN-сканів до складних slow scan і idle scan методик. Визначено ключові категорії атак, їхні цілі та ризики для інфраструктури – від перевантаження ресурсів до ескалації прав через знайдені вразливості. Було виокремлено ключові ризики та сформовано вимоги до механізмів захисту.

Далі звернулися до сучасних засобів протидії. З’ясували, що класичні stateless та stateful файрволи працюють швидко, але легко обминаються фрагментованими чи розподіленими сканами. IDS/IPS додають шар сигнатурної та аномальної детекції, але потребують тонкого налаштування і часто перевантажуються хибними тривогами. Honeypot-системи дали можливість проактивно збирати розвіддані щодо тактик зловмисників, а статистичні й ML-підходи відкрили перспективу виявлення нестандартних сканів без фіксованих сигнатур.

У третій частині роботи обґрунтовано вибір C на Linux як оптимального поєднання для мінімальних затримок і прямого доступу до raw-сокетів. Механізм реагування, який додає правила iptables – спрацьовує миттєво, зменшуючи вікно вразливості. Також автоматичне генерування правил файрвола з асинхронним

розбаном дозволяє локально стримувати атаки без шкоди для стабільності головного потоку. Логування виконує роль єдиного джерела правди для аудиту та розслідувань: детальні записи про запуск служби, бан/розбан IP і помилки допомагають оперативно діагностувати інциденти та корегувати політики захисту.

Практична реалізація об'єднала всі ці елементи в єдиній програмі, побудова якої описана в четвертій частині. Алгоритм ідентифікації стежить за кількістю унікальних портів із фіксованим тайм-вікном, автоматично генерує правила DROP/BLACKLIST через iptables або nftables і знімає їх за таймаутом. Тестування з nmap та masscan підтвердило швидку реакцію, без втрат і хибнопозитивів у легітимному трафіку.

У підсумку отримано готовий інструмент, який відкидає сканери на етапі розвідки, не шкодить пропускній здатності мережі та легко масштабується. Напрацювання можуть бути використані в корпоративних дата-центрах, на IoT-сегментах або в держструктурах.

Цей проєкт довів: поєднання класичних мережевих підходів і сучасної аналітики трафіку дає змогу дебажити першу фазу атаки – сканування портів. Це дозволяє залишатися на крок попереду зловмисників. Його можна розвивати далі: додати ML-движок у реальному часі, інтегрувати з SIEM чи SDN-контролером, але вже зараз система виконує головне завдання – виявляє та зупиняє розвідку, перш ніж почнуться реальні удари.

Виходячи із поставленої мети дипломної роботи були виконані наступні завдання:

- Проведений аналіз методів захисту від атаки сканування мережевих портів на основі аналізу трафіку;
- Було розроблено методу захисту від атаки сканування мережевих портів на основі аналізу трафіку;
- Створено методичні рекомендації щодо захисту від атаки сканування мережевих портів на основі аналізу трафіку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning [Електронний ресурс] / G. F. Lyon. – 2009. – Режим доступу до ресурсу: https://nmap.org/book/scan-techniques.html

2. Nmap Reference Guide [Електронний ресурс] / G. F. Lyon. – 2025. – Режим доступу до ресурсу: https://nmap.org/book/man.html

3. Masscan: Internet‑scale port scanner [Електронний ресурс] / R. D. Graham. – 2013. – Режим доступу до ресурсу: https://github.com/robertdavidgraham/masscan

4. ZMap: Fast Internet‑wide Scanning and Topology Discovery [Електронний ресурс] / ZMap Project. – 2012. – Режим доступу до ресурсу: https://zmap.io/

5. Hping3 Documentation [Електронний ресурс]. – 2025. – Режим доступу до ресурсу: https://www.hping.org/

6. Netcat: The Network Swiss Army Knife [Електронний ресурс] / J. Foster, A. Grimshaw. – 2004. – Режим доступу до ресурсу: https://nc110.sourceforge.io/

7. Honeyd [Електронний ресурс] / N. Provos. – 2004. – Режим доступу до ресурсу: https://www.honeyd.org/

8. Bro: A system for detecting network intruders in real‑time [Електронний ресурс] / V. Paxson. – 2003. – Режим доступу до ресурсу: https://www.bro.org/

9. Outside the closed world: On using machine learning for network intrusion detection [Електронний ресурс] / R. Sommer, V. Paxson. – 2010. – Режим доступу до ресурсу: https://ieeexplore.ieee.org/abstract/document/5504793

10. Anomaly detection: A survey [Електронний ресурс] / V. Chandola, A. Banerjee, V. Kumar. – 2009. – Режим доступу до ресурсу: https://dl.acm.org/doi/10.1145/1541880.1541882

11. An intrusion‑detection model [Електронний ресурс] / D. E. Denning. – 1987. – Режим доступу до ресурсу: https://ieeexplore.ieee.org/document/5608162

12. The Tao of Network Security Monitoring: Beyond Intrusion Detection [Електронний ресурс] / R. Bejtlich. – 2004. – Режим доступу до ресурсу: https://www.oreilly.com/library/view/the-tao-of/0596008955/

13. Advanced Penetration Testing: Hacking the World's Most Secure Networks [Електронний ресурс] / K. Kendall. – 2003. – Режим доступу до ресурсу: https://www.wiley.com/en-us/Advanced+Penetration+Testing%3A+Hacking+the+World%27s+Most+Secure+Networks-p-9780471785883

14. TCP/IP Illustrated, Volume 1: The Protocols [Електронний ресурс] / J. M. Stewart. – 2001. – Режим доступу до ресурсу: https://www.r-5.org/files/books/computers/internals/net/Richard_Stevens-TCP-IP_Illustrated-EN.pdf

15. UNIX Network Programming, Volume 1: The Sockets Networking API [Електронний ресурс] / W. R. Stevens. – 1998. – Режим доступу до ресурсу: https://dokumen.pub/unix-network-programming-volume-1-the-sockets-networking-api-3rd-edition-1-3rdnbsped-0-13-141155-1-978-0-13-141155-5.html

16. Linux Kernel Development [Електронний ресурс] / R. Love. – 2010. – Режим доступу до ресурсу: https://www.doc-developpement-durable.org/file/Projets-informatiques/cours-&-manuels-informatiques/Linux/Linux%20Kernel%20Development,%203rd%20Edition.pdf

17. Linux System Programming [Електронний ресурс] / R. Love. – 2007. – Режим доступу до ресурсу: https://www.oreilly.com/library/view/linux-system-programming/0596009587/

18. Linux Network Administrator's Guide [Електронний ресурс] / T. Bautts, T. Dawson, G. N. Purdy. – 2005. – Режим доступу до ресурсу: https://www.oreilly.com/library/view/linux-network-administrators/0596002554/

19. UNIX and Linux System Administration Handbook [Електронний ресурс] / E. Nemeth, G. Snyder, T. R. Hein. – 2010. – Режим доступу до ресурсу: https://www.pearson.com/us/higher-education/product/Nemeth-UNIX-and-Linux-System-Administration-Handbook/PGM310591.html

20. Systems Performance: Enterprise and the Cloud [Електронний ресурс] / B. Gregg. – 2013. – Режим доступу до ресурсу: https://www.oreilly.com/library/view/systems-performance/9780133390098/

21. iptables Pocket Reference [Електронний ресурс] / G. N. Purdy. – 2004. – Режим доступу до ресурсу: https://www.oreilly.com/library/view/iptables-pocket-reference/0596009949/

22. Iptables Tutorial [Електронний ресурс] / O. Andreasson. – 2025. – Режим доступу до ресурсу: https://fedoraproject.org/wiki/Iptables/Tutorial

23. Linux iptables by Example [Електронний ресурс] / Red Hat. – 2025. – Режим доступу до ресурсу: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/security_guide/chap-iptables

24. nftables Tutorial [Електронний ресурс] / Red Hat. – 2025. – Режим доступу до ресурсу: https://wiki.nftables.org/

25. systemd for Administrators [Електронний ресурс] / L. Poettering. – 2019. – Режим доступу до ресурсу: https://0pointer.de/blog/projects/systemd-for-admins.html

26. systemd Documentation [Електронний ресурс] / Freedesktop.org. – 2025. – Режим доступу до ресурсу: https://www.freedesktop.org/wiki/Software/systemd

27. Mastering systemd [Електронний ресурс] / P. Searle. – 2020. – Режим доступу до ресурсу: https://www.packtpub.com/product/mastering-systemd/9781783286017

28. The C Programming Language [Електронний ресурс] / B. W. Kernighan, D. M. Ritchie. – 1988. – Режим доступу до ресурсу: https://www.pearson.com/us/higher-education/product/Kernighan-The-C-Programming-Language-2nd-Edition/PGM263355.html

29. The GNU C Reference Manual [Електронний ресурс] / GNU Project. – 2025. – Режим доступу до ресурсу: https://www.gnu.org/software/gnu-c-manual

30. Machine learning for network‑based intrusion detection: A study [Електронний ресурс] / S. Murugesan et al. – 2017. – Режим доступу до ресурсу: https://doi.org/10.1109/JCOMPNET.2017.7983707

31. Scapy Documentation [Електронний ресурс] / Scapy Project. – 2025. – Режим доступу до ресурсу: https://scapy.net/

32. libpcap Packet Capture Library [Електронний ресурс] / Tcpdump.org. – 2025. – Режим доступу до ресурсу: https://www.tcpdump.org/

33. Windows of vulnerability: A case study analysis [Електронний ресурс] / W. A. Arbaugh, W. L. Fithen, J. McHugh. – 1997. – Режим доступу до ресурсу: https://ieeexplore.ieee.org/document/590176

34. Virtual Honeypots: From Botnet Tracking to Intrusion Detection [Електронний ресурс] / N. Provos, T. Holz. – 2007. – Режим доступу до ресурсу: https://www.oreilly.com/library/view/virtual-honeypots/0321173516/

35. Practical Packet Analysis [Електронний ресурс] / C. Sanders. – 2011. – Режим доступу до ресурсу: https://nostarch.com/packet

36. Network Security Through Data Analysis [Електронний ресурс] / M. Collins. – 2014. – Режим доступу до ресурсу: https://www.oreilly.com/library/view/network-security-through/9781491912773/

37. Hands‑On Machine Learning for Cybersecurity [Електронний ресурс] / J. Bamford. – 2019. – Режим доступу до ресурсу: https://www.packtpub.com/product/hands-on-machine-learning-for-cybersecurity/9781838827332

38. A survey of network‑based intrusion detection data sets [Електронний ресурс] / M. Ring et al. – 2019. – Режим доступу до ресурсу: https://doi.org/10.1016/j.cose.2019.02.005

39. A survey of data mining and machine learning methods for cybersecurity intrusion detection [Електронний ресурс] / A. L. Buczak, E. Guven. – 2015. – Режим доступу до ресурсу: https://doi.org/10.1109/COMST.2015.2475064

40. IDS benchmarking: A survey of methodologies [Електронний ресурс] / M. Z. Shafiq, A. Goyal, G. Kesavaraj. – 2019. – Режим доступу до ресурсу: https://doi.org/10.1109/COMST.2019.2912155

41. Evaluating the Effectiveness of Honeypots [Електронний ресурс] / D. Cardona, J. Zapata. – 2008. – Режим доступу до ресурсу: https://www.researchgate.net/publication/247804285

42. Intrusion detection in uncensored networks: Using machine learning approaches [Електронний ресурс] / P. Laskov et al. – 2005. – Режим доступу до ресурсу: https://doi.org/10.1007/11503560_13

43. Intrusion detection systems: A survey and taxonomy [Електронний ресурс] / S. Axelsson. – 2000. – Режим доступу до ресурсу: https://www.cs.dal.ca/~selinger/papers/IDSsurvey.pdf

44. A neural network based system for intrusion detection and classification [Електронний ресурс] / M. H. Moradi, M. Zulkernine. – 2004. – Режим доступу до ресурсу: https://ieeexplore.ieee.org/document/1350197

45. Toward credible IDS benchmark datasets: The KDD CUP 99 dataset [Електронний ресурс] / M. Tavallaee, N. Stakhanova, A. Ghorbani. – 2009. – Режим доступу до ресурсу: https://doi.org/10.1109/CISDAA.2009.5356528

46. BotMiner: Clustering analysis of network traffic for botnet detection [Електронний ресурс] / G. Gu et al. – 2006. – Режим доступу до ресурсу: https://www.usenix.org/legacy/event/sec06/tech/full_papers/gu/gu.pdf

47. A survey of botnet and botnet detection [Електронний ресурс] / M. Bailey et al. – 2005. – Режим доступу до ресурсу: https://www.cs.unm.edu/~strand/ns515/papers/bailey05survey.pdf

48. A survey of machine learning in wireless sensor networks [Електронний ресурс] / T. K. Borgohain, A. Phatak, D. Bhattacharyya. – 2017. – Режим доступу до ресурсу: https://doi.org/10.1016/j.jnca.2017.02.004

49. Deep Learning for Anomaly Detection: A Survey [Електронний ресурс] / R. Chalapathy, S. Chawla. – 2019. – Режим доступу до ресурсу: https://arxiv.org/abs/1901.03407

50. A survey and comparison of honeypots [Електронний ресурс] / E. Vasilomanolakis et al. – 2019. – Режим доступу до ресурсу: https://doi.org/10.1145/3318169

51. Comparing IPS and IDS Solutions: A Performance Study [Електронний ресурс] / D. Wandt et al. – 2020. – Режим доступу до ресурсу: https://doi.org/10.1016/j.jisa.2020.102486

52. Measuring the efficacy of network-based intrusion detection systems [Електронний ресурс] / A. Beddoes et al. – 2011. – Режим доступу до ресурсу: https://doi.org/10.1016/j.scico.2010.08.008

53. Survey of intrusion detection systems: Techniques, datasets and challenges [Електронний ресурс] / A. Khraisat et al. – 2019. – Режим доступу до ресурсу: https://doi.org/10.1186/s42400-019-0042-7

54. Detection of stealthy port scans using adaptive thresholding [Електронний ресурс] / J. Deng, X. Liu, S. Adams. – 2015. – Режим доступу до ресурсу: https://doi.org/10.1109/JCN.2015.148

55. NetDerived: Improving network anomaly detection by auto-generating signatures [Електронний ресурс] / R. Sommer, E. Eskin, V. Paxson. – 2010. – Режим доступу до ресурсу: https://doi.org/10.1145/1868447.1868478

56. Network intrusion detection and prevention systems: Classification and survey [Електронний ресурс] / B. Prakash, N. Padhy. – 2013. – Режим доступу до ресурсу: https://doi.org/10.1016/j.jnca.2012.07.009

57. Real-time network traffic classification using interpretable deep learning [Електронний ресурс] / M. Qiao, A. Albert, M. Chau. – 2019. – Режим доступу до ресурсу: https://doi.org/10.1109/TII.2018.2860658

58. Evaluation of packet reception buffer sizes for busyness monitoring applications [Електронний ресурс] / M. Steiner, G. Carle, G. Fairhurst. – 2002. – Режим доступу до ресурсу: https://doi.org/10.1145/964725.964742

59. Honeypots: Concepts, approaches, and challenges [Електронний ресурс] / I. Mokube, C. Adams. – 2007. – Режим доступу до ресурсу: https://doi.org/10.1109/IAS.2007.4382827

60. Anomaly detection of web‑based attacks with AppSensor [Електронний ресурс] / C. Kruegel, G. Vigna. – 2013. – Режим доступу до ресурсу: https://www.owasp.org/images/c/c3/AppSensor.pdf

61. Design and implementation of an intrusion detection system based on deep learning for IoT [Електронний ресурс] / X. Luo, J. Serge. – 2018. – Режим доступу до ресурсу: https://doi.org/10.1002/dac.3871

62. Artificial Intelligence‑based Network Security: Threat Detection and Forensics [Електронний ресурс] / S. Zhang et al. – 2020. – Режим доступу до ресурсу: https://doi.org/10.1109/MNET.001.1900158

63. IDS for the Cloud: A survey, taxonomy, and challenges [Електронний ресурс] / R. Sommer, Y. Liu, V. Paxson. – 2014. – Режим доступу до ресурсу: https://doi.org/10.1109/COMST.2012.6427542

64. Snort: Lightweight Intrusion Detection for High Speed Networks [Електронний ресурс] / T. Jager et al. – 2006. – Режим доступу до ресурсу: https://snort.org/

65. Snort: Lightweight Intrusion Detection for Networks [Електронний ресурс] / M. Roesch. – 1999. – Режим доступу до ресурсу: https://www.usenix.org/legacy/event/lisa99/full_papers/roesch/roesch_html/

66. Suricata Documentation [Електронний ресурс] / Suricata Project. – 2025. – Режим доступу до ресурсу: https://suricata.io/

67. DDoS attack detection using flow analysis [Електронний ресурс] / D. Zuev et al. – 2016. – Режим доступу до ресурсу: https://doi.org/10.1109/RUPEC.2016.7861615

68. RFC 791: Internet Protocol [Електронний ресурс] / IETF. – 1981. – Режим доступу до ресурсу: https://tools.ietf.org/html/rfc791

69. Linux Firewalls: Enhancing Security with nftables and Beyond [Електронний ресурс] / S. Suehring. – 2015. – Режим доступу до ресурсу: https://el.newoutlook.it/download/book/Linux-Firewalls-Enhancing-Security-with-nftables-and-Beyond.pdf

ДОДАТОК А

СПИСОК ОПУБЛІКОВАНИХ ПРАЦЬ ЗА ТЕМОЮ ДИСЕРТАЦІЇ

1. Шпильовий А.М., Лаптєв О.А. Методи захисту від сканування мережевих портів на основі аналізу трафіку. VIII Міжнародна науково-практична конференція «ПРОБЛЕМИ КІБЕРБЕЗПЕКИ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ СИСТЕМ».

ДОДАТОК Б
КОД ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

```
// Підключення стандартної бібліотеки введення-виводу
#include <stdio.h>
// Бібліотека для роботи з пам'яттю
#include <stdlib.h>
// Бібліотека для роботи з рядками
#include <string.h>
// Для роботи із системними викликами POSIX
#include <unistd.h>
// Для обробки сигналів
#include <signal.h>
// Для багатопоточності
#include <pthread.h>
// Для керування файловими дескрипторами
#include <fcntl.h>
// Для роботи з датою та часом
#include <time.h>
// Для роботи з IP-адресами та мережевими функціями
#include <arpa/inet.h>
// Для роботи з сокетами
#include <sys/socket.h>
// Визначення типів даних
#include <sys/types.h>
// Для роботи з файловою системою
#include <sys/stat.h>
// Структури IP-пакетів
#include <netinet/ip.h>
// Структури TCP-пакетів
```

```
#include <netinet/tcp.h>
// Структури UDP-пакетів
#include <netinet/udp.h>
// Надає засоби для перебору аргументів функції, кількість та типи яких заздалегідь
не відомі
#include <stdarg.h>
// Для роботи з обмеженнями
#include <limits.h>
// Для отримання імені каталогу файлу, що виконується
#include <libgen.h>
// Визначення розміру буфера прийому мережевих пакетів
#define BUF_SIZE 65536
// Максимальна кількість IP-адрес, що одночасно відстежуються
#define MAX_TRACKED_IPS 1024
// Максимальна кількість унікальних портів, що відстежуються для одного IP
#define MAX_TRACKED_PORTS 128
// Ім'я файлу конфігурації
#define CONFIG_FILE "config.ini"
// Файл з PID
#define PID_FILE "/var/run/scan_detector.pid"

// Структура для зберігання інформації про IP-адресу
typedef struct
{
   // IP-адреса джерела
   char ip[32];
   // Масив портів, куди IP відправляв пакети
   int ports[MAX_TRACKED_PORTS];
   // Кількість унікальних портів зафіксованих для цього IP
   int port_count;
```

```
    // Час першого пакету від цього IP
    time_t first_seen;
    // Час коли було заблоковано IP
    int banned;
} ip_tracker_t;

// Структура конфігурації, що завантажується з config.ini
typedef struct
{
    // Максимальна кількість портів, які IP може просканувати до блокування
    int max_ports;
    // Час у секундах, на який IP блокується
    int ban_time_seconds;
    // Шлях до файлу логування подій
    char log_file[128];
    // Назва файрволу: iptables або nftables
    char firewall[32];
} config_t;

// Масив всіх IP-адрес, що відстежуються.
ip_tracker_t tracked_ips[MAX_TRACKED_IPS];

// Структура зберігання поточних налаштувань, завантажених з конфігураційного
файла
config_t config;

// М'ютекс для захисту доступу до загальних структур у багатопотоковому
середовищі
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

```
// RAW-сокети для перехоплення TCP та UDP пакетів
int raw_sock_tcp, raw_sock_udp;

// Потоки для обробки TCP та UDP трафіку
pthread_t tcp_thread, udp_thread;

// Прапор для коректного завершення потоків
volatile sig_atomic_t stop_flag = 0;

// Видаляє файл PID, якщо він існує
void remove_pid()
{
   // Викликає системну функцію unlink() для видалення PID-файлу
   unlink(PID_FILE);
}

// Функція завантаження конфігурації із файлу
void load_config()
{
   // Буфер для зберігання абсолютного шляху до виконуваного файлу
   char path[PATH_MAX];
   // Отримуємо шлях до поточного виконуваного файлу через символічне посилання
/proc/self/exe
   ssize_t count = readlink("/proc/self/exe", path, PATH_MAX);
   // Якщо не вдалося отримати шлях — виводимо помилку та завершуємо програму
   if (count == -1)
   {
      perror("readlink");
      exit(1);
   }
```

```
// Завершуємо рядок нульовим символом після останнього символу шляху
path[count] = '\0';

// Отримуємо директорію з повної дороги (наприклад, /usr/bin -> /usr)
const char *dir = dirname(path);

// Сформуємо повний шлях до config.ini, який має лежати поруч із бінарником
char config_path[PATH_MAX];
snprintf(config_path, sizeof(config_path), "%s/%s", dir, CONFIG_FILE);

// Намагаємося відкрити файл конфігурації для читання
FILE *fp = fopen(config_path, "r");

// Якщо не вдалося відкрити файл - виводимо помилку та виходимо
if (!fp)
{
    printf("Cannot open config file: %s\n", config_path);
    exit(1);
}

// Значення за промовчанням, якщо параметри не вказані в config.ini

// Скільки різних портів може просканувати один IP за інтервал
config.max_ports = 10;
// На скільки секунд заблокувати зловмисника
config.ban_time_seconds = 120;
// Шлях до файлу логів
strcpy(config.log_file, "/var/log/scan_detector.log");
// Система файрвола за замовчуванням - iptables
```

```
strcpy(config.firewall, "iptables");
// Буфер для читання рядків із файлу
char line[256];
// Читаємо файл рядково
while (fgets(line, sizeof(line), fp))
{
   // Видаляємо символи нового рядка \r та \n
   line[strcspn(line, "\r\n")] = 0;
   // Пропускаємо коментарі (# або ;) та секції [section], а також порожні рядки
   if (line[0] == '#' || line[0] == ';' || line[0] == '[' || strlen(line) == 0)
      continue;
   // Буфери для ключа та значення
   char key[128], value[128];
   // Розбираємо рядок у форматі ключ = значення
   if (sscanf(line, "%127[^=]=%127s", key, value) == 2)
   {
      // Якщо ключ - це max_ports, перетворимо значення на число
      if (strcmp(key, "max_ports ") == 0)
         config.max_ports = atoi(value);
      // Аналогічно з іншими
      else if (strcmp(key, "ban_time_seconds") == 0)
         config.ban_time_seconds = atoi(value);
      else if (strcmp(key, "log_file") == 0)
         strncpy(config.log_file, value, sizeof(config.log_file));
      else if (strcmp(key, "firewall") == 0)
         strncpy(config.firewall, value, sizeof(config.firewall));
   }
}
// Закриваємо файл конфігурації
fclose(fp);
```

```
    // Перевірка, що програма запущена з root-правами (необхідні для raw-сокетів та
керування файрволом)
    if (geteuid() != 0)
    {
        fprintf(stderr, "Цей програма повинна бути як root!\n");
        exit(1);
    }
}

// Функція логування повідомлень у лог-файл
void log_event(const char *fmt, ...)
{
    // Відкриваємо файл додавання (append), шлях береться з config.log_file
    FILE *fp = fopen(config.log_file, "a");
    // Якщо файл не відкрився - просто виходимо з функції
    if (!fp) return;
    // Отримуємо поточний час у форматі time_t
    time_t now = time(NULL);
    // Перетворимо час на локальне уявлення
    const struct tm *tm_info = localtime(&now);
    // Буфер для зберігання форматованого рядка часу
    char time_buf[26];
    // Перетворимо дату/час у зручний формат
    strftime(time_buf, 26, "%Y-%m-%d %H:%M:%S", tm_info);
    // Друкуємо у файл логів рядок з позначкою часу
    fprintf(fp, "[%s] ", time_buf);
    // Ініціалізуємо змінний список аргументів
    va_list args;
    va_start(args, fmt);
    // Друкуємо у файл рядок відповідно до переданого формату
```

```
    vfprintf(fp, fmt, args);
    // Завершуємо обробку списку аргументів
    va_end(args);
    // Друкуємо символ нового рядка, завершивши лог-запис
    fprintf(fp, "\n");
    // Закриваємо лог-файл
    fclose(fp);
}

// Генерація команди для блокування або розблокування IP
void get_fw_command(char* cmd, size_t size, const char* ip, int block) {
    if (strcmp(config.firewall, "nftables") == 0) {
        const char* action = block ? "add" : "delete";
        snprintf(cmd, size, "nft %s element ip filter blacklist { %s }", action, ip);
    } else {
        if (block) {
            snprintf(cmd, size, "ipset add blacklist %s timeout %d", ip,
config.ban_time_seconds);
        } else {
            snprintf(cmd, size, "ipset del blacklist %s", ip);
        }
    }
}


// Функція блокує IP-адресу через iptables або nftables, а потім через певний час
автоматично розблокує її
void ban_ip(const char *ip)
{
    char cmd[256];
```

```
    get_fw_command(cmd, sizeof(cmd), ip, 1);
    system(cmd);

    // Записуємо в лог, що IP був заблокований
    log_event("Banned IP: %s", ip);

    // Виконуємо fork(), щоб створити окремий процес, який розблокує IP через
певний час
    if (!fork())
    {
        // Новий процес засинає на вказаний час блокування
        sleep(config.ban_time_seconds);
        // Після таймера формуємо команду для видалення правила з nftables
        get_fw_command(cmd, sizeof(cmd), ip, 0);
        system(cmd);
        // Записуємо в лог, що IP був розблокований
        log_event("Unbanned IP: %s", ip);
        // Завершуємо дочірній процес
        exit(0);
    }
}

// Функція перевіряє, чи вже відслідковувався зазначений порт для певного IP
int port_already_tracked(const ip_tracker_t *tracker, int port)
{

    int find = 0;
    // Проходимо по всіх записаних портах у структурі tracker для даного IP
    for (int i = 0; i < tracker->port_count; i++)
    {
```

```
        // Якщо знайдено порт, який вже був зареєстрований — find = 1
        if (tracker->ports[i] == port)
        {
            find = 1;
            break;
        }
    }
    // Повертаємо 0 find
    return find;
}

// Функція обробляє кожен вхідний пакет, записуючи IP та порт і приймає рішення
про блокування
void track_ip(const char *ip, int port)
{
    //
    time_t now = time(NULL);
    // Блокуємо доступ до спільної структури tracked_ips
    pthread_mutex_lock(&lock);
    // Перебираємо всі записані IP-адреси
    for (int i = 0; i < MAX_TRACKED_IPS; i++)
    {
        // Якщо запис не пустий і давно не оновлювався — очищаємо його (старіше 60
секунд)
        if (tracked_ips[i].ip[0] != '\0' && now - tracked_ips[i].first_seen > 60)
        {
            memset(&tracked_ips[i], 0, sizeof(ip_tracker_t)); // повне обнулення запису
            continue;
        }
        // Якщо IP уже відслідковується
```

```
    if (strcmp(tracked_ips[i].ip, ip) == 0)
    {
        // Якщо з моменту першої активності минуло понад 10 секунд — скидаємо лічильник портів
        if (now - tracked_ips[i].first_seen > 10)
        {
            tracked_ips[i].port_count = 0;
            tracked_ips[i].first_seen = now;
        }

        // Якщо новий порт і він ще не перевищив ліміт — додаємо в список
        if (!port_already_tracked(&tracked_ips[i], port) && tracked_ips[i].port_count < MAX_TRACKED_PORTS)
        {
            tracked_ips[i].ports[tracked_ips[i].port_count++] = port;
        }

        // Якщо кількість портів перевищує допустиму норму і IP ще не заблокований
        if (tracked_ips[i].port_count > config.max_ports && !tracked_ips[i].banned)
        {
            tracked_ips[i].banned = now; // Встановлюємо мітку часу блокування
            ban_ip(ip);  // Викликаємо блокування IP
        }

        // Якщо IP був в бані та час блокування пройшов - видаляємо даний IP
        if(tracked_ips[i].banned){
            if(tracked_ips[i].banned + config.ban_time_seconds < now) memset(&tracked_ips[i], 0, sizeof(ip_tracker_t));
        }
```

```
            // Розблокуємо мьютекс і виходимо з функції
            pthread_mutex_unlock(&lock);
            return;
        }
    }
    // Якщо IP не знайдено — шукаємо вільне місце для нового запису
    for (int i = 0; i < MAX_TRACKED_IPS; i++)
    {
        if (tracked_ips[i].ip[0] == '\0')
        {
            strcpy(tracked_ips[i].ip, ip);
            tracked_ips[i].ports[0] = port;
            tracked_ips[i].port_count = 1;
            tracked_ips[i].first_seen = now;
            tracked_ips[i].banned = 0;
            break;
        }
    }

    // Розблокуємо мьютекс після завершення всіх дій
    pthread_mutex_unlock(&lock);
}

// Функція обробляє один IP-пакет (TCP або UDP), витягує IP-джерело та порт
призначення
void process_packet(unsigned char *buffer, int size, int protocol)
{
    // Інтерпретуємо початок буфера як IP-заголовок
    struct iphdr *iph = (struct iphdr *)buffer;
    // Створюємо структуру для зберігання IP-адреси джерела
```

```
    struct sockaddr_in source;
    // Отримуємо IP-адресу відправника з IP-заголовка
    source.sin_addr.s_addr = iph->saddr;
    // Перетворюємо IP у рядковий формат
    const char *ip = inet_ntoa(source.sin_addr);
    // Ініціалізуємо змінну для збереження порту призначення
    int dest_port = 0;
    // Якщо протокол — TCP
    if (protocol == IPPROTO_TCP)
    {
        // Знаходимо TCP-заголовок
        struct tcphdr *tcph = (struct tcphdr *)(buffer + iph->ihl * 4);
        // Отримуємо порт призначення з TCP-заголовка
        dest_port = ntohs(tcph->dest);
        // Якщо протокол — UDP
    }
    else if (protocol == IPPROTO_UDP)
    {
        // Аналогічно: шукаємо UDP-заголовок
        struct udphdr *udph = (struct udphdr *)(buffer + iph->ihl * 4);
        // Отримуємо порт призначення
        dest_port = ntohs(udph->dest);
    }
    // Якщо порт визначено коректно, передаємо IP і порт на обробку
    if (dest_port > 0) track_ip(ip, dest_port);

}

// Потік прослуховування TCP
void *sniff_tcp(void *arg)
```

```
{
   // Виділяємо буфер для прийому пакетів
   unsigned char *buffer = malloc(BUF_SIZE);
   // Перевіряємо успішність виділення пам'яті
   if (!buffer)
      pthread_exit(NULL);
   // Реєструємо звільнення пам'яті при завершенні потоку
   pthread_cleanup_push(free, buffer);
   // Створюємо RAW-сокет для TCP
   raw_sock_tcp = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
   // Якщо не вдалося створити сокет - логуємо помилку та виходимо
   if (raw_sock_tcp < 0){
      perror("TCP socket error");
      pthread_exit(NULL);
   }
   // Отримуємо поточні прапори сокету
   int flags = fcntl(raw_sock_tcp, F_GETFL, 0);
   // Встановлюємо неблокуючий режим
   fcntl(raw_sock_tcp, F_SETFL, flags | O_NONBLOCK);
   // Основний цикл захоплення пакетів TCP
   while (!stop_flag)
   {
      // Читаємо пакет із RAW-сокету
      int data_size = recvfrom(raw_sock_tcp, buffer, BUF_SIZE, 0, NULL, NULL);
      // Якщо отримані дані - передаємо на обробку
      if (data_size > 0)
         process_packet(buffer, data_size, IPPROTO_TCP);
      // Короткий сон, щоб знизити навантаження на CPU
      usleep(10000);
   }
```

```
    // Звільняємо буфер при завершенні потоку
    pthread_cleanup_pop(1);
    return NULL;
}

// Потік прослуховування UDP
void *sniff_udp(void *arg)
{
    // Виділяємо буфер для прийому пакетів
    unsigned char *buffer = malloc(BUF_SIZE);
    // Перевіряємо успішність виділення пам'яті
    if (!buffer) pthread_exit(NULL);
    // Реєструємо звільнення пам'яті при завершенні потоку
    pthread_cleanup_push(free, buffer);
    // Створюємо RAW-сокет для UDP
    raw_sock_udp = socket(AF_INET, SOCK_RAW, IPPROTO_UDP);
    // Якщо не вдалося створити сокет - логуємо помилку та виходимо
    if (raw_sock_udp < 0){
        perror("UDP socket error");
        pthread_exit(NULL);
    }
    // Отримуємо поточні прапори сокету
    int flags = fcntl(raw_sock_udp, F_GETFL, 0);
    // Встановлюємо неблокуючий режим
    fcntl(raw_sock_udp, F_SETFL, flags | O_NONBLOCK);
    // Основний цикл захоплення пакетів UDP
    while (!stop_flag)
    {
        // Читаємо пакет із RAW-сокету
        int data_size = recvfrom(raw_sock_udp, buffer, BUF_SIZE, 0, NULL, NULL);
```

```
        // Якщо отримані дані - передаємо на обробку
        if (data_size > 0) process_packet(buffer, data_size, IPPROTO_UDP);
        // Короткий сон, щоб знизити навантаження на CPU
        usleep(10000);
    }
    // Звільняємо буфер при завершенні потоку
    pthread_cleanup_pop(1);
    return NULL;
}

// Обробник сигналу SIGTERM — викликається при завершенні служби
void handle_sigterm(int sig)
{
    // Прапор для завершення потоків
    stop_flag = 1;
    // Завершуємо потік, який перехоплює TCP-пакети
    pthread_cancel(tcp_thread);
    // Завершуємо потік, який перехоплює UDP-пакети
    pthread_cancel(udp_thread);
    // Закриваємо TCP RAW-сокет
    close(raw_sock_tcp);
    // Закриваємо UDP RAW-сокет
    close(raw_sock_udp);
    // Видаляємо PID-файл, щоб позначити, що служба більше не працює
    remove_pid();
    // Логуємо, що служба зупинена
    log_event("Service stopped");
    // Завершуємо програму
    exit(0);
}
```

```
// Функція записує PID поточного процесу в файл
void write_pid()
{
   // Відкриваємо файл для запису PID (створюється або перезаписується)
   FILE *f = fopen(PID_FILE, "w");
   // Якщо файл вдалося відкрити
   if (f){
      // Записуємо числовий ідентифікатор поточного процесу
      fprintf(f, "%d", getpid());
      // Закриваємо файл
      fclose(f);
   }
}

// Функція читає PID з файлу та повертає його як значення типу pid_t
pid_t read_pid()
{
   // Відкриває файл PID_FILE у режимі читання
   FILE *f = fopen(PID_FILE, "r");
   // Якщо файл не існує або не відкрився — повертаємо -1 як ознаку помилки
   if (!f) return -1;
   // Змінна для збереження зчитаного PID
   pid_t pid;
   // Зчитуємо ціле число з файлу у змінну pid
   fscanf(f, "%d", &pid);
   // Закриваємо файл після читання
   fclose(f);
   // Повертаємо зчитаний PID
   return pid;
```

```
}

// Функція запускає службу у фоновому режимі як демон
void start_service()
{
    // Зчитує PID з PID-файлу, якщо такий існує
    pid_t pid = read_pid();

    // Якщо PID дійсний та процес із цим PID реально існує — служба вже запущена
    if (pid > 0 && kill(pid, 0) == 0)
    {
        printf("Service already running with PID %d\n", pid);
        exit(1); // Вихід з помилкою — повторний запуск не допускається
    }
    // Створюємо дочірній процес через fork
    pid_t daemon_pid = fork();
    // Якщо fork повернув помилку — виводимо повідомлення і завершуємо
    if (daemon_pid < 0){
        perror("fork");
        exit(1);
    }
    // Якщо ми в основному процесі — виводимо PID і виходимо
    if (daemon_pid > 0){
        printf("Service started with PID %d\n", daemon_pid);
        exit(0); // Батьківський процес завершено — демон продовжить виконання
    }

    // Перетворюємо процес у сесію, щоб від'єднати від терміналу
    setsid();
```

```
    // Змінюємо поточну директорію на кореневу, щоб уникнути блокування
монтування
    chdir("/");
    // Скидаємо маску прав доступу
    umask(0);
    // Встановлюємо обробник сигналу SIGTERM для коректного завершення служби
    signal(SIGTERM, handle_sigterm);
    // Записуємо PID поточного процесу в файл
    write_pid();
    // Завантажуємо конфігурацію з файлу config.ini
    load_config();
    // Створюємо потік для перехоплення TCP-пакетів
    pthread_create(&tcp_thread, NULL, sniff_tcp, NULL);
    // Створюємо потік для перехоплення UDP-пакетів
    pthread_create(&udp_thread, NULL, sniff_udp, NULL);
    // Очікуємо завершення потоку TCP (на практиці ніколи не завершиться)
    pthread_join(tcp_thread, NULL);
    // Очікуємо завершення потоку UDP
    pthread_join(udp_thread, NULL);
    // При завершенні служби видаляємо PID-файл
    remove_pid();
}

// Зупиняє роботу сервісу шляхом надсилання сигналу SIGTERM
void stop_service()
{
    // Зчитує PID з PID-файлу
    pid_t pid = read_pid();
    // Якщо PID-файл не існує або не містить валідного PID
    if (pid <= 0){
```

```
        printf("Service is not running.\n");
        return;
    }
    // Надсилаємо сигнал SIGTERM процесу з цим PID
    kill(pid, SIGTERM) == 0 ? printf("Service stopped.\n") : perror("Failed to stop
service");
}

// Перезапускає службу: зупиняє і знову запускає через 1 секунду
void restart_service()
{
    stop_service();
    sleep(1);
    start_service();
}

// Перевіряє, чи працює сервіс у даний момент
void service_status()
{
    // Зчитуємо PID
    pid_t pid = read_pid();
    // Якщо PID дійсний і процес із цим PID існує
    pid > 0 && kill(pid, 0) == 0 ? printf("Service is running with PID %d\n", pid) :
printf("Service is not running.\n");
}

// Виводить список аргументів програми
void print_help()
{
    printf("Usage: ./scan_detector --start | --stop | --restart | --status | --help\n");
```

```
}

// Головна функція програми
int main(int argc, const char *argv[])
{
   // Перевірка кількості аргументів — очікується один параметр
   if (argc != 2){
      print_help();
      return 1;
   }
   // Порівнюємо переданий аргумент з підтримуваними командами
   if (strcmp(argv[1], "--start") == 0)
   {
      start_service();
   }else if (strcmp(argv[1], "--stop") == 0){
      stop_service();
   }else if (strcmp(argv[1], "--restart") == 0){
      restart_service();
   }else if (strcmp(argv[1], "--status") == 0){
      service_status();
   }else{
      // Якщо параметр невідомий - показати help
      print_help();
   }
   // Повертаємо 0 — нормальне завершення
   return 0;
}
```

ДОДАТОК В
КОНФІГУРАЦІЯ СЕРВІСУ

```
[Unit]
Description=Scan Detector Service
After=network.target

[Service]
Type=simple
ExecStart=/opt/my_service/scan_detector --start
ExecStop=/opt/my_service/scan_detector --stop
ExecReload=/opt/my_service/scan_detector --restart
Restart=always
User=root
WorkingDirectory=/opt/my_service/
StandardOutput=file:/var/log/scan_detector_stdout.log
StandardError=file:/var/log/scan_detector_stderr.log

[Install]
WantedBy=multi-user.target
```