МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ
ФАКУЛЬТЕТ МЕХАТРОНІКИ ТА КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

Розроблення програмного забезпечення для виявлення та ідентифікації загроз в кіберпросторі

Рівень вищої освіти другий (магістерський)
Спеціальність 122 Комп'ютерні науки
Освітня програма Комп'ютерні науки

Виконав: студент групи МгІТ1-23 Максим ЖЕНЖЕРА
Науковий керівник: к.т.н., доц. Тетяна ДЕМКІВСЬКА
Рецензент: к.т.н., доц. Віктор Чупринка

Київ 2024

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ

Факультет / Інститут	Мехатроніки та комп'ютерних технологій
Кафедра	Комп'ютерних наук
Рівень вищої освіти	Другий (магістерський)
Спеціальність	122 Комп'ютерні науки
Освітня програма	Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерних наук та технологій
_____ Наталія ЧУПРИНКА
(підпис)
«_____» _____ 2024р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
Женжері Максиму Олександровичу

1. Тема кваліфікаційної роботи: Розроблення програмного забезпечення для виявлення та ідентифікації загроз в кіберпросторі.
Науковий керівник роботи Демківська Тетяна Іванівна

затверджені наказом КНУТД від " 03 " 09 2024 року № 188-уч.

2. Вихідні дані до кваліфікаційної роботи Розробки кафедри комп'ютерних наук, рекомендована література.
3. Зміст кваліфікаційної роботи (перелік питань, які потрібно опрацювати) 1) Аналіз предметної області, 2) Проектування мережі підприємства, 3) Імплементація мережі підприємства
4. Дата видачі завдання 08.2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної кваліфікаційної роботи	Терміни виконання етапів	Примітка про виконання
1	Вступ	05.10.2024	
2	Розділ 1 (Аналіз предметної області)	05.10.2024	
3	Розділ 2 (Формування вимог до системи та аналіз існуючих рішень)	05.10.2024	
4	Розділ 3 (Реалізація системи)	25.10.2024	
5	Висновки	25.10.2024	
6	Оформлення кваліфікаційної роботи (чистовий варіант)	30.10.2024	
7	Подача кваліфікацйної роботи науковому керівнику для відгуку (за 14 днів до захисту)	4.11.2024	
8	Подача кваліфікаційної роботи для рецензування (за 12 днів до захисту)	18.11.2024	
9	Перевірка кваліфікаційної роботи на наявність ознак плагіату (за 10 днів до захисту)	23.11.2024	
10	Подання кваліфікаційної роботи завідувачу кафедри (за 7 днів до захисту)	28.11.2024	

Студент _____ Максим ЖЕНЖЕРА
(підпис) (Власне ім'я та ПРІЗВИЩЕ)

Науковий керівник _____ Тетяна ДЕМКІВСЬКА
(підпис) (Власне ім'я та ПРІЗВИЩЕ)

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПЗ — Програмне забезпечення

ШПЗ — Шкідливе програмне забезпечення

ОС — Операційна система

PUP (Potentially unwanted program) — Потенційно небажані програми

COFF (Common Object File Format) — Загальний формат об'єктних файлів

PE (Portable Executable) — Портативний виконуваний файл

DLL (Dynamic-link library) — Динамічно приєднувана бібліотека

ELF (Executable and Linkable Format) — Формат виконання та зв'язування

IOC (Indicator of compromise) — Індикатор компрометації

API (Application Programming Interface) — Прикладний програмний інтерфейс

GUI (Graphical User Interface) — Графічний інтерфейс користувача

C&C (Command and Control) — Центр управління і контролю

RVA (Relative Virtual Address) — Відносна віртуальна адреса

VA (Virtual Address) — Віртуальна адреса

CIDR (Classless Inter-Domain Routing) — Безкласова маршрутизація

UEFI (Unified Extensible Firmware Interface) — Уніфікований розширюваний інтерфейс прошивки

АНОТАЦІЯ

Женжера М. О. Розроблення програмного забезпечення для виявлення та ідентифікації загроз в кіберпросторі.

Кваліфікаційна робота за спеціальністю 122 – «Комп'ютерні науки». – Київський національний університет технологій та дизайну, Київ, 2024 рік.

У кваліфікаційній роботі було розроблено автоматизовану систему генерації сигнатур для виявлення шкідливого програмного забезпечення. Система спрощує процес аналізу, створення та управління сигнатурами, дозволяючи користувачам швидко отримувати результати у адаптивних форматах.

Реалізовано інтеграцію з базою даних MongoDB для зберігання та обробки даних, а також створено веб-інтерфейс для зручного візуального представлення сигнатур.

Система базується на сучасних технологіях, зокрема Python, що забезпечує високу продуктивність, безпеку та можливість легкого розширення функціоналу. Такий підхід сприяє ефективній інтеграції з іншими програмними продуктами для підвищення рівня кібербезпеки.

Ключові слова: шкідливе програмне забезпечення, антивірус, сигнатурний метод виявлення, сигнатури, автоматизована система, кібербезпека.

ABSTRACT

Zhenzhera Maksym. Development of software for detection and identification of threats in cyberspace.

Qualification work in specialty 122 - “Computer Science.” - Kyiv National University of Technology and Design, Kyiv, 2024.

In the qualification work, an automated signature generation system for malware detection was developed. The system simplifies the process of analyzing, creating, and managing signatures, allowing users to quickly obtain results in adaptive formats.

Integration with the MongoDB database for data storage and processing has been implemented, and a web interface for easy visual presentation of signatures has been created.

The system is based on modern technologies, in particular Python, which ensures high performance, security, and the ability to easily expand functionality. This approach facilitates effective integration with other software products to improve cybersecurity.

Keywords: malware, antivirus, signature detection method, signatures, automated system, cybersecurity.

Зміст

ВСТУП 8

РОЗДІЛ 1. Аналіз предметної області 10

1.1. Що таке шкідливе програмне забезпечення? 11

1.2. Windows та PE 20

1.3. Статичний та динамічний аналіз 29

1.4. Методи виявлення ШПЗ 32

Висновки до розділу 1 36

РОЗДІЛ 2. Формування вимог до системи та аналіз існуючих рішень 37

2.1. Вимоги до системи 37

2.2. Аналіз існуючих рішень 43

2.3. Обґрунтування унікальності запропонованого рішення 50

Висновки до розділу 2 52

РОЗДІЛ 3. Реалізація системи 53

3.1. Вибір технологій 53

3.2. Імплементація компонентів системи 56

Висновки до розділу 3 67

ВИСНОВКИ 68

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 69

ДОДАТОК А. ЛІСТИНГ КОДУ 71

ДОДАТОК Б. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ 78

ВСТУП

Актуальність роботи. Проблема ефективного виявлення шкідливого програмного забезпечення (ШПЗ) залишається однією з ключових у сфері кібербезпеки. Попри існування численних інструментів, аналоги автоматизованої системи генерації сигнатур із інтеграцією через веб-інтерфейси та API відсутні. Це створює потребу у розробці нових підходів, які забезпечують високу точність, адаптивність і зручність використання.

Мета дослідження. Довести ефективність автоматизованої системи генерації сигнатур для виявлення та аналізу шкідливого програмного забезпечення, що відповідає сучасним вимогам кібербезпеки.

Об’єкт дослідження. Шкідливе програмне забезпечення та методи його виявлення.

Предмет дослідження. Розроблення автоматизованої системи генерації сигнатур для виявлення шкідливого програмного забезпечення, включаючи інтеграцію з іншими інструментами.

Методи дослідження. Дослідження базується на аналізі наукових публікацій та існуючих рішень у сфері кібербезпеки, практичній розробці компонентів системи генерації сигнатур, тестуванні її ефективності на реальних даних, а також порівняльному аналізі запропонованого підходу з існуючими методами виявлення шкідливого програмного забезпечення.

Елементи наукової новизни. Запропоновано унікальний підхід до автоматизованої генерації сигнатур для ШПЗ із представленням у адаптивних форматах.

Практична значущість роботи. Результати дослідження дозволяють створювати адаптивні системи виявлення ШПЗ, які можуть бути використані в реальних умовах для захисту корпоративних та індивідуальних мереж. Система сприяє зниженню ризиків, пов'язаних зі ШПЗ, завдяки автоматизації процесів генерації сигнатур і інтеграції з іншими інструментами безпеки.

РОЗДІЛ 1. Аналіз предметної області

Для того щоб встановити, навіщо потрібні сигнатури виявлення шкідливого програмного забезпечення та яка користь від їх генерації, нам потрібно розглянути ключові моменти предметної області.

Підемо від о*сновного об'єкта вивчення проєкту* – **шкідливого програмного забезпечення** (**malware** – *malicious software*). Тут ми розглянемо що це взагалі таке, яка історія виникнення та проблематика.

Після цього ми приділимо увагу безпосередньо операційній системі та її формату виконуваного файлу. Мова йде про **ОС Windows** та формат **PE (exe)**. Ні для кого не секрет, що для кінцевих користувачів це найрозповсюдженіша ОС. А серед *malware* виконувані файли є найчастішими з їх можливих форм.

Тепер, маючи представлення проблеми та її можливого прояву у вигляді виконуваних файлів – ми зосередимося на тому, яким чином ми можемо ці файли **проаналізувати**. Розглянемо **статичний аналіз** та **динамічний аналіз**. На статичному аналізі зупинимося детальніше, так як це дуже важливо для реалізації проєкту.

Врешті-решт, ми дібрались моменту, коли зрозуміли і проблему, і деталі, і можливий аналіз, що ми можемо застосувати. Тепер час настав подивитися на **протидію** та **запобігання** від страшних „вірусів“. Як цього досягають *комплекси програмного забезпечення*, що націлені на боротьбу з *malware,* вони ж **антивіруси**. Тут ми дізнаємось про **сигнатурний метод виявлення** та потрібні для цього **сигнатури**.

Отже, генерація *таких* сигнатур є **предметом** проєкту.

В наступному розділі ми сформулюємо вимоги до реалізації проєкту та звернемо увагу на аналоги, що наявні на ринку захисту та відслідковування інцидендтів в кіберпросторі.

В останньому розділі ми імплементуємо таку систему та побачимо результат її роботи в реальних сигнатурах, що ми отримаємо після генерації. Саме ці сигнатури потім зможуть захистити вас від можливого *malware*.

1.1. Що таке шкідливе програмне забезпечення?

Це вже було зазначено вище, але розпочнемо ми, можна сказати, з одного з найважливіших моментів в контексті україномовного тексту. Це походження цього терміна та його первинного значення.

Шкідливий програмний засіб, ***шкідливе програмне забезпечення*** від англ. ***malware*** — скорочення від malicious — *зловмисний* і software — *програмне забезпечення*. Термін **malware** не один раз буде використаний нами в тексті, тому запам’ятаємо його відразу.

Також, це влучне місце на початку, щоб згадати про інший термін – ***вірус*** (virus). Детальніше про нього нижче, а зараз зазначимо те, що це один з типів ШПЗ (шкідливого програмного забезпечення). Цей термін є набагато більш популярним за класичний *malware*, але в якості доречності використання не відповідає істині.

Malware. Визначень в Інтернет є безліч, стислих та не сильно. Щоб зрозуміти, що ж це таке не вдаючись у подробиці, я дам наступне визначення.

Malware – *програмне забезпечення, яке може нанести шкоду користувачу без усвідомлення цього користувачем*. Юзер запускає програму, програма не інформує його про можливі шкідливі наслідки в ході роботи. Залежачи від цих наслідків та зовнішнього вигляду програми malware класифікується на різні типи. Це визначення є найбільш легким для розуміння та доступним для пояснення.

Тепер перейдемо до більш формальних.

Malware — програмне забезпечення, яке перешкоджає роботі комп'ютера, збирає конфіденційну інформацію або отримує доступ до комп'ютерних систем. Може проявлятися у вигляді коду, скрипту, активного контенту, і іншого програмного забезпечення. **Шкідливий** (malicious) — це загальний термін, який використовується для позначення різних форм ворожого або непроханого програмного забезпечення.

Отже, *шкідливе програмне забезпечення* — це загальний термін, який описує будь-яку шкідливу програму чи код, які завдають шкоди системам.

Мотиви зловмисного програмного забезпечення різні. Зловмисне програмне забезпечення може мати на меті заробляти на вас гроші, саботувати вашу здатність виконувати роботу, робити політичні заяви чи просто хвалитися. Хоча зловмисне програмне забезпечення не може пошкодити фізичне обладнання систем або мережевого обладнання, воно може викрасти, зашифрувати або видалити ваші дані, змінити чи захопити основні функції комп'ютера та стежити за діяльністю вашого комп'ютера. без вашого відома чи дозволу.

Ви знаєте, як щороку медичне співтовариство проводить кампанію, щоб усі отримали щеплення від грипу? Це пов'язано з тим, що спалахи грипу зазвичай мають свій сезон — час року, коли вони починають поширюватися та заражати людей. Навпаки, немає передбачуваних сезонних заражень для ПК, смартфонів, планшетів і корпоративних мереж. Для них завжди сезон грипу. Але замість того, щоб страждати від ознобу та болю в тілі, користувачі можуть захворіти на машинну хворобу — шкідливе програмне забезпечення.

1.1.1. Як дізнатися, чи заражений я шкідливим програмним забезпеченням?

Зловмисне програмне забезпечення може виявляти себе за допомогою багатьох різних ненормальних дій. Ось кілька яскравих ознак того, що у вашій системі зловмисне програмне забезпечення :

– **Ваш комп'ютер сповільнюється**. Одним із побічних ефектів зловмисного програмного забезпечення є зниження швидкості вашої операційної системи (ОС). Незалежно від того, чи перебуваєте ви в Інтернеті чи просто використовуєте локальні програми, використання ресурсів вашої системи виглядає ненормально високим. Ви навіть можете помітити, як вентилятор вашого комп'ютера крутиться на повній швидкості — хороший показник того, що щось забирає системні ресурси у фоновому режимі. Це зазвичай трапляється, коли ваш комп'ютер підключено до ботнету; тобто мережа поневолених комп'ютерів, які використовуються для здійснення DDoS- атак, розсилки спаму або майнингу криптовалюти .

– **Ваш екран завалений настирливою рекламою**. Неочікувані спливаючі рекламні оголошення є типовою ознакою зараження шкідливим програмним забезпеченням. Вони особливо пов’язані з формою зловмисного програмного забезпечення, відомого як рекламне ПЗ. Більше того, спливаючі вікна зазвичай постачаються разом із іншими прихованими загрозами зловмисного програмного забезпечення. Отже, якщо ви побачите щось схоже на «ВІТАЄМО! Ви виграли безкоштовне читання екстрасенсів!» у спливаючому вікні, не натискайте на нього. Який би безкоштовний приз не обіцяла реклама, вона коштуватиме вам багато.

– **Ваша система виходить з ладу**. Це може статися як зависання або **BSOD** (синій екран смерті), останній виникає в системах Windows після виявлення фатальної помилки.

– **Ви помітили загадкову втрату дискового простору**. Це може бути пов’язано з роздутим сквотером зловмисного програмного забезпечення, яке ховається на вашому жорсткому диску, відоме як bundleware .

– **Дивне збільшення активності вашої системи в Інтернеті**. Візьмемо, наприклад, трояни. Щойно троян потрапляє на цільовий комп’ютер, наступне, що він робить, — це звертається до командного та контрольного сервера (C&C) зловмисника, щоб завантажити вторинну інфекцію, часто програму-вимагач. Цим можна пояснити сплеск активності в Інтернеті. Те саме стосується ботнетів, шпигунського програмного забезпечення та будь-якої іншої загрози, яка потребує прямого зв’язку з серверами C&C.

– **Налаштування вашого браузера змінюються.** Якщо ви помітили, що ваша домашня сторінка змінилася або у вас встановлено нові панелі інструментів, розширення чи плагіни, можливо, ви заражені зловмисним програмним забезпеченням. Причини можуть бути різними, але зазвичай це означає, що ви натиснули спливаюче вікно «Вітаємо», яке завантажило небажане програмне забезпечення .

– **Ваш антивірусний продукт перестає працювати, і ви не можете його знову ввімкнути**, залишаючи вас незахищеними від прихованого шкідливого програмного забезпечення, яке вимкнуло його.

– **Ви втратите доступ до своїх файлів або всього комп'ютера**. Це симптом зараження програмою-вимагачем. Хакери повідомляють про себе, залишаючи повідомлення про викуп на вашому робочому столі або змінюючи шпалери робочого столу на повідомлення про викуп. У примітці зловмисники зазвичай повідомляють вам, що ваші дані були зашифровані, і вимагають викуп в обмін на розшифровку ваших файлів.

Навіть якщо здається, що у вашій системі все працює нормально, не заспокоюйтесь, тому що відсутність новин не обов'язково є гарною новиною. Потужне зловмисне програмне забезпечення може ховатися глибоко у вашому комп'ютері, уникаючи виявлення, і займатися своїми брудними справами, не викликаючи жодних тривог. Незважаючи на те, що ми надали короткий посібник із виявлення зловмисного програмного забезпечення, для виявлення зловмисного програмного забезпечення кожній системі дійсно потрібне непохитне око хорошої програми кібербезпеки.

1.1.2. Як я можу заразитися шкідливим ПЗ?

Два найпоширеніші способи потрапляння шкідливого ПЗ у вашу систему — це **Інтернет** та **електронна пошта**. Тобто, по суті, будь-який час, коли ви підключені до мережі, ви є вразливими.

Шкідливе ПЗ може проникнути на ваш комп'ютер, коли ви (глибокий вдих) переглядаєте зламані вебсайти, відвідуєте легітимні сайти, які показують шкідливу рекламу, завантажуєте заражені файли, встановлюєте програми або додатки від неперевірених постачальників, відкриваєте шкідливі вкладення в електронній пошті (malspam) або практично все інше, що ви завантажуєте з Інтернету на пристрій без якісного антивірусного захисту.

Шкідливі додатки можуть ховатися у програмах, які здаються легітимними, особливо якщо вони завантажені з вебсайтів або за прямими посиланнями (в електронному листі, текстовому або чат-повідомленні), а не з

офіційного магазину додатків. У таких випадках важливо звертати увагу на попереджувальні повідомлення під час встановлення програм, особливо якщо вони вимагають доступу до вашої електронної пошти або іншої персональної інформації.

1.1.3. Типи malware

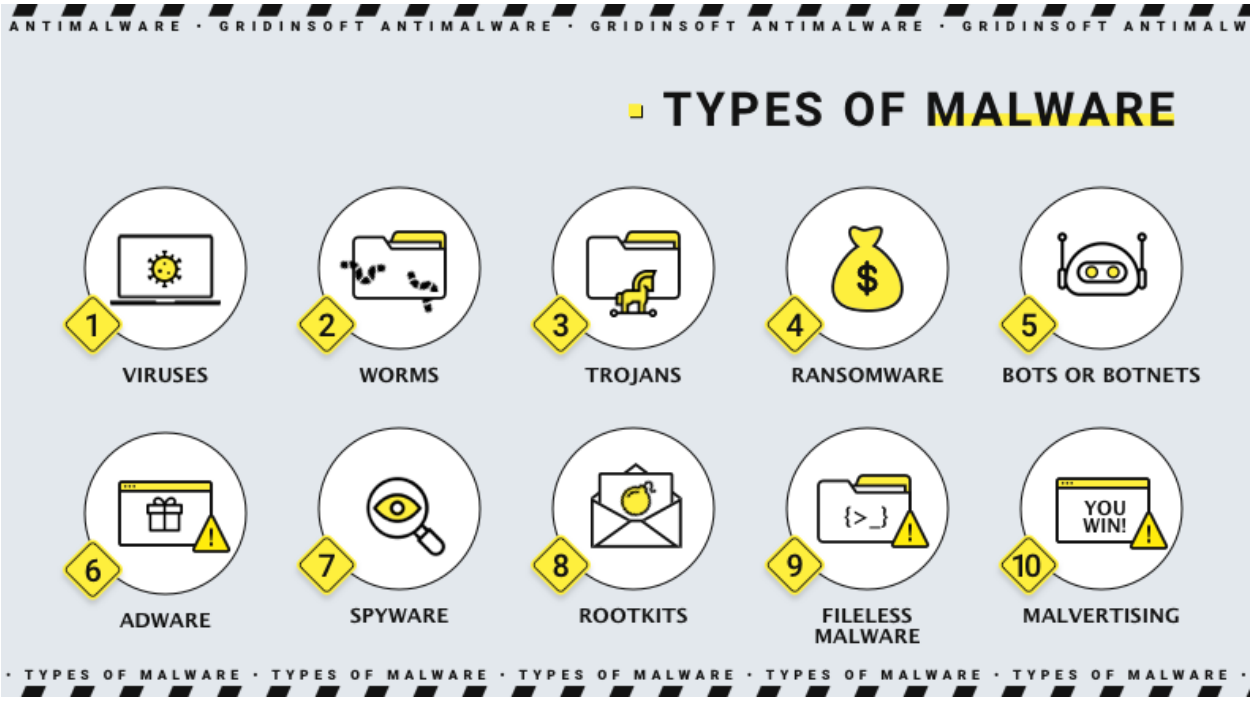


Рис. 1.1 Типи malware

Ось найпоширеніші представники галереї шкідливого ПЗ:

– **Adware** (рекламне ПЗ) — небажане програмне забезпечення, яке показує рекламу на вашому екрані, зазвичай у веббраузері. Часто воно маскується під легітимні програми або "чіпляється" до іншого ПЗ, щоб обманом змусити вас встановити його на ваш ПК, планшет чи мобільний пристрій.

– **Spyware** (шпигунське ПЗ) — це шкідливе ПЗ, яке таємно спостерігає за діяльністю користувача без його дозволу і передає отриману інформацію автору ПЗ.

– **Virus** (вірус) — шкідливе ПЗ, яке прикріплюється до іншої програми і, коли її запускає користувач (зазвичай ненавмисно), самовідтворюється, змінюючи інші програми та заражаючи їх своїм кодом.

– **Worm** (хробак) — вид шкідливого ПЗ, схожий на віруси. Як і віруси, черв’яки самовідтворюються. Головна відмінність — черв’яки можуть поширюватися між системами самостійно, без дій користувача.

– **Trojan, Trojan horse** (троян) — один із найнебезпечніших видів шкідливого ПЗ. Він маскується під щось корисне, щоб ввести вас в оману. Після потрапляння в систему зловмисники отримують несанкціонований доступ до комп’ютера. Трояни часто використовуються для крадіжки фінансової інформації або **встановлення іншого шкідливого ПЗ**, наприклад, програм-вимагачів.

– **Ransomware** (вимагач) — шкідливе ПЗ, яке блокує доступ до вашого пристрою або шифрує файли, а потім вимагає викуп за відновлення доступу. Це улюблена зброя кіберзлочинців, оскільки забезпечує швидку та вигідну оплату у криптовалюті, яку важко відстежити.

– **Rootkit** (руткіт) — вид шкідливого ПЗ, що надає зловмиснику привілеї адміністратора на зараженій системі (доступ root). Зазвичай воно також розроблене для приховування від користувача, інших програм і самої операційної системи.

– **Keylogger** (клавіатурний шпигун) — шкідливе ПЗ, яке записує всі натискання клавіш користувача, зберігаючи інформацію і відправляючи її зловмиснику. Підтип шпигунського ПЗ. Мета — отримання конфіденційних даних, таких як логіни, паролі або номери кредитних карток.

– **Cryptomining**, **cryptojacking** (шкідливий криптомайнінг) — це ПЗ, зазвичай встановлене через троян, яке використовує ваш комп’ютер для майнінгу криптовалют, таких як Bitcoin. Отримані монети надходять на рахунок зловмисника, а не ваш. По суті, такий майнінг краде ваші ресурси для власного збагачення.

– **Exploits** (експлойти) — вид шкідливого ПЗ, який використовує помилки або вразливості системи, щоб надати зловмиснику доступ. Отримавши доступ, зловмисник може викрасти дані або встановити інший вид

шкідливого ПЗ. **Zero-day exploit** означає вразливість, для якої ще не існує захисту або виправлення.

Окремої уваги заслуговують **потенційно небажані програми**.

1.1.4. Що таке потенційно небажані програми (PUPs)?

Потенційно небажані програми (Potentially Unwanted Programs, PUPs) — це тип програмного забезпечення, яке може потрапити на ваш пристрій без вашого відома. PUP часто розповсюджуються в комплекті з безкоштовними програмами, завантажуються ненавмисно або поширюються через оманливу рекламу. Хоча PUP не завжди є шкідливими, вони можуть порушувати вашу конфіденційність і безпеку, відстежуючи вашу онлайн-активність або показуючи цільову рекламу.

PUP можуть змінювати налаштування веббраузера, показувати небажану рекламу, встановлювати інше небажане програмне забезпечення або вносити небажані зміни у вашу систему. Деякі PUP також можуть сповільнювати роботу пристрою та викликати його збій.

PUP важко виявити, оскільки вони часто маскуються під легітимне програмне забезпечення або ховаються всередині інших програм. PUP часто встановлюються разом з іншим програмним забезпеченням, тому їх можна завантажити ненавмисно. **Важливо бути обережним при завантаженні програм з Інтернету та уважно читати умови використання, щоб уникнути ненавмисного встановлення PUP**.

1.1.5. Яка історія malware?

Зважаючи на різноманіття типів шкідливого програмного забезпечення та величезну кількість варіантів, що щодня з’являються в Інтернеті, повна історія шкідливих програм була б занадто довгою, щоб включити її сюди. Однак огляд тенденцій розвитку шкідливого програмного забезпечення в останні десятиліття є більш керованим. Ось основні тенденції в розвитку шкідливого програмного забезпечення.

1980-ті роки та наступні: Теоретичні основи "самовідтворювальних автоматів" (тобто вірусів) сягають лекції, яку в 1949 році прочитав Джон фон

Нейман, людина епохи Відродження XX століття. Однак історія сучасних вірусів починається з програми Elk Cloner, яка почала інфікувати системи Apple II в 1982 році.

Розповсюджуваний через інфіковані дискети, сам вірус був нешкідливим, але він поширювався на всі диски, підключені до системи, і настільки активно розповсюджувався, що його можна вважати першим великомасштабним спалахом комп'ютерних вірусів в історії. Зазначимо, що це сталося до появи шкідливих програм для Windows PC. Відтоді віруси та черв'яки стали широко поширюватися.

1990-ті роки: Microsoft Windows почала свою довгу епоху як найбільш популярна операційна система у світі (і не буде перевершена до появи Android від Google багато років по тому). Як тільки операційна система Windows і її вбудовані додатки стали популярними, так само почала зростати кількість вірусів, написаних для цієї платформи. Особливо шкідливі автори програм почали писати інфекційний код на макромовах Microsoft Word. Ці макровіруси заражали документи та шаблони, а не виконувані програми, хоча, строго кажучи, макроси в документах Word є формою виконуваного коду.

2002–2007 роки: Черв'яки для миттєвих повідомлень (IM) поширювалися через популярні мережі IM, включаючи AOL AIM, MSN Messenger та Yahoo Messenger. Більшість атак починалося з соціальної інженерії. Атакуючі могли надіслати повідомлення на кшталт "Хто з вами на цій фотографії?" або "О, Боже, я думаю, ви виграли в лотерею!" разом з посиланням на шкідливу програму для завантаження. Після того як ваша система інфікувалася, черв'як для миттєвих повідомлень поширював би шкідливі посилання на завантаження серед усіх ваших контактів.

2005–2009 роки: Атаки з використанням рекламного ПЗ (adware) поширилися, демонструючи небажану рекламу на екранах комп'ютерів, часто у вигляді спливаючих вікон або вікон, які не можна було закрити. Ці реклами часто використовували легітимне програмне забезпечення для свого поширення, але близько 2008 року видавці програмного забезпечення почали

подавати в суд на компанії, що займаються рекламним ПЗ, за шахрайство. Результатом стали мільйони доларів штрафів. Це зрештою призвело до закриття компаній з рекламним ПЗ. Сучасні шахрайства, пов'язані з технічною підтримкою, значною мірою зобов'язані цілим прийомам старих рекламних атак, наприклад, рекламі на весь екран, яку не можна закрити або вийти з неї.

2007–2009 роки: Шахраї з шкідливими програмами почали використовувати соціальні мережі, такі як Myspace, для доставки фальшивих рекламою, посилань на фішингові сторінки та шкідливих додатків. Після того, як популярність Myspace знизилася, Facebook і Twitter стали переважними платформами.

2013 рік: Новий тип шкідливого програмного забезпечення під назвою "ransomware" (вимагач) запустив атаку під назвою CryptoLocker, яка тривала з початку вересня 2013 до кінця травня 2014 року, націлену на комп'ютери, що працюють під Windows. CryptoLocker змусив жертв заплатити близько 3 мільйонів доларів, повідомляє BBC News. Більш того, успіх цього вимагача призвів до безперервного виникнення копій.

2013–2017 роки: Через трояни, експлойти та малвертизацію ransomware став королем шкідливого програмного забезпечення, досягнувши великих спалахів у 2017 році, що торкнулися бізнесів будь-яких типів.

2017 рік: Криптовалюта та способи її добування привернули широку увагу, що призвело до нового шахрайства з використанням шкідливого програмного забезпечення, відомого як cryptojacking, або таємне використання чужого пристрою для майнінгу криптовалюти з ресурсів жертви.

2018–2019 роки: Вимагачі знову набрали популярності. Цього разу кіберзлочинці змістили акцент з індивідуальних користувачів на бізнес-цілі. Завдяки хвилі заражень ransomware GandCrab і Ryuk атаки на бізнеси зросли на 365 відсотків з 2018 по 2019 рік. Станом на сьогодні немає ознак того, що атаки з використанням ransomware сповільняться.

1.2. Windows та PE

Windows стала популярною операційною системою з кількох основних причин, що пов'язані з її доступністю, підтримкою та зручністю для користувачів:

– Широка доступність: З моменту запуску Windows у 1985 році компанія Microsoft активно працювала над тим, щоб зробити її доступною для якнайбільшої кількості користувачів. Вона була випущена на безліч комп'ютерних систем, у тому числі на дешевших пристроях, що значно збільшило її поширеність.

– Інтерфейс користувача: Windows стала однією з перших операційних систем, яка запропонувала зручний графічний інтерфейс користувача (GUI), що зробило її набагато легшою для використання порівняно з іншими текстовими операційними системами, такими як MS-DOS. Це привернуло нових користувачів і зробило її популярною серед як новачків, так і досвідчених користувачів.

– Підтримка програмного забезпечення: Оскільки Windows стала домінуючою операційною системою, більшість програм розробляли саме для неї. Це створило великий екосистему програмного забезпечення, що ще більше збільшило її популярність.

– Масовість використання: Windows довго була найбільш поширеною операційною системою для персональних комп'ютерів.

1.2.1. Чому більшість ШПЗ на Windows?

Чим більше користувачів має операційну систему, тим більше розробників програмного забезпечення та кіберзлочинців звертаються до неї для створення додатків і атак.

– Масова популярність: **Оскільки Windows є найпоширенішою операційною системою, на неї націлено найбільше атак**. **Кіберзлочинці, хочуть досягти максимальної кількості заражених комп'ютерів**, і для цього обирають платформу з найбільшим числом користувачів.

– Відкритість та вразливості: Windows має величезну кількість компонентів і різноманітних функцій, що створює більше потенційних точок для вразливостей. У попередніх версіях операційної системи були численні недоліки в безпеці, які часто використовувалися для створення шкідливих програм. Хоча Microsoft постійно працює над покращенням безпеки в нових версіях, загальна складність системи та широке використання у різних пристроях створюють можливості для атак.

– Поширене використання додатків і підключення до мережі: Windows активно використовується для роботи з веб-браузерами, електронною поштою, чатами, а також для запуску додатків, що можуть бути заражені вірусами. Це означає, що більшість вірусів можуть потрапити до комп'ютера через електронну пошту, інтернет-завантаження чи заражені додатки.

– Мінімальна увага до безпеки в домашніх користувачів: Багато домашніх користувачів Windows не дотримуються належних заходів безпеки, не оновлюють свою систему або не використовують антивірусне програмне забезпечення. Це робить систему більш вразливою до атак.

– Залежність від сторонніх розробників: Багато програм, особливо старих або неофіційних, мають недоліки в безпеці, які можуть бути використані для поширення вірусів. Відомо, що в Windows часто використовуються додаткові програми, драйвери і розширення, які можуть мати вразливості.

Таким чином, комбінація широкого використання Windows, її складної структури та історичних вразливостей робить її основною ціллю для кіберзлочинців.

1.2.2. Формат файлів для виконуваних програм

В описі поняття ШПЗ було зазначено, що шкідливим може бути багато що. Це може бути, наприклад, текстовий документ з макросами всередині. Може бути скрипт – це файл, який може бути виконаний за допомогою іншої програми, наприклад, інтепретатора.

Але наш фокус буде саме на виконуваних файлах ОС, які всі знають за розширенням **.exe**. Виконувані файли можуть мати не тільки це розширення, але про це згодом.

Такі файли є бінарними (про **.exe**). Від скриптів вони відрізняються своїм наповненням. Скрипти – це буквально текстові файли, в яких описані інструкції для виконання, що підтримуються програмою, яка ці скрипти виконує.

Бінарні ж файли зовсім інші. Бінарний файл, також називають двійковим, (англ. binary file) — в широкому сенсі: файл, що містить послідовність довільних байтів. Назва пов'язана з тим, що байти складаються з біт, тобто двійкових (англ. binary) цифр.

Ще раз, виконуваний файл — це файл, який містить зрозумілі комп'ютеру спеціальні інструкції, і може бути виконаний (безпосередньо або через командний інтерпретатор операційної системи) як комп'ютерна програма.

Виконуваний файл є традиційно машинним кодом для певного фізичного процесора. Однак, файл, що містить байт-код чи скрипт для інтерпретатора, також можна розглядати як виконуваний. Точне трактування залежить від використання цього файлу, в той час як термін найчастіше вживається до файлів котрі містять саме машинний код. Сучасні операційні системи розпізнають виконувані файли по розширенні файлу (наприклад .com чи .exe в Windows) або ж за допомогою спеціальних ідентифікаторів, які знаходяться в метаданих цього файлу (наприклад, в усіх Unix-подібних системах).

Машиинний код — набір команд (інструкцій), які виконуються безпосередньо центральним процесором комп'ютера без транслятора. В кінці всіх перетворень саме цей машинний код і буде відпрацьовано на процесорі.

Формат виконуваного файлу на ОС Windows має назву **PE** (**Portable Executable**).

Формат Portable Executable (PE) — це формат файлів для виконуваних файлів, об’єктного коду, DLL та інших файлів, які використовуються в 32-розрядних і 64-розрядних версіях операційних систем Windows і в середовищах UEFI. Формат PE — **це структура даних, яка інкапсулює інформацію, необхідну завантажувачу ОС Windows для керування загорнутим виконуваним кодом**. Це включає динамічні бібліотечні посилання для зв’язування, таблиці експорту та імпорту API, дані керування ресурсами та дані локального зберігання потоків (TLS). В операційних системах NT формат PE використовується для EXE, DLL, SYS (драйвер пристрою), MUI та інших типів файлів. У специфікації Unified Extensible Firmware Interface (UEFI) зазначено, що PE є стандартним виконуваним форматом у середовищах EFI.

Простими словами, це структурований бінарний формат даних, в якому інформація щодо даних програми та коду програми можуть бути знайдені за рахунок правил їх знаходженння у файлі. Спеціальна програма імплементована в ОС завантажує ці файли та розбирає згідно відомої структури. Про цю структуру трохи пізніше.

Зараз перерахуємо можливі розширення для PE: .acm, .ax, .cpl, **.dll**, .drv, .efi, **.exe**, .mui, .ocx, .scr, **.sys**, .tsp, .mun, .msstyles.

Серед них ми можемо побачити вже неодноразово згаданий **.exe**. Виконуваний файл, що може бути запущений юзером безпосередньо.

Також тут **.sys**, що відіграє роль системних драйверів.

І насамкінець не менш відомий **.dll**. За допомогою цих файлів інші виконувані файли можуть підвантажують вже реалізований функціонал у кінцевий файл. Їх також називають бібліотеками.

Формати, аналогічні PE, це ELF (використовується в Linux і більшості інших версій Unix) і Mach-O (використовується в macOS і iOS).

1.2.3. Формат PE

З появою операційної системи Windows NT 3.1 корпорація Майкрософт перейшла до формату PE з 16-розрядних форматів NE. Усі пізніші версії

Windows, включаючи Windows 95/98/ME та додаток Win32s до Windows 3.1x, підтримують файлову структуру. Формат зберіг обмежену підтримку застарілих версій, щоб подолати розрив між системами на основі DOS і NT.

Наприклад, заголовки PE/COFF все ще включають виконувану програму DOS, яка за замовчуванням є заглушкою DOS, яка відображає повідомлення на кшталт «Цю програму неможливо запустити в режимі DOS» (або подібне), хоча це може бути повноцінна DOS версія програми (пізнішим відомим випадком є інсталятор Windows 98 SE).

PE також продовжує обслуговувати змінну платформу Windows. Деякі розширення включають формат .NET PE, версію з підтримкою 64-розрядного адресного простору під назвою PE32+ і специфікацію для Windows CE.

Чи є виконуваний код 32- чи 64-бітним, можна дізнатися, перевіривши певне поле в заголовку. Про це далі.

Файл PE/COFF складається з:

- Заголовок файлу MZ (MZ file header)
- Заголовок COFF (COFF header)
- Додатковий заголовок COFF (COFF optional header)
  - стандартна частина (the standard part)
  - частина Windows NT (the Windows NT part)
  - частина каталогів даних (the data directories part)
- таблиця секцій (section table)
- дані (data)
  - дані секцій (section data)
  - директорії даних (data directories)

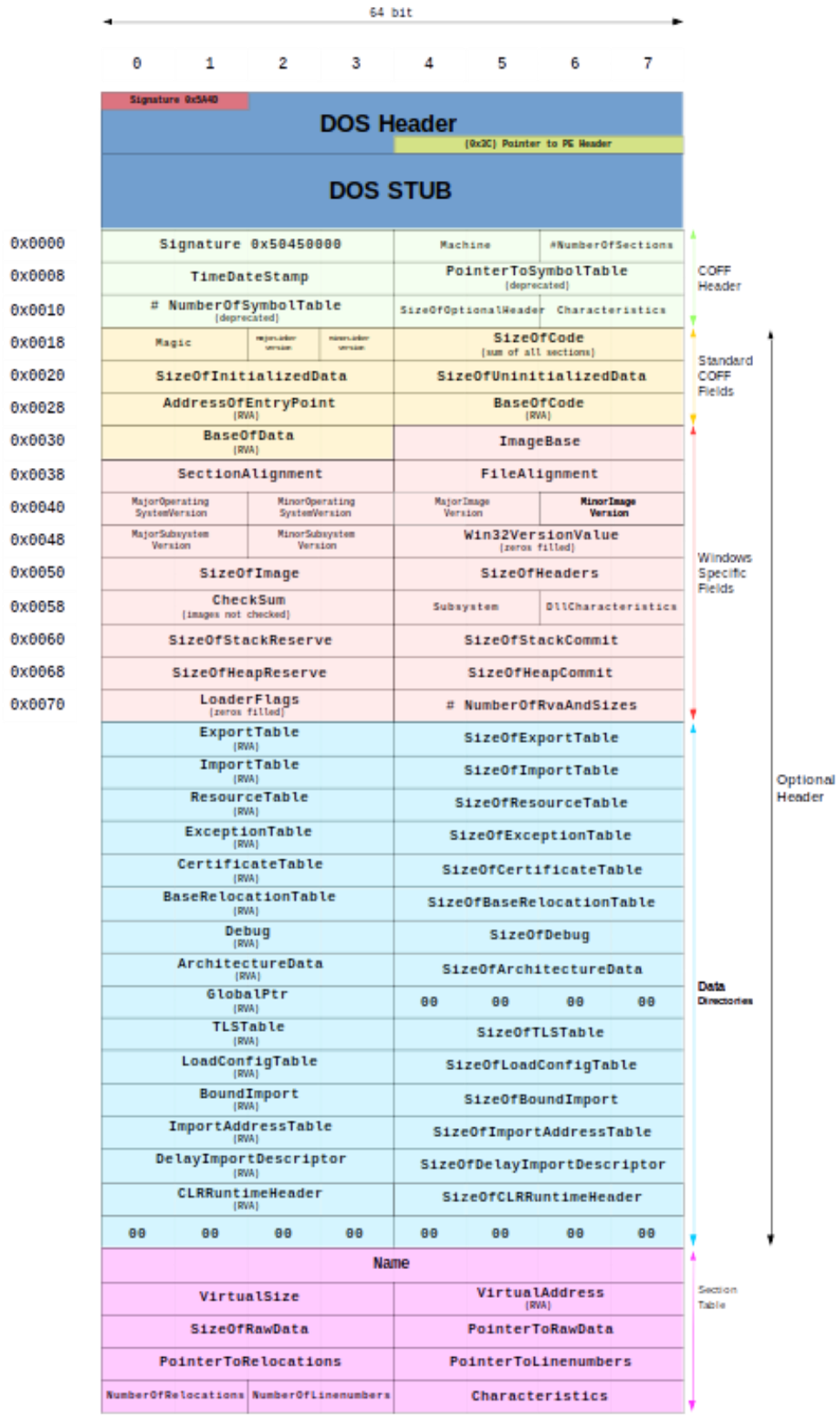


Рис. 1.2. Формат PE

Слід розглянути наступні концепії, що широко використовуються в форматі PE.

Відносна віртуальна адреса (Relative virtual address, RVA) - у файлі зображення це адреса елемента після його завантаження в пам'ять, з якої віднята базова адреса файлу зображення. RVA елемента майже завжди відрізняється від його позиції у файлі на диску (вказівник файлу). У об'єктному файлі RVA менш значуща, оскільки місця в пам'яті не призначаються. У цьому випадку RVA буде адресою в межах секції (описано нижче в таблиці), до якої пізніше застосовується переміщення під час лінкування. Для простоти компілятор має встановлювати перший RVA в кожній секції рівним нулю.

Секція (section) - основна одиниця коду або даних у файлі PE або COFF. Наприклад, весь код в об'єктному файлі може бути об'єднаний в одну секцію або (залежно від поведінки компілятора) кожна функція може займати свою окрему секцію. Чим більше секцій, тим більше накладних витрат на файл, але лінкер може більш вибірково пов'язувати код. Секція схожа на сегмент в архітектурі Intel 8086. Усі сирі дані в секції повинні завантажуватися послідовно. Крім того, файл зображення може містити кілька секцій, таких як .tls або .reloc, які мають спеціальне призначення.

Віртуальна адреса (Virtual Address, VA) - те саме, що й RVA, за винятком того, що базова адреса файлу зображення не віднімається. Адресу називають VA, оскільки Windows створює окремий простір VA для кожного процесу, незалежно від фізичної пам'яті. Для майже всіх цілей VA слід розглядати як просто адресу. VA не така передбачувана, як RVA, оскільки завантажувач може не завантажити зображення в його бажане місце.

Стислий опис формату:

– Магічне значення: Перші два байти файлу — це MZ (0x4D 0x5A), що є відсилкою до старого стандарту DOS. Це значення потрібно для вказівки, що файл є виконуваним, навіть якщо сам файл не має жодного відношення до DOS.

– DOS Header (Магічний заголовок DOS)

Після заголовка DOS йде сам PE заголовок, який починається з "PE\0\0" (0x50 0x45 0x00 0x00). Цей заголовок містить основні параметри виконуваного файлу, такі як:

Signature: Підпис, що містить значення "PE\0\0" для ідентифікації типу файлу.

File Header: Містить важливі характеристики, такі як кількість секцій, тип процесора, розмір заголовка та інші.

Machine: Тип процесора, для якого створено файл (наприклад, 0x14C для x86 або 0x8664 для x64).

Number of Sections: Кількість секцій в файлі.

Time Date Stamp: Час і дата компіляції файлу у вигляді Unix timestamp.

Pointer to Symbol Table: Вказівка на таблицю символів (не обов’язково для виконуваних файлів).

Size of Optional Header: Розмір додаткового заголовка (Optional Header).

Characteristics: Ознаки, які вказують на властивості файлу, наприклад, чи це стандартна виконувана програма, чи система.

– Optional Header

Це ще один важливий компонент, що містить інформацію для завантажувача. Він містить відомості про місце початку коду, точку входу, розмір і таблиці імпортів.

Magic: Вказує на тип файлу (наприклад, 0x10B для 32-бітних файлів або 0x20B для 64-бітних).

Address of Entry Point: Адреса, за якою програма починає виконання після завантаження.

Base of Code: Базова адреса для коду.

Base of Data: Базова адреса для даних.

Image Base: Базова адреса зображення в пам'яті.

Section Alignment: Визначає, як секції повинні вирівнюватися в пам'яті.

File Alignment: Визначає, як дані повинні вирівнюватися в файлі.

Subsystem: Вказує на підсистему, для якої створено файл (Windows GUI, Windows CUI, DOS, і т.д.).

Size of Image: Розмір зображення в пам'яті, коли воно завантажено.

Import Table: Таблиця, яка містить інформацію про імпортовані функції з інших DLL.

Export Table: Таблиця для експортованих функцій, які надаються іншими модулями.

Resource Table: Таблиця ресурсів, що містить різні ресурси (іконки, шрифти, графіка тощо).

Relocation Table: Таблиця, яка допомагає коригувати адреси у разі завантаження в іншу базову адресу.

– Section Headers

Файл PE поділяється на декілька секцій, кожна з яких містить різні типи даних, наприклад, код, дані, ресурси тощо.

Section Name: Ім’я секції (наприклад, .text, .data, .rsrc).

Virtual Size: Розмір секції в пам’яті після завантаження.

Virtual Address: Адреса секції у віртуальній пам'яті.

Size of Raw Data: Розмір даних в секції на диску.

Pointer to Raw Data: Вказівка на місце в файлі, де знаходяться дані цієї секції.

Characteristics: Описує властивості секції, наприклад, чи є вона виконуваною, читаною, записуваною чи спільною.

Основні флаги характеристик:

– IMAGE_FILE_EXECUTABLE_IMAGE: Вказує, що файл є виконуваним.

– IMAGE_FILE_DLL: Вказує, що файл є бібліотекою динамічного зв'язку (DLL).

– IMAGE_FILE_SYSTEM: Вказує, що файл є системним (драйвер).

– IMAGE_FILE_RELOCS_STRIPPED: Вказує, що переміщення не потрібне або було видалено.

– IMAGE_FILE_DEBUG_STRIPPED: Вказує, що налагоджувальні дані були видалені.

– IMAGE_FILE_32BIT_MACHINE: Вказує, що файл призначений для 32-бітних процесорів.

– IMAGE_FILE_64BIT_MACHINE: Вказує, що файл призначений для 64-бітних процесорів.

Секції в PE файлі:

– .text: Секція, що містить виконуваний код. Ця секція зазвичай позначена як доступна лише для читання та виконання (не записувана).

– .data: Секція, що містить глобальні змінні та дані програми.

– .rdata: Секція, що містить дані для читання, які часто використовуються для зберігання рядків або констант.

– .bss: Секція для неініціалізованих змінних.

– .rsrc: Секція для ресурсів, таких як зображення, іконки, діалогові вікна та інші ресурси.

– .reloc: Секція для даних переміщення, яка дозволяє змінювати адреси в коді, якщо зображення завантажується в інше місце пам’яті.

1.3. Статичний та динамічний аналіз

Коли справа доходить до автоматизованого аналізу зловмисного програмного забезпечення, домінують два підходи: **статичний** аналіз та **динамічний** аналіз. Розуміння обох методів з їх плюсами та мінусами може допомогти фахівцям із кібербезпеки приймати обґрунтовані рішення у боротьбі зі зловмисним програмним забезпеченням.

1.3.1. Що таке статичний аналіз ШПЗ?

Статичний аналіз шкідливого програмного забезпечення – це процес вивчення коду шкідливого програмного забезпечення **без його запуску**. Це означає, що експерти вивчають, як створено зловмисне програмне забезпечення та для чого воно призначене, і все це фактично не дозволяє йому

запускатися на комп’ютері. Це все одно, що зрозуміти рецепт із уже приготовленої страви, не знаючи рецепту.

Досліджуючи код зловмисного програмного забезпечення, експерти з безпеки можуть з’ясувати, яке призначення файлу, як він створений, які дії він може виконувати на пристрої та яка його загальна ціль. Нижче наведено деякі поширені методи, які використовуються для статичного аналізу зловмисного програмного забезпечення.

– Розбирання (**Disassembling**): на етапі розбирання статичний аналіз перетворює двійковий код на зрозумілі людині інструкції мови асемблера. Це дозволяє зрозуміти низькорівневі операції та логіку, які використовує зловмисне програмне забезпечення.

– Декомпіляція (**Decompiling**): ця техніка бере виконувані файли та перетворює їх у код високого рівня, який людям легше зрозуміти.

– Аналіз формату файлу (**File Format Analysis**): передбачає перевірку структури та вмісту файлу, щоб переконатися, що він відповідає очікуваному формату. Це допомагає виявити шкідливі зміни, прихований код або вбудовані загрози у файлі.

Статичний аналіз зловмисного програмного забезпечення допомагає експертам швидко ідентифікувати та розуміти зловмисне програмне забезпечення без ризику запуску його в системі. Звичайно, для цього існують обмеження, наприклад; ані виявлення поведінки під час виконання, ані динамічне генерування коду неможливо зробити. Він може бути неефективним проти сильно заплутаних або зашифрованих зловмисних програм і потребує досвіду для точної інтерпретації результатів, що займає багато часу.

1.3.2. Що таке динамічний аналіз ШПЗ?

Динамічний аналіз шкідливого програмного забезпечення – це процес запуску зловмисного програмного забезпечення в захищеному контрольованому середовищі, як-от пісочниця, для спостереження за його поведінкою в реальному часі. Подумайте про це як про поміщення

інфекційного вірусу в біолабораторію для безпечного вивчення його дій. Фактично запустивши зловмисне програмне забезпечення, експерти з безпеки можуть побачити повний спектр його діяльності та краще зрозуміти, як воно працює.

На відміну від статичного аналізу зловмисного програмного забезпечення, коли код зловмисного програмного забезпечення вивчається без його запуску, динамічний аналіз зловмисного програмного забезпечення показує, як зловмисне програмне забезпечення поводиться, коли воно активне. Цей метод дає чіткішу картину реального впливу зловмисного програмного забезпечення, полегшуючи виявлення та припинення його шкідливих наслідків. Деякі ключові речі, на які експерти звертають увагу під час динамічного аналізу зловмисного програмного забезпечення, можуть включати:

– Мережева активність (**Network Activity**) : підозрюваний файл може бути зловмисним програмним забезпеченням, яке надсилає важливі дані про жертву на командно-контрольний сервер, або шкідливим програмним забезпеченням типу завантажувача, яке завантажує додаткові шкідливі файли під час запуску. Відстежуючи ці з’єднання, аналітики можуть визначити місцезнаходження (IP-адреси), з якими зловмисне програмне забезпечення обмінюється даними, а також типи даних, які воно надсилає чи отримує.

– Зміни файлової системи (**File System Changes**) : зловмисне програмне забезпечення часто намагається приховати свою присутність, створюючи, змінюючи або видаляючи файли в зараженій системі. Під час динамічного аналізу зловмисного програмного забезпечення аналітики перевіряють, чи зловмисне програмне забезпечення створює нові файли, змінює існуючі або видаляє важливі дані. Це допомагає зрозуміти, чи намагається зловмисне програмне забезпечення викрасти інформацію, пошкодити файли чи встановити додаткове шкідливе програмне забезпечення.

– Маніпуляції процесами (**Process Manipulation**): зловмисне програмне забезпечення часто взаємодіє з іншими процесами, запущеними в системі,

такими як системні утиліти або програмне забезпечення безпеки. Під час динамічного аналізу шкідливого програмного забезпечення експерти спостерігають за тим, як шкідливе програмне забезпечення взаємодіє з іншими процесами. Він може спробувати замаскуватися, впровадивши інший процес, вимкнувши антивірусні програми або захопивши законні системні процеси, щоб уникнути виявлення.

Цей тип аналізу має вирішальне значення для виявлення широкого спектру зловмисних дій, які може виконувати зловмисне програмне забезпечення, і особливо корисний для виявлення складних загроз, які можуть приховувати або маскувати свою діяльність. Спостерігаючи за поведінкою зловмисного програмного забезпечення в контрольованому середовищі, експерти можуть збирати цінну інформацію для розробки стратегій захисту, таких як створення правил для блокування схожого зловмисного програмного забезпечення від проникнення в систему або вивчення того, як видалити його з уже заражених систем.

Динамічний аналіз зловмисного програмного забезпечення пропонує практичний спосіб зрозуміти, що зловмисне програмне забезпечення робить у режимі реального часу, що робить його важливим інструментом сучасної кібербезпеки.

1.4. Методи виявлення ШПЗ

Методи, що використовуються для виявлення ШПЗ, можна грубо поділити на дві категорії: на основі аномалій та на основі сигнатур. Метод виявлення на основі аномалій використовує знання про те, що є нормальною поведінкою, щоб визначити зловмисність програми, яка перевіряється. Особливий тип виявлення на основі аномалій називається специфікаційним виявленням. Техніки на основі специфікацій використовують набір правил або специфікацій, що визначають допустиму поведінку, для того, щоб вирішити, чи є програма зловмисною. Програми, що порушують специфікації, вважаються аномальними і зазвичай зловмисними.

Метод виявлення на основі сигнатур використовує характеристику відомого зловмисного програмного забезпечення для визначення зловмисності програми, що перевіряється. Очевидно, що ця характеристика або сигнатура зловмисної поведінки є ключовим елементом ефективності методу виявлення на основі сигнатур.

На рисунку 1.3 зображено взаємозв'язок між різними типами технік виявлення зловмисного програмного забезпечення. Кожен із методів виявлення може використовувати один із трьох підходів: статичний, динамічний або гібридний. Конкретний підхід або аналіз техніки на основі аномалій чи сигнатур визначається тим, як саме ця техніка збирає інформацію для виявлення зловмисного ПЗ.

Статичний аналіз використовує синтаксичні або структурні властивості програми (статично) або процесу (динамічно), що перевіряється, для визначення його зловмисності. Наприклад, статичний підхід до виявлення на основі сигнатур використовуватиме лише структурну інформацію (наприклад, послідовність байтів) для визначення зловмисності, тоді як динамічний підхід буде використовувати інформацію, отриману під час виконання (наприклад, системи, що з'являються у стеку виконання) ПЗ, що перевіряється.

Загалом статичний підхід намагається виявити зловмисне програмне забезпечення до його виконання. Навпаки, динамічний підхід намагається виявити зловмисну поведінку під час виконання програми або після її завершення. Існують гібридні методи, які поєднують обидва підходи. У цьому випадку для виявлення зловмисного програмного забезпечення використовуються як статична, так і динамічна інформація.

Статичні та динамічні типи аналізу вже було детально розглянуто вище.

1.4.1. Сигнатурний метод виявлення

Виявлення на основі сигнатур (**Signature-based detection**) намагається моделювати зловмисну поведінку програмного забезпечення і використовує цю модель для його виявлення. Сукупність усіх цих моделей представляє

знання методу виявлення на основі сигнатур. Ця модель зловмисної поведінки зазвичай називається сигнатурою.

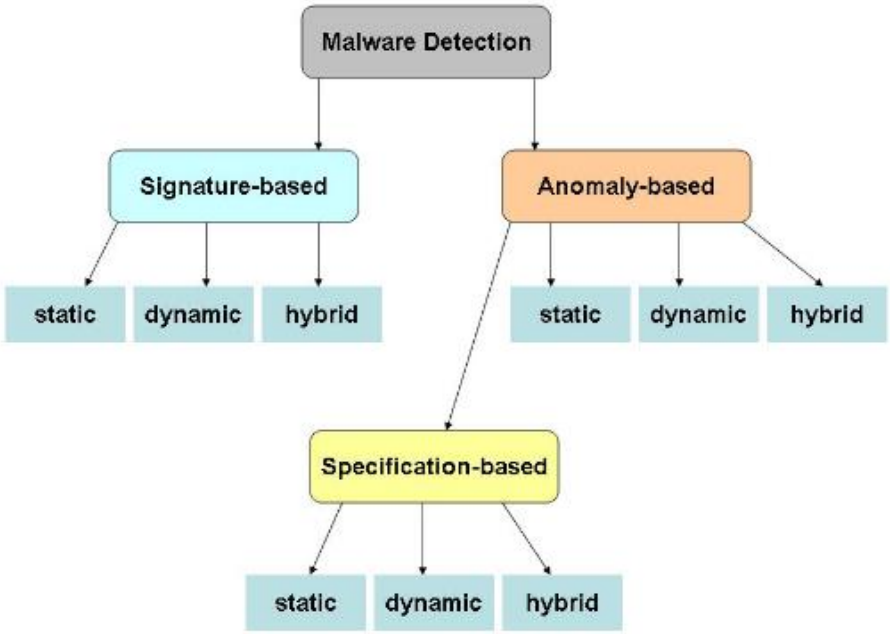


Рис 1.3.Класифікація методів виявлення ШПЗ

Ідеальною є ситуація, коли сигнатура може ідентифікувати будь-яке зловмисне програмне забезпечення, що демонструє поведінку, визначену цією сигнатурою. Як і будь-які дані, які існують у великій кількості та потребують зберігання, сигнатури вимагають репозиторію. Цей репозиторій представляє всі знання методу виявлення на основі сигнатур, які стосуються виявлення зловмисного ПЗ. Коли метод намагається оцінити, чи містить програма, що перевіряється, відому сигнатуру, здійснюється пошук у цьому репозиторії.

На даний момент ми здебільшого покладаємося на людський досвід у створенні сигнатур, які описують зловмисну поведінку програм. Після створення сигнатури її додають до знань методу виявлення на основі сигнатур (тобто до репозиторію).

Саме тому, актуальність автоматизованої системи генерації сигнатур виявлення складно переоцінити.

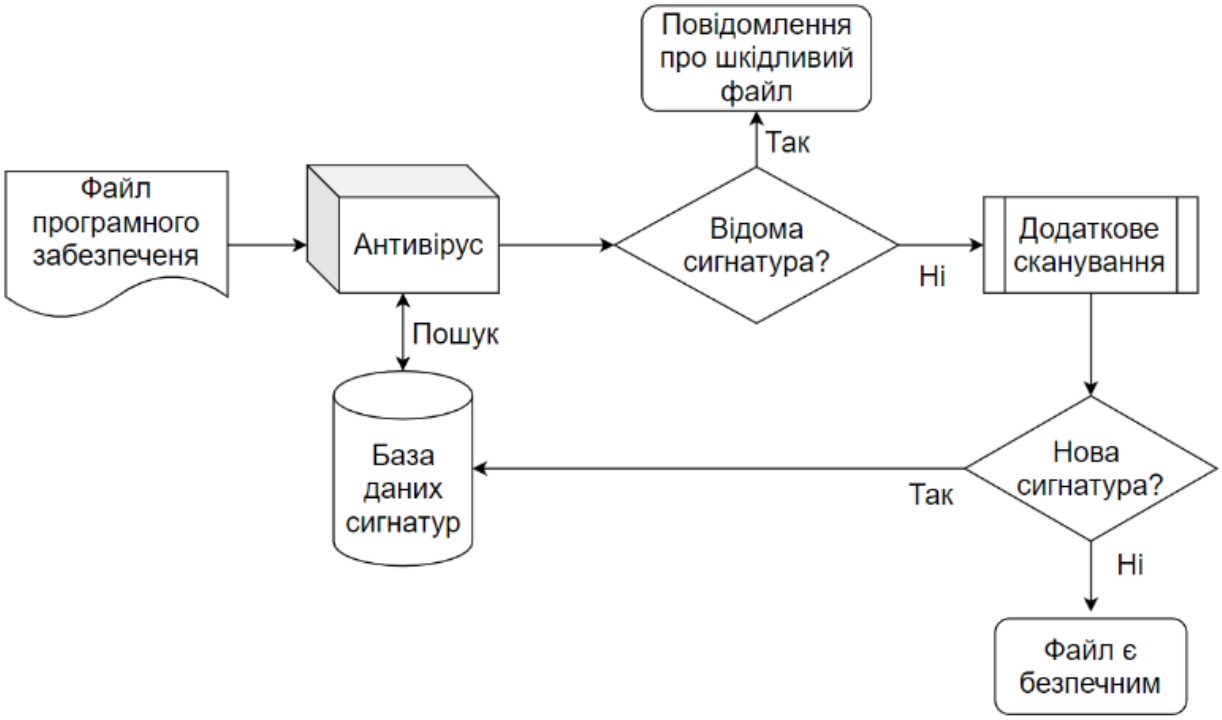


Рис. 1.4. Алгортим роботи сигнатурного методу

Схематично алгоритм сигнатурного методу виявлення ШПЗ зображено на рис. 1.4. Як можна побачити, логіка цього методу досить примітивна: є правило (сигнатура) в базі, правило перевіряється на істинність, якщо правило спрацювало, то файл з високою вірогідністю шкідливий.

Типів таких сигнатур може бути безліч. Такі сигнатури далеко не обмежені одними лише послідовностями двійкового коду.

Цей метод надає змогу виявити ШПЗ за будь-яким фактором, що може бути відомий. Буквально усі поля, що ми дістали після аналізу – як статичного, так і динамічного – можуть бути використані для сигнатур.

Це може бути хеш файлу. А може бути хеш певної структури виконуваного файлу, наприклад, секції. Це може бути наявність специфічного рядка в файлі. Може бути наявність підозрілого імпорту чи експорту. В метаданих, інформації про версію може бути використане значення, що характерне для якогось роду ШПЗ.

Фантазії є де розгулятись, адже такі сигнатури можуть поєднуватись у більш комплексні. Наприклад, за логічними операціями *ТА* і *АБО*.

Потенціал таких сигнатур не варто недооцінювати. Швидка автоматизована система може наповнювати базу сигнатур з вражаючою ефективністю. **Саме така система і є головною задачею проєкту.**

Висновки до розділу 1

Проведено аналіз предметної області дослідження, що стосується шкідливого програмного забезпечення (ШПЗ). Розглянуто визначення, класифікацію та основні загрози, які спричиняє ШПЗ.

Описано особливості операційної системи Windows та структуру файлів формату PE (Portable Executable), які є основною мішенню для ШПЗ.

Також проаналізовано підходи до дослідження програмного забезпечення, зокрема статичний і динамічний методи аналізу, їх переваги, недоліки та сферу застосування.

Окремо розглянуто сучасні методи виявлення ШПЗ, що включають сигнатурний аналіз, евристичний підхід та машинне навчання. Стисло оглянувши перелік методів, ми детально зупинились на сигнатурному методі виявлення.

Цей аналіз створює основу для подальших досліджень та розробки інструментів для ефективного виявлення та аналізу ШПЗ.

РОЗДІЛ 2. Формування вимог до системи та аналіз існуючих рішень

2.1. Вимоги до системи

Основною метою дослідницького проєкту є розроблення проєкту для генерації сигнатур виявлення шкідливого програмного забезпечення.

Таким чином, проєкт має складатися з 3 наступних основних компонентів:

– (**enrichment**) збагачення бази даних з різних джерел;

– (**generation**) генерація сигнатур на основі підготовленої бази даних;

– (**export**) експорт якісних згенерованих сигнатур у потрібному форматі.

На рисунку 2.1 ілюструється загальний алгоритм роботи системи, що охоплює всі етапи – від збагачення бази даних до експорту якісних сигнатур.

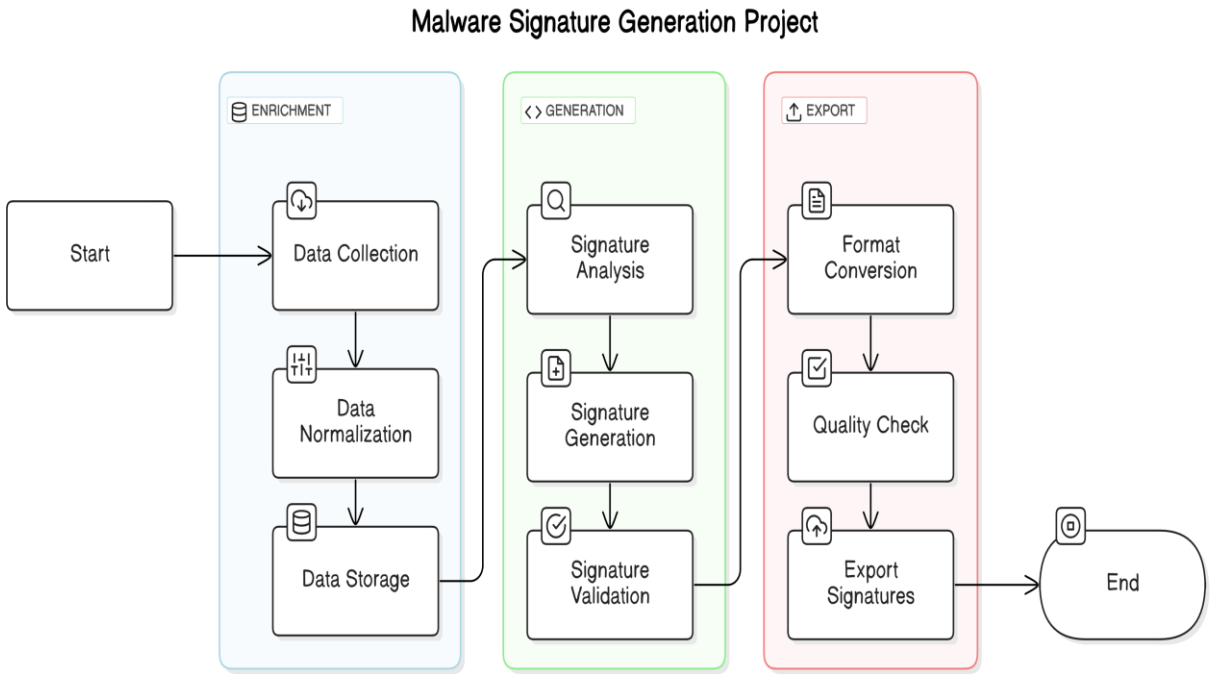


Рис. 2.1: Блок-схема алгоритму проєкту.

Важливим принципом у розробленні цього проєкту є забезпечення універсальності та гнучкості. Система має дозволяти легке налаштування та заміну джерел, форматів і додаткових правил. Це гарантує адаптивність до нових вимог і зручність інтеграції з різними типами даних та інструментів.

Розглянемо кожен компонент більш детально.

2.1.1. Enrichment

Основною моделлю даного проєкту є **шкідливі виконувані файли**, а також інші пов’язані артефакти, такі як мережева активність, ключі реєстру, конфігураційні файли тощо. Ключовою метою є створення універсального інструменту для збору, обробки та аналізу даних, який дозволяє виявляти характерні ознаки та формувати якісні сигнатури для ідентифікації загроз.

Головним джерелом даних є сервіси динамічного аналізу програмного забезпечення, які працюють із використанням віртуальних машин (sandbox). Такі сервіси дозволяють автоматично виконувати підозрілі файли в ізольованому середовищі, збираючи докладну інформацію про їхню поведінку. Однак архітектура проєкту передбачає можливість розширення та інтеграції з іншими джерелами, такими як:

– Пропрієтарні дані, зокрема журнали роботи систем (логи).

– Звіти в альтернативних форматах, наприклад, JSON, XML або специфічні формати, що використовуються внутрішніми системами організацій.

Це забезпечує гнучкість системи та її здатність адаптуватися до різних умов і джерел даних.

Резюмучи, етап збагачення даних включає такі ключові кроки.

Data collection (збір даних).

Дані збираються з різних джерел, таких як sandbox, журнали мережевої активності або інші зовнішні системи.

Data normalization (нормалізація даних).

Зібрані дані перетворюються у стандартизований формат, що дозволяє уніфікувати їх подальшу обробку. Наприклад, поля, які описують IP-адреси, часові мітки, хеші файлів, нормалізуються відповідно до єдиного стандарту.

Data storage (збереження даних).

Нормалізовані дані зберігаються в оптимізованій базі даних, яка забезпечує швидкий доступ і можливість масштабування для обробки великих обсягів інформації.

2.1.2. Generation

Генерація сигнатур є ключовим етапом, метою якого є створення точних і ефективних сигнатур для виявлення шкідливого програмного забезпечення. Цей компонент перетворює структуровані дані з бази, отриманої на етапі збагачення, у готові до використання сигнатури, які можуть бути інтегровані в системи виявлення загроз.

Особливу увагу приділяється забезпеченню якості сигнатур, щоб уникнути *помилкових спрацювань* (**false positives**), які можуть призвести до блокування доброякісних файлів чи дій, і *пропущених загроз* (**false negatives**), що можуть залишити шкідливе ПЗ невиявленим.

Процес генерації складається з таких основних підетапів.

Signature analysis (аналіз сигнатур).

На цьому етапі аналізуються дані з бази, щоб визначити основні характеристики, які можна використовувати для створення сигнатур.

Ідентифікація унікальних ознак: пошук властивостей, які чітко відрізняють шкідливий файл або поведінку від доброякісних. Це можуть бути:

– Статичні артефакти: наприклад, специфічні хеш-значення, унікальні строки, шаблони коду.

– Поведінкові ознаки: наприклад, спроби підключення до відомих шкідливих доменів, створення файлів у системних каталогах.

Кореляція між даними: виявлення закономірностей і взаємозв'язків між різними артефактами, наприклад, мережевою активністю та змінами у реєстрі.

Класифікація: визначення категорій загроз, наприклад, трояни, шифрувальники, експлойти, що допомагає оптимізувати наступний етап генерації.

Signature generation (Створення сигнатур).

Цей етап відповідає за безпосереднє формування сигнатур на основі проведеного аналізу.

Автоматична генерація: система автоматично створює сигнатури на основі виділених унікальних ознак. Наприклад:

– Метадані виконуваних файлів.

– YARA-правила: для виявлення файлів за їх вмістом (рядками, паттернами).

– Snort/Suricata сигнатури: для аналізу мережевого трафіку.

– Індикатори компрометації (IOCs): хеші файлів, IP-адреси, домени.

Підтримка гнучкості: можливість налаштовувати генерацію сигнатур залежно від потреб (детектування певного виду загроз або всіх підозрілих об'єктів).

Оптимізація: виключення надлишкових або дубльованих сигнатур, щоб зменшити навантаження на системи виявлення загроз.

Signature validation (Перевірка сигнатур).

Перевірка є важливим кроком, що гарантує якість і надійність згенерованих сигнатур.

Тестування на вибірках: перевірка сигнатур на великій базі файлів, яка включає як доброякісні, так і шкідливі зразки.

– Аналіз помилкових спрацювань: наприклад, якщо сигнатура виявляє доброякісні файли, вона потребує уточнення.

– Аналіз пропущених загроз: якщо сигнатура не детектує відомі загрози, її потрібно доопрацювати.

Динамічне тестування: сигнатури перевіряються у реальному середовищі (наприклад, у sandbox або системах моніторингу).

Ручна ревізія: важливі сигнатури проходять додатковий аналіз аналітиками, щоб підтвердити їхню якість.

Документування: збереження результатів перевірки, що дозволяє швидко вносити корективи у разі необхідності.

2.1.3. Export

Етап експорту забезпечує перетворення згенерованих сигнатур у необхідний формат для їх подальшого використання у системах виявлення загроз. Основна мета цього етапу — надати користувачу можливість швидко й ефективно отримати лише релевантні сигнатури у потрібному форматі, забезпечуючи високу якість результату.

Процес експорту складається з таких основних пунктів.

Format conversion (Конвертація формату).

Сигнатури, створені в системі, повинні бути адаптовані до вимог цільового середовища.

Підтримка різних форматів:

– YARA-правила для аналізу вмісту файлів.

– Snort/Suricata сигнатури для систем виявлення мережевих загроз.

– CSV/JSON/XML для інтеграції у зовнішні аналітичні інструменти.

Уніфікація структури: перетворення даних у стандартизовану форму, яка відповідає формату призначення. Наприклад, додавання заголовків, метаінформації чи опису сигнатури.

Конфігурація на вимогу: підтримка специфічних налаштувань формату залежно від вимог користувача чи цільової системи.

Quality check (Перевірка якості).

Перед експортом сигнатур важливо провести їх перевірку на відповідність заданим критеріям.

Перевірка коректності формату: сигнатури тестуються на синтаксичну правильність у цільових системах (наприклад, верифікація YARA-правил за допомогою YARA CLI).

Перевірка актуальності: відбір тільки перевірених і затверджених сигнатур, які успішно пройшли стадії аналізу та тестування.

Верифікація метаінформації: перевірка описів, категорій та інших полів, які можуть впливати на використання сигнатур у продуктивному середовищі.

Export signatures (Експорт сигнатур).

На завершальному етапі система надає користувачу зручні засоби для отримання потрібних даних.

Фільтрація: можливість експортувати лише релевантні сигнатури за критеріями:

– Тип загрози (троян, шифрувальник, шпигунське ПЗ тощо).

– Джерело даних або метод створення (поведінковий, статичний аналіз).

– Актуальність (дата створення або останньої перевірки).

Масштабованість: підтримка експорту великих обсягів сигнатур або окремих груп залежно від запиту.

Різні способи експорту:

– Файловий експорт: формування файлів для завантаження.

– API: можливість інтеграції з іншими системами для передачі сигнатур у реальному часі.

– Хмарне сховище: збереження сигнатур у централізованому репозиторії для віддаленого доступу.

У третьому розділі ми на власному досвіді побачимо кінцевий результат роботи всієї нашої системи саме на цьому етапі. Ці сигнатури будуть представлені у зручному форматі, зокрема відображатимуться у веб-інтерфейсі та у вигляді структурованих даних формату JSON, доступних через API.

2.2. Аналіз існуючих рішень

2.2.1. MISP

MISP – Платформа обміну інформацією про загрози.

MISP — це програмне рішення з відкритим кодом для збору, зберігання, розповсюдження та обміну індикаторами кібербезпеки та загрозами щодо аналізу інцидентів кібербезпеки та аналізу зловмисного програмного забезпечення. MISP розроблено аналітиками інцидентів, фахівцями з безпеки та ІКТ або реверсувальниками зловмисного програмного забезпечення для підтримки їхніх повсякденних операцій для ефективного обміну структурованою інформацією.

Метою MISP є сприяння обміну структурованою інформацією всередині співтовариства безпеки та за кордоном. MISP надає функціональні можливості для підтримки обміну інформацією, а також споживання зазначеної інформації системами виявлення вторгнень у мережу (NIDS), LIDS, а також інструментами аналізу журналів, SIEM.

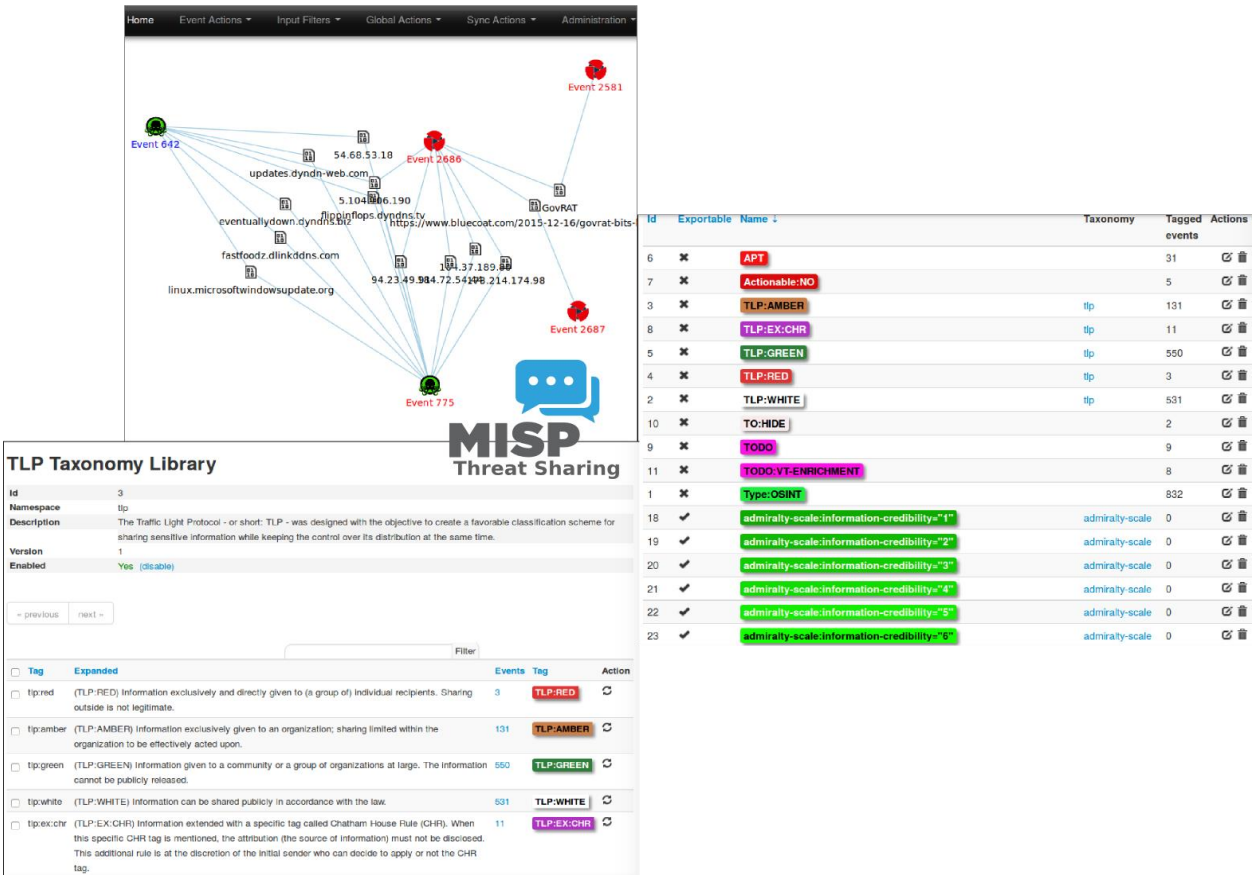


Рис. 2.2: Панорама MISP.

Основні функції:

– Ефективна база даних IOC і індикаторів, що дозволяє зберігати технічну та нетехнічну інформацію про зразки зловмисного програмного забезпечення, інциденти, зловмисників і розвідку.

– Автоматичний пошук кореляції зв'язків між атрибутами та індикаторами зловмисного програмного забезпечення, кампаній атак або аналізу. Механізм кореляції включає кореляцію між атрибутами та більш розширені кореляції, такі як кореляція нечіткого хешування (наприклад, ssdeep) або зіставлення блоків CIDR. Також можна ввімкнути кореляцію або вимкнути подію для кожного атрибута.

– Гнучка модель даних, у якій складні об'єкти можна виражати та зв'язувати разом, щоб виражати дані про загрози, інциденти чи пов'язані елементи.

– Вбудована функція обміну для спрощення обміну даними за допомогою різних моделей розподілу. MISP може автоматично синхронізувати події та атрибути між різними примірниками MISP. Розширені функції фільтрації можна використовувати для відповідності політиці спільного використання кожної організації, включаючи гнучку групу спільного доступу та механізми розподілу на рівні атрибутів.

– Інтуїтивно зрозумілий інтерфейс користувача для створення, оновлення та спільної роботи над подіями та атрибутами/індикаторами. Графічний інтерфейс для легкої навігації між подіями та їхніми кореляціями. Функція графіка подій для створення та перегляду зв'язків між об'єктами та атрибутами. Розширені функції фільтрації та списки попереджень, які допомагають аналітикам вносити події та атрибути та обмежують ризик помилкових спрацьовувань.

– зберігання даних у структурованому форматі (що дозволяє автоматизовано використовувати базу даних для різних цілей) із широкою підтримкою показників кібербезпеки разом із показниками шахрайства, як у фінансовому секторі.

– експорт: створення IDS, OpenIOC, звичайного тексту, CSV, MISP XML або JSON для інтеграції з іншими системами (мережеві IDS, хост-IDS, спеціальні інструменти), формат кешу (використовується для криміналістичних інструментів), STIX (XML і JSON) 1 і 2, експорт NIDS (Suricata, Snort і Bro/Zeek) або зона RPZ. Багато інших форматів можна легко додати за допомогою модулів misp.

– імпорт: груповий імпорт, пакетний імпорт, імпорт із OpenIOC, пісочниці GFI, ThreatConnect CSV, стандартного формату MISP або STIX 1.1/2.0. Багато інших форматів легко додаються за допомогою модулів misp.

– Гнучкий безкоштовний інструмент імпорту тексту для полегшення інтеграції неструктурованих звітів у MISP.

– Зручна система для спільної роботи над подіями та атрибутами, що дозволяє користувачам MISP пропонувати зміни або оновлення атрибутів/індикаторів.

– обмін даними: автоматичний обмін та синхронізація з іншими сторонами та довірчими групами за допомогою MISP.

– делегування спільного доступу: дозволяє використовувати простий, псевдоанонімний механізм делегування публікації подій/індикаторів іншій організації.

– Гнучкий API для інтеграції MISP із вашими власними рішеннями. MISP поєднується з PyMISP, яка є гнучкою бібліотекою Python для отримання, додавання або оновлення атрибутів подій, обробки зразків шкідливих програм або пошуку атрибутів. Повний API restSearch для легкого пошуку індикаторів у MISP і експорту їх у всіх форматах, які підтримує MISP.

– Регульована таксономія для класифікації та позначення подій відповідно до власних схем класифікації або існуючої класифікації. Таксономія може бути локальною для вашого MISP, але також доступною для інших примірників MISP.

– Словники розвідки, які називаються галактикою MISP і в комплекті з існуючими загрозами, зловмисними програмами, RAT, програмами-вимагачами або MITER ATT&CK, які можна легко пов’язати з подіями та атрибутами в MISP.

– Модулі розширення в Python для розширення MISP власними службами або активації вже доступних misp-модулів.

– Підтримка спостереження, щоб отримати спостереження від організацій щодо спільних показників і атрибутів. Спостереження можна надати через інтерфейс користувача MISP, API як документ MISP або документи спостереження STIX.

– Підтримка STIX: імпорт і експорт даних у форматі STIX версії 1 і версії 2.

– Вбудоване шифрування та підпис сповіщень через GnuPG та/або S/MIME залежно від уподобань користувача.

– Канал публікації та підписки в режимі реального часу в MISP, щоб автоматично отримувати всі зміни (наприклад, нові події, індикатори, спостереження або теги) у ZMQ (наприклад, misp-dashboard) або публікацію Kafka.

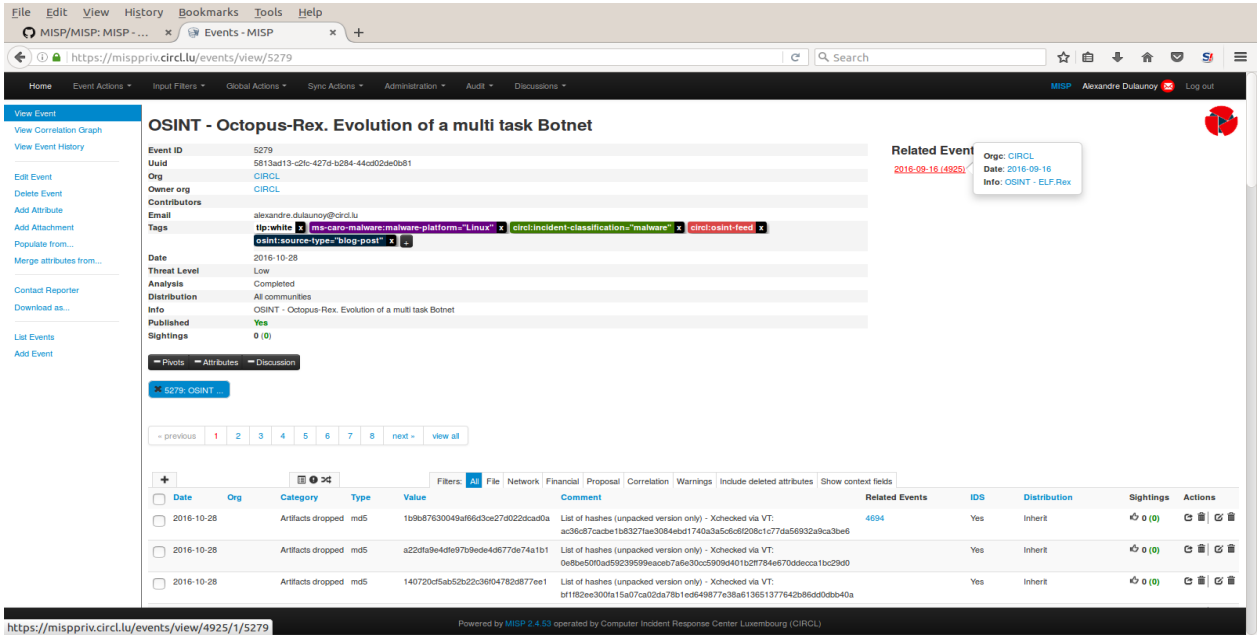


Рис. 2.3: Перегляд події MISP.

2.2.2. OpenCTI

OpenCTI — це платформа з відкритим вихідним кодом, яка дозволяє організаціям керувати своїми знаннями про кіберзагрози та спостереженнями. Він створений для структурування, зберігання, організації та візуалізації технічної та нетехнічної інформації про кіберзагрози.

Структурування даних виконується за допомогою схеми знань на основі стандартів STIX2. Його розроблено як сучасну веб-програму, що включає GraphQL API та UX-орієнтований інтерфейс. Крім того, OpenCTI можна інтегрувати з іншими інструментами та програмами, такими як MISP, TheHive, MITER ATT&CK тощо.

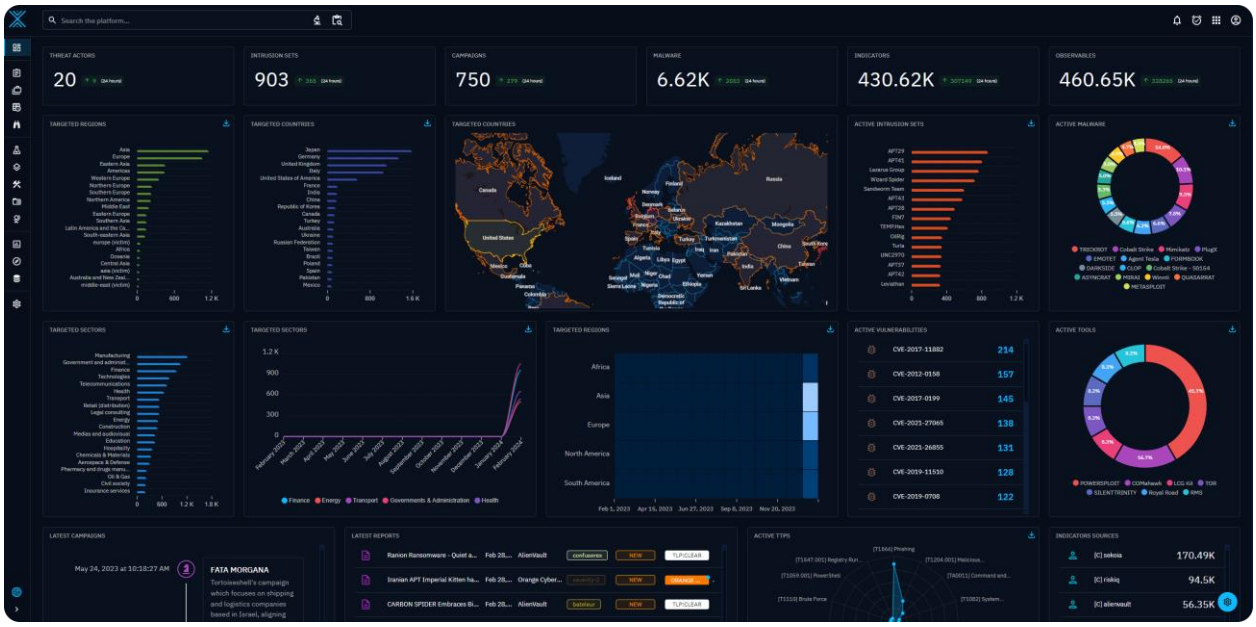


Рис. 2.4: Панорама OpenCTI.

Мета полягає в тому, щоб створити всеохоплюючий інструмент, який дозволить користувачам використовувати технічну (наприклад, TTP і спостережувані) і нетехнічну інформацію (наприклад, пропоновану атрибуцію, віктимологію тощо), одночасно пов’язуючи кожну частину інформації з її первинним джерелом (звітом, Подія MISP тощо), із такими функціями, як зв’язки між кожною інформацією, датами першого та останнього відвідування, рівнями довіри тощо. Інструмент може використовувати структуру MITRE ATT&CK (через спеціальний з’єднувач), щоб допомогти структурувати дані. Користувач також може вибрати реалізацію власних наборів даних.

Після того, як аналітики в OpenCTI капіталізують дані та опрацюють їх, нові зв’язки можуть бути виведені з існуючих, щоб полегшити розуміння та представлення цієї інформації. Це дозволяє користувачеві отримувати та використовувати важливі знання з необроблених даних.

OpenCTI дозволяє не тільки імпортувати, але й експортувати дані в різних форматах (CSV, пакети STIX2 тощо). Наразі розробляються конектори для прискорення взаємодії між інструментом та іншими платформами.

Екосистема конекторів (з’єднувачів), вони ж *OpenCTI Connectors*, дуже розвинута. Кількість підтримуваних інтеграцій величезна та постійно актуалізується. Конектори є наріжним каменем платформи OpenCTI і дозволяють організаціям легко отримувати, збагачувати або експортувати дані. Відповідно до функціональних можливостей і варіантів використання вони поділяються на наступні класи.

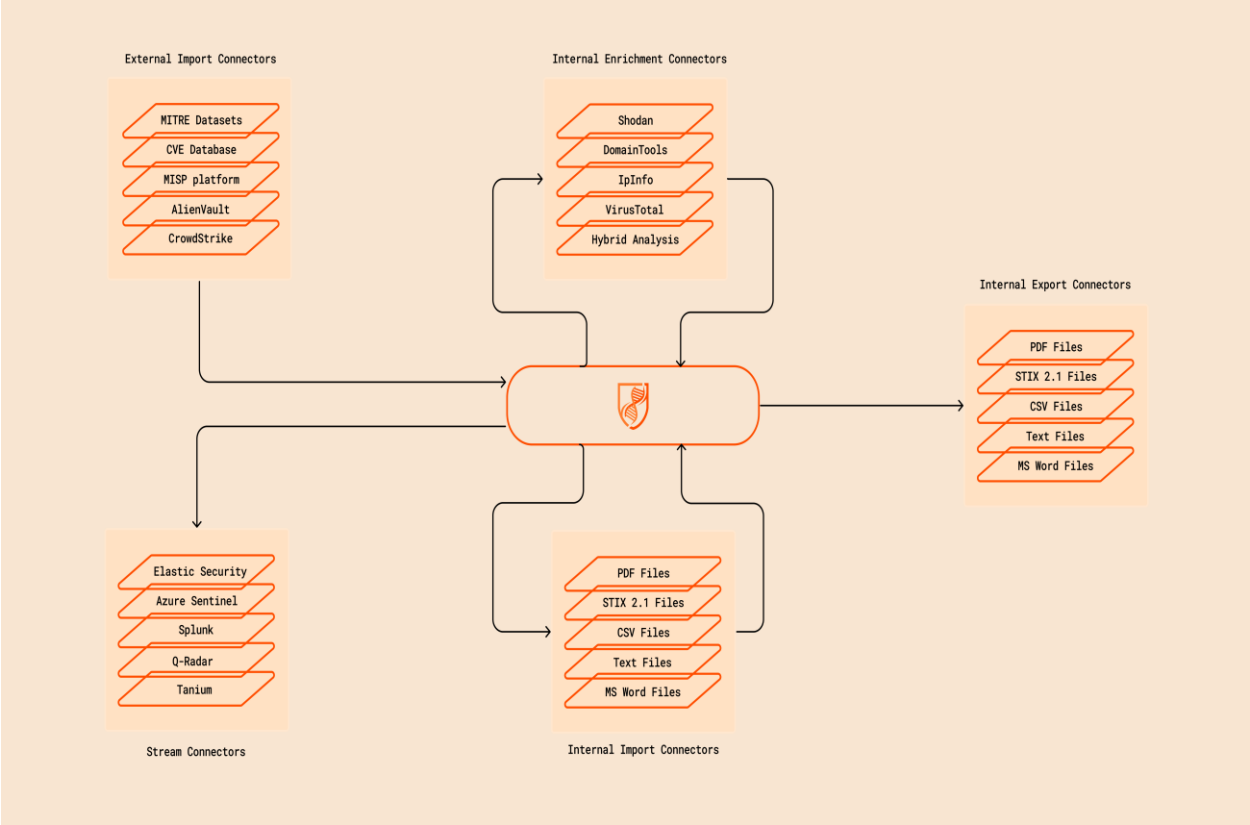


Рис. 2.5: Типи з’єднувачів OpenCTI.

Основні функції:

– Управління знаннями про загрози: OpenCTI дозволяє централізовано зберігати і структурувати знання про кіберзагрози. У платформі створюються об’єкти, які відображають загрози, TTPs (тактики, техніки, процедури), інциденти, та взаємозв’язки між ними. Це забезпечує користувачів можливістю будувати повні картини загроз для швидкого розуміння їхньої природи.

– Виявлення зв’язків між загрозами: завдяки використанню графічного представлення даних, OpenCTI дозволяє аналізувати зв’язки між групами кіберзлочинців, інфраструктурою атак, зловмисними файлами та індикаторами компрометації. Такий підхід сприяє формуванню глибших аналітичних висновків і пріоритизації заходів захисту.

– Імпорт даних із різних джерел: платформа підтримує автоматичний збір інформації з популярних OSINT-джерел (наприклад, MITRE ATT&CK, VirusTotal, Shodan), звітів sandbox (Cuckoo, Any.Run) та власних внутрішніх баз даних. Дані інтегруються через конектори, які можна адаптувати під специфічні потреби організації.

– Аналіз індикаторів компрометації (IoCs): OpenCTI надає інструменти для управління життєвим циклом IoCs — від виявлення до видалення з бази, коли вони втрачають актуальність. Крім цього, IoCs прив’язуються до інших об’єктів у системі, наприклад до кампаній чи шкідливого ПЗ, що дозволяє швидше виявляти залежності між компонентами загроз.

– Спільна робота з даними: платформа надає можливість спільної роботи для команд із різних відділів. Завдяки розмежуванню прав доступу, адміністратори можуть налаштовувати, хто має доступ до певних даних, наприклад, чутливої інформації або інформації про клієнтів.

– Інтерактивні дашборди та звіти: вбудовані інструменти дозволяють створювати візуалізації у вигляді графів, гістограм, та карт розташування атак. Ці дашборди допомагають керівникам безпеки приймати рішення на основі даних, а аналітикам швидше знаходити проблемні місця.

– Експорт даних у стандартизованих форматах: OpenCTI підтримує експорт даних у формати STIX 2.1, CSV та інші стандарти, що широко використовуються у сфері кіберзахисту. Це дозволяє легко передавати інформацію в SIEM-системи, платформи EDR або інші інструменти аналізу загроз.

– Автоматизація обробки даних: платформа використовує конектори для автоматичного збору, обробки та оновлення інформації. Конектори можуть запускатися за розкладом або працювати в режимі реального часу, залежно від потреб. Наприклад, вони можуть автоматично підтягувати останні оновлення баз MITRE ATT&CK.

– Гнучкість та налаштування: OpenCTI адаптується під специфічні потреби організації завдяки відкритому API та можливості створення кастомних конекторів і правил для автоматизації.

– Безпека та відповідність вимогам: для забезпечення відповідності правилам організації та стандартам (наприклад, GDPR), платформа підтримує мітки конфіденційності, аудит дій користувачів, а також сегрегацію даних між різними підрозділами.

2.3. Обґрунтування унікальності запропонованого рішення

Більшість існуючих систем для аналізу та управління кіберзагрозами, такі як **MISP** і **OpenCTI**, мають вузькоспеціалізовані завдання, орієнтовані на колекціонування, структурування, та інтерактивне відображення інформації про загрози. Основний акцент у цих платформах робиться на:

– Роботу з інцидентами: відстеження, реєстрація, аналіз і зв'язок між інцидентами.

– Графічний інтерфейс: спрощення аналізу загроз за допомогою інтерактивних візуалізацій і дашбордів.

– Інтеграцію з різними джерелами: збирання та агрегування даних із зовнішніх та внутрішніх джерел.

Проте вони не спрямовані на глибокий аналіз окремих файлів і не враховують особливості файлів, які представлені у звітах динамічного аналізу (sandbox). Зокрема, такі системи:

– Не фокусуються на детальному статичному аналізі файлів (метадані, секції, рядки, хеші).

– Не обробляють інформацію з кожного окремого файлу в sandbox-звітах. **Їхній акцент — це макрорівень загроз, а не деталізація малварного об'єкта**.

– Не надають інструментів для генерації сигнатур, які можуть бути використані для автоматичного виявлення шкідливого ПЗ в реальних системах.

Чому цей проєкт важливий та потрібний:

– Практичність: генеровані сигнатури можуть бути інтегровані в реальні інструменти для виявлення загроз, забезпечуючи пряме застосування аналітичної роботи.

– Деталізація: цей підхід враховує всі рівні даних — від метаданих до аналізу внутрішніх структур файлів.

– Недоступність таких можливостей у MISP і OpenCTI: ці платформи добре працюють для стратегічного рівня аналізу, але не пропонують засобів для тактичного використання детальних даних файлу.

На сьогоднішній день відсутні open-source системи, які б автоматично генерували сигнатури для виявлення шкідливого ПЗ на основі статичного аналізу файлів. Існуючі рішення, такі як MISP і OpenCTI, є найближчими аналогами, але їхня функціональність сфокусована на інших аспектах.

Ці системи не підтримують автоматизовану роботу зі статичним аналізом окремих файлів і не створюють сигнатури для реального використання в детектуванні.

Проєкт, спрямований на створення таких сигнатур із врахуванням метаданих, рядків, хешів та інших характеристик, заповнює цю прогалину, пропонуючи унікальну функціональність, недоступну у відкритих рішеннях.

Висновки до розділу 2

Сформовано вимоги до розроблюваної системи, які враховують специфіку аналізу шкідливого програмного забезпечення, включаючи ефективність, масштабованість, інтеграцію з іншими інструментами та зручність використання.

Проведено огляд існуючих рішень у цій галузі, їх переваг і недоліків. Особливу увагу приділено методам виявлення та аналізу ШПЗ, реалізованим у популярних інструментах. На основі цього аналізу було визначено недоліки, які не враховуються в поточних рішеннях, і потенційні напрями для вдосконалення.

Обґрунтовано унікальність запропонованого рішення, яке поєднує сучасні методи аналізу, зокрема створення сигнатур, їх інтеграцію через API та відображення у зручних форматах. Такий підхід забезпечує адаптивність системи до нових загроз і значно розширює її функціональні можливості порівняно з існуючими альтернативами.

РОЗДІЛ 3. Реалізація системи

3.1. Вибір технологій

У практичному розділі реалізації системи генерації сигнатур буде розглянуто розроблення MVP проєкту. Вибір технологій обумовлений швидкістю написання коду, легкою розширюваністю та масштабованістю.

3.1.1. База даних – MongoDB

MongoDB — це сучасна документно-орієнтована NoSQL база даних, яка зберігає дані у вигляді документів у форматі BSON (розширений JSON). Її функціональні можливості роблять її чудовим вибором для реалізації системи генерації сигнатур.

Переваги використання MongoDB для цього проєкту:

– Зручна структура зберігання даних: MongoDB дозволяє зберігати складні структури даних, такі як метадані файлів, хеші, секції та рядки, в одному документі. Це полегшує роботу з інформацією, зберігаючи її у зрозумілому й логічному вигляді.

– Швидкість та масштабованість: швидке читання та запис, MongoDB оптимізована для обробки великих обсягів даних, що важливо для роботи з файлами з sandbox-звітів.

– Горизонтальне масштабування: шардінг (розподіл даних між серверами) дозволяє працювати з великими масивами інформації без втрати продуктивності.

– Легка інтеграція з мовами програмування: MongoDB має бібліотеки та драйвери для більшості популярних мов, включаючи Python. Це спрощує розробку, оскільки інструменти для роботи з базою даних вже доступні.

– Гнучкість у зміні схеми: відсутність жорсткої схеми дозволяє легко адаптувати структуру збережених даних до нових типів метаданих чи інших артефактів без необхідності оновлення всієї бази. Це важливо для проєкту, де дані з різних джерел (наприклад, sandbox-звітів) можуть мати різні формати.

– Запити та агрегація: MongoDB підтримує потужні запити та агрегаційні функції, що дозволяють легко фільтрувати дані (наприклад,

вибрати всі файли з певними хешами чи секціями) та генерувати статистику для подальшого аналізу.

– Вбудована реплікація та резервування: MongoDB забезпечує високу доступність і захист даних завдяки реплікації (створення копій бази на кількох серверах). Це мінімізує ризик втрати даних у разі збоїв.

– Підтримка великих обсягів даних: MongoDB добре працює з великими обсягами неструктурованої інформації, що важливо для проєкту, де зберігаються різноманітні дані про файли.

Рис. 3.1: Структура бази даних MongoDB.

MongoDB ідеально підходить для цього проєкту завдяки своїй гнучкості, масштабованості та простоті роботи з різнотипними даними. Використання цієї бази даних дозволить ефективно зберігати, обробляти та масштабувати інформацію, необхідну для генерації сигнатур.

3.1.2. Мова програмування – Python

Python — високорівнева мова програмування загального призначення, яка є ідеальним вибором для реалізації проєкту генерації сигнатур. Її популярність, багата екосистема бібліотек і простота у використанні забезпечують ефективну реалізацію ключових завдань.

Переваги використання Python для цього проєкту:

– Простота та швидкість розробки: Python має зрозуміний синтаксис, що знижує час написання коду. Це дозволяє швидко розробляти та тестувати функціонал MVP (мінімально життєздатного продукту).

– Багата екосистема бібліотек: Для Python доступні потужні бібліотеки, що охоплюють усі аспекти, необхідні для проєкту, такі як *статичний аналіз* файлів (pefile, lief) для роботи з PE, ELF, та іншими форматами файлів, робота з базами даних (PyMongo) для інтеграції з MongoDB, обробка даних (pandas, numpy) для аналізу та маніпуляції великими масивами даних, криптографія та хешування (hashlib, cryptography) для обчислення хешів.

– Кросплатформеність: Python працює на всіх основних операційних системах (Windows, Linux, macOS), що спрощує адаптацію проєкту до різних середовищ.

– Широка спільнота та підтримка: Python має одну з найбільших спільнот розробників у світі. Це забезпечує доступ до великої кількості документації, прикладів та рішень для вирішення технічних проблем.

– Легка інтеграція: Python легко інтегрується з іншими мовами та системами, що дозволяє додавати підтримку нових функцій або взаємодіяти з іншими компонентами інфраструктури.

– Автоматизація процесів: завдяки бібліотекам на кшталт argparse та click, Python зручний для створення скриптів автоматизації, що спрощує виконання завдань, таких як обробка даних із sandbox-звітів чи підготовка сигнатур.

– Масштабованість: Python підходить для прототипування (MVP) та може бути масштабований за допомогою оптимізацій або переписування критичних компонентів на швидших мовах (C, C++) з використанням розширень (ctypes, cython).

– Інструменти тестування та розгортання: Python підтримує сучасні інструменти для тестування (unittest, pytest) і розгортання (наприклад, через Docker, Kubernetes), що полегшує підтримку та розвиток проєкту.

3.2. Імплементація компонентів системи

3.2.1. Enrichment

Відобразимо найбільш простий підхід до **збору даних**. За допомогою універсального формату завдання для обробки будемо працювати через колекцію бази даних як через чергу. Покласти завдання можна з будь-якого ресурсу, вказаваши мінімальні поля потрібні для подальшого аналізу.

Формат завдання для аналізу:

```
from __future__ import annotations

from dataclasses import dataclass

__all__ = ["StaticReportTask"]

@dataclass(kw_only=True)
class StaticReportTask:
    id: str
    """ SHA256. """
    link: str
    name: str | None
    positives: int
```

Наступним кроком буде скачування файлу для аналізу та його аналіз.

Інтерфейс класу для тимчасового скачування:

```
from __future__ import annotations

from contextlib import AbstractContextManager
from pathlib import Path
from typing import Protocol


__all__ = ["FileDownloader"]


class FileDownloader(Protocol):
    def tmp_download(self, link: str) -> AbstractContextManager[Path]:
        ...
```

Далі ми маємо зробити статичний аналіз файлу.

Поверхнево деталі виконуваного файлу мають наступний вигляд:

```
@dataclass
class PEInfo:
    rich_header: RichHeader | None
    file_header: FileHeader
    optional_header: OptionalHeader
    version_info: VersionInfo | None
    export_info: ExportInfo
    import_info: ImportInfo
    resource_directory: ResourceDirectory
    debug_info: DebugInfo
    signature: SignatureInfo
```

Процес нормалізації та статичного аналізу відбувається в рівні реалізації інтерфейсів.

Більш детальний код парсингу виконуваного файлу та зведення його до заданого формату буде наведено в додатках.

Наостанок проаналізовані дані потрібно зберігти.

Наведемо інтерфейс для взаємодії з базою даних:

```
from __future__ import annotations

from typing import Protocol

from ....domain.static_report.entities import StaticReport, StaticReportTask


__all__ = ["StaticReportDao"]


class StaticReportDao(Protocol):
    def get_one(self, id_: str) -> StaticReport | None:
        ...

    def update_last_seen(self, report: StaticReport) -> None:
        ...

    def create_one(self, report: StaticReport) -> None:
        ...
```

Загальний процес виконання етапу збагачення наведено в класі прикладного архітектурного рівня, що використовує інтерфейси для взаємодії з зовнішніми компонентами:

```
from __future__ import annotations

import logging
from dataclasses import dataclass

from ..interfaces import (
    FileDownloader,
    StaticAnalyser,
    StaticReportDao,
)
from ....domain.static_report.entities import StaticReport, StaticReportTask

__all__ = ["StaticReportProcessTaskInteractor"]

logger = logging.getLogger(__name__)


@dataclass
class StaticReportProcessTaskInteractor:
    dao: StaticReportDao
    downloader: FileDownloader
    analyser: StaticAnalyser

    def execute(self, task: StaticReportTask) -> None:
        logger.info("Run static report process task interactor")

        report = self.dao.get_one(task.id)
        if report is not None:
            report.mark_as_seen()
            self.dao.update_last_seen(report)
            return

        with self.downloader.tmp_download(task.link) as filepath:
            analysis = self.analyser.analyse(filepath)

        report = create_static_report(analysis, task)
        self.dao.create_one(report)
```

3.2.2. Generation

Найголовнішою деталлю цього етапу безумовно є групування даних проаналізованих файлів.

Реалізація цього зроблено за допомогою використаної бази даних. У цьому проєкті реалізація такого підходу здійснена за допомогою MongoDB, яка відмінно підходить для обробки великих обсягів даних та забезпечення швидкого доступу до них.

Розберемо генерацію сигнатур для цифрового підпису файлів. Цифровий підпис є одним з найважливіших факторів, що забезпечує автентичність файлів та програмного забезпечення, а також гарантує, що вони не були змінені після підписання. Розглянемо імплементацію на прикладі наступного коду.

Цей код описує клас *SignatureSignerGeneratorMongoImpl*, який реалізує генерацію сигнатур за допомогою MongoDB. Ось детальне пояснення кожної частини коду.

Опис класу:

```
class SignatureSignerGeneratorMongoImpl(
    BaseDao[StaticReport],
    SignatureGenerator[SignatureSigner],
):
```

Клас *SignatureSignerGeneratorMongoImpl* поєднує два інтерфейси:

– *BaseDao* працює з MongoDB, забезпечуючи базові функції для взаємодії з базою даних (наприклад, збереження та запити).

– *SignatureGenerator* — інтерфейс для генерації сигнатур, визначає, як повинна виглядати логіка створення сигнатур.

Атрибути класу:

```
collection_metadata = static_report_collection
signature_type = SignatureSigner
```

– *collection_metadata* вказує на колекцію в MongoDB, з якої будуть витягуватись статичні звіти.

– *signature_type* визначає тип об'єкта, що буде створюватися в результаті — це *SignatureSigner*, клас для представлення згенерованих підписів.

Метод *generate*:

```
def generate(
    self,
    min_count: int,
    date_after: datetime | None,
    date_before: datetime | None,
) -> list[SignatureSigner]:
```

Метод відповідає за безпосередню генерацію сигнатур на основі даних з MongoDB. Параметри:

– *min_count*: мінімальна кількість зразків для кожної сигнатури.

– *date_after* та *date_before*: фільтри для обмеження вибірки даних за датою.

Логіка формування **pipeline**:

```
pipeline: ListStrAnyDict = [
    *build_date_filter_stages(date_after, date_before),
    {"$match": {"signer": {"$ne": None}}},
    {
        "$group": {
            "_id": "$signer",
            "version_info_s": {"$addToSet": "$version_info"},
            **COMMON_GROUP_CLAUSE,
        }
    },
    *build_count_filter_stages(min_count),
    COMMON_ADD_FIELDS_FILTER_NAMES_NOT_NONE_STAGE,
    ...
    COMMON_ADD_FIELDS_LIMIT_ARRAYS_SIZE_STAGE,
]
```

pipeline — це набір етапів агрегації для MongoDB, який визначає, як будуть оброблятися дані перед їх поверненням. Включає:

– Фільтрація за датою: *build_date_filter_stages* додає умови для обмеження вибірки за діапазоном дат.

– Фільтрація по полю "*signer*": виключаються записи, де поле "signer" є None.

– Групування: дані групуються за полем *signer*, при цьому для кожного підпису збираються всі пов'язані версії через $addToSet, уникаючи дублювання.

– Фільтрація за мінімальною кількістю зразків: b*uild_count_filter_stages* додає фільтр, щоб включати лише підписи з мінімальною кількістю зразків.

Функції *build_date_filter_stages* та *build_count_filter_stages* використовуються для побудови етапів фільтрації в пайплайні агрегації MongoDB, що дозволяють налаштувати пошук даних відповідно до певних критеріїв:

```
def build_date_filter_stages(
    after: datetime | None,
    before: datetime | None,
) -> ListStrAnyDict:
    ops = []
    if after is not None:
        ops.append({"$gt": after})
    if before is not None:
        ops.append({"$lt": before})
    filters = [{"created_at": op} for op in ops]
    if not ops:
        return []
    return [{"$match": {"$and": filters}}]


def build_count_filter_stages(min_count: int) -> ListStrAnyDict:
    if min_count < 2:
        return []
    return [{"$match": {"count": {"$gt": min_count}}}]
```

У наступному фрагменті коду представлено клас з прикладного рівня для взаємодії з генератором сигнатур у контексті проєкту. Основним компонентом є клас *SignatureGenerateInteractor*, який використовує генератор сигнатур і DAO (Data Access Object) для обробки та збереження згенерованих сигнатур.

Клас *SignatureGenerateInteractor* має два основних компоненти:

– **generator**: об'єкт, який реалізує інтерфейс *SignatureGenerator* і відповідає за сам процес генерації сигнатур. Він використовує методи для отримання згенерованих сигнатур, в залежності від заданих параметрів. Реалізація генерації в MongoDB наведена вище.

– **dao**: об'єкт, який реалізує інтерфейс *SignatureDao*, і використовується для збереження або оновлення згенерованих сигнатур у базі даних. Це може бути будь-який механізм для роботи з даними, наприклад, MongoDB, SQL бази даних тощо.

```
@dataclass
class SignatureGenerateInteractor(Generic[SignatureT]):
    generator: SignatureGenerator[SignatureT]
    dao: SignatureDao[SignatureT]

    def execute(self, params: SignatureGenerateParams) -> None:
        logger.info(
            "Run signature generate service",
            extra={"ctx": {"dao": repr(self.dao), "params": params}},
        )

        signatures = self._generate(params)
        self.dao.upsert_many(signatures)

    def _generate(self, params: SignatureGenerateParams) -> list[SignatureT]:
        signatures = self.generator.generate(
            min_count=params.min_count,
            date_after=params.date_after,
            date_before=params.date_before,
        )
        return signatures
```

Цей код ілюструє архітектурний підхід до розробки через використання принципів розділення відповідальності та інтерфейсів. Клас *SignatureGenerateInteractor* відповідає за координацію між генератором сигнатур і DAO, не втручаючись у деталі їх реалізації. Такий підхід спрощує розширюваність та тестування системи.

3.2.3. Export

На етапі експорту буде розглянуто два ключові підходи для роботи з результатами генерації сигнатур:

– **JSON** через **API**: експорт сигнатур у форматі JSON дозволить інтегрувати дані з іншими системами або автоматизованими скриптами. Це забезпечує легку передачу структурованих даних і можливість налаштування відповідно до вимог інтеграції. Такий формат є універсальним, добре підтримується сучасними API та дозволяє ефективно працювати з великими обсягами інформації.

– Рендеринг у **HTML**: для зручності користувачів, дані можуть бути представлені у вигляді візуального HTML-звіту. Це дає змогу переглядати згенеровані сигнатури в читабельному форматі через браузер, із можливістю пошуку, сортування та навігації. HTML-рендеринг також може бути адаптований для інтерактивності, наприклад, для використання фільтрів або інших елементів керування.

Цей код реалізує простий ендпоінт для отримання та відображення підписів (signers) через Flask Blueprint. Розглянемо кожен компонент детальніше:

```
@bp.get("/signers")
def get_signers() -> str:
    dao: SignatureDao[SignatureSigner] = g.get("signer_dao")
    query = validate_query(SignatureQuery)
    signatures = dao.get_many(query.dict())
    return render_template(
        "signatures/signers.html",
        signatures=signatures,
    )
```

– Шлях ендпоінту: Цей декоратор створює HTTP GET-ендпоінт за адресою “/signers”. Він використовується для отримання списку сигнатур.

– Доступ до DAO (Data Access Object): Об'єкт dao (реалізація *SignatureDao*) дістається із контексту Flask (g), що дозволяє зберігати об'єкти,

специфічні для кожного запиту. DAO відповідає за взаємодію з базою даних, яка містить сигнатури підписів.

– Валідація запиту: Функція validate_query перевіряє параметри запиту, передані клієнтом, за допомогою класу *SignatureQuery*. Це дозволяє гарантувати, що запит сформовано коректно.

– Отримання даних із бази: Метод *dao.get_many* використовує параметри запиту, перетворені у формат словника (query.dict()), щоб отримати потрібні записи з бази даних.

– Рендеринг шаблону HTML: Отримані підписи передаються в шаблон signatures/signers.html, який генерує HTML-сторінку для їх відображення. Шаблон дозволяє зручно візуалізувати дані у браузері.

Цей декоратор створює HTTP GET-ендпоінт за адресою /signers. Він використовується для отримання списку підписів.

Експорт в JSON ще простіше, тому що він не потребує додаткового рендерингу. Цей код реалізує API ендпоінт для отримання сигнатур в JSON форматі за допомогою HTTP запиту:

```
@bp.get("/api/signers")
def get_signers() -> list[dict]:
    dao: SignatureDao[SignatureSigner] = g.get("signer_dao")
    query = validate_query(SignatureQuery)
    signatures = dao.get_many(query.dict())
    return [signature.serialize() for signature in signatures]
```

Головною відмінністю тут є інший шлях ендпоінту та виклик серіалізації в об'єктах сигнатур.

– Шлях ендпоінту: цей декоратор створює HTTP GET-ендпоінт, який відповідає на запити за адресою "*/api/signers*". Метою ендпоінту є повернення серіалізованих даних у форматі JSON.

– Серіалізація об'єктів: кожен об'єкт *SignatureSigner*, отриманий із бази, перетворюється у словник (серіалізується) за допомогою методу *serialize()*. У результаті ендпоінт повертає список словників, що легко передається у форматі JSON.

Результат рендерингу для веб сторінки зображено на рис. 3.2.

Окрім виведення сигнатур ми також можемо побачити набір можливих фільтрів, за допомогою яких можна вивести потрібні та актуальні записи.

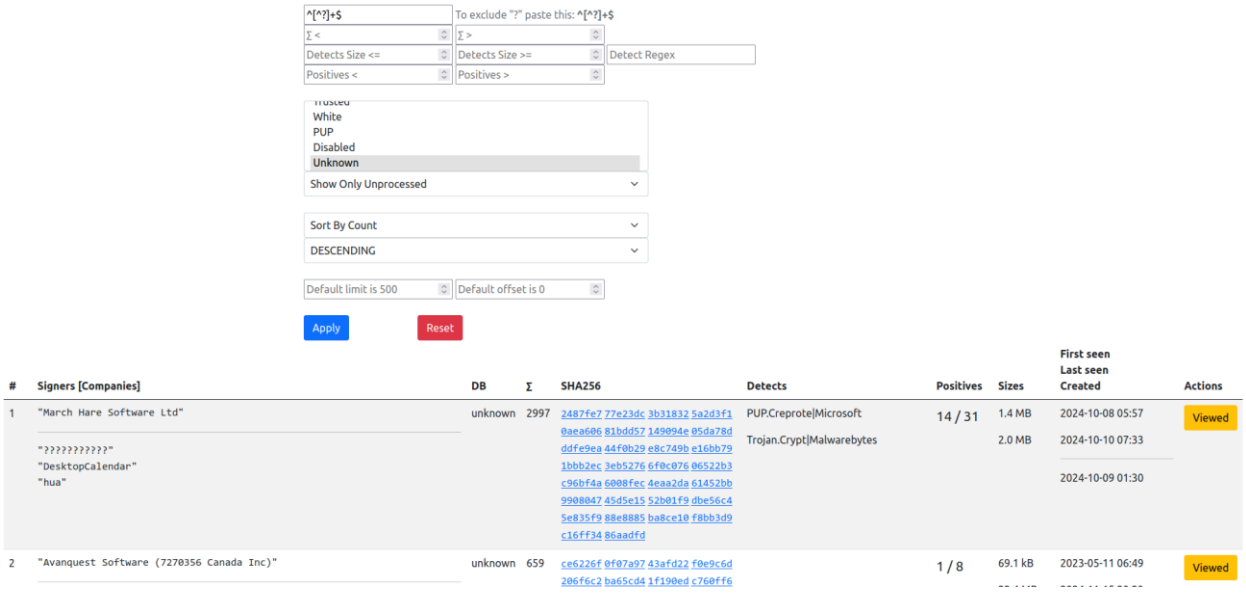


Рис. 3.2: HTML відображення сигнатур.

Спільні та найважливіші поля сигнатур, що експортуються, наведено далі:

```
@dataclass(kw_only=True)
class BaseSignature(CreatedAtMixin):
    id: str
    """ Signature. Must have the format of real DB signature. """
    count: int
    """ Count of matched static reports on generation. """
    hashes: list[str]
    """ List of SHA256 matched static reports. """
    names: list[str]
    """ List of `Type.Family|Vendor`. """
    min_size: int
    max_size: int
    """ Min/max size in bytes. """
    min_positives: int
    max_positives: int
    """ Min/Max positives. """
    first_seen_at: datetime
    last_seen_at: datetime
    """ Min/Max of `StaticReport.created_at`. """
    viewed_at: datetime | None = None
    """ Set when analyst manually viewed signature. """
```

Приклад реальної згенерованої сигнатури в JSON форматі зображено на рис. 3.3. Скриншот зроблено у додатку MongoDB Compass, що дозволяє зручно маніпулювати даними в MongoDB через GUI.

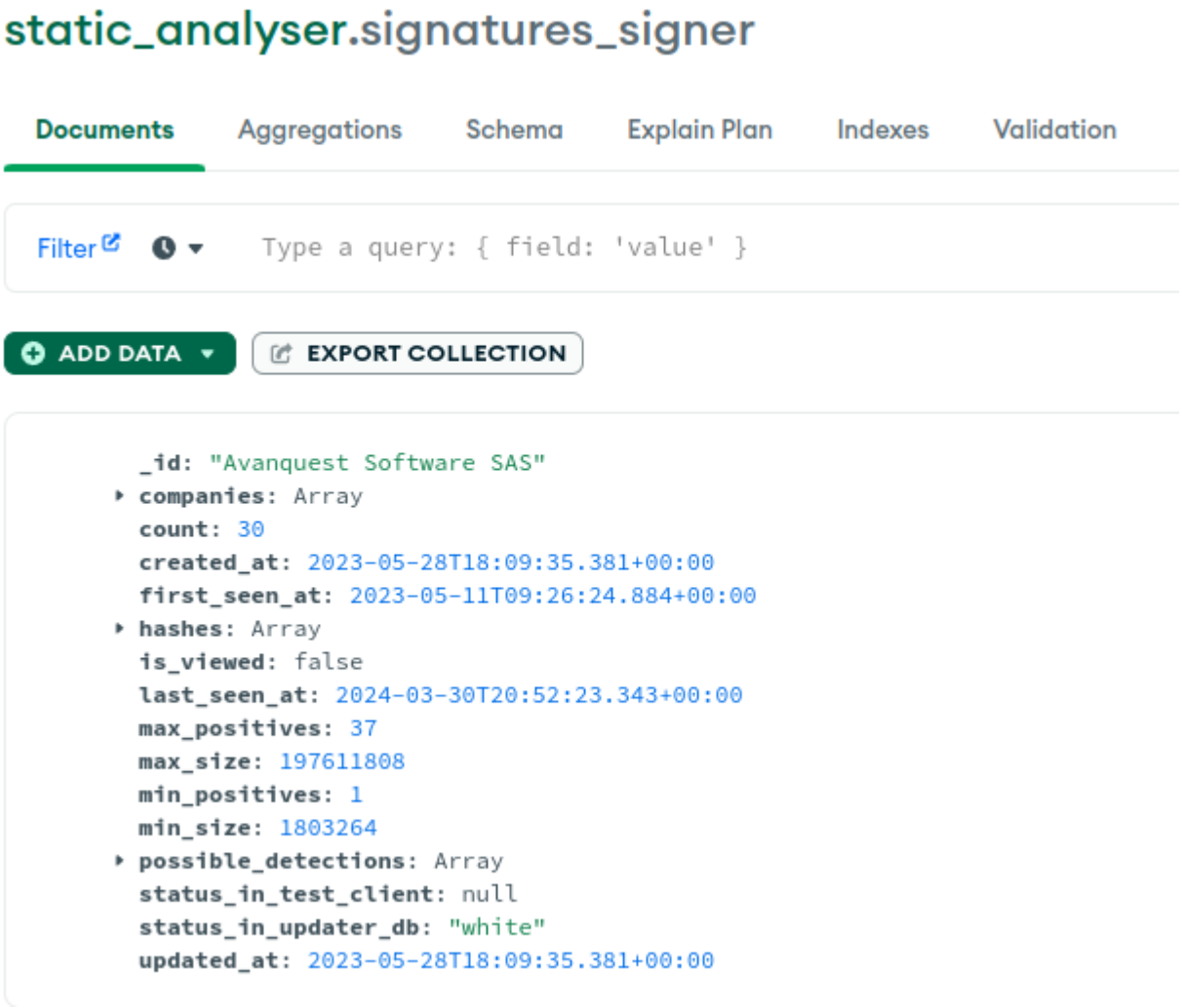


Рис. 3.3: Приклад реальної сигнатури у додатку MongoDB Compass.

Висновки до розділу 3

Досліджено процес реалізації системи, що включає вибір відповідних технологій і імплементацію компонентів відповідно до сформованих у попередньому розділі вимог.

Для зберігання даних обрано MongoDB, яка забезпечує гнучкість у роботі зі структурованою та напівструктурованою інформацією, необхідною для аналізу шкідливого програмного забезпечення. Як основну мову програмування використано Python, що дозволило ефективно реалізувати алгоритми аналізу, обробки даних та інтеграції компонентів системи.

У рамках реалізації системи виконано розробку ключових модулів, таких як генерація сигнатур, API для доступу до даних, а також інтеграція веб-інтерфейсу для зручного представлення результатів. Імплементація базується на вимогах, сформованих у другому розділі, що забезпечило відповідність системи поставленим завданням і високий рівень функціональності.

ВИСНОВКИ

Актуальність проблеми боротьби зі шкідливим програмним забезпеченням зумовила проведення даного дослідження. У даній кваліфікаційній роботі вирішено завдання розробки автоматизованої системи генерації сигнатур для виявлення шкідливого програмного забезпечення (ШПЗ). Робота охоплює аналіз предметної області, формування вимог до системи, огляд існуючих рішень та їх недоліків, а також практичну реалізацію та тестування розробленої системи.

У першому розділі було досліджено загальну характеристику шкідливого програмного забезпечення, наведено основні підходи до його аналізу, зокрема статичний і динамічний методи, та описано різні типи методів виявлення ШПЗ. Особливу увагу приділено сигнатурному методу як одному з найефективніших у сучасній практиці.

Другий розділ присвячено визначенню вимог до автоматизованої системи генерації сигнатур, а також аналізу існуючих рішень у цій галузі. Проведено порівняння методів і інструментів, доступних на ринку, та обґрунтовано унікальність запропонованого підходу, який поєднує ефективність, інтеграцію через API та зручність використання.

Третій розділ охоплює практичну реалізацію системи. Для розробки обрано технології MongoDB і Python, які забезпечують гнучкість та продуктивність. Реалізовано автоматизовану систему, що генерує сигнатури, зберігає їх у базі даних, надає доступ через API та демонструє результати у веб-інтерфейсі. Наведено приклади згенерованих сигнатур, які підтверджують функціональність системи.

Загалом, виконана кваліфікаційна робота демонструє, що запропонована автоматизована система генерації сигнатур є ефективним інструментом для виявлення шкідливого програмного забезпечення. Вона дозволяє автоматизувати рутинні процеси аналізу та виявлення, покращує швидкість і точність виявлення ШПЗ, а також забезпечує можливість інтеграції з іншими системами кібербезпеки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software / ed. by H. Andrew. San Francisco : No Starch Press, 2012. 800 p.

2. Saxe J., Sanders H. Malware Data Science: Attack Detection and Attribution. No Starch Press, 2018. 272 p.

3. Marak V. Windows Malware Analysis Essentials: Master the fundamentals of malware analysis for the Windows platform and enhance your anti-malware skill set. Packt Publishing, 2015. 330 p.

4. Malware forensics field guide for Windows systems: Digital forensics field guides / ed. by C. Eoghan, A. J. M. Waltham, MA : Syngress, 2012. 560 p.

5. Malware Detection / C. Wang et al. Springer London, Limited, 2007. 312 p.

6. Mohanta A., Saldanha A. Malware Analysis and Detection Engineering. Berkeley, CA : Apress, 2020. URL: https://doi.org/10.1007/978-1-4842-6193-4 (date of access: 18.11.2024).

7. Mastering Malware Analysis: The Complete Malware Analyst's Guide to Combating Malicious Software, APT, Cybercrime, and IoT Attacks. de Gruyter GmbH, Walter, 2019.

8. Aycock J. Computer Viruses and Malware (Advances in Information Security Book 22). Springer, 2006. 228 p.

9. The art of computer virus research and defense. Upper Saddle River, NJ : Addison-Wesley, 2005. 713 p.

10. Barker D. Malware Analysis Techniques: Tricks for the Triage of Adversarial Software. de Gruyter GmbH, Walter, 2021.

11. Thuraisingham B., Khan L., Masud M. Data Mining Tools for Malware Detection. Auerbach Publishers, Incorporated, 2016. 450 p.

12. What is Malware? Malware Definition, Types and Protection. Malwarebytes. URL: https://www.malwarebytes.com/malware (date of access: 18.11.2024).

13. Waliulu, Raditya Faisal, and Teguh Hidayat Iskandar Alam. "Reverse Engineering Analysis Statis Forensic Malware Webc2-Div." Insect (Informatics and Security): Jurnal Teknik Informatika 4, no. 1 (August 23, 2019): 15. http://dx.doi.org/10.33506/insect.v4i1.223.

14. Потенційно небажане ПЗ (PUA) | Gridinsoft. ТОВ "Грідінсофт". URL: https://gridinsoft.ua/unwanted-program (date of access: 18.11.2024).

15. Що таке Зловмисні Програми та як вони працюють? Визначення та приклади | Gridinsoft. ТОВ "Грідінсофт". URL: https://gridinsoft.ua/malware (date of access: 18.11.2024).

16. Що таке шкідливі програми? Симптоми та рішення. NordVPN. URL: https://nordvpn.com/uk/cybersecurity/what-is-malware/ (date of access: 18.11.2024).

17. Учасники проектів Вікімедіа. Виконуваний файл – Вікіпедія. Вікіпедія. URL: https://uk.wikipedia.org/wiki/Виконуваний_файл (date of access: 18.11.2024).

18. PE Format - Win32 apps. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format (date of access: 18.11.2024).

19. YILMAZ F. PE Format. Malware-Reverse Blog. URL: https://onlyf8.com/pe-format (date of access: 18.11.2024).

20. Static Malware Analysis vs Dynamic Malware Analysis - Comparison Chart. Malwation. URL: https://www.malwation.com/blog/static-malware-analysis-vs-dynamic-malware-analysis-comparison-chart (date of access: 18.11.2024).

21. What Is Signature-Based Detection? | Corelight. Corelight: Evidence-Based NDR and Threat Hunting Platform. URL: https://corelight.com/resources/glossary/signature-based-detection (date of access: 18.11.2024).

ДОДАТОК А. ЛІСТИНГ КОДУ

pe_parser.py

```
from __future__ import annotations

import struct
from ti.application.dto.static_file_analyser import PEInfo, RichHeader, FileHeader, OptionalHeader, VersionInfo, ExportInfo, ImportInfo, Flag, FlagCharacteristics
from abc import abstractmethod
from pathlib import Path
import pefile
from pefile import retrieve_flags, IMAGE_CHARACTERISTICS
import signify
from dnfile import dnPE

OPTIONAL_HEADER_MAGIC_PE_STRING = 'PE'
PE_PLUS = 'PE+'
PE_TYPE = {
    pefile.OPTIONAL_HEADER_MAGIC_PE: OPTIONAL_HEADER_MAGIC_PE_STRING,
    pefile.OPTIONAL_HEADER_MAGIC_PE_PLUS: PE_PLUS,
}


class PEParser:
    @abstractmethod
    def get_info(self, filepath: Path) -> PEInfo:
        pass


class PEParserPythonPEFileImpl(PEParser):
    pe: pefile.PE

    def _parse_rich_header(self) -> RichHeader | None:
        if self.pe.RICH_HEADER is None:
            return None
        return RichHeader(
            md5=self.pe.get_rich_header_hash(),
        )

    def _parse_file_header(self) -> FileHeader:
```

```
                structure = self.pe.FILE_HEADER
                return FileHeader(
                    machine=Flag(
                        name=pefile.MACHINE_TYPE[structure.Machine],
                        value=structure.Machine,
                    ),
                    number_of_sections=structure.NumberOfSections,
                    time_date_stamp=structure.TimeDateStamp,
                    pointer_to_symbol_table=structure.PointerToSymbolTable,
                    number_of_symbols=structure.NumberOfSymbols,
                    size_of_optional_header=structure.SizeOfOptionalHeader,
                    characteristics=self._retrieve_file_header_characteristics(),
                )

            def _retrieve_file_header_characteristics(self) -> FlagCharacteristics:
                """
                image_flags = retrieve_flags(IMAGE_CHARACTERISTICS, "IMAGE_FILE_")
                flags = []
                for flag in sorted(image_flags):
                    if getattr(self.pe.FILE_HEADER, flag[0]):
                        flags.append(flag[0])
                return flags
                """

                image_flags = retrieve_flags(IMAGE_CHARACTERISTICS, "IMAGE_FILE_")
                flags = []
                for flag_name, flag_value in sorted(image_flags):
                    if getattr(self.pe.FILE_HEADER, flag_name):
                        flags.append(Flag(name=flag_name, value=flag_value))
                return FlagCharacteristics(
                    value=self.pe.FILE_HEADER.Characteristics,
                    flags=flags,
                )

            def _parse_optional_header(self) -> OptionalHeader:
                structure = self.pe.OPTIONAL_HEADER
                return OptionalHeader(
                    magic=Flag(PE_TYPE[structure.Magic],  structure.Magic),    #  TODO:
`rom` is not handled
                    major_linker_version=structure.MajorLinkerVersion,
                    minor_linker_version=structure.MinorLinkerVersion,
```

```
                size_of_code=structure.SizeOfCode,
                size_of_initialized_data=structure.SizeOfInitializedData,
                size_of_uninitialized_data=structure.SizeOfUninitializedData,
                address_of_entry_point=structure.AddressOfEntryPoint,
                base_of_code=structure.BaseOfCode,
                image_base=structure.ImageBase,
                section_alignment=structure.SectionAlignment,
                file_alignment=structure.FileAlignment,
                major_os_version=structure.MajorOperatingSystemVersion,
                minor_os_version=structure.MinorOperatingSystemVersion,
                major_image_version=structure.MajorImageVersion,
                minor_image_version=structure.MinorImageVersion,
                major_subsystem_version=structure.MajorSubsystemVersion,
                minor_subsystem_version=structure.MinorSubsystemVersion,
                size_of_image=structure.SizeOfImage,
                size_of_headers=structure.SizeOfHeaders,
                check_sum=structure.CheckSum,
                subsystem=Flag(
                    pefile.SUBSYSTEM_TYPE[structure.Subsystem],
                    structure.Subsystem,
                ),
                dll_characteristics=self._parse_dll_characteristics(),
                size_of_stack_reserve=structure.SizeOfStackReserve,
                size_of_stack_commit=structure.SizeOfStackCommit,
                size_of_heap_reserve=structure.SizeOfHeapReserve,
                size_of_heap_commit=structure.SizeOfHeapCommit,
                number_of_rva_and_sizes=structure.NumberOfRvaAndSizes,
                actual_check_sum=self.pe.generate_checksum(),
            )

        def _parse_dll_characteristics(self) -> FlagCharacteristics:
            dll_flags         =         retrieve_flags(pefile.DLL_CHARACTERISTICS,
"IMAGE_DLLCHARACTERISTICS_")
            flags = []
            for flag_name, flag_value in sorted(dll_flags):
                if getattr(self.pe.OPTIONAL_HEADER, flag_name):
                    flags.append(Flag(name=flag_name, value=flag_value))
            return FlagCharacteristics(
                value=self.pe.OPTIONAL_HEADER.DllCharacteristics,
                flags=flags,
            )
```

```
            def _parse_version_info(self) -> VersionInfo | None:
                data = self._get_versioninfo(self.pe)
                to_dict = {d['name']: d['value'] for d in data}
                return VersionInfo(
                    comments=to_dict.get('Comments', ''),
                    company_name=to_dict.get('CompanyName', ''),
                    file_description=to_dict.get('FileDescription', ''),
                    file_version=to_dict.get('FileVersion', ''),
                    internal_name=to_dict.get('InternalName', ''),
                    legal_copyright=to_dict.get('LegalCopyright', ''),
                    legal_trademarks=to_dict.get('LegalTrademarks', ''),
                    original_filename=to_dict.get('OriginalFilename', ''),
                    product_name=to_dict.get('ProductName', ''),
                    product_version=to_dict.get('ProductVersion', ''),
                )

            def _get_versioninfo(self, pe: pefile.PE) -> list[dict] | None:

                peresults = []

                if not hasattr(pe, "FileInfo"):
                    return None
                import codecs

                for infoentry in pe.FileInfo:
                    for entry in infoentry:
                        if hasattr(entry, "StringTable"):
                            for st_entry in entry.StringTable:
                                for str_entry in st_entry.entries.items():
                                    entry  =  {"name":  str_entry[0].decode("latin-1"),
"value": str_entry[1].decode("latin-1")}
                                    if    entry["name"]    ==    "Translation"    and
len(entry["value"]) == 10:
                                        entry["value"]   =   f"0x0{entry['value'][2:5]}
0x0{entry['value'][7:10]}"
                                    peresults.append(entry)
                        elif hasattr(entry, "Var"):
                            for var_entry in entry.Var:
                                if hasattr(var_entry, "entry"):
                                    name = list(var_entry.entry.keys())[0]
```

```
                                value = list(var_entry.entry.values())[0]
                                if isinstance(name, bytes):
                                    name = name.decode("latin-1")
                                if isinstance(value, bytes):
                                    value = value.decode("latin-1")
                                entry = {
                                    "name": name,
                                    "value": value,
                                }
                                if    entry["name"]    ==    "Translation"    and
len(entry["value"]) == 10:
                                    entry["value"]   =   f"0x0{entry['value'][2:5]}
0x0{entry['value'][7:10]}"
                                peresults.append(entry)

            return peresults

        def _parse_exports(self) -> ExportInfo:
            symbols = self.get_exported_symbols(self.pe)
            exports = [symbol["name"] for symbol in symbols]
            return ExportInfo(
                exports=exports,
                exported_dll_name=self.get_exported_dll_name(self.pe),
            )

        def get_exported_dll_name(self, pe: pefile.PE) -> str:
            """Gets exported DLL name, if any
            @return: exported DLL name as string or None.
            """
            if not pe:
                return None

            if hasattr(pe, "DIRECTORY_ENTRY_EXPORT"):
                dllname                                                    =
pe.get_string_at_rva(pe.DIRECTORY_ENTRY_EXPORT.struct.Name)
                return dllname.decode("latin-1")
            return None

        def get_exported_symbols(self, pe: pefile.PE) -> List[dict]:
            """Gets exported symbols.
            @return: list of dicts of exported symbols or None.
```

```
            """
            if not pe:
                return None

            exports = []

            if hasattr(pe, "DIRECTORY_ENTRY_EXPORT"):
                exports.extend(
                    {
                        "address":        hex(pe.OPTIONAL_HEADER.ImageBase        +
exported_symbol.address),
                        "name":      exported_symbol.name.decode("latin-1")      if
exported_symbol.name else "",
                        "ordinal": exported_symbol.ordinal,
                    }
                    for exported_symbol in pe.DIRECTORY_ENTRY_EXPORT.symbols
                )
            return exports

        def _parse_import_info(self) -> ImportInfo:
            symbols = self.get_imported_symbols(self.pe)
            imports = [symbol["name"] for symbol in symbols]
            return ImportInfo(
                imphash=self.pe.get_imphash(),
                imported_dll_count=len(symbols),
                imports=imports,
            )

        def get_imported_symbols(self, pe: pefile.PE) -> dict[str, dict]:
            """Gets imported symbols.
            @return: imported symbols dict or None.
            """
            if not pe:
                return None
            imports = {}
            if not hasattr(pe, "DIRECTORY_ENTRY_IMPORT"):
                return imports

            for entry in pe.DIRECTORY_ENTRY_IMPORT:
                try:
                    symbols = [
```

```
                        {"address":     hex(imported_symbol.address),     "name":
imported_symbol.name.decode("latin-1")}
                        for imported_symbol in entry.imports
                        if imported_symbol.name and imported_symbol.address
                    ]
                    dll_name = entry.dll.decode("latin-1").split(".", 1)[0]
                    if dll_name in imports:
                        imports[dll_name]["imports"].extend(symbols)
                    else:
                        imports[dll_name] = {
                            "dll": entry.dll.decode("latin-1"),
                            "imports": symbols,
                        }
                except Exception as e:
                    # log.error(e, exc_info=True)
                    continue
            return imports

        def get_pdb_path(self, pe: pefile.PE) -> str:
            if not pe or not hasattr(pe, "DIRECTORY_ENTRY_DEBUG"):
                return None

            try:
                for dbg in pe.DIRECTORY_ENTRY_DEBUG:
                    dbgst = dbg.struct
                    dbgdata      =      pe.__data__[dbgst.PointerToRawData      :
dbgst.PointerToRawData + dbgst.SizeOfData]
                    if dbgst.Type == 2:  # CODEVIEW
                        if dbgdata[:4] == b"RSDS":
                            return dbgdata[24:].decode("latin-1").rstrip("\0")
                        elif dbgdata[:4] == b"NB10":
                            return dbgdata[16:].decode("latin-1").rstrip("\0")
                    elif dbgst.Type == 4:  # MISC
                        if len(dbgdata) == 9:
                            length = struct.unpack_from("IIB", dbgdata)[1]
                            return                dbgdata[12:length].decode("latin-
1").rstrip("\0")
            except Exception as e:
                log.error(e, exc_info=True)

            return None
```

ДОДАТОК Б. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

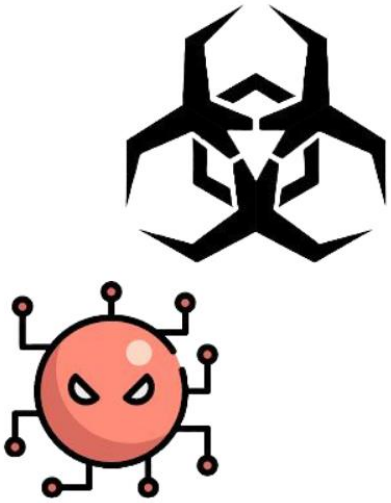
РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ВИЯВЛЕННЯ ТА ІДЕНТИФІКАЦІЇ ЗАГРОЗ В КІБЕРПРОСТОРІ

Виконавець - Максим Женжера

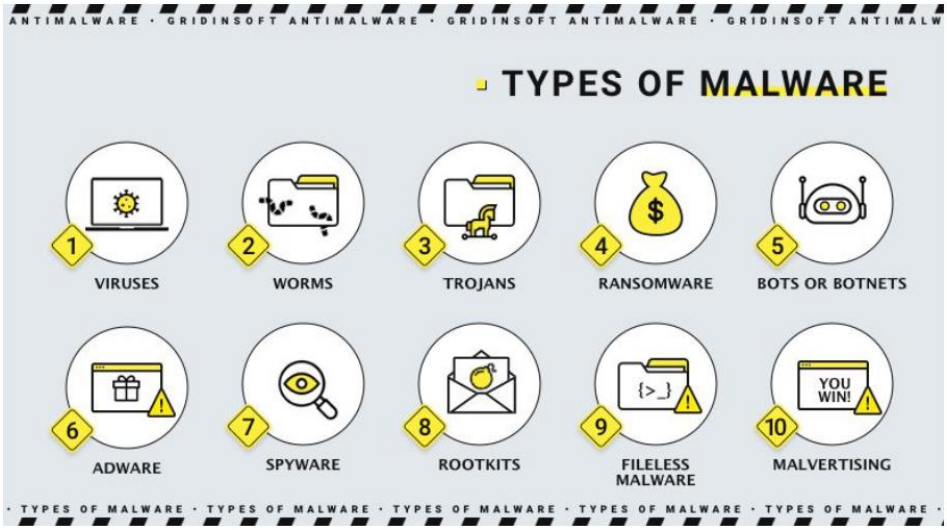
Науковий керівник - к.т.н., доцент Тетяна Демківська

Мета роботи

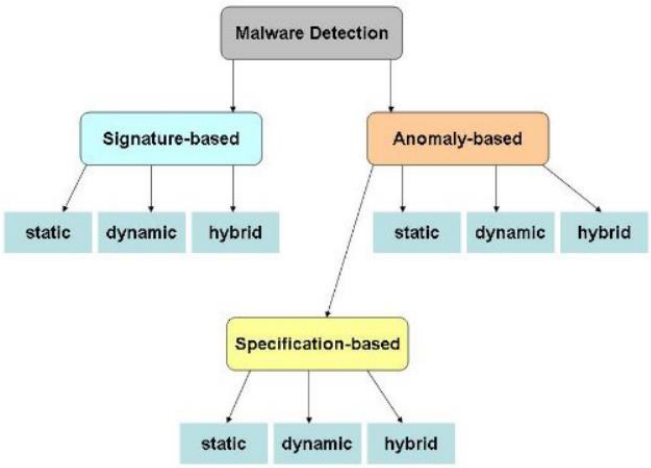
- Розгляд загальної характеристики ШПЗ
- Класифікація та аналіз методів виявлення
- Розроблення системи автоматизованої генерації сигнатур
- Демонстрація згенерованих сигнатур



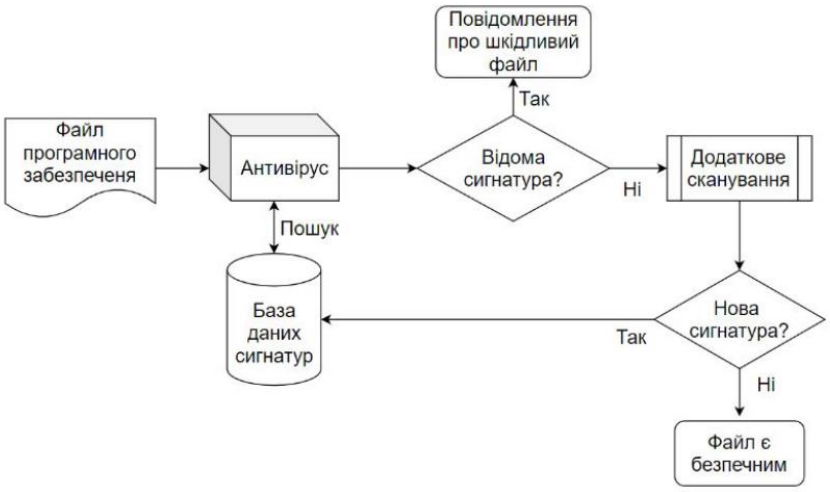
ШПЗ та його типи



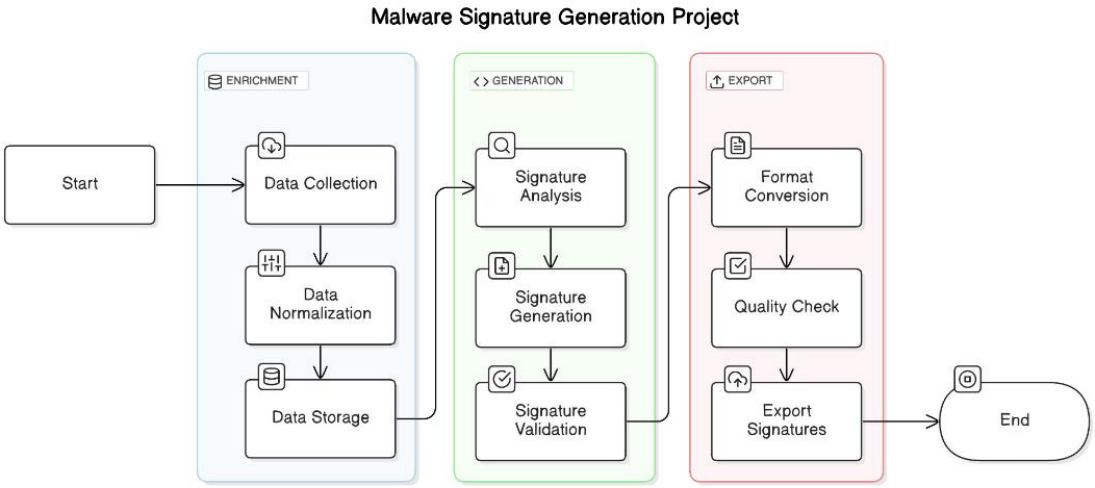
Класифікація методів виявлення



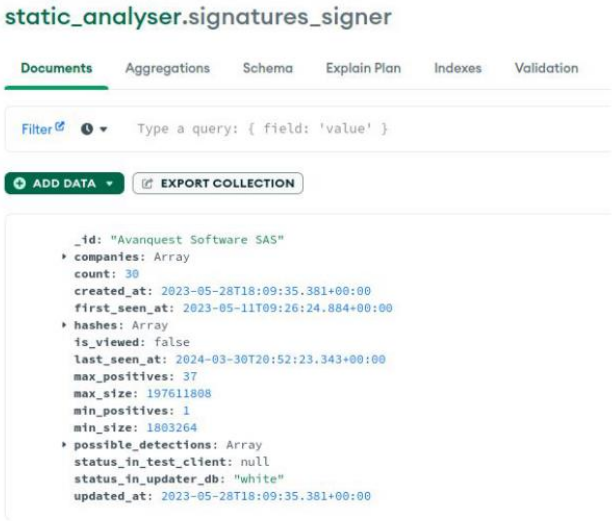
Сигнатурний метод виявлення



Блок-схема алгоритму проекту



Демонстрація згенерованої сигнатури



Висновки

В роботі вирішено завдання розроблення системи автоматизованої генерації сигнатур. Проектування та розроблення системи виконано згідно з наданими вимогами. Загалом, виконана кваліфікаційна робота демонструє, що запропонований проект є ефективним інструментом для виявлення шкідливого програмного забезпечення.

В процесі виконання завдання були використані:

- мова програмування **Python**
- документно-орієнтована NoSQL база даних **MongoDB**