МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
Факультет електроніки та інформаційних технологій

Кафедра комп’ютерних наук

Кваліфікаційна робота магістра

Інформаційна технологія забезпечення захисту виконуваних файлів операційної системи

Здобувач освіти гр. ІК.м – 11 Денис РУБАН

Науковий керівник,
кандидат фізико-математичних наук,
доцент кафедри комп’ютерних наук Надія ТИРКУСОВА

В.о. завідувача кафедри
кандидат технічних наук,
доцент кафедри комп’ютерних наук Ігор ШЕЛЕХОВ

Суми 2022

Сумський державний університет
(назва вузу)

Факультет *ЕлІТ* Кафедра *Комп'ютерних наук*
Спеціальність *«122 - Комп'ютерні науки»*

Затверджую:
В.о. зав.кафедри _____
_____
"_____" _____ 20___р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ МАГІСТРА

Рубану Денису Ігоровичу
(прізвище, ім'я, по батькові)

1. Тема роботи *Інформаційна технологія забезпечення захисту виконуваних файлів операційної системи*

затверджую наказом по інституту від "____" _____ 20 ___ р. № _____

2. Термін здачі здобувачем вищої закінченого роботи_____

3. Вхідні дані до роботи _____
_____
_____
_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
1) Аналіз існуючих алгоритмів; 2) Пошук інструментів для подальшої розробки; 3) Проектування алгоритму роботи інформаційної технології; 4) Розробка інформаційної технології забезпечення захисту виконуваних файлів від модифікації.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) _____
_____

6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання _____

Керівник _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів дипломного проекту (роботи)	Термін виконання проекту (роботи)	Примітка
1.	Огляд алгоритмів захисту		
2.	Постановка задачі та формування завдань дослідження.		
3.	Проектування алгоритму роботи інформаційної технології		
4.	Розробка інформаційної технології		
5.	Оформлення пояснювальної записки до кваліфікаційної магістерської роботи		

Студент – дипломник _____
(підпис)

Керівник проекту _____

(підпис)

РЕФЕРАТ

Записка: 41 стор., 10 рис., 2 табл., 1 додаток, 15 джерел.

Об'єкт дослідження — інформаційна технологія захисту виконуваних файлів.

Мета роботи — дослідження існуючих алгоритмів захисту виконуваних файлів, створення власного алгоритму захисту, а також розробка інформаційної технології з використанням створеного алгоритму та його тестування.

Методи дослідження — метод емпіричного дослідження.

Результати — досліджено існуючі алгоритми захисту виконуваних файлів Windows. Було проаналізовано декілька мов програмування та їх бібліотек для розбору виконуваних файлів Windows. Результатом проведеної роботи є інформаційна технологія для захисту 64-розрядних виконуваних файлів Windows шляхом модифікації машинних інструкцій. Розроблена інформаційна технологія була написана мовою програмування Python 3, а також знайдених модулів для парсингу виконуваних файлів та їх коду.

PYTHON, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, СТРУКТУРА ВИКОНУВАНИХ ФАЙЛІВ WINDOWS, МАШИННИЙ КОД X64.

ЗМІСТ

ВСТУП

Розробники програмного забезпечення витрачають багато часу на розробку, вдосконалення та підтримку їхніх продуктів. Нерідко причиною цього являється те що користувачі витрачають гроші купуючи програмні рішення для їхніх потреб, проте в цьому немає нічого поганого адже розробники повинні отримувати якусь нагороду за їх працю.

І все було б добре якби не було піратів. Вони використовують або розповсюджують програмне забезпечення безкоштовно що погано впливає на продаж його ліцензованих копій. З піратством розробники борються вже давно і одним із самих розповсюджених способів є додавання активації за ліцензійним ключем, коли щоб почати використовувати програмне забезпечення потрібно ввести ключ який спочатку потрібно придбати. Але зазвичай надовго такого захисту не вистачає так як пірати навчились вирізати перевірку ключа ніби її і не було в програмі.

Причини по яким розробник може не хотіти щоб його програми модифікували можуть бути різними. Одною з них є ускладнення розповсюдження програмного продукту без ліцензії, проте без сумніву така проблема існує і в наш час.

Також з розвитком кіберспорту онлайн ігри стали суперницькими як ніколи, а там де є суперництво є і люди які хочуть перемогти любою ціною, навіть якщо перемога дісталась нечесним способом. Запобігання використання різного виду програм для модифікації ігор щоб отримати перевагу над суперниками вже давно стало необхідністю. Ускладнення процесу модифікації коду ігор безсумнівно допоможе в боротьбі з нечесними гравцями та розробниками такого виду програмного забезпечення.

Для вирішення даної проблеми було вирішено провести дослідження на тему розробки програмного забезпечення для захисту виконуваних файлів. Так як найчастіше проблеми з розповсюдженням продуктів з вирізаною ліцензією

трапляються на платформі Windows її було обрано основою виконуваних файлів для захисту.

Метою даної роботи є дослідження методів захисту виконуваних файлів від модифікації, вибір оптимального, а також розробка та тестування програмного забезпечення після застосування обраного методу для дослідження його впливу на швидкість виконання.

1 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1 Заплутування шляху виконання програми

Модифікація готового виконуваного файлу не така проста навіть без захисних механізмів, так як потрібно знати в якому місці знаходиться код який потрібно змінити[1]. В сучасних виконуваних файлах знаходяться сотні тисяч інструкцій, а в деяких навіть мільйони, і знайти в такій купі те що треба може бути меншій мірі проблематично.

Після того як місце яке потрібно модифікувати буде знайдено далі зазвичай залишається тільки розібратися як оптимальніше це зробити, і зазвичай це не займає багато часу. Ідея заплутування[2] шляху виконання програми в тому щоб як можна більше ускладнити[3, 4] пошук потрібної частини коду.

Заплутування відбувається за допомогою таких технік як непрямий стрибок, використання Structured exception handler(SEH), код який модифікує сам себе[5] та інших[6, 7]. Використовуючи даний метод зазвичай застосовують одразу декілька технік для збільшення рівня захисту. Основна їх суть полягає в тому що при перегляді графу виконання програми він буде відображатися не повним або відрізнятиметься від того як він буде виконуватися при запуску програми.

Переваги даного методу полягають в тому що він майже не впливає на швидкість виконання програми, а також доволі велика різноманітність в виборі технік заплутування які можна використовувати одночасно. Хоча цей метод має свої переваги він доволі обмежений в рівні захищеності який він може надати, а також ніяк не захищає данні програми від модифікації.

1.2 Використання охоронців

Концепція охоронців полягає в тому що до існуючого коду програми додається код з охоронцями в довільних місцях програми які будуть перевіряти чи не було змінено код основної програми. Кожен із охоронців буде відповідати за

свою частину програми, і якщо він помітить зміни то відновить код до оригінального або завершить виконання програми. Перевірка коду на модифікацію виконується шляхом хешування[8] коду і порівняння з хешем оригінального коду.

Плюсом охоронців є те що вони можуть захистити він найменшої зміни в коді, і роблять вони це в момент виконання програми, а це означає що навіть при модифікації коду напряму в пам'яті запущеного процесу не чіпаючи сам файл охоронці все одно спрацюють.

Але процес додавання охоронців є доволі проблематичним. Основною проблемою є те що хеші для перевірки коду потрібно прораховувати коли програма вже повністю скомпільована, тому перед компіляцією потрібно залишати місце для охоронців і вже після компіляції обраховувати хеші та поміщати їх на раніше підготовлені місця. Якщо програма вже повністю скомпільована можна також додавати охоронців на місце існуючого коду перемістивши його в інше місце виконуваного файлу але тут потрібно бути обережним і враховувати варіанти коли переміщення коду можуть зламати існуючу програму.

При додаванні охоронців з виділенням місця перед компіляцією треба мати вихідний код програми, а це не завжди є можливим. По цій причині варіант з додаванням охоронців в готову програму є більш універсальним, хоча і більш складним в реалізації.

Підсумовуючи можна сказати що використання охоронців робить можливим захист від модифікації навіть після запуску програми що важко повторити використавши інші методи. Недоліком такого захисту є складність реалізації.

1.3 Власна віртуальна машина

Віртуальні машини використовуються вже давно, одною із найпопулярніших є JVM(Java Virtual Machine). У кожної віртуальної машини є власний список інструкцій, і його можна запускати всюди де працюватиме ця віртуальна машина. Проте кросплатформність не є причиною використання її для захисту від

модифікації. При розробці власної віртуальної машини її особливості будуть відомі тільки тому хто її створив. Це ускладнить життя тому хто захоче зрозуміти як працює програма запущена в віртуальній машині так як вона написана використовуючи незнайомі йому команди.

Використовуючи цей метод весь код програми яку потрібно захистити повинен бути конвертований[9, 10] в інструкції віртуальної машини. Якщо робити віртуальну машину яка буде підтримувати процесорну архітектуру x86 то робити альтернативу кожній інструкції може бути трудомістким процесом так як дана архітектура налічує декілька тисяч інструкцій. Щоб не витрачати час на інструкції які використовуються рідко можна робити конвертацію тільки найбільш розповсюджених інструкцій, а інструкції які залишаться оставити як є.

Якщо створювати віртуальну машину з метою заплутування алгоритму виконання програм то це може стати доволі хорошим захистом від небажаних модифікацій. Мінусом цього методу є те що якщо розібратися як працює якась конкретна віртуальна машина то всі програми які зроблені з її використанням будуть по суті беззахисні, а в деяких випадках навіть більш вразливі якщо інструкції віртуальної машини буде модифікувати легше ніж оригінальної програми без конвертації.

1.4 Виконуваний файл обгортка

Суть файлу обгортки полягає в тому що оригінальний файл буде запускатися при запуску обгортки. Сама по собі обгортка яка дістає із себе оригінальний файл та запускає його захисту не допоможе, проте якщо архівувати або зашифрувати оригінальний файл щоб обгортка перед запуском розшифровувала або розпаковувала його то це не дозволить так просто дістати оригінал із обгортки. Шифрування файлу очевидно буде краще для захисту так як для розпаковки файлу витягнутого з обгортки потрібно лиш знати алгоритм яким було архівовано оригінальний файл.

Також треба враховувати що для того щоб запустити файл його потрібно спочатку записати на диск так як просто так із пам’яті виконувані файли запускати не можна. Windows не дозволить так просто видалити файл доки файл запущено тому він буде залишатися в файловій системі що зробить його знаходження доволі простим тому що можна подивитися шлях до файлу запущеного процесу. Ця проблема вирішується використанням методу запуску під назвою Process Hollowing. Його суть полягає в тому щоб запустити будь який файл в призупиненому стані щоб він був у стані паузи, потім перезаписати його в пам’яті новим файлом який потрібно запустити без запису на диск, і після цього продовжити виконання програми.

Проте навіть використавши Process Hollowing можна буде дістати файл із пам’яті запущеного процесу, але потрібно буде виправити файл щоб його можна було запустити. Підсумовуючи можна сказати що цей метод сам по собі не зможе захистити оригінальний файл від досвідченої людини.

1.5 Постановка задачі

Після проведення аналізу існуючих методів захисту від модифікації було вирішено що для майбутньої інформаційної технології буде використано алгоритм з додаванням охоронців без необхідності в попередній компіляції. Інформаційна технологія буде використовуватися на виконуваному файлі Windows і в результаті користувач отримуватиме той самий по функціональності виконуваний файл але з доданими охоронцями. Кількість охоронців та обсяг коду що вони будуть перевіряти можна буде змінювати за бажанням.

2 ВИБІР ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1 Вибір мови програмування

Для розробки програмного забезпечення яке буде модифікувати виконуваний файл додаючи до нього охоронців потрібно обрати мову програмування яку зручно використовувати для розбору та аналізу виконуваного файлу. Вибір буде проводитися з таких мов програмування: C++, Python, Java, C#.

C++ є досить хорошим варіантом через його швидкість та можливість працювати з пам’яттю напряму. Також з пошуком необхідних бібліотек проблем не повинно бути через його популярність в сфері системного програмування.

Python має багато переваг. Перше це його простота в розробці, завдяки цьому розробка програмного забезпечення займає менше часу, що дозволить швидше перейти до тестування прототипу. Друге це дуже великий вибір модулів(бібліотек) для майже будь яких потреб. Приємним бонусом є те що його можна запустити без проблем на інших платформах де можна встановити Python.

Java та C# дуже схожі за концепцією мови. У них не буде проблем з кросплатформністю, зручність розробки вища за C++ але менша за Python. З вибором бібліотек для розбірки та аналізу виконуваних файлів можуть виникнути проблеми тому що ці мови зазвичай використовуються для інших областей розробки.

В результаті проаналізувавши перечислені вище мови програмування було обрано мову програмування Python через її швидкість розробки[11] та різноманітність модулів(бібліотек).

2.2 Пошук необхідних бібліотек

В ході розробки даного програмного забезпечення часто буде потрібно конвертувати послідовності байтів в числа і навпаки. Для цієї задачі було обрано стандартний модуль Python під назвою struct.

Щоб вирішити проблему аналізу виконуваного файлу Windows було обрано модуль під назвою pefile. Він дозволяє отримувати інформацію про структуру виконуваних файлів Windows формату PE32 та PE32-64. З його допомогою можна легко подивитися заголовки, секції, таблицю релокацій та багато іншого що може знадобитися в подальшій розробці.

І останнім модулем але не за значенням було обрано distorm3. Він може розбирати інструкції x86 та x86-64 процесорної архітектури щоб діставати з них операнди, а також визначати тип інструкцій та інші їх параметри. Даний модуль допоможе визначати куди можна буде розміщувати охоронців щоб не змінити алгоритм роботи програми.

2.3 Пошук інструментів для аналізу виконуваних файлів

Вибір зручного інструментарію досить важливо адже це прискорить швидкість розробки. Додавання охоронців в виконувані файлів Windows може пошкодити їх і щоб дивитися вміст цих файлів, а також знаходити помилки було обрано програма з відкритим вихідним кодом під назвою PE-bear. Вона допоможе переглядати та редагувати вміст виконуваних файлів у зручному інтерфейсі, а також вказувати на помилки якщо файл після модифікації не завантажуватиметься через помилки в його структурі. З її допомогою можна переглядати такі частини виконуваних файлів як таблиця імпорту та експорту, таблиця релокацій, дерево ресурсів, список секцій, список виключень та різні типи заголовків. Інтерфейс програми можна побачити на рисунку 2.1 разом з відкритою вкладкою списку секцій.

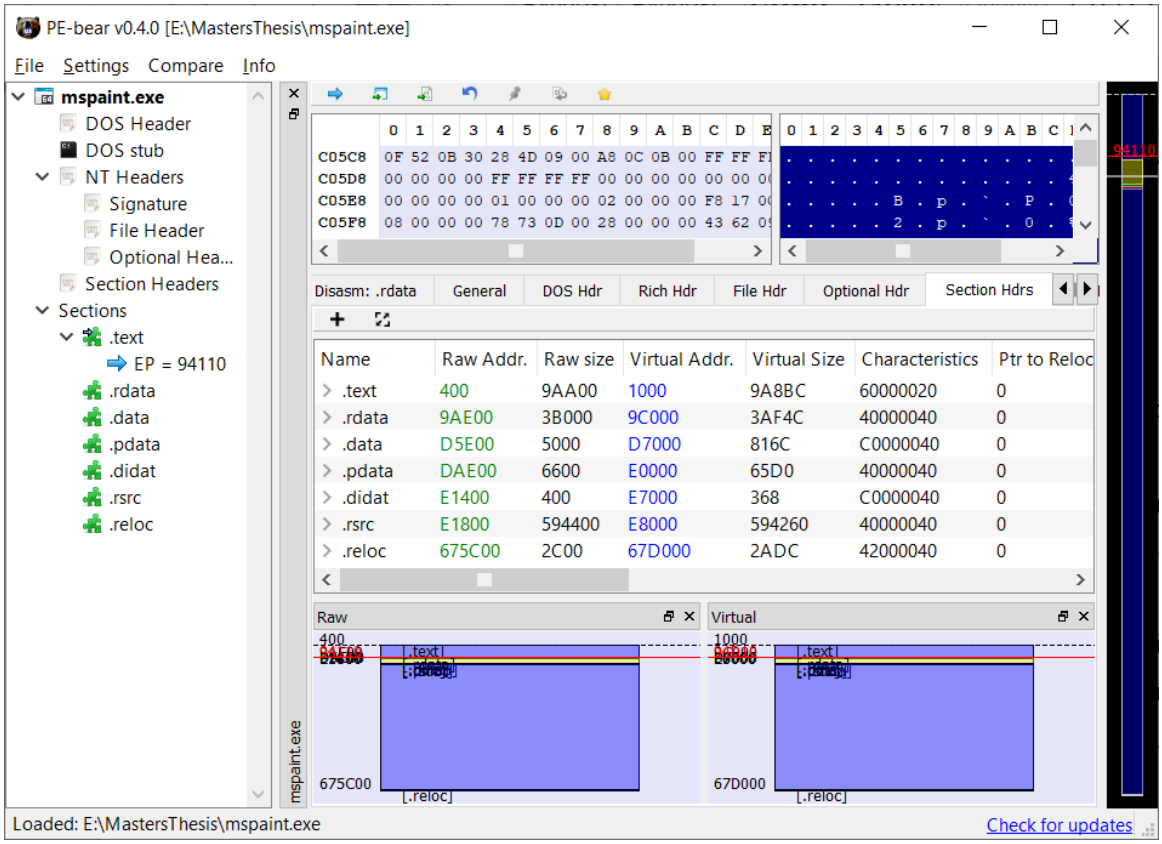

Рисунок 2.1 – Список секцій в програмі PE-bear

Щоб спостерігати за виконанням файлу коли він вже запущений, знаходити помилки та тестувати виконувані файли потрібно дебагер. У ролі дебагера було обрано програму під назвою x64dbg. Вона також програмою з відкритим вихідним кодом та має широкий список інструментів для нагляду за процесом виконання коду, знаходження помилок, можливість модифікації коду в програмах в процесі їх виконання та багато інших. Інтерфейс програми разом з графом виконання можна побачити на рисунку 2.2.

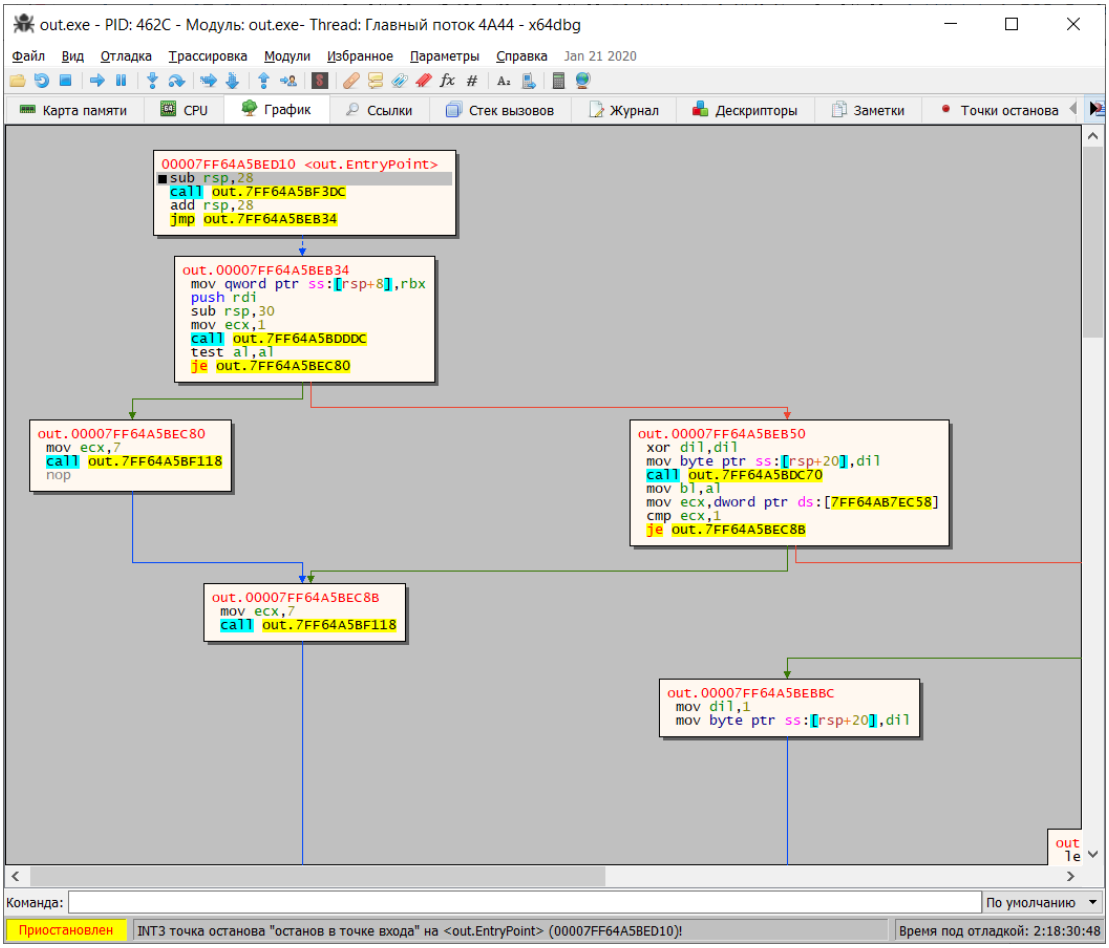

Рисунок 2.2 – Граф виконання в програмі x64dbg

3 ПРОЕКТУВАННЯ АЛГОРИТМУ РОБОТИ

3.1 Вибір алгоритму хешування

Щоб охоронці змогли перевіряли чи була модифікована програма вони повинні порівнювати результат отриманий після хешування тієї частини програми з значенням порахованим в момент створення охоронців. Алгоритм хешування повинен бути достатньо складним щоб знаходження колізій було дуже складним, але він повинен бути і в міру швидким щоб це не сповільнювало виконання програми. Для визначення відносної швидкості між алгоритмами було проведено невеликий тест, для тесту було обрано 5 алгоритмів хешування та виміряно за який час вони закінчать обробку 8 мегабайт даних. Тест проводився 1000 разів і потім було знайдено середнє арифметичне часу кожного алгоритму, результати тестів розміщено в таблиці 1.1.

Таблиця 3.1 – Швидкість алгоритмів хешування

Алгоритм	**Час виконання в мікросекундах**
MD5	18980
SHA-1	13945
SHA-256	29874
SHA-384	20946
SHA-512	19983

Щоб всі перечисленні вище критерії задовольнялися було обрано алгоритм хешування під назвою SHA-1. Він є безпечнішим за MD5, і найшвидший із представлених алгоритмів.

3.2 Типи охоронців

Якщо просто розмістити охоронців в програмі щоб вони порівнювали результат хешування з значенням обчисленим на основі оригіналу і при неправильному хеші зупиняли виконання програми то достатньо буде просто

знайти всіх охоронців та видалити їх і програма знову буде вразливою до модифікації.

Щоб ускладнити процес видалення охоронців було вирішено додати другий тип охоронців які будуть заміняти існуючі в програмі інструкції на такі ж самі але у якості операндів у них будуть значення хешів. Таким чином якщо просто видалити такого охоронця то інструкція яку було замінено буде з неправильним операндом. Якщо обчислити хеші та замінити операнди інструкцій щоб повернути їх до того як вони були перед встановленням охоронців то видалення охоронців буде знову можливим, проте це набагато складніше зробити ніж видалення першого типу охоронців. Окрім того другий тип охоронців буде захищати перший тип, і в результаті перший тип охоронців також не вийде так просто видалити з програми без наслідків.

Також слід сказати що так як другий тип охоронців повинен знаходити результат хешу щоб дізнатися оригінальне значення операнду то він не буде порівнювати отриманий хеш так як щоб порівняти потрібно знати його значення, а це зробить відновлення модифікованих другим типом охоронців інструкцій легшим тому що результуючий хеш який повинен бути отриманий вже буде в коді. Через це у випадку якщо програма була модифікована після встановлення охоронців і при виконанні першим в програмі зустрінеться охоронець другого типу то він порахує неправильний операнд через що програма скоріш за все завершиться якоюсь випадковою помилкою так як отриманий операнд буде мати непередбачуване значення.

3.3 Розміщення охоронців

В коді програми яку потрібно буде захистити від модифікації вільного міста для охоронців в більшості випадків не буде тому потрібно буде створювати нове місце в файлі. Виконувані файли Windows формату PE32 та PE32-64 поділені на секції, для розміщення коду охоронців буде створюватися нова секція, а в

основному коді будуть додані інструкції стрибків які будуть перенаправляти виконання програми на код потрібного охоронця. В кінці коду кожного охоронця буде стрибок назад в основний код програми, що можна побачити на рисунку 3.1.

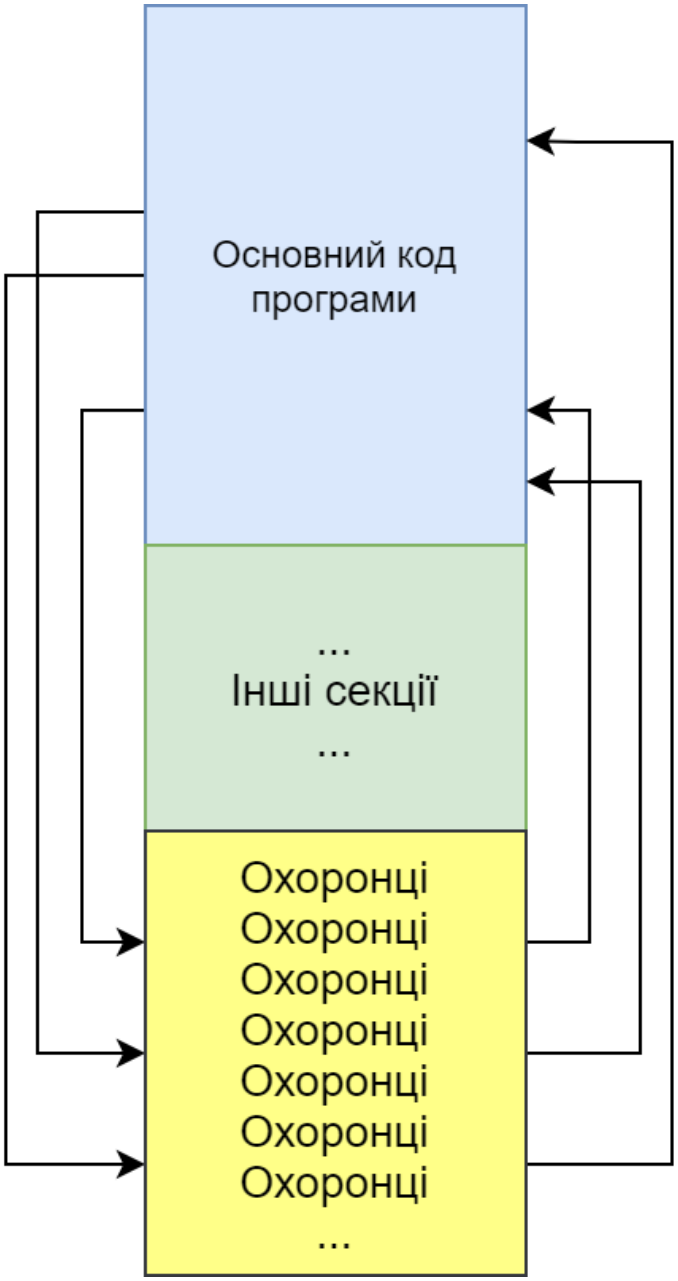

Рисунок 3.1 – Алгоритм роботи стрибків до охоронців

В основному коді програми будуть вирізані інструкції щоб розмістити стрибки до охоронців, проте вирізані інструкції будуть переміщені в охоронців щоб не ламати хід виконання програми.

3.4 Алгоритм роботи

Взявши до уваги типи та розміщення охоронців алгоритм їх додавання можна поділити на 5 основних етапів:

1) Створення міста в виконуваному файлі для розміщення охоронців, для цього буде створена нова секція у виконуваному файлі
2) В першу чергу в новій секції буде розміщено функцію хешування SHA1 так як для обраховування хешів охоронці будуть її викликати замість того щоб не розміщувати код функції в кожному з охоронців та тим самим з економити місце.
3) Далі будуть розміщені охоронці другого типу.
4) Після цього починається пошук інструкцій на місто яких можна розмістити стрибки до охоронців першого типу, а як інструкції будуть знайдені охоронці першого типу будуть розміщені в новій секції
5) І фінальний етап це обрахунок хешів для всих охоронців. Це важливо робити в самому кінці коли код програми вже не буде модифіковано щоб обраховані хеші не стали застарілими.

Перечисленні етапи зображено на рисунку 3.2 у вигляді блок-схеми.

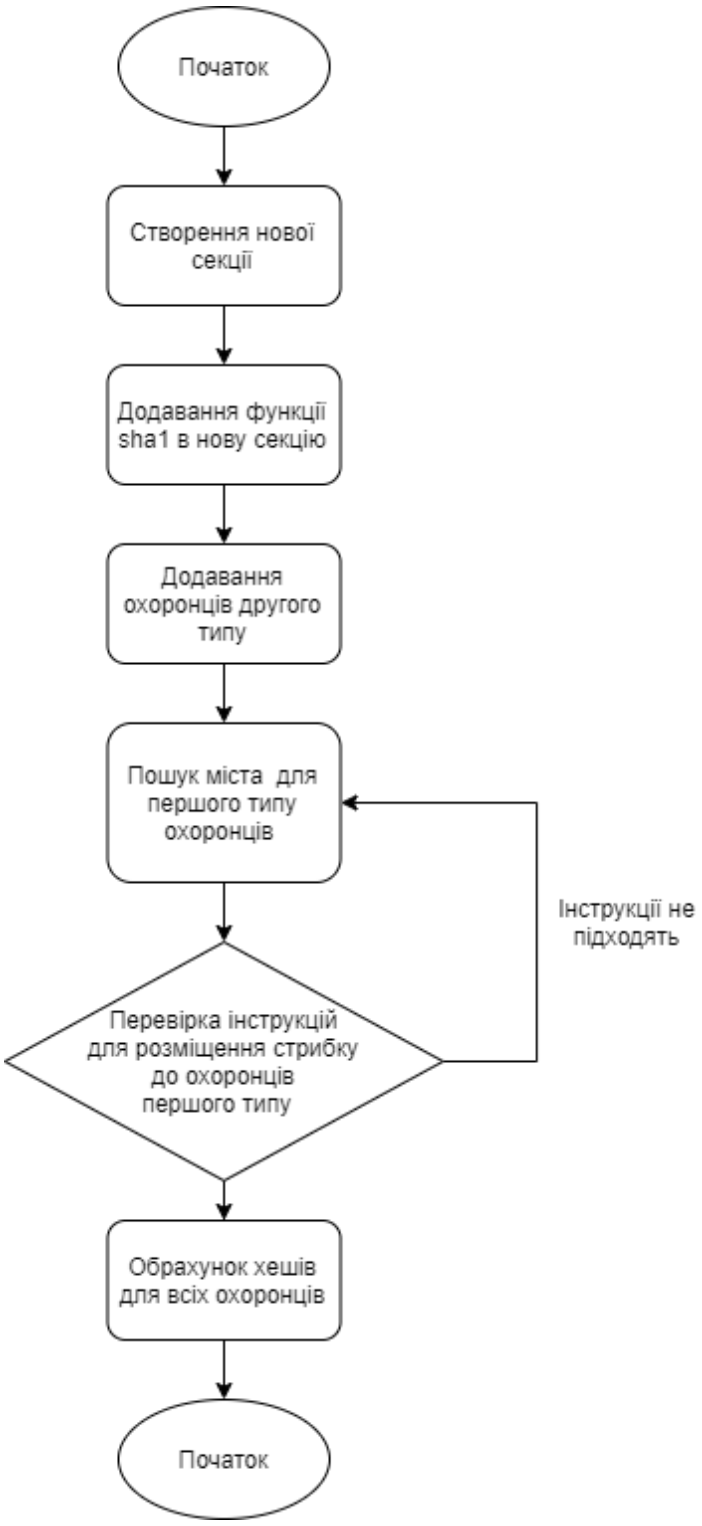

Рисунок 3.2 – Повний алгоритм додавання охоронців

4 ПРОГРАМНА РЕАЛІЗАЦІЯ

4.1 Реалізація створення нової секції

Щоб створити нову секцію спочатку потрібно записати в заголовок списку секцій[12, 14] інформацію про нову секцію та її характеристики по типу її розміру, адреса даних, а також прав доступу до секції коли вона буде завантажена в пам’ять при запуску програми.

Список полів заголовку секції та відступ кожного поля в байтах можна побачити на рисунку 4.1, але для створення нової секції важливі будуть тільки 5 із них які позначені золотистим кольором. Віртуальний розмір[15] та адреса відповідають за розташування даних в адресному просторі програми коли вона буде запущена. Розмір та адреса в файлі відповідають за розташування даних секції безпосередньо в виконуваному файлі. Права доступу відповідають за можливі операції з даними секції такі як читання, запис та виконання.

Відступ	Назва поля
0+8	Назва секції
8+4	Віртуальний розмір
12+4	Віртуальна адреса
16+4	Розмір секції в файлі
20+4	Адреса секції в файлі
24+4	Покажчик до релокацій
28+4	Покажчик до номерів рядків
32+2	Кількість релокацій
34+2	Кількість нмерів рядків
36+4	Права доступу

Рисунок 4.1 – Список полів заголовку секції

Назву секції можна обрати довільну тому що це ніяк не вплине на виконання програми. Покажчики до релокацій та номерів рядків та їх кількість потрібно заповнювати нулями тому що ці поля не використовуються в виконуваних файлах формату PE32 та PE32-64.

Спочатку для того щоб вписати заголовок нової секції потрібно знайти кількість секцій, потім знайти де знаходиться PE заголовок тому що заголовки секцій йдуть зразу після PE заголовку. Коли початок заголовків з секціями знайдено потрібно помножити кількість секцій на розмір заголовку секції і отримати адресу де буде знаходитися новий заголовок.

Перед створенням секції потрібно враховувати що віртуальна адреса та адреса в файлі повинні буде вирівняні до певного значення. Для віртуальної адреси стандартне значення становить 4096, а для адреси в файлі 512. На рисунку 4.2 зображено код отримання адреси де буде розміщена нова секція, а також вирівнювання адрес секції до їхніх стандартних значень.

```
last_section_va_end = PEObj.sections[-1].section_max_addr
last_sectoin_ra = PEObj.sections[-1].get_file_offset()
start_of_new_section_header = last_sectoin_ra + SIZE_OF_SECTION_HEADER
new_section_va = last_section_va_end
if new_section_va % 0x1000 != 0:
    new_section_va = new_section_va + (0x1000 - new_section_va % 0x1000)
new_section_ra = len(file_data)
if new_section_ra % 0x200 != 0:
    new_section_ra = new_section_ra + (0x200 - new_section_ra % 0x200)
    file_data += b"A" * (0x200 - new_section_ra % 0x200)
```

Рисунок 4.2 – Знаходження адреси для заголовку нової секції

Після додавання заголовку нової секції можна додавати код охоронців по адресі зазначеній в заголовку секції. Але після цього виконуваний файл все ще не зможе запуститися тому що треба змінити поле розміру образу яке находиться в Optional заголовку. Воно відповідає за розмір всіх секцій та заголовків які будуть

завантажені в пам’ять програми при запуску. Щоб програма знову стала працездатною потрібно додати до поля розміру образу розмір створеної секції.

Після додавання нової секції можна побачити всі її характеристики на рисунку 4.3 в програмі PE-bear. Нова секція має назву “.guards”, її розмір складає 0xF000 байт, вона розміщена в кінці фалу та має права доступу на читання, запис та виконання.

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics
.text	400	4DE00	1000	4DC0A	60000020
.rdata	4E200	1C400	4F000	1C2EE	40000040
.data	6A600	800	6C000	2CB8	C0000040
.pdata	6AE00	6200	6F000	612C	40000040
.rsrc	71000	800	76000	6E0	40000040
.reloc	71800	1000	77000	E06	42000040
.guards	72800	F000	78000	F000	E0000000
>	81800	^	87000	^	rwx

Рисунок 4.3 – Нова секція

4.2 Підготовка функції SHA1

Для того щоб обраховувати хеш потрібна відповідна функція, код який буде додаватися в виконуваний файл повинен складатися з машинних інструкцій. Так як знайти дану функцію на асемблері не вдалося, було взято код на C++ та скомпільовано без лінкування щоб отримати тільки інструкції потрібної функції без зайвої інформації. Код функції також був модифікований щоб повертати перші 64 біти результуючого хешу, так як для охоронців не потрібні всі 160 біт хешу тому що вони будуть тільки сповільнювати роботу програми. Команда з допомогою якої була скомпільована функція:

```
gcc.exe sha1.c -c -Os
```

Після компіляції протестувавши код функції в x64dbg було визначено що не всі регістри які будуть змінені в ході виконання функції зберігаються на стеку щоб

відновити їх після обрахунку хешу. Змінені регістри після виконання функції підсвічуються червоним кольором що можна побачити на рисунку 4.4.

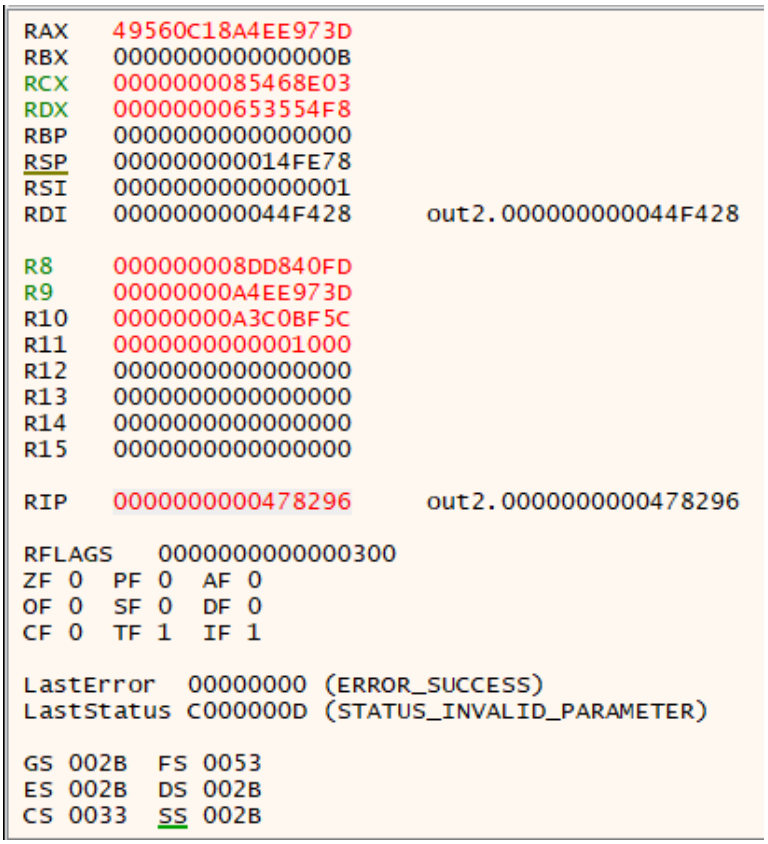

Рисунок 4.4 – Змінені регістри після виконання функції SHA1

Хоча регістри RAX, RCX та RDX також було змінено вони будуть зберігатися самими охоронцями тому що ці регістри використовуються в конвенції виклику fastcall. В результаті потрібно було зберегти на стеку регістри R8, R9, R10, R11. Щоб вирішити цю проблему на початку функції та в кінці були додані інструкції для вставки та діставання цих регістрів зі стеку.

4.3 Реалізація першого типу охоронців

Перший тип охоронців буде перевіряти всі існуючі секції файлу, і якщо результуючий хеш буде відрізнятися він завершить виконання програми. Так як основний код охоронця буде знаходитися в новій створеній секції, потрібно знайти інструкції які можна перезаписати інструкцією стрибка яка займає 5 байт. Для цього

треба врахувати що перезаписані[13] інструкції при переносі їх в тіло коду охоронця повинні коректно виконатися щоб зберегти цілісність коду програми.

Завадити виконанню переміщених інструкцій може тип їх операндів, якщо операнд охоронця буде відносним до поточного адресу виконання то переміщені інструкції не зможуть виконатися як було в оригіналі програми. Щоб цього уникнути можна при переміщенні інструкцій з відносними операндами обчислювати їх операнди знову щоб враховувати зміну місця розташування. Проте є другий варіант який є набагато простіший і він полягає в пропуску таких інструкцій поки не буде знайдено ті які можна без проблем перемістити.

В сучасних програмах через часті проблеми з безпекою виконуваних файлів по стандарту в них додається таблиця релокацій. Вона потрібна для використання такої технології як ASLR(Address space layout randomization). Ця технологія дозволяє розміщувати виконувану програму у випадковому місці в пам’яті кожного разу. Це робить експлуатацію вразливостей таких як переповнення буферу набагато важчим. Але ця технологія буде заважати охоронцям тому що щоб програма працювала як і раніше після переміщення її по випадковій адресі використовується таблиця релокацій щоб замінити в коді частини в яких є адреси. Так як адреси змінюються після запуску на випадкові то хеші пораховані охоронцями стануть не дійсними. Щоб вирішити цю проблему потрібно щоб охоронці не перевіряли секції де є релокації, так як інших варіантів немає це і стало рішенням даної проблеми.

Основний код охоронця можна побачити на рисунку 4.5. Слід зауважити що перевіряти секцію в якій будуть знаходитися охоронці потрібно не в повному обсязі, а до місця в якому буде починатися код самого охоронця. При спробі порахувати хеш самого себе потрібно буде замінити хеш для перевірки в коді охоронця, це призведе до того що хеш коду який перевіряється зміниться і так вийде замкнуте коло тому що хеш коду охоронця буде залежати від самого себе.

Інструкція	Пояснення
push rax	Збереження регістрів на стекові тому що вони будуть модифіковані
push rcx	
push rdx	
lea rcx , адреса_1_секції	Завантаження адреси та розміру першої секції як аргументів для виклику функції sha1
mov rdx , довжина_1_секції	
call sha1	Виклик функції sha1
cmp rax , хеш_1_секції	Порівняння з правильним хешем
jne 0x0	Якщо хеш не вірний аварійно закрити програму
lea rcx , адреса_2_секції	Завантаження адреси та розміру першої секції як аргументів для виклику функції sha1
mov rdx , довжина_2_секції	
call sha1	Виклик функції sha1
cmp rax , хеш_2_секції	Порівняння з правильним хешем
jne 0x0	Якщо хеш не вірний аварійно закрити програму
...	Перевірка інших секцій
push rdx	Відновлення регістрів зі стеку
push rcx	
push rax	
Перезаписані інструкції	Інструкції які були перезаписані щоб помістити стрибок в код охоронця
...	
jmp адреса_повернення	Повернення до основного коду програми

Рисунок 4.5 – Код охоронця першого типу

4.4 Реалізація другого типу охоронців

Основною ціллю другого типу охоронців є їх розміщення в багатьох містах в коді програми для того щоб їх було не так легко видалити. Вони на відміну від першого типу будуть швидкими але будуть перевіряти тільки невеликі частини коду по 4 кілобайти. За рахунок швидкості їх кількість може бути набагато більшою ніж першого типу та мінімально впливати на швидкодію програми.

Стрибки до їх коду будуть встановлюватися на місці існуючих стрибків, таким чином достатньо буде знайти тільки інструкції стрибків і змінити адресу їх призначення на адресу коду охоронця. Код цього типу охоронців можна побачити на рисунку 4.6.

Інструкція	Пояснення
push rax	Збереження регістрів на стекові тому що вони будуть модифіковані
push rcx	
push rdx	
lea rcx , адреса_для_хешування	Завантаження адреси для хешування та розміру як аргументів для виклику функції sha1
mov rdx , довжина_даних	
call sha1	Виклик функції sha1
add eax , раніше_обраховане_значення	Сума хешу з раніше обрахованим значенням
mov [адреса_стрибка_нижче] , eax	Запис результату в адресу стрибка
push rdx	Відновлення регістрів зі стеку
push rcx	
push rax	
jmp адреса_для_перезапису	Повернення до основного коду програми

Рисунок 4.6 – Код охоронця другого типу

Так як кожен охоронець другого типу перевіряє невелику частину коду це зменшує шанси того що зміни в коді будуть цим типом охоронця. Для того щоб збільшити ці шанси декілька охоронців будуть перевіряти одну і ту ж саму частину коду, таким чином кожна чистина коду матиме більше шансів на перевірку другим типом охоронців.

4.5 Проведення тестів виконуваних файлів після додавання охоронців

Для тестування зміни швидкодії програм після додавання охоронців до виконуваних файлів було обрано 3 програми для перевірки: 7zip, Audacity, ffmpeg. Програма 7zip буде тестуватися обчислюючи хеш великого файлу, Audacity буде вимірюватися час запуску програми, а ffmpeg буде перевірятися конвертуючи wav файл в mp3. Після проведення тестів було отримано таблицю 4.1.

Таблиця 4.1 – Швидкість алгоритмів хешування

Протестована програма	**Оригінальна швидкість**	**Швидкість після додавання охоронців**
7zip	24.72 сек.	35.61 сек.
Audacity	3.71 сек.	4.08 сек.
ffmpeg	5.65 сек.	14.04 сек.

Після закінчення тестів можна явно побачити що додавання охоронців в програми які виконують багато обчислювальних операцій дуже погано впливає на їх швидкість роботи. Проте на швидкість роботи звичайних програм це не такого сильного впливу.

ВИСНОВОК

Виконавши аналіз існуючих методів захисту виконуваних файлів від модифікації було обрано концепт охоронців в різних місцях програми з хешуванням для ускладнення зворотного процесу, для більшого степеню захищеності було вирішено зробити два типи охоронців. Далі було розглянуто декілька мов програмування таких як C++, Python, C#, Java для розробки інформаційної технології, в результаті мова Python була обрана як сама оптимальна для даної задачі.

Обравши мову програмування потрібно було знайти для неї необхідні бібліотеки для розробки, а також інструментів для аналізу виконуваних файлів. В якості інструментів було обрано PE-bear та x64dbg, а у якості бібліотек distorm3 та pefile. Після цього було обрано проведено тестування алгоритмів хешування щоб обрати оптимальний, SHA1 по результатах тестів виявився самим оптимальним для перевірки цілісності. На основі обраного алгоритму хешування було спроектовано алгоритм роботи інформаційної технології.

По завершенню етапу проектування було розпочата розробка інформаційної технології. Спочатку було імплементовано створення нової секції в виконуваному файлі для додавання туди коду охоронців. Потім був імплементований алгоритм створення охоронців обох видів, і нарешті перевірено на реальних виконуваних файлах та вплив охоронців на швидкодію програм.

По результатам тестів можна точно сказати що вплив охоронців на швидкість програм які роблять високоінтенсивні обчислення сильно погіршує їх темп виконання, і може призводити навіть до зменшення швидкості в декілька разів. З іншого боку зміна швидкості звичайних програм буде майже не помітна для звичайних користувачів. Але варти сказати що вплив на швидкість може сильно варіюватись в різних програм тому що все залежить від розміщення інструкцій на місце яких будуть встановлені охоронці.

Проведене дослідження буде актуально і в майбутньому тому що виконувані файли будуть продовжувати використовуватися в операційні системі Windows, і поки повноцінної їм альтернативи немає. Необхідність розробників захистити своє програмне забезпечення від модифікації також нікуди не зникне поки будуть існувати пірати.

СПИСОК ЛІТЕРАТУРИ

1. Schrittwieser S. Protecting software through obfuscation: can it keep pace with progress in code analysis? / S. Schrittwieser, S. Katzenbeisser, J. Kinder[et al.] // ACM Computing Surveys. — 2016. — Vol. 49, No. 1. — P. 40.
2. Balakrishnan G. WYSINWYX: what you see is not what you execute / G. Balakrishnan, T. Reps, D. Melski, T. Teitelbaum. — 2008. — 11 p.
3. Linn C. Obfuscation of executable code to improve resistance to static disassembly / C. Linn, S. Debray. — 2003. — 15 p.
4. Banescu S. A tutorial on software obfuscation / S. Banescu, A. Pretschner // Advances in Computers. — 2018. — P. 52.
5. Bonfante G. A computability perspective on self-modifying programs / G. Bonfante, J. Y. Marion, D. Reynaud-Plantey. — 2009. — 10 p.
6. Hai N. M. Obfuscation code localization based on cfg generation of malware / N. M. Hai, M. Ogawa, Q. T. Tho. — 2016.
7. Yao X. A method and implementation of control flow obfuscation using seh / X. Yao, J. Pang, Y. Zhang[et al.] // Proceedings - 2012 4th International Conference on Multimedia and Security, MINES 2012. — 2012. — P. 336–339.
8. Chang H. Protecting software code by guards / H. Chang, M. J. Atallah. — 2002. — 15 p.
9. Haijiang X. Nightingale: translating embedded vm code in x86 binary executables / X. Haijiang, Z. Yuanyuan, L. Juanru, G. Dawu. — 2017. — 18 p.
10. Fang H. Multi-stage binary code obfuscation using improved virtual machine / H. Fang, Y. Wu, S. Wang, Y. Huang. — 2011. — 14 p.
11. Sweigart A. Automate the boring stuff with python : practical programming for total

beginners / A. Sweigart // P. 547.

12. Webster G. D. Finding the needle: a study of the pe32 rich header and respective malware triage / G. D. Webster, B. Kolosnjaji, C. Von Pentz[et al.] // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). — 2017. — Vol. 10327 LNCS. — P. 119–138.

13. Halladay K. X64 function hooking by example / K. Halladay. — 2020.

14. Qasha R. Controlling and protecting windows applications by analyzing and manipulating pe file format / R. Qasha, Z. Monther // AL-Rafidain Journal of Computer Sciences and Mathematics. — 2012. — Vol. 9, No. 1. — P. 23–33.

15. Mohanta A. Virtual memory and the portable executable (pe) file / A. Mohanta, A. Saldanha // Malware Analysis and Detection Engineering. — 2020. — P. 71–122.

ДОДАТОК

Лістинг коду

```
import distorm3 as ds
import pefile
import struct, hashlib, ctypes, time, random

target_file_name = "7z.exe"
SECTION_HEADER_STRUCT = struct.Struct("QIIIIIIHHI")

SIZE_OF_SECTION_HEADER = 0x28
PEObj = pefile.PE(target_file_name)

file = open(target_file_name, "rb")
file_data = file.read()
file.close()

text_section = None
for section in PEObj.sections:
    if section.Name == b".text\x00\x00\x00":
        text_section = section

if text_section == None:
    print("Text section not found")
    input()
    exit()


file_data = bytearray(file_data)
code = bytes(
    file_data[
        text_section.PointerToRawData : text_section.PointerToRawData
        + text_section.SizeOfRawData
    ]
)

instructions_generator = ds.DecomposeGenerator(
    text_section.section_min_addr, code, ds.Decode64Bits
)
jmp_instructions_array = []

tm = time.time()
for i in instructions_generator:
    if (
        len(i.operands) == 1
        and i.operands[0].size == 32
        and i.operands[0].type == "Immediate"
        and i.mnemonic == "JMP"
    ):
        jmp_instructions_array.append(i)

print("TEXT SECTION PARSING END", time.time() - tm)
print("FOUND JMPS", len(jmp_instructions_array))

JMP_RATIO = 1
```

```python
def insert_new_section(file_data, PEObj, new_section_vs, new_section_rs):
    last_section_va_end = PEObj.sections[-1].section_max_addr
    last_sectoin_ra = PEObj.sections[-1].get_file_offset()
    start_of_new_section_header = last_sectoin_ra + SIZE_OF_SECTION_HEADER
    new_section_va = last_section_va_end
    if new_section_va % 0x1000 != 0:
        new_section_va = new_section_va + (0x1000 - new_section_va % 0x1000)
    new_section_ra = len(file_data)
    if new_section_ra % 0x200 != 0:
        new_section_ra = new_section_ra + (0x200 - new_section_ra % 0x200)
        file_data += b"A" * (0x200 - new_section_ra % 0x200)

    section_header_bytes = SECTION_HEADER_STRUCT.pack(
        int.from_bytes(b".guards\x00", "little"),
        new_section_vs,
        new_section_va,
        new_section_rs,
        new_section_ra,
        0,
        0,
        0,
        0,
        0xE0000000,
    )
    file_data[
        start_of_new_section_header : start_of_new_section_header
        + SIZE_OF_SECTION_HEADER
    ] = section_header_bytes

    sections_count_address = PEObj.FILE_HEADER.get_file_offset() + 2
    file_data[sections_count_address : sections_count_address + 2] = int(
        PEObj.FILE_HEADER.NumberOfSections + 1
    ).to_bytes(2, "little")

    size_of_image_address = PEObj.OPTIONAL_HEADER.get_file_offset() + 0x38
    file_data[size_of_image_address : size_of_image_address + 4] = int(
        PEObj.OPTIONAL_HEADER.SizeOfImage + new_section_vs
    ).to_bytes(4, "little")
    file_data += b"\x90" * new_section_rs
    return new_section_va, new_section_ra

new_section_rs = 45 * (len(jmp_instructions_array) // JMP_RATIO)
new_section_vs = new_section_rs
new_section_va, new_section_ra = insert_new_section(
    file_data, PEObj, new_section_vs, new_section_rs
)
```

```
image_base = PEObj.OPTIONAL_HEADER.ImageBase
text_section_va = text_section.section_min_addr + image_base
text_section_end_va = text_section.section_max_addr + image_base

instruction_ptr = 0

sha1_func_code = b"\x41\x50\x41\x51\x41\x52\x41\x53\x41\x57\x41\x56\x41\x5
    7\x48\x8D\x42\x08\x49\x89\xD3\x48\x89\x8C\x24\x30\x02\x00\x00\xB9\x1E\
    x1C\xD5\x00\x00\x00\x00\x89\x44\x24\x04\x89\xC2\x31\xC0\x48\x89\x44\x2
    2\xF9\x48\x89\xD9\x4C\x8D\xA4\x24\xA0\x00\x00\x00\x48\xC1\xE9\x30\x88\
    x42\xFC\x48\x89\xD9\x48\xC1\xE9\x18\x88\x4C\x04\x20\x8D\x42\xFD\x48\x8
    0\x00\x48\x89\x44\x24\x10\x48\x8D\x84\x24\x60\x01\x00\x00\x89\x14\x24\
    xE2\x4D\x89\xE0\xB8\x18\x00\x00\x00\x85\xC0\x89\xE9\x78\x21\x4C\x39\xD
    1\x41\x89\xEA\x85\xC9\x78\x18\x44\x89\xD3\x41\xFF\xC2\x4C\x29\xDB\x0F\
    xA0\x4C\x89\xE1\x8B\x41\x34\x48\x83\xC1\x04\x33\x41\x1C\x33\x41\x04\x3
    0\x02\x89\x42\x7C\x48\x39\x54\x24\x18\x75\xE3\x44\x8B\x44\x24\x0C\x45\
    x31\xC2\x21\xC2\x44\x31\xC2\xEB\x43\x8D\x4B\xEC\x83\xF9\x13\x77\x10\x4
    1\x41\xBE\xDC\xBC\x1B\x8F\x21\xC2\x44\x21\xC1\x09\xCA\xEB\x0B\x31\xC2\
    xF1\x44\x01\xF9\x48\x83\xFB\x50\x45\x89\xC7\x74\x0D\x45\x89\xD0\x89\xC
    F\xFF\x48\xC1\xE6\x20\x44\x89\xC8\x48\x09\xF0\x48\x81\xC4\x40\x02\x00\
rel_jmp_opcode = b"\xe9"  # 4 bytes
mov_rcx_opcode = b"\x48\xB9"  # 8 bytes
mov_rdx_opcode = b"\x48\xBA"  # 8 bytes
mov_edx_opcode = b"\xBA"  # 4 bytes
call_opcode = b"\xe8"  # 4 bytes
lea_in_rcx_opcode = b"\x48\x8D\x0D"  # 4 bytes
add_to_ptr_from_eax_opcode = b"\x01\x05"
add_to_eax_opcode = b"\x05"
mov_eax_in_addr_opcode = b"\x89\x05"

push_rax_opcode = b"\x50"
pop_rax_opcode = b"\x58"

push_rcx_opcode = b"\x51"
pop_rcx_opcode = b"\x59"

push_rdx_opcode = b"\x52"
pop_rdx_opcode = b"\x5A"


def overwrite_instruction(barray, offset, instruction):
    instruction_len = len(instruction)
    barray[offset : offset + instruction_len] = instruction
    return len(instruction)


func_code = b"\xC3"
func_code = sha1_func_code
overwrite_instruction(file_data, new_section_ra + instruction_ptr, func_code)
instruction_ptr += len(func_code)
```

```
jmp_array = (
    []
)  # [raw_address_of_add_operand , offset_from_jump_to_original_destination]

new_jmp_instructions_array = []
for i in range(len(jmp_instructions_array) // JMP_RATIO):
    chosen = random.choice(jmp_instructions_array)
    new_jmp_instructions_array.append(chosen)
    jmp_instructions_array.remove(chosen)

for i in new_jmp_instructions_array:
    if (
        len(i.operands) == 1
        and i.operands[0].size == 32
        and i.operands[0].type == "Immediate"
        and i.mnemonic == "JMP"
    ):

        # replace original jmp
        instruction_ra = (
            i.address - text_section.section_min_addr + text_section.PointerToRawData
        )
        replacement_jmp = rel_jmp_opcode + struct.pack(
            "i", (new_section_va + instruction_ptr) - 5 - i.address
        )
        overwrite_instruction(file_data, instruction_ra, replacement_jmp)
        #######

        # push rax
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, push_rax_opcode
        )
        # push rcx
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, push_rcx_opcode
        )
        # push rdx
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, push_rdx_opcode
        )

        # lea into rcx first arg (address)
        va_of_hashing_target = (
            new_section_va + instruction_ptr + len(lea_in_rcx_opcode) + 4
        )
        ra_of_hashing_target = new_section_ra + instruction_ptr + len(lea_in_rcx_opcode)
        lea_of_target = lea_in_rcx_opcode + struct.pack(
            "i", text_section.section_min_addr - (new_section_va + instruction_ptr) - 7
        )
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, lea_of_target
        )
```

```
        # edx second arg (length)
        mov_in_edx = mov_edx_opcode + struct.pack("I", 0x1000)
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, mov_in_edx
        )

        # hashing func call
        test_call = call_opcode + struct.pack("i", -instruction_ptr - 5)
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, test_call
        )

        jmp_array.append([new_section_ra + instruction_ptr + len(add_to_eax_opcode)])
        hash_result_add = add_to_eax_opcode + struct.pack("i", 0)
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, hash_result_add
        )

        mov_new_jmp_addr = mov_eax_in_addr_opcode + struct.pack(
            "i", len(push_rax_opcode) * 3 + len(rel_jmp_opcode)
        )
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, mov_new_jmp_addr
        )

        # pop rdx
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, pop_rdx_opcode
        )
        # pop rcx
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, pop_rcx_opcode
        )
        # pop rax
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, pop_rax_opcode
        )

        # create detour jmp
        jump_address = i.operands[0].value
        detour_jmp = rel_jmp_opcode + struct.pack("i", 0)
        jmp_array[-1].append(jump_address - (new_section_va + instruction_ptr) - 5)
        jmp_array[-1].append(va_of_hashing_target)
        jmp_array[-1].append(ra_of_hashing_target)
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, detour_jmp
        )

        if (instruction_ptr - 0x100) >= new_section_vs:
            print("LIMIT")
            break
```

```
for i in guards:
    # replace original jmp
    instruction_ra = (
        i.address - text_section.section_min_addr + text_section.PointerToRawData
    )
    replacement_jmp = rel_jmp_opcode + struct.pack(
        "i", (new_section_va + instruction_ptr) - 5 - i.address
    )
    overwrite_instruction(file_data, instruction_ra, replacement_jmp)
    #######

    # push rax
    instruction_ptr += overwrite_instruction(
        file_data, new_section_ra + instruction_ptr, push_rax_opcode
    )
    # push rcx
    instruction_ptr += overwrite_instruction(
        file_data, new_section_ra + instruction_ptr, push_rcx_opcode
    )
    # push rdx
    instruction_ptr += overwrite_instruction(
        file_data, new_section_ra + instruction_ptr, push_rdx_opcode
    )

    for i in PEObj.sections:

        # lea into rcx first arg (address)
        va_of_hashing_target = (
            new_section_va + instruction_ptr + len(lea_in_rcx_opcode) + 4
        )
        ra_of_hashing_target = new_section_ra + instruction_ptr + len(lea_in_rcx_opcode)
        lea_of_target = lea_in_rcx_opcode + struct.pack(
            "i", text_section.section_min_addr - (new_section_va + instruction_ptr) - 7
        )
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, lea_of_target
        )

        # edx second arg (length)
        mov_in_edx = mov_edx_opcode + struct.pack("I", 0x1000)
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, mov_in_edx
        )

        # hashing func call
        test_call = call_opcode + struct.pack("i", -instruction_ptr - 5)
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, test_call
        )
```

```
            jmp_array.append([new_section_ra + instruction_ptr + len(add_to_eax_opcode)])
            hash_result_add = add_to_eax_opcode + struct.pack("i", 0)
            instruction_ptr += overwrite_instruction(
                file_data, new_section_ra + instruction_ptr, hash_result_add
            )

            cmp_hash = cmp_eax_in_addr_opcode + struct.pack(
                "i", len(push_rax_opcode) * 3 + len(rel_jmp_opcode)
            )
            instruction_ptr += overwrite_instruction(
                file_data, new_section_ra + instruction_ptr, cmp_hash
            )

            jmp_after_cmp_addr = mov_eax_in_addr_opcode + struct.pack(
                "i", len(push_rax_opcode) * 3 + len(rel_jmp_opcode)
            )
            instruction_ptr += overwrite_instruction(
                file_data, new_section_ra + instruction_ptr, jmp_after_cmp_addr
            )

        # pop rdx
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, pop_rdx_opcode
        )
        # pop rcx
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, pop_rcx_opcode
        )
        # pop rax
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, pop_rax_opcode
        )

        # create detour jmp
        jump_address = i.operands[0].value
        detour_jmp = rel_jmp_opcode + struct.pack("i", 0)
        jmp_array[-1].append(jump_address - (new_section_va + instruction_ptr) - 5)
        instruction_ptr += overwrite_instruction(
            file_data, new_section_ra + instruction_ptr, detour_jmp
        )

        if (instruction_ptr - 0x100) >= new_section_vs:
            print("LIMIT")
            break


def compute_hash_32(data):
    hash_obj = hashlib.sha1(data)
    return struct.unpack(">i", hash_obj.digest()[4:8])[0]
```

```
print("created jmps:", len(jmp_array))


addresses_to_protect = [
    0x1000 * (i + 1) for i in range(text_section.Misc_VirtualSize // 0x1000)
]
counter = 0

for i in jmp_array:
    (
        add_to_eax_data_offset,
        rel_va_of_original_destination,
        va_of_hashing_target,
        ra_of_hashing_target_data,
    ) = i

    address_to_protect = addresses_to_protect[counter]
    protection_section = PEObj.get_section_by_rva(address_to_protect)
    if not protection_section:
        print("Not found virtual address to protect")
        continue
    ra_protection_address = (
        protection_section.PointerToRawData
        + addresses_to_protect[counter]
        - protection_section.VirtualAddress
    )
    hash_result_32 = compute_hash_32(
        file_data[ra_protection_address : ra_protection_address + 0x1000]
    )
    #   print("HASH" , hex(hash_result_32))

    new_add_data = ctypes.c_int32(rel_va_of_original_destination - hash_result_32).value
    if abs(new_add_data) > 0x80000000:
        print("OUTOFBOUNDS", new_add_data, hex(new_add_data), i)
    new_add_data_bytes = struct.pack("i", new_add_data)
    overwrite_instruction(file_data, add_to_eax_data_offset, new_add_data_bytes)

    new_lea_data = struct.pack("i", address_to_protect - va_of_hashing_target)
    overwrite_instruction(file_data, ra_of_hashing_target_data, new_lea_data)

    new_lea_data = struct.pack("i", address_to_protect - va_of_hashing_target)
    overwrite_instruction(file_data, ra_of_hashing_target_data, new_lea_data)

    if (counter + 1) >= len(addresses_to_protect):
        print(i)

    counter += 1
    if counter >= len(addresses_to_protect):
        print("COUNTER RESET")
        counter = 0
```

```
f = open("out.exe", "wb")
f.write(file_data)
f.close()
```