НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра кібербезпеки

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«ДОПОВНЕННЯ ЗАХИСТУ ВЕБСЕРВЕРА ЗА ДОПОМОГОЮ СТВОРЕННЯ ХИБНИХ МІШЕНЕЙ»

Виконав: здобувач 4 курсу
Рівень бакалавр
Галузь знань 12 «Інформаційні технології»,
спеціальність 125 «Кібербезпека»
Ніякий Олександр Олександрович

Керівник: к.т.н., доцент
Бойко Віктор Дмитрович

Рецензент: к.т.н, доцент
Ахмаметьева Ганна Валеріївна

Одеса - 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»
Факультет кібербезпеки та інформаційних технологій
Кафедра **кібербезпеки**
Галузь знань: **12 Інформаційні технології**
Спеціальність: **125 Кібербезпека**
Назва освітньої програми: **Кібербезпека**

ЗАТВЕРДЖУЮ
Завідувач кафедри кібербезпеки
професор С.А. Горбаченко
_____ «____ »_____ 20__ року

З А В Д А Н Н Я
на кваліфікаційну (бакалаврську) роботу
здобувачу Ніякому Олександру Олександровичу

1. Тема роботи: «Доповнення захисту вебсервера за допомогою створення хибних мішеней»
Керівник роботи: к.т.н, доцент Віктор Дмитрович Бойко
затверджені наказом НУ ОЮА від «18» березня 2025 року № 548-6
2. Строк подання здобувачем роботи «19» травня 2025 року
3. Дата видачі завдання«16 » вересня 2024 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка

Здобувач_____
(підпис) (прізвище та ініціали)
Керівник роботи_____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Ніякий О. О. Доповнення захисту вебсервера за допомогою створення хибних мішеней - Кваліфікаційна робота бакалавра за спеціальністю 125 «Кібербезпека», освітньою програмою «Кібербезпека». Кваліфікаційна робота викладена на 33 сторінках основного тексту і 49 сторінок загального тексту, містить 3 розділи, 11 рисунків, 1 додаток та 15 використаних джерел.

Мета дослідження є побудування модульного додатоку створення пасток для кіберзлочинців, які будуть імітувати реальні вебсервери або їх додаткові інструменти.

У кваліфікаційній роботі виконано реалізацію отримання та аналізу кіберінцидентів за допомогою хибних мішеней.

Розроблену систему можна використовувати для доповнення захисту вебсерверів, додаючи отриману інформацію до стандартних засобів захисту.

Можливими напрямками подальших досліджень є додавання додаткових можливостей для використання більшої кількості протоколів та сервісів, створення нових функцій для імітації серверів, а також розширення функціональності самої системи.

ІНЦИДЕНТ КІБЕРБЕЗПЕКИ, ОТРИМАННЯ ТА АНАЛІЗ ІНЦИДЕНТІВ, МОДУЛЬНИЙ HONEYPOT, ІМІТАЦІЯ ВЕБСЕРВЕРУ, ЗАХИСТ ВЕБСЕРВЕРУ, РЕАЛІЗАЦІЯ ПАСТОК, PYTHON, PARAMIKO, SSH, OOP, SOLID.

ABSTRACT

Niiakyi O. O. Supplementing the protection of a web server by creating false targets - Bachelor's thesis in the specialty 125 “Cybersecurity”, educational program “Cybersecurity”. The qualification work is set out on 33 pages of the main text and 53 pages of the general text, contains 3 sections, 11 figures, 1 appendix and 15 references.

The purpose of the research is to build a modular application for creating traps for cybercriminals that will imitate real web servers or their additional tools.

In the qualification work, the implementation of obtaining and analyzing cyber incidents using false targets was performed.

The developed system can be used to supplement the protection of web servers by adding the information obtained to standard security tools.

Possible areas for further research include adding additional features to use more protocols and services, creating new functions to simulate servers, and expanding the functionality of the system itself.

CYBERSECURITY INCIDENT, INCIDENT ACQUISITION AND ANALYSIS, MODULAR HONEYPOT, WEB SERVER SIMULATION, WEB SERVER PROTECTION, TRAP IMPLEMENTATION, PYTHON, PARAMIKO, SSH, OOP, SOLID.

ЗМІСТ

ВСТУП

Актуальність теми дослідження обумовлена зростанням кількості та складності кібератак на вебсервери та інші структури. Тільки у 2024 році кількість кіберінцедентів на Україну збільшилися у 69,8% у порівнянні із 2023 роком. Ворог продовжує використовувати всі можливі методи отримання інформації, треба реалізовувати ефективний захист та аналіз методів атак [1]. Вебсервери є особливо привабливою ціллю для зловмисників через їх публічну доступність та зберігання важливих даних. успішна атака на вебсервер може призвести до серйозних наслідків, таких як витік даних, фінансові втрати, порушення репутації та ін. Традиційні методи як Firewall, IDS/IPS, хоч і є дуже важливими для забезпечення захисту, але мають обмеження у виявленні складних атак та сильно залежать від бази даних відомих атак, які переводяться у сигнатури для виявлення їх.

У зв’язку з цим, методом реалізації спеціальних пасток, які імітують системи з вразливостями, можна не чекати серйозних атак на реальні вебсервери, тощо, а отримувати інформацію про атаку від ізольованих спеціально налаштованих пасток, після чого проаналізувати данні й краще захищати справжні системи. Отримувати данні для аналізу можна за допомогою систем логування, що часто використовуються [2]. Система-пастка називається honeypot, яка і буде реалізована у даній роботі.

Головним напрямом є захищення вебсерверів. Через велику кількість різновидів використання, треба реалізовувати систему, яка буде легко розшируватся та модифікуватися від потреби, для кращої адаптації ті використання будь-яких протоколів та сервісів. Реалізувати таких honeypot можна за допомогою модульної архітектури.

Модульна архітектура реалізується за допомогою методів Об’єктно-орієнтованого програмування та принципів SOLID для кращої реалізації.

Мета дослідження: Побудувати модульний додаток створення пасток для кіберзлочинців, які будуть імітувати реальні вебсервери або їх додаткові інструменти.

Для досягнення мети було сформовано такі завдання:

1. Провести аналіз методів захисту вебсерверів;
2. Здійснити теоретичну розробку модульної системи для захисту вебсерверу;
3. Обрати інструменти та методи реалізації додатку;
4. Побудувати та запустити модульний Honeypot;
5. Проаналізувати отриманні данні;
6. Провести аналіз покращень;

Об’єкт дослідження: кіберінциденти та атаки на вебсервери та їх додатки.

Предмет дослідження: отримання і аналіз кібератак та інцидентів на вебсервери

Реалізація даного додатка дасть змогу отримати зручний інструмент для отримання і аналізу методів кібератак та інцидентів на вебсервери для подальшого покращення стандартних систем захисту основних активів.

Основні висновки роботи були опубліковані та апробовані на науковій конференції «V Міжнародної науково-практичної конференції» [3].

1 АНАЛІЗ МЕТОДІВ ЗАХИСТУ ВЕБСЕРВЕРІВ

1.1 Огляд традиційних методів захисту

Вебсервери є одними з найпривабливіших цілей для зловмисників, оскільки вони забезпечують публічний доступ до ресурсів та взаємодію з користувачами через відкриті мережі. Це обумовлює потребу в комплексному та надійному захисті вебінфраструктури. Протягом останніх десятиліть були розроблені різні засоби безпеки, які стали основою інформаційного захисту організацій. Найбільш поширеними серед них є фаєрволи (firewall), системи виявлення вторгнень (IDS) та системи запобігання вторгнень (IPS).

Firewall (Брандмауер) - це апаратний або програмний пристрій мережевої безпеки, який відстежує весь вхідний і вихідний трафік і на основі визначеного набору правил безпеки приймає, відхиляє або відкидає певний трафік. Він діє як охоронець, який допомагає захистити ваш цифровий простір від небажаних відвідувачів і потенційних загроз. Правила безпеки налаштовуються всередині організації відповідно поставленим нормам [5].

Два основних типів фаєрволів:

1. Мережевий фаєрвол - розташовуються між мережами (наприклад, між Інтернетом і внутрішньою мережею) і аналізують пакети за IP-адресами, портами та протоколами. Наприклад, закрити всі порти крім 80 та 443, або ще блокувати пакети від доданих до таблиці ір адрес.

2. Прикладний фаєрвол (WAF) - Брандмауер прикладного рівня може перевіряти та фільтрувати пакети на будь-якому рівні OSI, аж до прикладного. Він має здатність блокувати певний контент, а також розпізнавати, коли певні програми та протоколи (такі як HTTP/HTTPS , FTP) використовуються неналежним чином.

Фаєрволи можуть бути програмні та апаратні (рисунок 1.1).

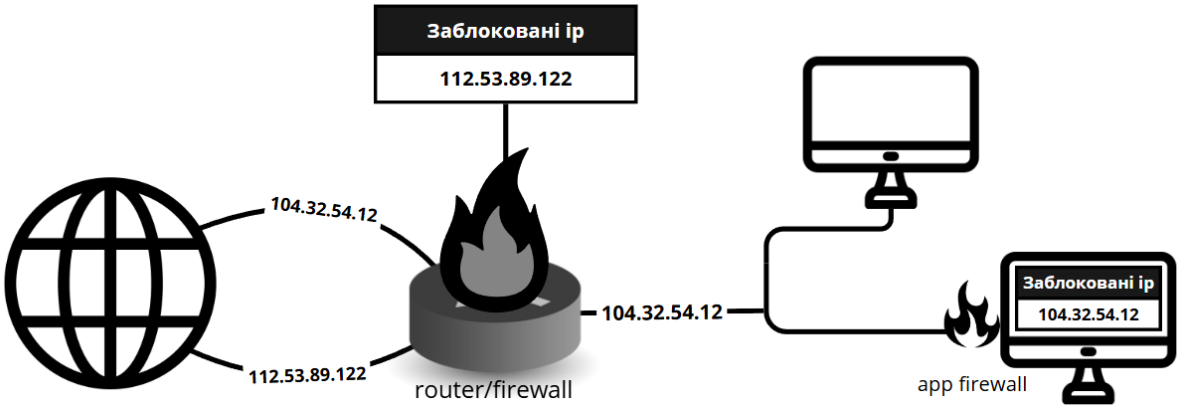

Рисунок 1.1 - Схема роботи firewall

Проте фаєрволи мають обмеження: вони не виявляють атак, які проходять через дозволений трафік або маскуються під легітимну активність.

Intrusion Detection System (IDS - Системи виявлення вторгнень) - це системи, які аналізують трафік або поведінку системи для виявлення ознак несанкціонованої активності. Вони аналізують шаблони мережевого трафіку, лог-файли, системні події та інші джерела. Кожна незаконна діяльність або порушення реєструється централізовано за допомогою системи SIEM, або повідомляється адміністрації [6].

Робота IDS поділяється на два основні підходи:

1. Виявлення сигнатур (signature-based) - перевіряє мережеві пакети на наявність відомих шаблонів, пов'язаних з певними загрозами. Система виявлення вторгнень на основі сигнатур порівнює пакети з базою даних сигнатур атак і видає сповіщення, якщо знайдено збіг. Для виявлення нових загроз необхідні регулярні оновлення, але невідомі атаки без сигнатур можуть обійти цю систему.

2. Виявлення аномалій (anomaly-based) - виявляє відхилення від нормальної поведінки користувачів, вхідних та вихідних пакетів, адміністраторів.

Основним недоліком IDS є те, що вони не блокують атаки, а лише сповіщають про них і тільки після аналізу. Це може призводити до затримки в реакції та до великої кількості хибних спрацювань. Також головну, що цій системі потрібно постійне додавання сигнатур, а їх треба десь брати.

Intrusion Prevention System (IPS - Системи запобігання вторгнень) - це логічне продовження IDS, яке поєднує моніторинг з автоматичним реагуванням на загрози. Зазвичай ця система інтегрується з IDS, тому виявлення займається система виявлення, яка при знаходженні загрози посилає його IPS для реагування. Часто ще називають IDPS [7].

На відміну від IDS, IPS має можливість:

- блокувати шкідливий трафік у реальному часі;
- перервати сесію;
- вносити зміни до фаєрволу на основі виявлених загроз.

IPS часто інтегрується з фаєрволом як єдина система (наприклад, NGFW - next-generation firewall).

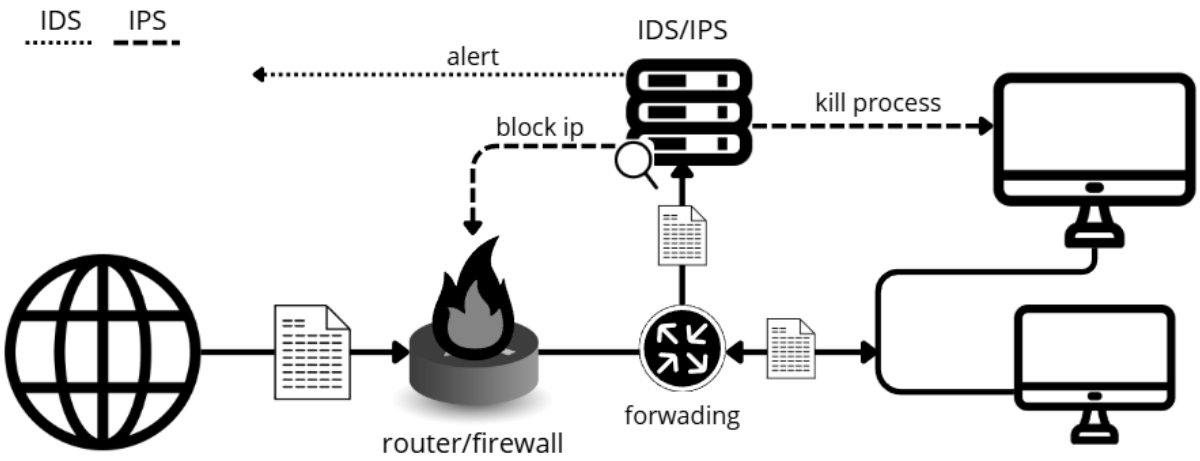

Рисунок 1.2 - Схема роботи IDS/IPS

Для максимального захисту використовують всі ці методи разом. Однак, незважаючи на багаторівневий захист, всі ці системи мають пасивний характер: вони діють тільки тоді, коли атака вже розпочата. Вони не здатні ініціювати взаємодію з потенційним зловмисником, щоб отримати додаткову інформацію про його інструменти, мотиви чи техніки. Саме тому виникає потреба в альтернативних підходах, таких як використання систем-«пасток» (honeypots), які дозволяють проводити активне дослідження загроз.

1.2 Альтернативний спосіб захисту. Honeypot

На відміну від класичний способів захисту вебсерверів, які загалом залежать від того, чи мали ми вже справу з тією чи іншою загрозою і обробляли її. Існують інші методи, які не захищають сервер напряму, а скоріше відволікають та аналізують способи зламу зловмисників. Один з таких методів захисту називають honeypot.

Honeypot пропонує проактивну модель оборони. Її сутність - створити спеціальну «пастку», котра здається зловмиснику законним і вразливим ресурсом, але насправді ізольована та повністю контрольована службою безпеки. Увесь трафік, що надходить до такого сервера, апріорі розцінюється як підозрілий, тому кожна взаємодія ретельно журналюється й аналізується [8].

Сама назва “honeypot” - горщик з медом походить з аналогії до принади, яка виглядає привабливо, але є пасткою. У природі мед - це те, що приваблює комах, зокрема бджіл або ос, які можуть у результаті потрапити в пастку. У сфері кібербезпеки honeypot - це фальшива, але спеціально приваблива ціль, яка імітує справжній сервер чи сервіс, щоб заманити зловмисників.

Першою приманкою для кіберзлочинців часто є вразливість системи безпеки, тоді як всередині знаходяться підроблені цифрові активи, програмні додатки або дані, щоб зайняти зловмисника. Ось деякі поширені тактики приманки, які використовують honeypots для залучення хакерів: слабкі паролі, відкриті порти, фальшиві IP-адреси, фальшиві електронні листи, підроблені файли, фальшиві папки, фальшиві URL-адреси.

Існує багато видів приманок, але найчастіше їх поділяють на приманки з низьким та високим рівнем взаємодії:

Низький рівень взаємодії (low‑interaction) - вид приманки, яка не реалізує повного функціоналу системи, використовується тільки для отримання розвідданих. Наприклад це може бути легка імітація вебсерверу на якомусь ip, основні клієнти не можуть знати про нього, тому коли хтось спробує підключитися та ввести логін і пароль, ми збираємо всі данні про ного, які можемо. Це може бути як айпі, так і логін

з паролем, який зловмисник використовував. Це все, далі немає реалізації нічого, це вся наша пастка, далі ми можемо заблокувати бота або зловмисника.

Високий рівень взаємодії (high‑interaction) - вид приманки, яка реалізує повну копію функціоналу системи. Наприклад, реалізація SSH з легким паролем (або приймати всі паролі) яка повністю імітує реальний SSH та дає зловмиснику реалізовувати напад на систему. При цьому ми фіксуємо кожну дію зловмисника для подальшого аналізу. Цей рівень взаємодіє дає більше інформації, крім розвідданих, можемо ще отримати методи та засоби атаки. Але це потребує більшої потужності.

Середній рівень взаємодії (mid‑interaction) - це компромісний варіант між низьким та високим рівнями взаємодії. Ханіпоти середнього рівня емулюють лише найбільш важливі сервіси та протоколи. Наприклад, ханіпот середнього рівня може емулювати лише певну підмножину команд SSH або SMTP. Він може приймати з'єднання, реагувати на певні команди та реєструвати дії зловмисника, але не надавати повного функціоналу справжнього сервісу.

Реалізація honeypot вимагає дуже сурової реалізації безпеки середовища, де встановлюється пастка. Якщо буде якісь недолік, зловмисним може через це спробувати попасти на основну систему. Тому часто використовуються або віртуальні середовища, або взагалі інший легкий сервер. Це забезпечує повну ізоляцію [9].

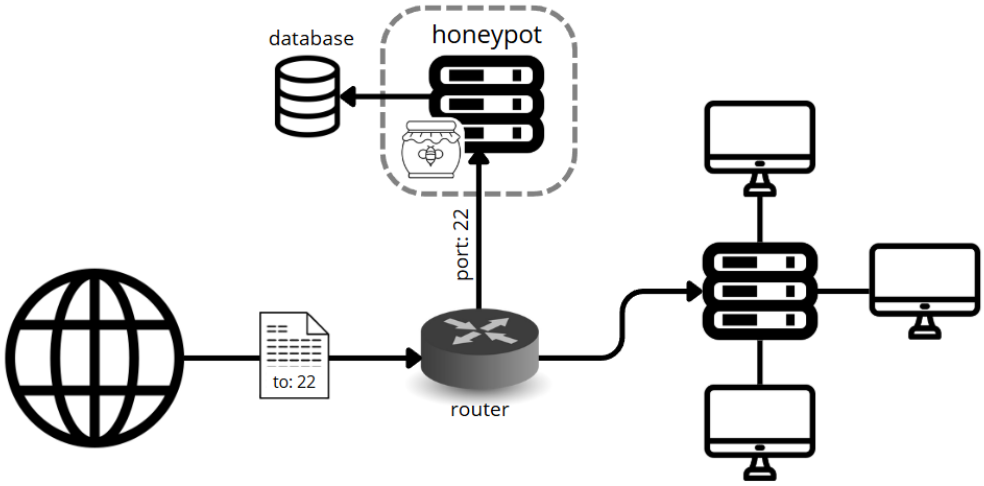

Рисунок 1.3 - Наочне відображання роботи honeypot

Також для мінімізації ризиків і підвищення користі honeypot необхідно проектувати з урахуванням кількох архітектурних вимог: жорстка ізоляція (контейнери, VLAN, гіпервізор), мінімальний набір активних сервісів, централізований збір логів у режимі append‑only та автоматичне відновлення стану після завершення сесії.

Крім того, що приманка відволікає ботів/злочинців від реального серверу, після чого може їх блокувати на основному. Головне, що вона нам дає, це дії атакуючих. Часто можна отримати нові данні, зробити за допомогою них сигнатуру та додати до бази, яку будуть використовувати IDP/IPS для виявлення вторгнень. Це показує гарну взаємодію різних методів захисту, доповнюють один одного (рисунок 1.4).

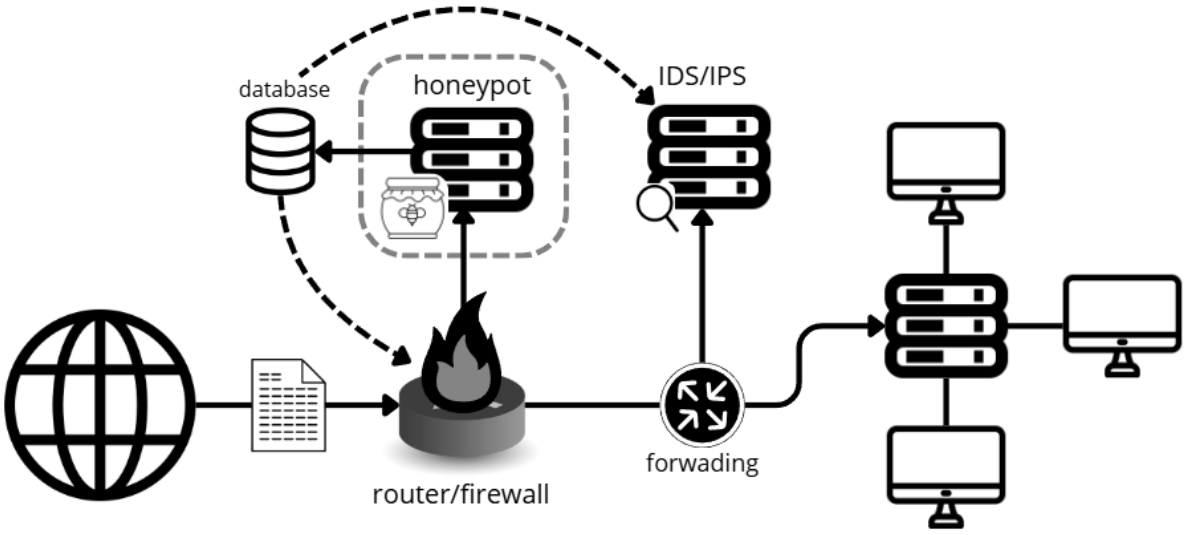

Рисунок 1.4 - Взаємодія honeypot з іншими методами захисту

Попри суттєві переваги нуль хибнопозитивних спрацювань, здатність фіксувати zero‑day‑ експлойти, економічність у стані очікування honeypot має низку обмежень. Він «бачить» лише той трафік, що спрямований безпосередньо на нього, не блокує атаки на інші вузли та, у разі високого рівня взаємодії, може стати трампліном для подальшого проникнення, якщо ізоляція реалізована неналежно. До того ж досвідчені зловмисники інколи здатні розпізнати неприродну поведінку сервера й оминати його.

1.3 Аналіз реалізованих пасток

У світі кібербезпеки існує низка готових рішень‑honeypot, кожне з яких вирішує вузьке коло завдань. Наприклад, Cowrie - це mid‑interaction SSH‑емулятор, який швидко розгорнути за допомогою Docker, і який детально логгує кожну введену команду та з'єднання в pcap‑форматі. Проте він обмежує вас лише SSH/Telnet‑сервісами та використовує жорстко закодовані банери, що ускладнює адаптацію до нових протоколів [10].

Dionaea вміє "ловити" шкідливе ПЗ через кілька стандартних протоколів (SMB, HTTP, FTP), автоматично збираючи й зберігаючи зразки експлойтів. Однак для підтримки кожного нового сервісу потрібні окремі налаштування, а емуляція прикладного рівня залишається обмеженою [11].

Honeyd дозволяє емулювати тисячі віртуальних хостів із різними ОС‑банерами, що вигідно для масштабу, але практична цінність таких пасток знижується через дуже примітивну поведінку "хостів" - глибокі сценарії атаки не відпрацьовуються [12].

У всіх вищеописаних проектах основною перешкодою є монолітна архітектура: щоб додати підтримку нового протоколу, доводиться глибоко занурюватися в код або форкати репозиторій. Більшість платформ мають власні механізми логування, часто потребують додаткових скриптів для інтеграції з SIEM, і не передбачають єдиного API для підключення нових емуляторів чи лог‑синків.

Саме ця обмеженість визначає необхідність створення модульного honeypot‑фреймворку. Він поєднає переваги існуючих рішень (детальні логи, запис сесій) з можливістю підключення будь‑якого протоколу чи механізму візуалізації/аналізу. Такий підхід дозволить швидко розгорнути нову пастку для HTTP/2, WebSocket, SSH, та інші, підключивши відповідний модуль без зміни ядра системи.

Підсумовуючи, традиційні фаєрволи, IDS/IPS і готові honeypot‑системи забезпечують різні рівні захисту, але жодне з них самостійно не дає універсального

рішення: перші блокують відоме, другі лише сигналізують про підозрілу активність, а останні часто «прихильні» лише до одного протоколу.

Проведений аналіз показав, що найбільш цінним є поєднання детального логування та гнучкості розгортання, яке пропонують mid‑interaction‑пастки. Водночас сучасні вимоги бізнесу вимагають швидкого додавання нових сервісів і централізованої інтеграції з SIEM.

Отже, логічним продовженням є розробка модульного honeypot‑фреймворку, здатного адаптуватися під будь‑які протоколи та способи аналізу. Саме ця концепція ляже в основу теоретичної частини роботи і стане основою практичної реалізації у розділі 3.

2 ТЕОРЕТИЧНА РОЗРОБКА МОДУЛЬНОЇ СИСТЕМИ ДЛЯ ЗАХИСТУ ВЕБСЕРВЕРУ

2.1 Вибір інструментальної бази та технологій

Проаналізувавши існуючи системи honeypot, було прийнято рішення створити свого власного фреймворку для реалізації пасток з урахуванням недоліків інших. Головними особливостями будуть:

– Модульність - можливість використання будь-якого протоколу та рівня емуляції

– Розширюваність - можливість додавання модулів, як додатків, щоб будь хто міг їх створювати

Система має бути розроблена таким чином, щоб розробники могли легко розуміти код, вносити зміни та додавати нові функціональні можливості. Це забезпечить прискорення процесу розробки та полегшить подальшу підтримку системи. Гнучкість та розширюваність архітектури є критично важливими для можливості інтеграції нових модулів, які підтримують різні протоколи та сценарії використання honeypot. Це дозволить системі адаптуватися до нових загроз та вимог. Не менш важливими є надійність та безпека системи. Обрані інструменти повинні забезпечувати стабільну роботу системи та мінімізувати потенційні ризики, пов'язані з безпекою самої honeypot.

Зважаючи на зазначені вимоги, було прийнято рішення використовувати мову програмування Python. Python є інтерпретованою, об'єктно-орієнтованою мовою програмування високого рівня, яка відзначається простотою синтаксису, що значно полегшує читання та розуміння коду. Крім того, Python має потужну стандартну бібліотеку та велику кількість сторонніх бібліотек, які спрощують розробку мережевих застосунків, обробку даних та виконання інших завдань, важливих для функціонування honeypot. Зокрема, існують зручні інструменти для роботи з різноманітними протоколами, представлені у вигляді готових бібліотек. Важливим

фактором є також широка популярність Python у спільноті розробників, що забезпечує доступність великої кількості ресурсів для навчання та підтримки.

Для реалізації окремих функціональностей системи було використано ряд бібліотек та модулів Python:

Paramiko - Бібліотека використовується для емуляції протоколу SSH, що є важливим для створення правдоподібної пастки для зловмисників, які намагаються отримати несанкціонований доступ до системи [13].

Loguru - для ведення логів системи використовується бібліотека за її простоту та функціональність [14].

Socket - модуль забезпечує можливість роботи з мережевими з'єднаннями, що є основою для взаємодії honeypot з атакуючими.

Pty - для емуляції терміналу використовується модуль, який дозволяє створити псевдо-термінал для імітації взаємодії з операційною системою.

Os - модуль надає засоби для взаємодії з операційною системою, що необхідно для виконання певних команд та операцій.

Threading - Для реалізації багатопоточності при обробці з'єднань використовується модуль, який дозволяє одночасно обробляти декілька підключень до honeypot.

Uuid - Генерація унікальних ідентифікаторів.

Yaml - для роботи з файлами конфігурації використовується модуль, який дозволяє зручно зберігати та змінювати налаштування системи.

Файли конфігурації системи використовують формат YAML. YAML є людино-читаємим форматом серіалізації даних, який забезпечує зручне зберігання та зміну налаштувань, таких як параметри мережі, вибір модулів та шляхи до файлів.

Вибір зазначених інструментів та технологій обумовлений їхньою здатністю забезпечити необхідний рівень модульності, розширюваності та простоти розробки, що є критично важливим для створення ефективної системи honeypot.

2.2 Методологія реалізації модульності

Головна ідея створення додатку honeypot, що відрізняє від інших, це створення модульності. Це реалізується за допомогою використання Об'єктно-орієнтованого програмування (ООП) та принципів SOLID які є продовженням ООП.

Щоб було легше розуміти задумку, проведемо аналогію використання модульності у моделі мережі. Використаємо модель середню від TCP/IP та OSI як приклад. Ми знаємо, що у мережі є певна кількість рівнів для передавання інформації через неї. Кожен рівень виконує свою функцію, а реалізація цих рівнів виконується за допомогою протоколів. Протоколів є багато дуже різних і на кожен рівень свої. Головна особливість, яка нас цікавить, це можливість використання та зміна будь-якого протоколу, не змінювавши попередні або слідуючи.

Тобто, хоча в нас і багато протоколів одного рівня, ми можемо вибрати один і він буде вірно функціонувати, бо кожен протокол зроблено модульно і незалежно один від іншого (рисунок 2.1)

Рисунок 2.1 - Модульна модель OSI-TCP/IP

Для забезпечення гнучкості та розширюваності системи honeypot було розроблено модульну архітектуру. Модульність досягається шляхом поділу функціональності системи на окремі компоненти, кожен з яких відповідає за певну задачу. Це дозволяє легко додавати, видаляти або замінювати модулі, не впливаючи на роботу інших частин системи.

В основі модульної архітектури лежить принцип використання інтерфейсів. Кожен модуль реалізує певний інтерфейс, який визначає набір методів та властивостей, що модуль повинен надавати. Це дозволяє різним модулям взаємодіяти один з одним, не знаючи конкретної реалізації кожного модуля.

Наприклад, для обробки різних протоколів можуть бути розроблені окремі модулі, кожен з яких реалізує інтерфейс "Сервер". Модуль "SSHServer" відповідає за обробку протоколу SSH, а для обробки інших протоколів можуть бути створені аналогічні модулі.

Аналогічно, для емуляції терміналу використовуються модулі, які реалізують інтерфейс "EmulateTerminal". Модуль "BashEmulateTerminal" емулює термінал Bash, але можуть бути розроблені й інші модулі для емуляції інших типів терміналів.

Для виконання команд використовуються модулі, які реалізують інтерфейс "Executor". Модуль "LocalExecutor" виконує команди локально в операційній системі, на якій запущено honeypot.

Реєстрація та конфігурація модулів здійснюється за допомогою файлу конфігурації у форматі YAML. Це дозволяє легко налаштовувати систему, вибираючи, які модулі використовувати для кожної задачі.

Такий підхід забезпечує ряд переваг:

Гнучкість: Можливість легкої заміни модулів дозволяє адаптувати систему до різних потреб та сценаріїв використання.

Розширюваність: Додавання нових модулів дозволяє розширити функціональність системи без необхідності зміни існуючого коду.

Підтримуваність: Модульна архітектура полегшує розробку, тестування та підтримку системи, оскільки кожен модуль може розроблятися та тестуватися незалежно.

Ця методологія реалізації модульності є ключовою для забезпечення гнучкості та розширюваності розробленої системи honeypot.

Після аналізу було обґрунтовано вибір інструментальної бази та методології розробки модульної системи honeypot. Зокрема, було визначено, що мова програмування Python та ряд спеціалізованих бібліотек є оптимальним вибором для реалізації системи, забезпечуючи необхідну гнучкість, розширюваність та простоту розробки.

Крім того, була розроблена методологія модульності, яка базується на використанні інтерфейсів, розподілі функціональності на окремі модулі та конфігурації. Такий підхід дозволяє легко адаптувати систему до різних потреб та сценаріїв використання, додавати нові функціональні можливості та спрощує процес розробки та підтримки.

3 ПОБУДОВА ТА АНАЛІЗ МОДУЛЬНОГО HONEYPOT

3.1 Архітектура та реалізація програмного продукту

Перед написанням коду, було би не погано мати уяву про те, які саме модулі будуть у нашому додатку. Так як ідея у тому, щоб створювати пастки з різних протоколів, один із модулів буде називатися “Server” та мати роль реалізації самого протоколу або сервісу. Це основний модуль навколо якого будуть реалізовуватися всі інші.

Так, як нам треба імітувати сервіси різних рівнів взаємодії, треба реалізувати такий модуль, якого буде приймати наш Server і в залежності від реалізації у середині буде імітувати по різному. Такий модуль буде мати назву “Emulate_terminal” поки що. У подальшому краще назвати його більш загально.

У більшості випадків, наш модуль емуляції взаємодії, має виконувати команди, які захоче ввести зловмисним. Краще виділити виконання і аналіз в інший модуль, який буде мати назву “Executor”. В залежності від реалізації, він теж зможе мати вплив на рівень взаємодії.

Далі нашу пастку краще всього запускати з іншого модуля “Starter”, це дає нам більшої гнучкості, легкість, та запуск декількох пасток зразу, якщо це буде потрібно.

Тепер саме головне. Наш honeypot не буде нести користі, якщо все, що в ньому трапляється не записувати, тобто логувати. Тому обов’язковим є наявність модуля “Logger”, який буде виконуватися у всіх інших модулях.

І додаткова дуже корисна можливість створити модуль відправки вузької кількості даних до інших джерел приймання за необхідністю. Наприклад, до пастки зайшов хтось, це зразу показує, що це або бот, або зловмисним, тому ми б хотіли за допомогою отриманих даних (IP) зробити наш сервер більш захищеним. Можна передавати айпі через модуль “Output” наприклад до сокету, а потім блокувати його на брандмауері. Це як варіант, загалом цей модуль можна використовувати для багато чого ще, наприклад у телеграм повідомляти. Головна мета, повідомляти про щось.

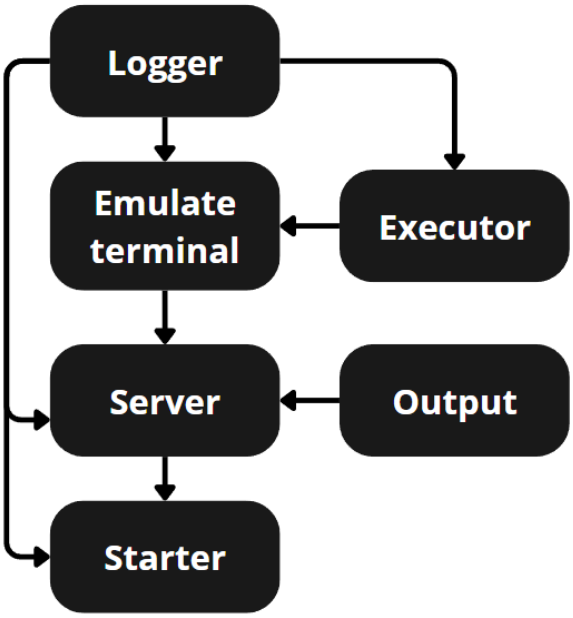

Рисунок 3.1 - Основні модулі додатку honeypot

Після того, як основний вигляд додатку створено, можна починати писати код. Важливо притримуватися правил ООП та концепцій SOLID для створення правильної модульності. Як саме?

Перш за все зробимо чіткі визначення:

Модуль - це самостійна частина системи, яка інкапсулює певну функціональність. Модуль може містити одну або декілька служб.

Служба - це конкретна реалізація певної функціональності в межах модуля. Служба надає певну функцію або набір функцій, які можуть бути використані іншими частинами системи. Загалом буде реалізовано за допомогою класів.

Інтерфейс - дозволені методи служби, за допомогою яких інші модулі або служби можуть взаємодіяти з цією. Служби одного модуля мають однакові інтерфейси.

Всі служби будуть реалізовані за допомогою класів, які можна легко розширювати додаванням методів або, якщо буде необхідність створити іншу версію служби, то можна їх успадковувати.

Створення різних реалізацій виконується за допомогою абстрактних класів. Абстрактний клас - правила створення класу, тобто те, які інтерфейси мають бути і правила реалізація. Саме на цих абстракціях зроблена найголовніша частина модульності. За допомогою них, можна розширювати honeypot використовуючи будь-які протоколи, логери, емуляції. Головне, щоб реалізація служби відповідала правилам. Дуже важливо створення всіх модулів повністю незалежно від інших.

Перш за все треба створити папку проекту. Буде називатися honeypot, до якої ми додаємо директорії, щоб простіше було компанувати модулі. Структура:

```
honeypot/
├── config/
├── emulate_terminal/
├── executor/
├── logger/
├── output/
├── server/
├── starter/
├── main.py
├── README.md
├── requirements.txt
└── utils.py
```

По черзі будемо реалізовувати кожен модуль. Config не вважається модулем, просто зручне налаштування. Почнемо із server

Модуль Server має мати абстрактний клас BaseServer у файлі base_server.py:

```
from abc import ABC, abstractmethod
import socket

class BaseServer(ABC):
    @abstractmethod
    def handle_client(self, client: socket.socket):
        pass
    @abstractmethod
    def get_name(self) -> str:
        pass
```

Головним інтерфейсом (методом) є handle_client який буде використовуватися у стартері модуля Starter. Він має обробляти сокет підключення. Get_name просто повертає назву служби, наприклад SSH, HTTP.

Так, як першим сервісом паски буде реалізовано SSH, для якого потрібно кілька додаткових файлів та ключів, зробимо додаткову директорію для зручності ssh/.

За допомогою модуля paramiko створюється ServerInterface для ssh. Просто успадкувавши paramiko.ServerInterface, створивши свій SSHServerInterface перевизначаються методи які конфігурують обробку підключень ssh. Наприклад :

```
    class SSHServerInterface(paramiko.ServerInterface):
        def __init__(self, logger: BaseLogger, login: str = '',
password: str = ''):
            self.login = login
            self.password = password
            self.logger = logger

            self.event = threading.Event()
        def get_allowed_auths(self, username):
            return "password"
```

Get_allowed_auths визначає яким способом буде автентифікація виконуватися, ставимо завжди паролем. Також зауважимо, що при ініціалізації приймається об’єкт BaseLogger. Це наш логер, який буде реалізовано.

Далі вже створюємо свою реалізацію обробки з’єднання по ssh, створивши клас SSHServer успадкований від BaseServer у файлі ssh_server.py, який приймає для ініціалізації об’єкти BaseLogger, BaseEmulateTerminal, BaseOutput. Для використання цих модулів всередині.

Код файлу відображено у Додатку А

Структура модулів стає більш розвиненою:

```
├── server
│   ├── base_server.py
│   ├── __init__.py
│   └── ssh
│       ├── __init__.py
│       ├── ssh_interface.py
```

```
│       ├── ssh_keys
│       │   ├── fake_ssh_host_key
│       │   └── fake_ssh_host_key.pub
│       └── ssh_server.py
```

Для ініціалізації paramiko треба ще згенеровані ключі за допомогою команди ssh_keygen.

Перший модуль створено, з іншими виконуються ті самі методи, але під свої потреби. Коди всіх файлів у Додатку А

Далі для кожного модуля створюються свої Абстрактні класи, які починаються зі слова Base… BaseLogger, BaseStarter, BaseExecutor, BaseOutput, BaseEmulateTerminal. Описуються інтерфейси для кожного класу та створюється своя реалізація. Наприклад, для BaseLogger створено декілька реалізацій. Для логування тексту до файлу за допомогою loguru у класі FileLogger, логування у базу даних sqlite DataBaseLogger, та об’єднаний, де дві ці реалізації виконуються разом DataBaseFileLogger.

Після реалізації всіх модулів налаштовується файл main.py для запуску всього проекту підключивши файл конфігурації config.yaml. У конфігі можна вибрати які саме служби служби використовувати для кожного модуля і ввести додаткові параметри, наприклад на якому порті запустити додаток або де зберігати файли логів.

Частина config.yaml:

```
client:
    login: null
    password: null
server:
    host: "0.0.0.0"
    port: 2222
    listen_number: 1 # no need
service:
    emulate_terminal: "BashEmulateTerminal"
    executor: "LocalExecutor" # SimpleExecutor no need
    output: "FileOutHandler" # FileOutHandler,
SocketOutHandler | in future:(TelegramOutHandler)
```

```
        logger: "DataBaseFileLogger" # FileLogger,
DataBaseLogger
        server: "SSHServer" # in future: (HTTPServer, FTPServer)
        starter: "Starter"
```

Можна побачити, що yaml простий формат для розуміння. Далі все переходить у main.py. Де створюються об’єкти класів і з’єднуються один з одним.

Як тільки всі файли на своїх місцях і все вірно налаштовано можна запускати honeypot.

3.2 Сценарії використання та функціонування системи

Один з найголовніших принципів розгортання honeypot, це його ізольованість. Додаток створений відкритим, без внутрішньої ізоляції. Тому треба подумати де його запускати. Головне використовувати або контейнери, віртуальну машину або віддалений сервер.

Контейнер - легкий та швидкий варіант розгортання, але немає повної системи, від чого емуляція може бути не повною. Також захист та ізоляція не абсолютна, для ssh цей варіант не підходить.

Віртуалізація - при вірних налаштуваннях є повністю ізольованою, та має повною систему, яку встановимо. Гарний варіант для пастки, але буде працювати тільки поки працює машина.

Сервер - мається на увазі чистий ізольований сервер. Орендувати наприклад на server.ua та встановити пастку туди. Повна ізоляція, повна система і постійна робота. Найкращий варіант.

Було орендовано сервер на digitalocean.com та перенесено додаток через команду `scp`. Після чого порт для реального ssh змінений на 1022, а для honeypot у конфігурації поставлено 22.

Далі йде встановлення бібліотек за допомогою менеджера pip. Після чого можна запустити пастку за допомогою команди `python3 main.py`.

Після запуску додатка він буде чекати підключення по ssh. Для перевірки підключимося з іншого терміналу (рисунок 3.2)

```
etrig@debian:~$ ssh root@127.0.0.1
root@127.0.0.1's password:
Welcome to Ubuntu 22.04.3 LTS (GNU/Linux 5.15.0-89-generic x86_64)
root@debian:/home#
```

Рисунок 3.2 - Підключення до honeypot по ssh

Зразу запитує пароль, та який б він не був, завжди впускає. Дії підключення та введення паролю логуються як у файл так і базу (рисунок 3.3).

```
2025-05-19 02:35:39.025 | WARNING  | logger.file_logger:log:23 - message: New co
nnection: 127.0.0.1:46842, client - <socket.socket fd=8, family=2, type=1, proto
=0, laddr=('127.0.0.1', 22), raddr=('127.0.0.1', 46842)>, data: [ {'ip': '127.0.
0.1', 'port': 46842, 'session_id': '722f6ac0-9158-4c28-9cc4-1c6ff4d07946', 'even
t_type': 'SSH connect', 'command': None, 'timestamp': '2025-05-18T23:34:38.04422
7'} ]
2025-05-19 02:35:43.224 | INFO     | logger.file_logger:log:23 - message: Bot en
tered username: root, password: test, data: [ {'ip': '127.0.0.1', 'port': 46842,
 'session_id': '722f6ac0-9158-4c28-9cc4-1c6ff4d07946', 'event_type': 'SSH login'
, 'command': None, 'timestamp': '2025-05-18T23:35:43.220232'} ]
```

Рисунок 3.3 - Логування підключення та введеного паролю

Даля бот або зловмисник має можливість використовувати будь-які команди, бо це високий рівень взаємодії (рисунок 3.4).

```
root@debian:/home# cd ..
root@debian:/# ls
bin   etc         initrd.img.old  lost+found  opt   run   sys  var
boot  home        lib             media       proc  sbin  tmp  vmlinuz
dev   initrd.img  lib64           mnt         root  srv   usr  vmlinuz.old
root@debian:/# whoami
root
root@debian:/# echo "hello"
hello
root@debian:/#
```

Рисунок 3.4 - Введення команд до honeypot

Кожна команда теж відстежується (рисунок 3.5).

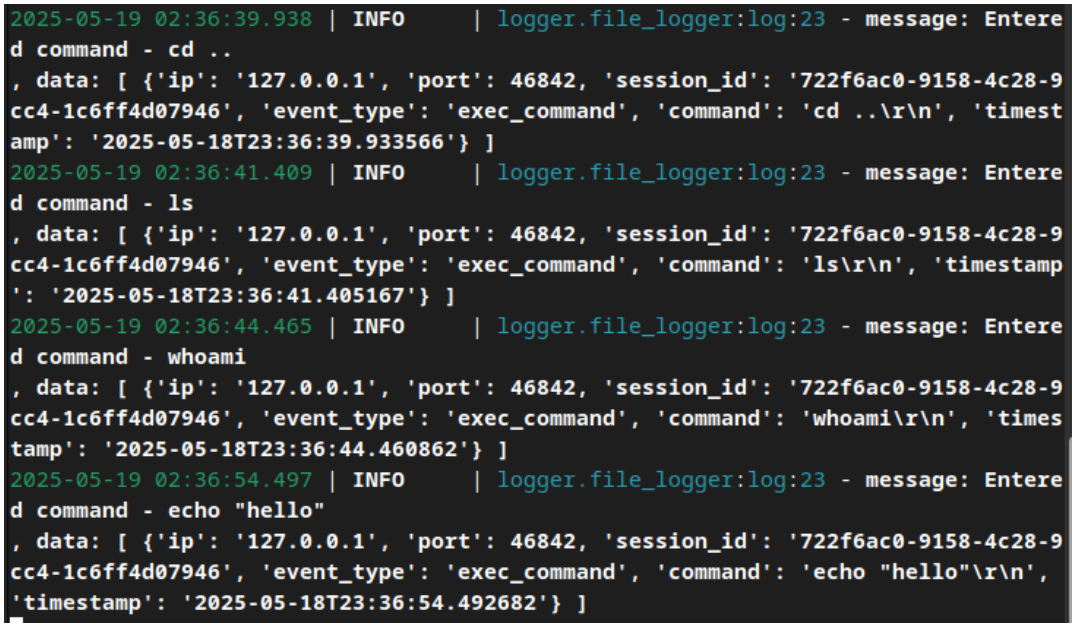

Рисунок 3.5 - Запис введених команд

Все працює, відстежується кожна дія зловмисника. Тепер треба вирішити одну проблему. Всі логи зберігаються на сервері, де можуть бути видалені, треба їх у реальному часі передавати кудись на зберігання, бо сервер може зламатися від дій злочинця. Було використано віддільний програмний додаток написаний на Python для постійного передавання бази даних на свою машину. Але можна біло використати bash скрипт або інші методи.

Залишаємо honeypot активним для збору інформації.

3.3 Аналіз спроб проникнення ботів

Після активної роботи honeypot. Було отримано 4407 записів у базі даних за 24 години, з них 243 унікальних ip адрес.

Більшість ботів просто заходити підбиравши пароль та зразу від’єднувалися, а саме 4402 записи. Тільки 5 записів були з командами, які боти намагалися виконати. Один за них був з айпі 134.122.95.85, можна подивитися данні про нього на abuseipdb.com (рисунок 3.6)

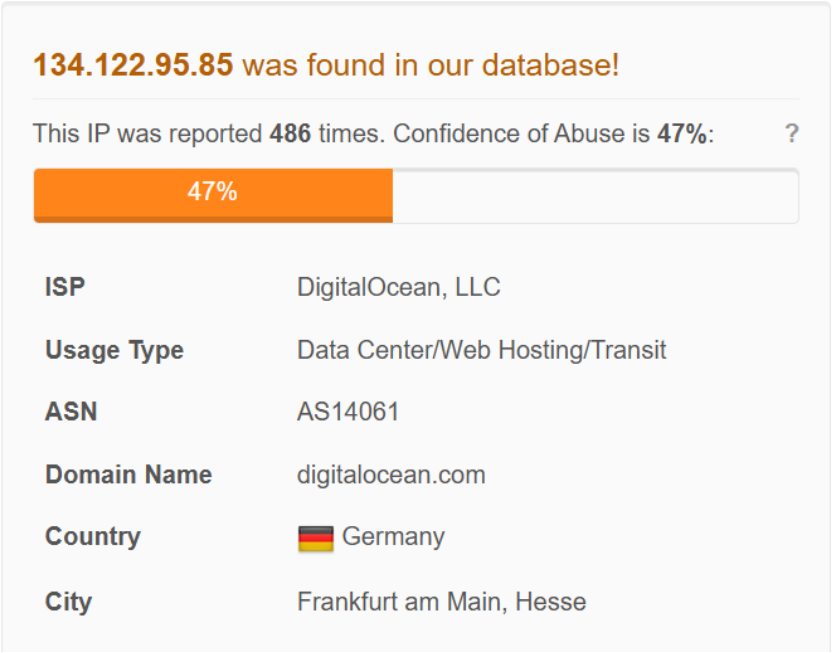

Рисунок 3.6 - Данні про ip адресу на abuseipdb.com

Розглянемо один з цікавих випадків виконання команд на honeypot: `nohup bash -c "exec 6<>/dev/tcp/134.122.95.85/60139 && echo -n 'GET /linux' >&6 && cat 0<&6 > /tmp/nuGE2b0fPO && chmod +x /tmp/nuGE2b0fPO && /tmp/nuGE2b0fPO BkY0WwYFRy5MAhlNNU8BB0QzTBcPWzJEDhlHMEcZBUQ1TwEHRDBCFw9bMkQHGUQwTRkGQTdPAQdEM0MXA0IuRwQBWzFDBBlHNUUNAUUxRQcXRDFMGQNALkcHBVs1RQ0BRTFBAxdBN1sFBEIuQgcZRDVNDQFFMUcDF0E3WwUEQy5EAgRbMUIBDUMwRAUOVTFMAxlFLkQOA1sxRwcNQzBEAwdVNEIZBUY3WwUHWzFMAg1DMEQDBFU4WwUGQi5EBgVbMUUBDUMwRAUDVTRCGQVGNlsFB0EuRA8HTzZFBgZEgKbN07cF" &`

Проаналізувавши команду, можна зробити висновки, що бот намагається підключитися до себе, отримати данні, зберегти у тимчасовий файл та зробити його виконавчим. Після чого запустити його із заданим параметром у фоновому режимі, щоб його складніше було знайти.

Ця спроба злому показує стандартну техніку отримання доступу. Більшість ботів, які виконують команди, а не тільки сканують порти, виконують саме такі дії, бо це найкращий спосіб створити backdoor на сервер.

Це тільки невеличка частина, що може зібрати honeypot. Якщо залишити його на довго і зробити вхід більш реалістичним, можна отримати більше різноманітних даних, які будуть цікаві для забезпечення захищеності серверів.

Було показано, що додаток виконує свої функції, але поки що тільки із протоколом ssh, що є тільки початком. У подальшому є необхідністю розширювати використання інших протоколів та систем вебсерверів для більшого отримання різноманітної інформації, яка покращить захист стандартних систем. Крім того, доступна лише високий рівень взаємодії пастки, що добре, але не завжди є необхідністю, додаткові реалізації модулів виконання команд у подальшому дадуть вибір рівнів. Також важно зауважити, що отримання даних не повністю автономне, так як вони зберігаються на системі емуляції, їх треба запирати у реальному часі, що є першочерговою проблемою, яку треба вирішити.

Підсумовуючи, було розроблено та використано модульний додаток honeypot для створення ssh пасток. Розміщення додатку було на віддаленому сервері з великим рівнем взаємодії. Пастка працювала 24 години, за цей час вдалося зібрати інформацію про проникнення до сервісу, а саме підбір паролів та введення шкідливих команд до терміналу. Далі було проаналізовано ip адреси ботів та команди. Для більшої різноманітності даних треба розташовувати пастку на більший період часу. Було зроблено аналіз покращень.

ВИСНОВКИ

Захист вебсерверів є необхідністю. З кожним роком кількість та складність кіберінцедентів та атак збільшується, що визиває необхідність отримання та аналізу нових методів зламу. Було проаналізовано стандартні засоби захисту вебсерверів, як Firewall, IDS та IPS, які є найпоширенішими. Було знайдено недолік у необхідності оновлення бази вже виявлених атак, створених з них сигнатур для кращого захисту. Головне питання де брати такі записи. Тому було проаналізовані додаткові методи доповнення захисту вебсерверів, а саме використання пасток honeypot.

Використання пасток надає можливість отримувати нові методи та данні про атаки та інциденти, які доповнюють захист стандартними системами захисту. Проаналізувавши альтернативні додатки honeypot, було виявлено, що майже всі з них вузько направлені та не надають можливість змінювати одним інтерфейсом використання протоколів та серверів. Для отримання найновішої та найбільшої кількості даних, додаток має буди розширюванним, щоб підтримувати будь-які протоколі та нові теж. Також для більшого налаштування додаток має бути модульним. Це дозволить розбити обов’язки кожної частини додатку, що дає більшу гнучкість та варіації реалізацій.

Модульність досягається за допомогою концепцій ООР та SOLID. За допомогою чого було створено модульну архітектуру додатку із можливістю розширення. Після чого додаток було випробувано та встановлено на сервер з ізоляцією для прийому ssh з’єднань.

Було отримано певну кількість даних на проникнення та виконання команд ботів/зловмисників. Команди було проаналізовано та отримано висновки щодо методів атак.

Після повного випробування було проаналізовані можливі покращення та додатки до реалізованого модульного honeypot. Система вже може гарно функціонувати та має потенціал.

ПЕРЕЛІК ПОСИЛАНЬ

1. Державна служба спеціального зв’язку та захисту інформації України. URL: https://cip.gov.ua/ua/news/cert-ua-minulogo-roku-opracyuvala-4315-kiberincidentiv.

2. Boiko V., Vasilenko N., Slatvinska V. Distributed Systems Log Protection from Cyberattacks by Verkle Trees // Information Technology for Education, Science, and Technics. Springer Nature Switzerland, 2024. P. 221-234.

3. Бойко В.Д., Ніякий О.О. Методи аналізу та захисту від кібератак. Методи аналізу поведінки кіберзлочинця // Кібербезпека в сучасному світі: актуальні виклики: матеріали V Міжнародної науково-практичної конференції (м. Одеса 22 листопада 2024р.). 2024. P. 233-235.

4. Бойко В., Горбаченко С. Тестування на проникнення як ефективний інструмент менеджменту кібербезпеки // Інформаційні технології та суспільство. 2023. № 3 (9). P. 23-29.

5. What is a firewall?. *Cisco*. URL: https://www.cisco.com/site/us/en/learn/topics/ security/what-is-a-firewall.html

6. GeeksforGeeks. Intrusion detection system (IDS) - geeksforgeeks. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/intrusion-detection-system-ids/

7. GeeksforGeeks. Intrusion prevention system (IPS) - geeksforgeeks. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/intrusion-prevention-system-ips/

8. What is a honeypot and how does it work?. *Norton*. URL: https://us.norton.com/blog/iot/what-is-a-honeypot

9. What is a honeypot? Meaning, types, benefits, and more | fortinet. URL: https://www.fortinet.com/resources/cyberglossary/what-is-honeypot.

10. Welcome to Cowrie's documentation! - cowrie 2.6.1 documentation. URL: https://docs.cowrie.org/en/latest/

11. The honeynet project. URL: https://www.honeynet.org/

12. Honeyd general information. Honeyd. URL: https://www.honeyd.org/general/

13. Welcome to paramiko! - paramiko documentation. URL: https://www.paramiko.org/

14. Table of Contents - loguru documentation. URL: https://loguru.readthedocs.io/en/stable/overview.html

15. Honeypot diploma Niiakyi Oleksandr https://github.com/sasha-niiakyi/honeypot

Додаток А. Повна структура проєкту

```
.
├── config
│   ├── config.py
│   ├── config.yaml
│   └── __init__.py
├── emulate_terminal
│   ├── base_emul_term.py
│   ├── bash_emul_term.py
│   └── __init__.py
├── executor
│   ├── base_executor.py
│   ├── __init__.py
│   ├── local_executor.py
│   ├── simple_executor.py
│   └── ssh_executor.py
├── logger
│   ├── base_logger.py
│   ├── database
│   │   └── honey.db
│   ├── data_log.py
│   ├── db_file_logger.py
│   ├── db_logger.py
│   ├── file_logger.py
│   ├── __init__.py
│   ├── log_config.py
│   └── logs
│       └── debug.log
├── output
│   ├── base_out.py
│   ├── file_out.py
│   ├── __init__.py
│   ├── out.txt
│   ├── socket_out.py
│   ├── telegram_out.py
│   └── terminal_out.py
├── server
│   ├── base_server.py
│   ├── __init__.py
│   └── ssh
│       ├── __init__.py
│       ├── ssh_interface.py
│       ├── ssh_keys
│       │   ├── fake_ssh_host_key
│       │   └── fake_ssh_host_key.pub
│       └── ssh_server.py
├── starter
│   ├── base_starter.py
│   ├── __init__.py
```

```
│   └── starter.py
├── dockerfile
├── main.py
├── README.md
├── requirements.txt
└── utils.py
```

Файл config.yaml

```
client:
    login: null
    password: null

server:
    host: "0.0.0.0"
    port: 2222
    listen_number: 1 # no need

service:
    emulate_terminal: "BashEmulateTerminal"
    executor: "LocalExecutor" # SimpleExecutor no need
    output: "FileOutHandler" # FileOutHandler, SocketOutHandler | in
future:(TelegramOutHandler)
    logger: "DataBaseFileLogger" # FileLogger, DataBaseLogger
    server: "SSHServer" # in future: (HTTPServer, FTPServer)
    starter: "Starter"

emulate_terminal:
    color: False #no need
    time_to_block: 3600 # seconds

socket_out:
    server_ip: "127.0.0.1"
    server_port: 2525

path:
    ssh_keys_path: "server/ssh/ssh_keys/fake_ssh_host_key"
    file_out_path: "output/out.txt"
    logging_path: 'logger/logs/honey.db'
```

Файл base_starter.py

```
from abc import ABC, abstractmethod

class BaseStarter:
@abstractmethod
def start_server(self):
     pass
```

Файл base_server.py

```
from abc import ABC, abstractmethod
import socket


class BaseServer(ABC):

@abstractmethod
def handle_client(self, client: socket.socket):
     pass

@abstractmethod
def get_name(self) -> str:
     pass
```

Файл base_emule_term.py

```
from abc import ABC, abstractmethod


class BaseEmulateTerminal(ABC):
@abstractmethod
def send(self, data: str):
     pass

@abstractmethod
def run_term(self, pwd: str):
     pass
```

Файл base_logger.py

```
from typing import Optional, Union
from abc import ABC, abstractmethod

from .data_log import DataLog

class BaseLogger(ABC):

@abstractmethod
def log(self, message:Optional[str], data:Optional[Union[DataLog,
dict]] = None, level:Optional[str] = 'INFO'):
     pass
```

```
    @abstractmethod
    def set_datalog(self, datalog: DataLog):
        pass

    @abstractmethod
    def update(self, **kwargs):
        '''Update DataLog'''
        pass

    @abstractmethod
    def get_session_id(self) -> str:
        pass
```

Файл data_log.py

```
    from dataclasses import dataclass, asdict, field
    from typing import Optional
    from datetime import datetime
    import uuid

    @dataclass
    class DataLog:
    ip: str
    port: int
    session_id: uuid.UUID = field(default_factory=uuid.uuid4)
    event_type: Optional[str] = None
    command: Optional[str] = None
    timestamp: str = datetime.utcnow().isoformat()

    def update(self, event_type: Optional[str] = None, command:
Optional[str] = None):
        self.event_type = event_type or self.event_type
        self.command = command or self.command
        self.timestamp = datetime.utcnow().isoformat()

    def get_session_id(self) -> str:
        return str(self.session_id)

    def to_dict(self):
        return {
            **asdict(self),
            "session_id": str(self.session_id)
        }

    def __str__(self):
        return f'''{self.timestamp} | {self.event_type} from
{self.ip}:{self.port}
```

```
          | session: {self.session_id} | command:
{self.command}'''.replace('\n', '').replace('\t', '')
```

Файл base_executor.py

```
from abc import ABC, abstractmethod

class BaseExecutor:

@abstractmethod
def execute(self, command: str):
     pass
```

Файл base_output.py

```
from abc import ABC, abstractmethod
import socket

class BaseOutHandler(ABC):

@abstractmethod
def notify(self, data: str, end: str = '\n'):
     pass

@abstractmethod
def log_ip(self, ip: str, end: str = '\n'):
     pass

@abstractmethod
def log_session_id(self, session_id: str, client: socket.socket):
     pass
```

Файл starte.py

```
import socket
import threading

from .base_starter import BaseStarter
from server import BaseServer
from logger import BaseLogger, DataLog

class Starter(BaseStarter):
def __init__(self, server: BaseServer, logger: BaseLogger,
host="0.0.0.0", port=2222):
```

```
            self.host = host
            self.port = port
            self.socket = (host, port)
            self.server = server
            self.logger = logger

            self.session_lock = threading.Semaphore(1)

      def start_server(self, listen_number: int = 100):
            server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,
1)
            server_socket.bind(self.socket)
            server_socket.listen(listen_number)

            #self.logger.log(f"Server {self.server.get_name()} running at
{self.host}:{self.port}", level='DEBUG')

            while True:
                  client, addr = server_socket.accept()

                  #logging
                  datalog = DataLog(ip=addr[0], port=addr[1], event_type='SSH
connect')
                  self.logger.set_datalog(datalog)
                  self.logger.log(f"New connection: {addr[0]}:{addr[1]},
client - {client}", level='WARNING')

                  threading.Thread(
                        target=self._handle_with_lock,
                        args=(client,)
                  ).start()

      def _handle_with_lock(self, client):
            with self.session_lock:
                  self.server.handle_client(client)
```

Файл ssh_server.py

```
      import socket
      import paramiko
      import threading
      import uuid

      from .ssh_interface import SSHServerInterface
      from emulate_terminal import BaseEmulateTerminal
      from executor import BaseExecutor
```

```
from output import BaseOutHandler
from server import BaseServer
from logger import BaseLogger


class SSHServer(BaseServer):
def __init__(self,
     path_key: str,
     emul_term: BaseEmulateTerminal,
     executor: BaseExecutor,
     out: BaseOutHandler,
     logger: BaseLogger,
     login: str = '',
     password: str = '',
     server_banner: str = None,
     protocol_banner: str = None
):
     self.emul_term = emul_term
     self.executor = executor
     self.out = out
     self.logger = logger
     self.host_key = paramiko.RSAKey(filename=path_key)
     self.server_interface = SSHServerInterface(logger, login=login,
password=password)
     self.name = 'SSH'

     if server_banner:
          self.server_banner = server_banner
     else:
          self.server_banner = "Welcome to Ubuntu 22.04.3 LTS
(GNU/Linux 5.15.0-89-generic x86_64)\n"

     if protocol_banner:
          self.protocol_banner = protocol_banner
     else:
          self.protocol_banner = "SSH-2.0-OpenSSH_8.9p1 Ubuntu-
3ubuntu0.5"

def get_name(self):
     return self.name

def set_protocol_banner(self, new_banner: str):
     self.protocol_banner = new_banner

def get_protocol_banner(self):
     return protocol_banner

def set_server_banner(self, new_banner: str):
     self.server_banner = new_banner
```

```
    def get_server_banner(self):
        return self.server_banner

    def _setup_transport(self, client: socket.socket):
        transport = paramiko.Transport(client)
        transport.add_server_key(self.host_key)
        transport.local_version = self.protocol_banner

        return transport

    def _start_server_interface(self, transport: paramiko.Transport):
        transport.start_server(server=self.server_interface)

        return self.server_interface

    def _accept_channel(self, transport: paramiko.Transport, seconds_wait:
int = 20) -> paramiko.Channel:
        # wait client
        return transport.accept(seconds_wait)

    def _wait_shell(self, server_interface: SSHServerInterface, timeout:
int):
        server_interface.event.wait(timeout)
        return server_interface.event.is_set()

    def gen_log(self, session_id: str, data: str) -> str:
        return f"Client: {session_id}, data: {data}"

    # # create another class Session
    # def generate_session_id(self, socket: list):
    #     return uuid.uuid5(uuid.NAMESPACE_DNS, f'{socket[0]}:{socket[1]}')

    def handle_client(self, client: socket.socket, seconds_wait: int = 20):
        socket = client.getpeername()
        self.out.log_ip(socket[0])
        session_id = self.logger.get_session_id()
        self.out.log_session_id(session_id, socket)

        transport = self._setup_transport(client)
        server_interface = self._start_server_interface(transport)

        channel = self._accept_channel(transport, seconds_wait)
        if channel is None:
            return

        if not self._wait_shell(server_interface, seconds_wait):
            return
```

```
        self.emul_term.set_channel(channel)
        self.emul_term.send(self.server_banner)
        self.emul_term.run_term(pwd='/home/{server_interface.login}')

        self.logger.update(event_type='SSH disconnect', command=None)
        self.logger.log(f'Bot disconnected')

        channel.close()
        transport.close()
```

Файл bash_emul_term.py

```
import pty
import os,sys
import select
import termios
import tty
import time

import paramiko

from .base_emul_term import BaseEmulateTerminal
from logger import BaseLogger

class BashEmulateTerminal(BaseEmulateTerminal):

def __init__(
        self,
        logger: BaseLogger,
        channel: paramiko.Channel = None,
        time_to_block: int = 600,
        color: bool = False
):
        self.pwd = '/home/root'
        self.channel = channel
        self.master_fd = None
        self.logger = logger
        self.time_to_block = time_to_block
        self.now = time.time()
        self.color = color
        self.buffer = ""

def set_channel(self, new_channel: paramiko.Channel):
        self.channel = new_channel

def send(self, data: str):
        self.channel.send(f"{data}\r")

def clear_buffer(self):
```

```
            self.buffer = ""

      def run_term(self, pwd: str = None):
            if pwd:
                  self.pwd = pwd

            pid, self.master_fd = pty.fork()

            if pid == 0:
                  self._run_child()
            else:
                  self._run_parent()

      def _run_child(self):
            '''slave - run commands'''
            try:
                  os.chdir(self.pwd)
            except FileNotFoundError:
                  os.chdir("/home")  # fallback, если нет home

            if self.color:
                  os.execvp("bash", ["bash"])

            else:
                  os.environ["PS1"] = r"\u@\h:\w\$ "
                  os.environ["LS_COLORS"] = ""

                  os.execvp("bash", ["bash", "--norc", "--noprofile", "-i"])

      def _get_log_buffer(self, data: bytes):
            self.buffer += data.decode()
            if data == b'\r': # enter
                  self.buffer += '\n'

            if data == b'\x7f': # backspace
                  self.buffer = self.buffer[:-2]

            if '\n' in self.buffer:
                  self.logger.update(event_type='exec_command',
command=self.buffer)
                  self.logger.log(f'Entered command - {self.buffer}')
                  self.clear_buffer()

      def _check_time_block(self, rlist: list) -> bool:
            if rlist:
                  self.now = time.time()

            if (time.time() - self.now) > self.time_to_block:
                  return True
```

```
            return False

      def _run_parent(self):
            '''master - get and send data for slave'''
            try:
                  while True:
                        rlist, _, _ = select.select([self.master_fd,
self.channel], [], [], 1)

                        if self.master_fd in rlist:
                              try:
                                    output = os.read(self.master_fd, 2048)

                                    if not output:
                                          break

                                    self.channel.send(output)
                              except OSError:
                                    break

                        if self.channel in rlist:
                              try:
                                    data = self.channel.recv(1024)
                                    self._get_log_buffer(data)

                                    if not data:
                                          break

                                    os.write(self.master_fd, data)
                              except OSError:
                                    break

                        if self._check_time_block(rlist):
                              break

            finally:
                  os.close(self.master_fd)
```

Файл local_executor.py

```
      import os
      import pty

      from .base_executor import BaseExecutor

      class LocalExecutor(BaseExecutor):
```

```
    def execute(self, command: str, pwd: str):
        pid, fd = pty.fork()
        if pid == 0:
            try:
                os.chdir(pwd)
            except FileNotFoundError:
                os.chdir("/tmp")  # fallback, если нет home

            os.execvp("bash", ["bash", "-c", command])
        else:
            output = b""
            while True:
                try:
                    data = os.read(fd, 1024)
                    if not data:
                        break
                    output += data
                except OSError:
                    break
            return output.decode()
```

Файл file_logger.py

```
from typing import Optional, Union

from loguru import logger

from .base_logger import BaseLogger
from .data_log import DataLog


class FileLogger(BaseLogger):

    def __init__(self, datalog: Optional[DataLog] = None):
        self.datalog = datalog

    def set_datalog(self, datalog: DataLog):
        self.datalog = datalog

    def log(self, message: Optional[str], data: Optional[Union[DataLog,
dict]] = None, level: Optional[str] = 'INFO'):
        if data is None:
            data = self.datalog
        if isinstance(data, DataLog):
            data = data.to_dict()

        logger.log(level, f"message: {message}, data: [ {data} ]")

    def update(self, **kwargs):
```

```
            if self.datalog:
                self.datalog.update(**kwargs)

    def get_session_id(self) -> str:
            return self.datalog.get_session_id()
```

Файл file_out.py

```
    import sys, os
    import socket

    from .base_out import BaseOutHandler


    class FileOutHandler(BaseOutHandler):
    def __init__(self, path: str = 'output/out.txt'):
            self.path = os.path.join(sys.path[0], path)

    def _write(self, text: str, end: str = '\n'):
            with open(self.path, 'a') as file:
                file.write(f"{text}{end}")

    def log_session_id(self, session_id: str, socket: list):
            self.notify(f'{session_id} - {socket[0]}:{socket[1]}')

    def notify(self, data: str, end: str = '\n'):
            self._write(data, end)

    def log_ip(self, ip: str, end: str = '\n'):
            self._write(ip, end)
```

Файл main.py

```
    from starter import Starter
    from server.ssh import SSHServer
    from output import FileOutHandler, SocketOutHandler
    from executor import SimpleExecutor, LocalExecutor
    from emulate_terminal import BashEmulateTerminal
    from config import config
    from logger import FileLogger, DataLog, DataBaseLogger,
DataBaseFileLogger


    service = {
    "BashEmulateTerminal": BashEmulateTerminal,
        "LocalExecutor": LocalExecutor,
        "SimpleExecutor": SimpleExecutor,
        "SocketOutHandler": SocketOutHandler,
        "FileOutHandler": FileOutHandler,
```

```
        "FileLogger": FileLogger,
        "DataBaseLogger": DataBaseLogger,
        "DataBaseFileLogger": DataBaseFileLogger,
        "SSHServer": SSHServer,
        "Starter": Starter,
    }

    #emul_term = service[config.data.service.emulate_terminal]()
    executor = service[config.data.service.executor]()
    logger = service[config.data.service.logger]()
    emul_term = BashEmulateTerminal(
    logger=logger,
    time_to_block=config.data.emulate_terminal.time_to_block,
    color=config.data.emulate_terminal.color)

    if config.data.service.output == "SocketOutHandler":
    out = SocketOutHandler(config.data.socket_out.server_ip,
config.data.socket_out.server_port)

    if config.data.service.output == "FileOutHandler":
    out = FileOutHandler(config.data.path.file_out_path)

    if config.data.service.server == "SSHServer":
    server = SSHServer(
         config.data.path.ssh_keys_path,
         emul_term,
         executor,
         out,
         logger,
         login=config.data.client.login,
         password=config.data.client.password,
    )

    starter = service[config.data.service.starter](
    server,
    logger,
    host=config.data.server.host,
    port=config.data.server.port
    )
    starter.start_server(listen_number=config.data.server.listen_number)
```

Файл ssh_interface.py

```
    import paramiko
    import threading

    from logger import BaseLogger
```

```
    class SSHServerInterface(paramiko.ServerInterface):
        def __init__(self, logger: BaseLogger, login: str = '', password:
str = ''):
            self.login = login
            self.password = password
            self.logger = logger

            self.event = threading.Event()

        def check_channel_request(self, kind, chanid):
            if kind == 'session':
                return paramiko.OPEN_SUCCEEDED
            return paramiko.OPEN_FAILED_ADMINISTRATIVELY_PROHIBITED

        def check_auth_password(self, username, password):
            self.logger.update(event_type='SSH login')
            self.logger.log(f'Bot entered username: {username}, password:
{password}', level='INFO')

            if not self.login or not self.password:
                self.login = username
                return paramiko.AUTH_SUCCESSFUL

            if self.login == username and self.password == password:
                return paramiko.AUTH_SUCCESSFUL

            return paramiko.AUTH_FAILED

        def check_auth_publickey(self, username, key):
            return paramiko.AUTH_FAILED

        def get_allowed_auths(self, username):
            return "password"

        def check_channel_shell_request(self, channel):
            self.event.set()
            return True

        def check_channel_pty_request(self, channel, term, width, height,
pixelwidth, pixelheight, modes):
            return True
```

Файл config.py

```
    import yaml

    class DataConfig:
```

```
def __init__(self, *args, **kwargs):
     for key, value in kwargs.items():
          if isinstance(value, dict):
               setattr(self, key, DataConfig(**value))
          else:
               setattr(self, key, value)

def to_dict(self):
     return self.__dict__

def __str__(self):
     return f"{self.__dict__}"

def __repr__(self):
     return f"{self.__dict__}"


class Config:
_instance = None

def __new__(cls, *args, **kwargs):
     if not cls._instance:
          cls._instance = super().__new__(cls)
     return cls._instance

def __init__(self, config_path: str = 'config/config.yaml'):
     self.config_path = config_path
     temp_dict_data = self._load_config()
     self.data = self._dict_to_attrs(temp_dict_data)

def _load_config(self) -> dict:
     with open(self.config_path) as file:
          return yaml.safe_load(file)

def _dict_to_attrs(self, dict_attrs: dict):
     return DataConfig(**dict_attrs)


config = Config()
```