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

«На правах рукопису»

УДК 004.056.5

«До захисту допущено»
Завідувач кафедри
_____ Дмитро ЛАНДЕ
“___”_____2023 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою «Системи, технології та математичні методи кібербезпеки»

зі спеціальності 125 «Кібербезпека»

на тему: ТЕХНІКА ЕМУЛЯЦІЇ ВИКЛИКІВ WINAPI У ЗАСТОСУНКАХ WINDOWS ЯК МЕТОД ПРОТИДІЇ ЗВОРОТНОЇ РОЗРОБКИ

Виконав здобувач ступеня магістра II курсу, групи ФБ-11мн

Рейценштейн Кирило Сергійович_____ _____
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник доцент, к.т.н.,

Гальчинський Л. Ю._____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент_____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Здобувач ступеня магістра _____
(підпис)

Київ – 2023 року

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

Рівень вищої освіти – другий (магістерський)
Спеціальність – 125 «Кібербезпека»
Освітньо-наукова програма «Системи, технології та математичні методи кібербезпеки»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Дмитро ЛАНДЕ
(підпис)

«___»_____2023 р.

ЗАВДАННЯ
на магістерську дисертацію здобувачу ступеня магістра

Рейценштейн Кирилу Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема дисертації: Техніка емуляції викликів WinAPI у застосунках Windows як метод протидії зворотної розробки,
науковий керівник дисертації Гальчинський Леонід Юрійович, к.т.н., доцент кафедри інформаційної безпеки
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «11» квітня 2023 р. №1482-с

2. Термін подання здобувачем дисертації 12.05.2023 р.

3. Об'єктом дослідження є застосунки Windows.

4. Предметом дослідження є техніка емуляції викликів WinAPI у застосунках Windows як метод протидії зворотній розробки.

5. Перелік завдань, які потрібно розробити 1. Розглянути теоретичні матеріали методів та технік для протидії зворотній розробці та налагодженню. 2. Розглянути перехід виклику WinAPI з простору користувача до простору ядра. 3. Розробити бібліотеку для емуляції викликів WinAPI. 4. Протестувати бібліотеку на застосунках Windows. 5. Проаналізувати результати.

6. Орієнтовний перелік ілюстративного матеріалу – презентація.

7. Опублікований перелік публікацій – “Grail of Science №27 May, 2023" Секція XX. ТЕХНІКА ЕМУЛЯЦІЇ ВИКЛИКІВ WINAPI У ЗАСТОСУНКАХ WINDOWS ЯК МЕТОД ПРОТИДІЇ ЗВОРОТНОЇ РОЗРОБКИ Рейценштейн К., Гальчинський Л. (ст. 361-367) https://archive.journal-grail.science/index.php/2710-3056/issue/view/12.05.2023_____

8. Дата видачі завдання 01.01.2023

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Вибір тематики роботи	01.01.2023	Виконано
2	Огляд літератури	01.01.2023 – 01.02.2023	Виконано
3	Розробка бібліотеки для емуляції викликів WinAPI	15.02.2023 – 28.02.2023	Виконано
4	Тестування бібліотеки	01.03.2023 – 01.04.2023	Виконано
5	Проведення експериментального дослідження і аналіз результатів	01.03.2023 – 01.04.2023	Виконано
6	Оформлення дипломної роботи	01.04.2023 – 30.04.2023	Виконано

Здобувач ступеня магістра _____ Кирило РЕЙЦЕНШТЕЙН
(підпис) (власне ім’я, ПРІЗВИЩЕ)

Науковий керівник _____ Леонід ГАЛЬЧИНСЬКИЙ
(підпис) (власне ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ

Робота обсягом 146 сторінок містить 122 ілюстрації, 23 таблиці, 25 джерел за переліком посилань та 4 додатка.

У даній роботі було досліджено техніки та методи для протидії зворотної розробки у застосунках Windows.

Було запропоновано емуляцію викликів WinAPI, як покращення технік, які базуються на використанні викликів WinAPI та реалізовано бібліотеку для емуляції викликів WinAPI.

Отримана бібліотека може використовуватися у захисті програмного забезпечення будь-яким розробником, а також для тестування поведінки застосунку при використанні WinAPI.

Ключові слова: WinAPI, PEB, TEB, алгоритми мутації, алгоритми віртуалізації, алгоритми компресії, алгоритми заплутування, техніки та методи протидії налагодженню.

ABSTRACT

The 146-page work contains 122 illustrations, 23 tables, 25 sources according to the list of references and 4 appendices.

This paper explored techniques and methods for countering reverse engineering in Windows applications.

Emulation of WinAPI calls was proposed as an improvement of techniques based on the use of WinAPI calls and a library for emulating WinAPI calls was implemented.

The resulting library can be used in software protection by any developer, as well as for testing application behavior when using WinAPI.

Keywords: WinAPI, PEB, TEB, mutation algorithms, virtualization algorithms, compression algorithms, obfuscation algorithms, anti-debugging techniques and methods.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

PEB – Process Environment Block

TEB – Thread Environment Block

WinAPI – Windows API

ВСТУП

У світі, де кіберзлочинність стає все більшою проблемою, забезпечення безпеки програмного забезпечення є однією з найважливіших задач в галузі кібербезпеки. Код, що міститься у програмах, може бути піддано зворотній розробці, що може призвести до витоку конфіденційної інформації, або створення шкідливих програм.

У даній роботі буде розглянуто різні методи та техніки протидії зворотної розробки, а також протидії налагодженню. Будуть розглянуті переваги та недоліки кожного методу або техніки, а також буде проведене експериментальне дослідження на застосунках Windows.

Актуальність дослідження. На сьогоднішній день існує багато технік та методів протидії зворотної розробки або налогодженю. Головна проблема методів та технік, які базуються на використані викликів WInAPI для перевірки чи застосунок під налагодженням це використання викликів WinAPI, що дає зловмиснику можливість перехоплювати функції та маніпулювати як вхідними так і вихідними даними. Тема є актуальною оскільки результат роботи може допомогти вирішити дану проблему та покращити вже існуючи методи та техніки.

Мета дослідження. Метою дослідження є пошук найефективнішого поєднання технік і методів для захисту застосунків від зворотної розробки.

Завдання дослідження. Завданням дослідження є аналіз існуючих методів та технік протидії зворотної розробки та налагодженню, детальний аналіз викликів WinAPI з user-mode до kernel-mode, розробка власної бібліотеки для емуляції викликів WinAPI, тестування біблотеки на застосунках Windows.

Об’єкт дослідження. Об'єктом дослідження є техніка емуляції викликів WinAPI у застосунках Windows як метод протидії зворотної розробки. В рамках дослідження будуть розглянуті різні методи та техніки захисту програмного забезпечення від зворотної розробки, а також будуть

досліджені можливості техніки емуляції викликів WinAPI для захисту від такого типу атак. Основними завданнями дослідження є вивчення теоретичних основ емуляції викликів WinAPI, розробка бібліотеки для емуляції викликів, порівняння ефективності захисту програмного забезпечення за допомогою емуляції викликів WinAPI та прямих викликів, а також аналіз можливостей комерційних рішень для захисту програмного забезпечення.

Наукова новизна. Науковою новизною даної дипломної роботи є покращення технік та методів протидії налагодженню, які базуються на викликах WinAPI завдяки емуляції цих викликів.

Предмет дослідження. Предметом дослідження є техніка емуляції викликів WinAPI у застосунках Windows як метод протидії зворотній розробці.

Практичне значення. Практичне значення даної дипломної роботи полягає в розробці бібліотеки для емуляції викликів WinAPI, яка може бути використана розробниками програмного забезпечення для забезпечення захисту від зворотної розробки та аналізу їх програм. Розробка даної бібліотеки може допомогти уникнути втрат даних, викрадення інтелектуальної власності та втрати конкурентної переваги. Також результати порівняння застосування бібліотеки з прямими викликами WinAPI та з емульованими викликами WinAPI можуть бути корисні для вирішення проблем захисту програмного забезпечення.

Апробація результатів та публікації. Результати цієї роботи були представлені в “Grail of Science №27 May, 2023" Секція XX. ТЕХНІКА ЕМУЛЯЦІЇ ВИКЛИКІВ WINAPI У ЗАСТОСУНКАХ WINDOWS ЯК МЕТОД ПРОТИДІЇ ЗВОРОТНОЇ РОЗРОБКИ Рейценштейн К. Гальчинський Л. (ст. 361-367) https://archive.journal-grail.science/index.php/2710-3056/issue/view/12.05.2023

1 АНАЛІЗ ІСНУЮЧИХ ТЕХНІК ТА МЕТОДІВ ПРОТИДІЇ ЗВОРОТНОЇ РОЗРОБКИ

У цьому розділі буде розглянуто широкий спектр методів та технік, які використовуються для захисту програм від зловмисників, які намагаються отримати доступ до джерела програмного коду. У розділі проводиться аналіз переваг та недоліків кожного методу та техніки. Досліджується можливість їх використання в реальних умовах та ефективність в захисті програмного забезпечення від зворотної розробки. До методів та технік, що аналізуються, можна віднести запакування коду, використання анти-дизасемблерів, змінення імен функцій, використання криптографічних методів та інших.

1.1 Алгоритми заплутування

Обфускація коду входить до числа найбільш популярних технік забезпечення додатків від хакерських атак. Це є однією з найбільш рекомендованих ініціатив забезпечення додатків від зловмисних атак для фахівців з кібербезпеки у всьому світі, і часто використовується для забезпечення мінімальних потреб у безпеці додатку. Ця техніка, як правило, виступає як основний механізм захисту від спроб хакерських атак і захищає від поширених атак, таких як ін’єкція коду, зворотна розробка та втручання у персональні дані клієнтів і користувачів додатку.

Обфускація коду - це зміна виконувального коду, щоб його було неможливо зрозуміти, інтерпретувати та виконувати.[1] Сам код джерела є затуманеним, щоб він став незрозумілим і неможливим для сторонньої особи зрозуміти його, не кажучи вже про виконання. Обфускація коду не впливає на інтерфейс додатку, який призначений для кінцевого користувача або на передбачені результати коду. Це просто заходи попередження, які роблять код непридатним для використання потенційним хакером, який може потрапити у руки виконувального коду додатка. Обфускація коду особливо корисна для програм із відкритим вихідним кодом, які створюють величезний недолік з точки зору можливості злому коду для особистої

вигоди. Роблячи програму складною для зворотного проектування, розробники гарантують, що інтелектуальна власність їхнього продукту захищена від загроз безпеки, несанкціонованого доступу та виявлення вразливостей програми. Цей процес обмежує зловмисний доступ до вихідного коду та, залежно від типу застосованої техніки обфускації, забезпечує різні рівні захисту коду. Фактори часу, вартості та ресурсів схиляють терези на користь відмови від коду, коли він обфускований, оскільки декомпільований код стає незрозумілим. Обфускація діє на кількох рівнях – вона застосовується або на рівні семантичної структури/структури лексичного коду, або на рівні структури даних/потоку керування. Методи обфускації також відрізняються залежно від операції, яку вони виконують над кодом. По суті, команда безпеки, консультуючись із командою розробників, вирішує, який тип обфускації використовувати в коді.

Обфускація шляхом зміни імен. Ця техніка передбачає заплутане іменування змінних, щоб оригінальний намір їх використання розумно маскувався. Методи та змінні перейменовуються з використанням різних символів і чисел, що ускладнює декомпіляторам розуміння потоку керування. Ця техніка обфускації зазвичай використовується для обфускації коду програми, розробленого на платформах Java, .NET. Це підпадає під загальну категорію обфускації макета, спрямованої безпосередньо на вихідний код, щоб створити рівень захисту для програми. Нижче можна побачити як виглядає не обфускований код:

```
using System;

namespace ExampleNamespace
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Рисунок 1.1 – Не обфускований код

Після обфускації код буде виглядати так:

```
using System;

namespace a
{
    class b
    {
        static void c(string[] d)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Рисунок 1.2 – Обфускований код

Як можна побачити, всі імена змінних, функцій та класів змінені на випадкові рядки символів, що ускладнює зворотню розробку коду.

Обфускація даних. Ця техніка націлена на структури даних, які використовуються в коді, так що зловмисник не може побачити фактичний

намір програми. Це може передбачати зміну способу збереження даних у програмі в пам’яті та способу інтерпретації збережених даних для відображення кінцевого результату.[2] Існують різні варіації цієї техніки:

Обфускація агрегації. Змінюється спосіб зберігання даних у програмі. Наприклад, масиви можна розбити на багато підмасивів, на які потім можна посилатися в різних місцях програми. Код до та після обфускації:

```cpp
#include <iostream>
using namespace std;

int add(int a, int b)
{
    return a + b;
}

int subtract(int a, int b)
{
    return a - b;
}

int main()
{
    int a = 5;
    int b = 3;
    int c = add(a, b);
    int d = subtract(a, b);
    cout << "a + b = " << c << endl;
    cout << "a - b = " << d << endl;
    return 0;
}
```

```cpp
#include <iostream>
using namespace std;

int f1(int a, int b)
{
    return a + b;
}

int f2(int a, int b)
{
    return a - b;
}

int f3(int a, int b)
{
    return a * b;
}

int f4(int a, int b)
{
    return a / b;
}

void main()
{
    int a = 5;
    int b = 3;
    int c = f1(a, b);
    int d = f2(a, b);
    int e = f3(a, b);
    int f = f4(a, b);
    cout << "a + b = " << c << endl;
    cout << "a - b = " << d << endl;
    return;
}
```

Рисунок 1.3 – Не обфускований/Обфускований код

Як можна побачити, всі функції злиті в одну, змінений порядок виклику функцій та додані додаткові функції, що нічого не роблять, але ускладнюють зворотний розробку коду.

Обфускація поведінки в пам’яті. Це змінює сам спосіб зберігання даних у пам’яті. Наприклад, розробники можуть перемішувати між локальним і глобальним зберіганням змінних, щоб справжня природа поведінки змінної була прихована. Код до та після обфускації:

```
#include <iostream>
using namespace std;

int main()
{
    int a = 5;
    int b = 3;
    int c = a + b;
    cout << "a + b = " << c << endl;
    return 0;
}
```

Рисунок 1.4 – Не обфускований код

```cpp
#include <iostream>
using namespace std;

int f1(int x, int y)
{
    int z = x * y;
    int w = z + x;
    return w;
}

int f2(int x, int y)
{
    int a = x - y;
    int b = y + 1;
    int c = a + b;
    return c;
}

void main()
{
    int x1 = 5;
    int y1 = 3;
    int x2 = 4;
    int y2 = 2;
    int x3 = 3;
    int y3 = 5;
    int z1 = f1(x1, y1);
    int z2 = f2(x2, y2);
    int z3 = f1(x3, y3);
    cout << "z1 = " << z1 << endl;
    cout << "z2 = " << z2 << endl;
    cout << "z3 = " << z3 << endl;
    return;
}
```

Рисунок 1.5 – Обфускований код

Обфускація порядку даних. Цей метод змінює порядок упорядкування даних, не змінюючи поведінку програми/фрагмента коду. Розробники досягають цього шляхом розробки окремого модуля, який викликається для всіх екземплярів посилання на змінну.

Обфускація текстових даних. Цей метод шифрує всі рядки, які можна прочитати, і, отже, призводить до нечитабельного коду.[3] Їх потрібно розшифрувати під час виконання програми.

Приклад:

```
#include "xorstr.hpp"

int main() {
    std::puts(xorstr_("an extra long hello_world"));
}
```

Рисунок 1.6 – Приклад використання бібліотеки xorstr

У цьому випадку строка буде зашифрована при компіляції застосунку та розшифрована під час виконання програми.

Обфускація коду/потоку керування. Управління, яке передається від однієї функції до іншої, грає вирішальну роль у визначенні мети програми. Заплутування цього потоку зазвичай є одним з найбільш ефективних способів заплутати зловмисників, які намагаються зворотно розробити програму. Цей метод обфускації може допомогти утримати зловмисників на відстані, не дозволяючи їм зрозуміти, як і чому код приймає певний потік. Один з найпоширеніших способів реалізації цього методу обфускації полягає у використанні довільних та неочікуваних операторів, додаванні непотрібних операторів перемикання регістру (мертвий код), які ніколи не будуть виконані. Це може включати у себе додавання непотрібних гілок if або switch, які ніколи не виконуються, використання непотрібних goto та додавання викликів функцій, які не використовуються в програмі. Використання цих методів може зробити код більш складним для зворотного розроблення та ускладнити процес зрозуміння логіки програми для зловмисників, що намагаються зламати її.

Приклад:

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

int main()
{
    // Генеруємо випадкове число
    srand(time(NULL));
    int randomNumber = rand() % 100;

    // Додаємо непотрібну гілку коду
    if (randomNumber > 50)
    {
        std::cout << "This code block will never execute" << std::endl;
    }

    // Головний блок коду
    std::cout << "This code block will always execute" << std::endl;

    return 0;
}
```

Рисунок 1.7 – Приклад обфускації потоку керування

У цьому прикладі, якщо випадкове число, згенероване за допомогою функції rand(), є більшим за 50, то умова if виконається і надрукований рядок "This code block will never execute". Однак, якщо число менше або дорівнює 50, то ця умова не буде виконуватися, і програма перейде до наступного блоку коду. Це додаткове розгалуження не має ніякої корисної функції в програмі, і його можна вважати непотрібним, але такий код може стати перешкодою для зловмисників, що намагаються зрозуміти логіку програми. Це може бути використано для заплутування потоку управління кодом і унеможливлення зловмисників отримати доступ до важливих ділянок коду.[4]

Обфускація даних налагодження. Інформація, яка отримується під час налагодження, часто містить корисну інформацію про перебіг програми та допомагає виявляти недоліки в програмі. Однак, ця інформація може стати легкою мішенню для зловмисників, які намагаються проникнути до програмного коду. З цієї причини важливо маскувати ідентифікаційну

інформацію, що міститься в налагоджувальній інформації, змінюючи ідентифікатори, номери рядків або припиняючи доступ до налагоджувальних інструментів. Наприклад, можна змінити імена змінних та функцій, які використовуються в програмі, щоб збільшити складність зворотного розроблення програми. Також можна заборонити доступ до налагоджувальних інструментів за допомогою захисту паролем або виключити налагоджувальний режим повністю. Маскування інформації про налагодження може допомогти зберегти конфіденційність вихідного коду програми та унеможливити зловмисникам отримати доступ до важливих ділянок коду. Такі заходи можуть бути важливим елементом комплексної системи захисту програмного коду від зламу.

Приклад:

```
#include <iostream>

// Видалення директиви препроцесора #line
#pragma push_macro("line")
#undef line
#define line 0

int main()
{
    // Виведення тексту
    std::cout << "Hello, world!" << std::endl;

    return 0;
}
```

Рисунок 1.8 – Модифікація debug інформації

У цьому прикладі директива препроцесора #line була видалена за допомогою директиви #undef, а після цього замінена на значення 0 за допомогою директиви #define. Це допомагає унеможливити зловмисникам використовувати інформацію про номер рядка та назву файлу вихідного коду для дослідження програмного коду.

Обфускація простору адрес. Атаки, які використовують помилки програмування пам'яті, є одними з найпоширеніших методів злому програм. Наприклад, використання неперевіреного доступу до масиву може призвести до вразливості безпеки, яка може бути використана зловмисниками для отримання незаконного доступу до важливих даних. Одним із методів захисту програм від таких атак є метод обфускації адреси. Цей метод полягає в тому, що кожен раз, коли виконується перетворений код, віртуальні адреси програмного коду та даних програми рандомізуються. Це ускладнює процес зворотного проектування, оскільки зловмисники не можуть зрозуміти, які саме адреси використовуються в програмі. Цей метод робить ефект більшості експлойтів помилок пам'яті недетермінованим, тобто зловмисник не може точно передбачити, яка адреса буде використана в програмі. Це зменшує шанси на успішну атаку та робить програмний код більш стійким до злому. Обфускація адреси може бути важливим елементом системи захисту програмного коду від різних атак та вразливостей.[5]

Приклад:

```cpp
#include <iostream>

// Обфускація адреси пам'яті змінної
void* ObfuscateAddress(void* address)
{
    return reinterpret_cast<void*>(reinterpret_cast<std::intptr_t>(address) ^ 0xCAFEBABE);
}

// Функція, яка використовує обфусковану адресу пам'яті
void PrintAddress(void* address)
{
    void* obfuscatedAddress = ObfuscateAddress(address);
    std::cout << "Address: " << obfuscatedAddress << std::endl;
}

int main()
{
    int a = 10;

    // Виклик функції, яка використовує обфусковану адресу пам'яті
    PrintAddress(&a);

    return 0;
}
```

Рисунок 1.9 – Обфускація адреси

У цьому прикладі функція ObfuscateAddress приймає адресу пам'яті змінної та обфускує її, використовуючи операцію XOR зі значенням 0xCAFEBABE. Далі функція PrintAddress викликає функцію ObfuscateAddress, щоб отримати обфусковану адресу пам'яті та вивести її на екран.

Алгоритми заплутування є важливим елементом захисту програмного коду від зломів та вразливостей. Проте, як і в будь-якому заході безпеки, існують як позитивні, так і негативні аспекти використання обфускації коду. Нижче наведено деякі з плюсів та мінусів використання алгоритмів заплутування.

Плюси:

1. Захист від реверс-інженерії: Може ускладнити процес зворотного проектування та захистити програмний код від взлому.
2. Захист від крадіжки інтелектуальної власності: Якщо програмний код містить важливі деталі, такі як алгоритми, ідеї або алгоритми шифрування, обфускація коду може запобігти їхньому викраденню.
3. Зменшення ризику вразливості програмного коду: Якщо зловмисники не можуть зрозуміти, як працює програма, це може зменшити ризик вразливості програмного коду.

Мінуси:

1. Складність відлагодження: Обфускований код може бути складним для відлагодження та тестування.
2. Збільшення розміру програми: Обфускація коду може збільшити розмір програми та збільшити час завантаження.
3. Вплив на продуктивність: Обфускація коду може вплинути на продуктивність програми, зменшуючи її швидкість та збільшуючи споживання ресурсів.

Отже, код обфускації має як позитивні, так і негативні аспекти. Вибір використання цього методу залежить від конкретних потреб програми та її розробників.

1.2 Алгоритми мутації

Алгоритми мутації - це методи обфускації коду, що використовуються для генерації випадкового коду, що еквівалентний вихідному коду, але важчий для аналізу і зрозуміння.

Основні алгоритми мутації:

1. Заміна ідентифікаторів: заміна назв змінних, функцій та класів на випадкові назви, що не мають логічного зв'язку зі значенням, яке вони зберігають;
2. Перестановка фрагментів коду: перестановка фрагментів коду випадковим чином, зберігаючи логічну структуру програми, але роблячи його важчим для аналізу;
3. Додавання мертвого коду: додавання непотрібних конструкцій, таких як не використовувані змінні та функції, що змінюють структуру програми та роблять її важчою для аналізу;
4. Заміна операторів: заміна стандартних операторів на аналогічні, але менш зрозумілі, що може змінити логіку програми та зробити її важчою для аналізу;
5. Додавання відмінностей: додавання відмінностей у вихідний код, що не впливають на його функціональність, але роблять його важчим для аналізу.

Метою алгоритмів мутації є збільшення складності аналізу програмного коду та ускладнення зворотного проектування. Однак, їх використання може зробити код більш заплутаним та важким для розуміння, що може вплинути на продуктивність та ефективність програми.

Ось приклад мутації коду на мові C++, в якому здійснюється заміна назв змінних на випадкові назви:

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

int main()
{
    int num1, num2, result;
    std::cout << "Enter two numbers: ";
    std::cin >> num1 >> num2;
    srand(time(nullptr));
    int random_num = rand() % 10 + 1;
    for (int i = 0; i < random_num; i++)
    {
        result = num1 + num2;
        num1 = num2;
        num2 = result;
    }
    std::cout << "Result: " << result << std::endl;
    return 0;
}
```

Рисунок 1.10 – Не обфускований код

У цьому коді змінні num1, num2 та result були замінені на випадкові назви, що складаються з 10 символів:

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

int main()
{
    int zcqYfblfbJ, IjKUSWugMT, qUgfIAeuSL;
    std::cout << "Enter two numbers: ";
    std::cin >> zcqYfblfbJ >> IjKUSWugMT;
    srand(time(nullptr));
    int CyMvSgnSZG = rand() % 10 + 1;
    for (int XqceYaXcKP = 0; XqceYaXcKP < CyMvSgnSZG; XqceYaXcKP++)
    {
        qUgfIAeuSL = zcqYfblfbJ + IjKUSWugMT;
        zcqYfblfbJ = IjKUSWugMT;
        IjKUSWugMT = qUgfIAeuSL;
    }
    std::cout << "Result: " << qUgfIAeuSL << std::endl;
    return 0;
}
```

Рисунок 1.11 – Обфускований код

Тепер код є більш заплутаним та важким для аналізу, що може ускладнити процес зворотного проектування.

Плюси алгоритмів мутації:

1. Підвищення складності зворотного проектування. Мутація змінює код програми таким чином, що ускладнює аналіз та зрозуміння коду. Наприклад, змінення імен змінних на випадкові може ускладнити зрозуміння функції цих змінних;
2. Зниження ризику виявлення вразливостей у програмі. Мутація може змінити програмний код таким чином, що вразливості, що були присутні у вихідному коді, стають менш очевидними або навіть повністю приховуються;
3. Збільшення надійності програм. Мутація може підвищити стійкість програми до атак та помилок, що можуть спричинити відмову програми.

Мінуси алгоритмів мутації:

1. Потенційна втрата функціональності. Мутація може призвести до зміни поведінки програми або викликати її збій. Це може статися внаслідок неправильної реалізації алгоритмів мутації або недостатньої їх тестування;
2. Погіршення читабельності коду. Мутація може зробити код програми складнішим для розуміння та підтримки, особливо якщо використовуються складні та випадкові трансформації коду;
3. Збільшення вартості та складності розробки програм. Мутація може вимагати від розробників додаткових зусиль для тестування, налагодження та підтримки програми. Також може бути потрібно використовувати спеціальні інструменти та бібліотеки для здійснення мутації коду, що може збільшити вартість проекту.

Алгоритми мутації можуть допомогти у захисті програмного коду від зворотного проектування та атак зловмисників. Вони є ефективними

засобами для ускладнення аналізу та зрозуміння коду, зниження ризику виявлення вразливостей та підвищення надійності програм. Однак, використання алгоритмів мутації може призвести до погіршення читабельності коду, втрати функціональності та збільшення вартості розробки програм.[6]

Тому, вибір оптимальних методів захисту залежить від потреб програми та її розробки, а також потреби забезпечення балансу між захистом і читабельністю коду. Ретельне вивчення особливостей алгоритмів мутації та їхніх обмежень допоможе розробникам забезпечити ефективний захист програмного коду від потенційних загроз.

1.3 Алгоритми віртуалізації

Алгоритми віртуалізації є однією з технік обфускації коду, яка застосовується для захисту програмного забезпечення від атак зловмисників. Ця техніка полягає в тому, що програмний код замість прямого виконання перетворюється віртуальний код, який виконується на спеціальному віртуальному процесорі.Одним з головних переваг використання коду віртуалізації є те, що він робить аналіз та зворотне проектування коду набагато складнішим. Код віртуалізації розбиває оригінальний код на дрібні блоки та віртуалізує їх, щоб зменшити можливість зворотного проектування. Техніка віртуалізації коду є особливо корисною для програм, які містять важливі дані, наприклад, програми зі шкідливим кодом, антивірусні програми, програми для захисту мережі та інші. Одним з недоліків коду віртуалізації є те, що він може знизити продуктивність програми, оскільки виконання коду відбувається на віртуальному процесорі, а не на реальному. Крім того, код віртуалізації може бути складним для підтримки та оновлення. Загалом, код віртуалізації є важливим елементом захисту програмного забезпечення від атак зловмисників, оскільки він робить зворотне проектування програмного коду набагато складнішим. Проте, перед

використанням цієї техніки необхідно зрозуміти, як вона може вплинути на продуктивність програми та складність її підтримки.

Техніка обфускації віртуалізації перетворює оригінальний код у віртуальний код, який може інтерпретувати структура віртуалізації. Компоненти структури для перетворення, отримання, декодування та виконання віртуального коду є такими:

1. Оригінальний код: послідовність інструкцій, які потрібно захистити в цільовій програмі;
2. Віртуальний код: послідовність інструкцій, перетворена з оригінального коду, так що лише згенерована віртуальна машина (тобто диспатчер) може її декодувати;
3. Код обробника: послідовність інструкцій цільової машини, яка відповідає за фактичну функціональність віртуального коду;
4. Диспатчер: компонент у віртуальній машині, який відповідає за декодування віртуального коду;
5. Віртуальний регістр: регістр, який використовується в структурі віртуалізації та зберігається в певній області пам’яті або області стека;

Обфускація віртуалізації зазвичай застосовується наступним чином:

1. Обфускатор віртуалізації перетворює оригінальний код на віртуальний код;
2. Обфускатор віртуалізації перетворює віртуальний код на код обробки;
3. Якщо обфускатор віртуалізації надає відповідні параметри, він заплутує або впорядковує послідовності кодів обробника.

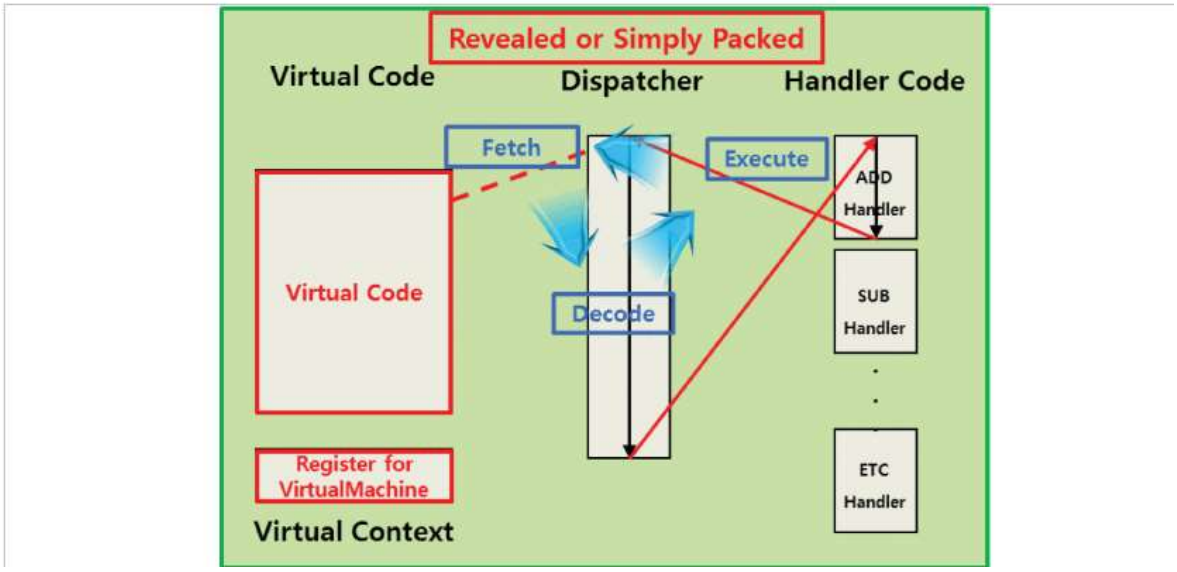


Рисунок 1.12 – Приклад віртуальної машини

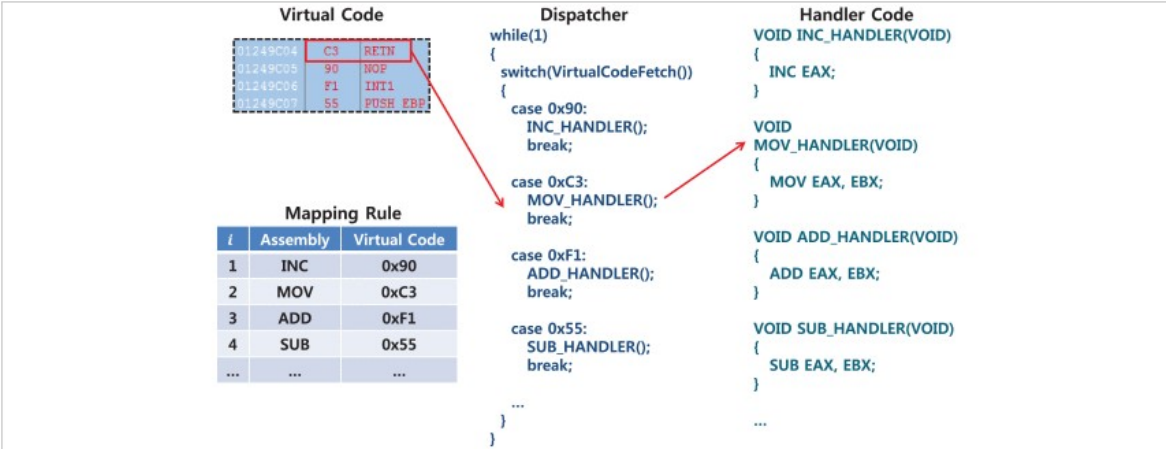


Рисунок 1.13 – Приклад віртуальної машини

Приклад:

```cpp
#include <iostream>

typedef int (*func_ptr)(int);

int virtual_func(int x)
{
    return x + 1;
}

void virtualize_func(void* ptr)
{
    unsigned char* code = reinterpret_cast<unsigned char*>(ptr);
    const size_t code_size = 16;
    unsigned char backup[code_size];
    memcpy(backup, code, code_size);

    const unsigned char replacement_code[] = {
        0x55,                   // push ebp
        0x89, 0xe5,             // mov ebp, esp
        0x83, 0xec, 0x08,       // sub esp, 8
        0x8b, 0x45, 0x08,       // mov eax, [ebp+8]
        0x83, 0xc0, 0x01,       // add eax, 1
        0xc9,                   // leave
        0xc3                    // ret
    };
    memcpy(code, replacement_code, sizeof(replacement_code));

    func_ptr virtual_func_ptr = reinterpret_cast<func_ptr>(ptr);
    int result = virtual_func_ptr(5);
    std::cout << "Virtualized function result: " << result << std::endl;

    memcpy(code, backup, code_size);
}

int main()
{
    void* func_ptr = reinterpret_cast<void*>(virtual_func);
    virtualize_func(func_ptr);
    int result = virtual_func(5);
    std::cout << "Original function result: " << result << std::endl;
    return 0;
}
```

Рисунок 1.14 – Приклад віртуалізації

У цьому прикладі функція virtualize_func приймає вказівник на функцію virtual_func, яка зберігається як беззнаковий символьний масив. Код функції віртуалізується шляхом заміни початкового коду новим кодом, який має той самий ефект. Функція memcpy використовується для збереження копії

оригінального коду. Після того, як новий код був записаний в пам'ять, функція virtualize_func викликається через вказівник на функцію, що виртуалізується. Змінний вказівник virtual_func_ptr потім використовується для виклику віртуалізованої функції з параметром 5, і результат повертається в змінну result. На завершення функція virtualize_func копіює оригінальний код з резервної копії в пам'ять.[7]

Переваги використання техніки віртуалізації коду для захисту програмного забезпечення:

1. Захист від декомпіляції: віртуалізація коду дозволяє ускладнити процес декомпіляції програми і отримання зрозумілого вихідного коду. Це збільшує рівень захисту програмного забезпечення від зловмисних атак;
2. Рівень складності: віртуалізація коду може зробити програмне забезпечення складнішим для аналізу та розуміння, що зменшує ризики виявлення слабких місць у програмному забезпеченні;
3. Рівень захисту: віртуалізація коду дозволяє додатково захистити програмне забезпечення від різних видів атак, таких як виконання з кодуванням, перевірка цілісності і витік інформації.

Недоліки використання техніки віртуалізації коду для захисту програмного забезпечення:

1. Висока вартість: віртуалізація коду може бути складною і витратною, особливо для великих програмних продуктів;
2. Витрата продуктивності: віртуалізація коду може зменшити продуктивність програмного забезпечення через збільшення розміру програми і додаткові вимоги до ресурсів;
3. Ризик зламу: незважаючи на те, що віртуалізація коду може зробити програмне забезпечення складнішим для аналізу, зловмисники можуть все ще знайти спосіб обійти захист.

1.4 Алгоритми компресії програмного коду

Алгоритми компресії програмного коду допомагають зменшити розмір файлів з програмним кодом і забезпечують більш ефективне зберігання і передачу цих файлів. Ось деякі з найбільш популярних алгоритмів компресії програмного коду:

1. LZ77/LZ78: ці алгоритми використовуються для стиснення даних шляхом заміни повторюваних фрагментів даних посиланнями на раніше вже використані фрагменти даних.
2. Huffman: цей алгоритм використовується для стиснення даних, зменшуючи розмір символів, які використовуються у даних.
3. Arithmetic coding: цей алгоритм використовується для стиснення даних, зменшуючи розмір даних, які передаються, шляхом закодування символів змінною довжиною.
4. Burrows-Wheeler transform: цей алгоритм використовується для стиснення даних, перетворюючи послідовність символів у нову послідовність, яка має більше повторюваних фрагментів.
5. LZW: цей алгоритм використовується для стиснення даних, створюючи словник для зберігання повторюваних фрагментів даних, що дозволяє замінити довгі послідовності даних на коротші посилання на словник.

UPX (Universal Packer for eXecutables).

Це вільний і відкритий засіб для стиснення виконуваних файлів різних операційних систем, включаючи Windows, Linux, Mac OS X та інші. UPX дозволяє зменшити розмір виконуваних файлів шляхом видалення непотрібних розділів, оптимізації таблиць символів, видалення дебаг-інформації та інших операцій. UPX зазвичай використовується розробниками програмного забезпечення для зменшення розміру програми, що дозволяє

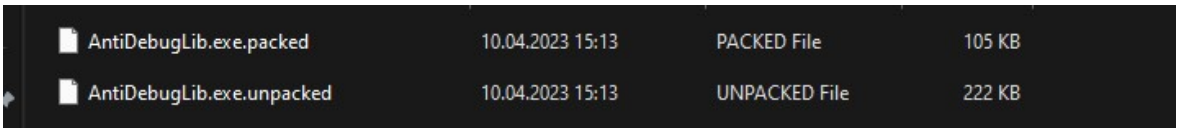
зберегти місце на диску та зменшити час завантаження програми. UPX також може використовуватися для зменшення розміру шкідливого програмного забезпечення, що дозволяє швидше виявляти та усувати загрози безпеці.[8]

UPX стискає виконувані файли, змінюючи їхню структуру та видаляючи зайву інформацію, що не впливає на роботу програми. UPX може виконувати різні види стискання, включаючи:

1. Факторизація: UPX аналізує виконуваний файл та знаходить повторювані фрагменти, які можна замінити посиланнями на вже наявні фрагменти.
2. Заміна: UPX замінює деякі фрагменти виконуваного файлу на менші або більш ефективні аналоги.
3. Зміщення: UPX зміщує деякі частини виконуваного файлу, що дозволяє більш ефективно використовувати пам'ять та зменшити розмір файлу.
4. Комбінування: UPX використовує комбінації факторизації, заміни та зміщення для досягнення найбільш ефективного стискання.

UPX також включає можливості розширення, що дозволяють розробникам визначати власні методи стискання та оптимізації.

Приклад використання:



Як можна побачити запакований розмір файлу 105 КБ в той час як не запакований 222 КБ.

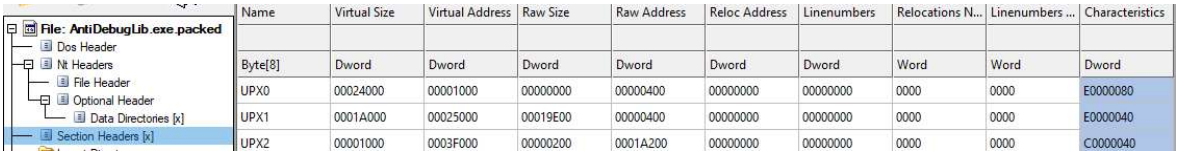
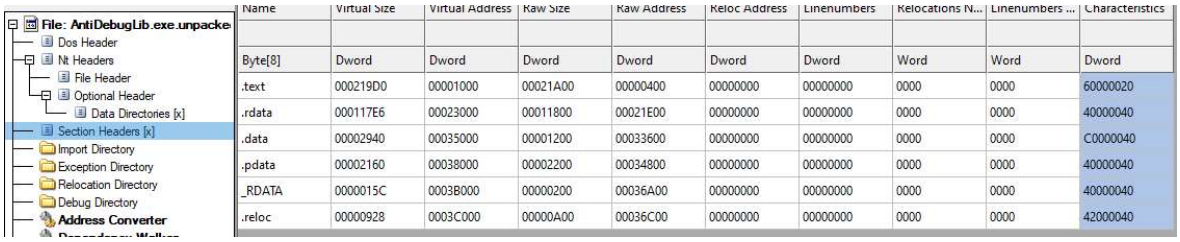


Рисунок 1.15 – CFF Explorer з компресованим файлом у UPX



Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	000219D0	00001000	00021A00	00000400	00000000	00000000	0000	0000	60000020
.rdata	000117E6	00023000	00011800	00021E00	00000000	00000000	0000	0000	40000040
.data	00002940	00035000	00001200	00033600	00000000	00000000	0000	0000	C0000040
.pdata	00002160	00038000	00002200	00034800	00000000	00000000	0000	0000	40000040
_RDATA	0000015C	0003B000	00000200	00036A00	00000000	00000000	0000	0000	40000040
.reloc	00000928	0003C000	00000A00	00036C00	00000000	00000000	0000	0000	42000040

Рисунок 1.16 – CFF Explorer з чистим файлом

MPRESS.

MPRESS - це інструмент для стискання виконуваних файлів, розроблений Андрієм Міхальчуком. Він застосовує різні методи стискання, включаючи видалення зайвих розділів та об'єднання різних секцій, що дозволяє зменшити розмір файлу. MPRESS використовує методи стискання, що піддають виконуваний файл додатковій обробці та модифікації, що може ускладнити реверс-інженерні дослідження та забезпечити певний рівень захисту від деобфускації. MPRESS відомий своєю високою ефективністю та здатністю стискати файли до дуже малих розмірів, що дозволяє економити місце на диску та зменшувати час завантаження програми. Однак, стискання файлів з MPRESS може призвести до збоїв під час роботи програми та інших проблем, що варто враховувати при використанні цього інструменту. MPRESS використовує методи стискання, які дозволяють зменшити розмір виконуваного файлу. [9]

Процес стискання зазвичай включає наступні етапи:

1. Аналіз виконуваного файлу: MPRESS аналізує виконуваний файл та знаходить зайві розділи та інші елементи, які можуть бути видалені або об'єднані.
2. Видалення зайвих елементів: MPRESS видаляє зайві розділи та інші елементи, які не впливають на роботу програми.
3. Об'єднання різних секцій: MPRESS може об'єднувати різні секції виконуваного файлу, що дозволяє зменшити розмір файлу та забезпечити більш ефективне використання пам'яті.

4. Модифікація: MPRESS може вносити зміни до виконуваного файлу, що дозволяє забезпечити додатковий рівень захисту від деобфускації та зменшити ймовірність виявлення можливих вразливостей.
5. Стиснення: MPRESS використовує різні методи стискання для зменшення розміру виконуваного файлу, включаючи методи, які піддають файл додатковій обробці та модифікації.

Після проходження всіх етапів стискання, MPRESS створює новий стиснутий файл, який можна використовувати замість оригінального файлу.

Приклад використання:

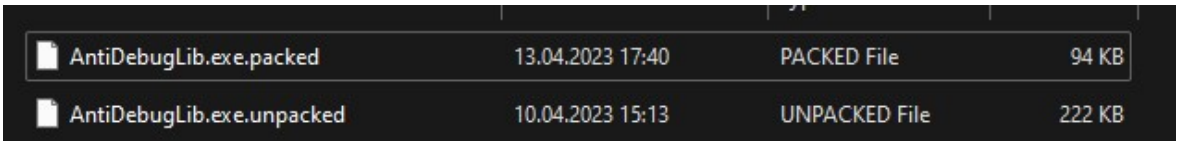


Рисунок 1.17 – Результат виконання MPRESS

Як можна побачити запакований розмір файлу 94 КБ в той час як не запакований 222 КБ. Можна зробити висновок, що MPRESS стискає краще, ніж UPX.

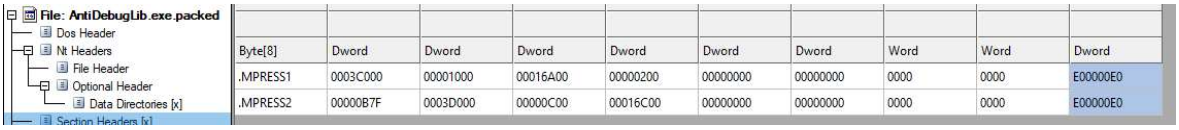


Рисунок 1.18 – CFF Explorer MPRESS

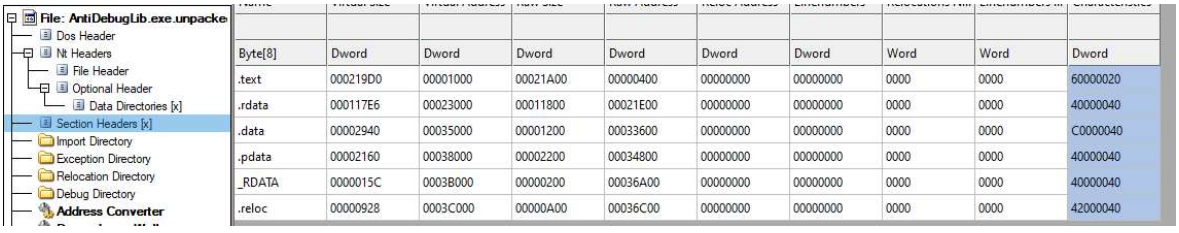


Рисунок 1.19 – CFF Explorer чистий файл

Плюси алгоритмів компресії коду:

1. Зменшення розміру файлу: компресія дозволяє зменшити розмір виконуваного файлу, що поліпшує ефективність його розповсюдження та зменшує витрати на зберігання.

2. Підвищення швидкодії: зменшення розміру файлу дозволяє зменшити час, необхідний для завантаження та запуску програми.
3. Захист від зворотного проектування: деякі алгоритми компресії коду можуть ускладнити процес зворотного проектування, що дозволяє забезпечити додатковий рівень захисту від несанкціонованого доступу до програмного коду.

Мінуси алгоритмів компресії коду:

1. Збільшення часу виконання програми: розпакування стисненого файлу може зайняти додатковий час, що може вплинути на швидкодію програми.
2. Загроза безпеці: стиснення файлу може викликати проблеми з безпекою, так як стиснений файл може містити вразливості, які можуть бути використані для злому системи.
3. Обмеження розширеності: деякі стиснені файли можуть бути нерозпаковані на певних системах, що може обмежити розповсюдження програми.
4. Витрати на розробку: створення та підтримка алгоритмів компресії може вимагати додаткових зусиль та ресурсів, що може збільшувати витрати на розробку програмного продукту.

1.5 Методи та техніки протидії налагодженню

Методи та техніки протидії налагодженню є важливою складовою процесу захисту програмного коду. Ці методи та техніки використовуються для ускладнення налагодження програм та запобігання виявленню вразливостей.

До основних методів та технік протидії налагодженню можна віднести:

1. Видалення інформації про налагодження. Цей метод передбачає видалення налагоджувальних символів, відповідних директив і бібліотек, які можуть допомогти в розумінні коду;
2. Запобігання дії налагоджувальних засобів. Цей метод полягає у використанні різних технік, щоб запобігти запуску

налагоджувальних засобів, таких як підміна функцій, які викликають налагодження, і перехоплення сигналів налагоджувальних засобів;

3. Обфускація коду. Цей метод передбачає застосування різних технік, таких як перейменування змінних та функцій, вставки непотрібного коду та розбиття функцій на менші фрагменти, щоб ускладнити розуміння та аналіз коду;
4. Застосування захисту від підміни коду. Цей метод полягає у використанні різних технік для захисту коду від підміни, таких як захист від модифікації пам'яті, контроль цілісності коду та використання антивірусних програм.

Загалом, методи та техніки протидії налагодженню використовуються для підвищення безпеки програмного коду та зниження ризику виявлення вразливостей. Вибір оптимального методу залежить від конкретного випадку та потреб програми.

IsDebuggerPresent.

Мабуть, найпростішим методом протидії налагодженню є виклик функції IsDebuggerPresent. Ця функція визначає, чи процес виклику налагоджується налагоджувачем режиму користувача. Код нижче показує приклад елементарного захисту:

```
int main()
{
    if (IsDebuggerPresent())
    {
        std::cout << "Stop debugging program!" << std::endl;
        exit(-1);
    }
    return 0;
}
```

Рисунок 1.20 – Приклад коду IsDebuggerPresent

Якщо ми подивимось асемблерний вигляд функції IsDebuggerPresent, то побачимо для x86:

```
0:000< u kernelbase!IsDebuggerPresent L3
KERNELBASE!IsDebuggerPresent:
751ca8d0 64a130000000    mov     eax,dword ptr fs:[00000030h]
751ca8d6 0fb64002        movzx   eax,byte ptr [eax+2]
751ca8da c3              ret
```

Рисунок 1.21 – Дизасемблювання IsDebuggerPresent

Для x64:

```
0:000< u kernelbase!IsDebuggerPresent L3
KERNELBASE!IsDebuggerPresent:
00007ffc`ab6c1aa0 65488b042560000000 mov   rax,qword ptr gs:[60h]
00007ffc`ab6c1aa9 0fb64002           movzx eax,byte ptr [rax+2]
00007ffc`ab6c1aad c3                 ret
```

Рисунок 1.22 – Дизасемблювання IsDebuggerPresent x64

Тобто ми бачимо що функція використовує PEB (Process Environment Block) та зчитує поле BeingDebugged.

```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  PVOID                         Reserved4[3];
  PVOID                         AtlThunkSListPtr;
  PVOID                         Reserved5;
  ULONG                         Reserved6;
  PVOID                         Reserved7;
  ULONG                         Reserved8;
  ULONG                         AtlThunkSListPtr32;
  PVOID                         Reserved9[45];
  BYTE                          Reserved10[96];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved11[128];
  PVOID                         Reserved12[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

Рисунок 1.23 – Структура PEB

Щоб обійти цю перевірку, достатньо помістити 0 у поле BeingDebugged, тобто зробити зворотню операцію функції IsDebuggerPresent.

Для x86:

```
mov eax, dword ptr fs:[0x30]
mov byte ptr ds:[eax+2], 0
```

Рисунок 1.24 – Патчинг IsDebuggerPresent

Для x64:

```
DWORD64 dwpeb = __readgsqword(0x60);
*((PBYTE)(dwpeb + 2)) = 0;
```

Рисунок 1.25 – Патчинг IsDebuggerPresent x64

Використання функції IsDebuggerPresent є найпростішим механізмом захисту від налаштовування і не є достатнім для захисту програмного забезпечення, адже IsDebuggerPresent є WINAPI функцією, а отже її з легкістю можна перехопити.[10]

TLS Callback.

Перевірка наявності налагоджувача в основній функції — не найкраща ідея, оскільки це перше місце, куди дивиться атакуючий під час налагодження. Перевірки, реалізовані в функції main, можна стерти інструкціями NOP, таким чином знявши захист. Якщо використовується бібліотека CRT, головний потік вже матиме певний стек викликів до передачі керування головній функції. Таким чином, хорошим місцем для виконання перевірки присутності налагоджувача є TLS Callback. Функція зворотного виклику буде викликана перед викликом точки входу виконуваного модуля.

Приклад:

```
#pragma section(".CRT$XLY", long, read)
__declspec(thread) int var = 0xDEADBEEF;
VOID NTAnopPI TlsCallback(PVOID DllHandle, DWORD Reason, VOID Reserved)
{
    var = 0xB15BADB0; // Required for TLS Callback call
    if (IsDebuggerPresent())
    {
        MessageBoxA(NULL, "Stop debugging program!", "Error", MB_OK | MB_ICONERROR);
        TerminateProcess(GetCurrentProcess(), 0xBABEFACE);
    }
}
__declspec(allocate(".CRT$XLY"))PIMAGE_TLS_CALLBACK g_tlsCallback = TlsCallback;
```

Рисунок 1.26 – Приклад TLS Callback

В даному випадку зловмиснику дещо складніше буде зрозуміти як перевіряється наявність налагоджувача, але суть при цьому не змінюється, так як все одно використовується WINAPI, є можливість перехопити цей виклик.

NtGlobalFlag.

У Windows NT існує набір прапорів, які зберігаються в глобальній змінній NtGlobalFlag, яка є спільною для всієї системи. Під час завантаження глобальна системна змінна NtGlobalFlag ініціалізується значенням із розділу системного реєстру:

[HKEY_LOCAL_MACHINE\\SYSTEM\\Current\\ControlSet\\ControlSession\\ \\ManagerGlobalFlag]

Це значення змінної використовується для трасування системи, налагодження та контролю. Прапори змінних не задокументовані, але SDK містить утиліту gflags, яка дозволяє редагувати значення глобального прапора. Структура PEB також включає поле NtGlobalFlag, і його бітова структура не відповідає глобальній системній змінній NtGlobalFlag. Під час налагодження в полі NtGlobalFlag встановлюються такі позначки:

```
FLG_HEAP_ENABLE_TAIL_CHECK (0x10)
FLG_HEAP_ENABLE_FREE_CHECK (0x20)
FLG_HEAP_VALIDATE_PARAMETERS (0x40)
```

Рисунок 1.27 – Перелік флагів HEAPS

Щоб перевірити, чи запущено процес за допомогою налагоджувача, перевірте значення поля NtGlobalFlag у структурі PEB. Це поле розташоване

за зміщенням 0x068 і 0x0bc для систем x32 і x64 відповідно відносно початку структури PEB.[11]

Наступний фрагмент коду є прикладом захисту від налагодження на основі перевірки прапорів NtGlobalFlag:

```
#define FLG_HEAP_ENABLE_TAIL_CHECK   0x10
#define FLG_HEAP_ENABLE_FREE_CHECK   0x20
#define FLG_HEAP_VALIDATE_PARAMETERS 0x40
#define NT_GLOBAL_FLAG_DEBUGGED (FLG_HEAP_ENABLE_TAIL_CHECK | FLG_HEAP_ENABLE_FREE_CHECK | FLG_HEAP_VALIDATE_PARAMETERS)
void CheckNtGlobalFlag()
{
    PVOID pPeb = GetPEB();
    PVOID pPeb64 = GetPEB64();
    DWORD offsetNtGlobalFlag = 0;
#ifdef _WIN64
    offsetNtGlobalFlag = 0xBC;
#else
    offsetNtGlobalFlag = 0x68;
#endif
    DWORD NtGlobalFlag = *(PDWORD)((PBYTE)pPeb + offsetNtGlobalFlag);
    if (NtGlobalFlag & NT_GLOBAL_FLAG_DEBUGGED)
    {
        std::cout << "Stop debugging program!" << std::endl;
        exit(-1);
    }
    if (pPeb64)
    {
        DWORD NtGlobalFlagWow64 = *(PDWORD)((PBYTE)pPeb64 + 0xBC);
        if (NtGlobalFlagWow64 & NT_GLOBAL_FLAG_DEBUGGED)
        {
            std::cout << "Stop debugging program!" << std::endl;
            exit(-1);
        }
    }
}
```

Рисунок 1.28 – Приклад коду який базується на флагах HEAPS

Щоб обійти перевірку NtGlobalFlag, просто виконайте зворотні дії, які виконуються до перевірки; іншими словами, необхідно встановити для поля NtGlobalFlag структури PEB налагодженого процесу значення 0, перш ніж це значення буде перевірено захистом від налагодження.[12][13]

CheckRemoteDebuggerPresent та NtQueryInformationProcess (WINAPI).

На відміну від функції IsDebuggerPresent, CheckRemoteDebuggerPresent перевіряє, чи процес налагоджується іншим паралельним процесом. Ось приклад техніки перевірки налагодження на основі CheckRemoteDebuggerPresent:

```
int main(int argc, char* argv[])
{
    BOOL isDebuggerPresent = FALSE;
    if (CheckRemoteDebuggerPresent(GetCurrentProcess(), &isDebuggerPresent))
    {
        if (isDebuggerPresent)
        {
            std::cout << "Stop debugging program!" << std::endl;
            exit(-1);
        }
    }
    return 0;
}
```

Рисунок 1.29 – Приклад перевірки CheckRemoteDebuggerPresent

Усередині CheckRemoteDebuggerPresent викликається функція NtQueryInformationProcess:

```
0:000> uf kernelbase!CheckRemotedebuggerPresent
KERNELBASE!CheckRemoteDebuggerPresent:
...
75207a24 6a00            push    0
75207a26 6a04            push    4
75207a28 8d45fc          lea     eax,[ebp-4]
75207a2b 50              push    eax
75207a2c 6a07            push    7
75207a2e ff7508          push    dword ptr [ebp+8]
75207a31 ff151c602775    call    dword ptr [KERNELBASE!_imp__NtQueryInformationProcess (7527601c)]
75207a37 85c0            test    eax,eax
75207a39 0f88607e0100    js      KERNELBASE!CheckRemoteDebuggerPresent+0x2b (7521f89f)
...
```

Рисунок 1.30 – Дизасемблювання CheckRemoteDebuggerPresent

Якщо ми подивимося на документацію NtQueryInformationProcess, цей асамблерний лістинг покаже нам, що функції CheckRemoteDebuggerPresent присвоєно значення DebugPort, оскільки значення параметра ProcessInformationClass (друге) дорівнює 7. Наступний приклад коду протидії налагодженню базується на виклику NtQueryInformationProcess:

```
typedef NTSTATUS(NTAPI* pfnNtQueryInformationProcess)(
    _In_      HANDLE           ProcessHandle,
    _In_      UINT             ProcessInformationClass,
    _Out_     PVOID            ProcessInformation,
    _In_      ULONG            ProcessInformationLength,
    _Out_opt_ PULONG           ReturnLength
    );
const UINT ProcessDebugPort = 7;
int main(int argc, char* argv[])
{
    pfnNtQueryInformationProcess NtQueryInformationProcess = NULL;
    NTSTATUS status;
    DWORD isDebuggerPresent = 0;
    HMODULE hNtDll = LoadLibrary(TEXT("ntdll.dll"));

    if (NULL != hNtDll)
    {
        NtQueryInformationProcess = (pfnNtQueryInformationProcess)GetProcAddress(hNtDll, "NtQueryInformationProcess");
        if (NULL != NtQueryInformationProcess)
        {
            status = NtQueryInformationProcess(
                GetCurrentProcess(),
                ProcessDebugPort,
                &isDebuggerPresent,
                sizeof(DWORD),
                NULL);
            if (status == 0x00000000 && isDebuggerPresent != 0)
            {
                std::cout << "Stop debugging program!" << std::endl;
                exit(-1);
            }
        }
    }
    return 0;
}
```

Рисунок 1.31 – Обхід перевірки CheckRemoteDebuggerPresent

Щоб обійти CheckRemoteDebuggerPresent і NTQueryInformationProcess, замініть значення, яке повертає функція NtQueryInformationProcess. Для цього можна використовувати метод перехоплення WINAPI функцій. Щоб налаштувати перехолпення, підгрузіть власну DLL у налагоджений процес і та перехопіть необхідну WINAPI функцію.[14]

Ось приклад використання методу перехоплення:

```
#include <Windows.h>
#include "mhook.h"
typedef NTSTATUS(NTAPI* pfnNtQueryInformationProcess)(
    _In_      HANDLE           ProcessHandle,
    _In_      UINT             ProcessInformationClass,
    _Out_     PVOID            ProcessInformation,
    _In_      ULONG            ProcessInformationLength,
    _Out_opt_ PULONG           ReturnLength
    );
const UINT ProcessDebugPort = 7;
pfnNtQueryInformationProcess g_origNtQueryInformationProcess = NULL;
NTSTATUS NTAPI HookNtQueryInformationProcess(
    _In_      HANDLE           ProcessHandle,
    _In_      UINT             ProcessInformationClass,
    _Out_     PVOID            ProcessInformation,
    _In_      ULONG            ProcessInformationLength,
    _Out_opt_ PULONG           ReturnLength
)
{
    NTSTATUS status = g_origNtQueryInformationProcess(
        ProcessHandle,
        ProcessInformationClass,
        ProcessInformation,
        ProcessInformationLength,
        ReturnLength);
    if (status == 0x00000000 && ProcessInformationClass == ProcessDebugPort)
    {
        *((PDWORD_PTR)ProcessInformation) = 0;
    }
    return status;
}
```

Рисунок 1.32 – Обхід перевірки CheckRemoteDebuggerPresent

```
DWORD SetupHook(PVOID pvContext)
{
    HMODULE hNtDll = LoadLibrary(TEXT("ntdll.dll"));
    if (NULL != hNtDll)
    {
        g_origNtQueryInformationProcess = (pfnNtQueryInformationProcess)GetProcAddress(hNtDll, "NtQueryInformationProcess");
        if (NULL != g_origNtQueryInformationProcess)
        {
            Mhook_SetHook((PVOID*)&g_origNtQueryInformationProcess, HookNtQueryInformationProcess);
        }
    }
    return 0;
}
BOOL WINAPI DllMain(HINSTANCE hInstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
    switch (fdwReason)
    {
    case DLL_PROCESS_ATTACH:
        DisableThreadLibraryCalls(hInstDLL);
        CreateThread(NULL, NULL, (LPTHREAD_START_ROUTINE)SetupHook, NULL, NULL, NULL);
        Sleep(20);
    case DLL_PROCESS_DETACH:
        if (NULL != g_origNtQueryInformationProcess)
        {
            Mhook_Unhook((PVOID*)&g_origNtQueryInformationProcess);
        }
        break;
    }
    return TRUE;
}
```

Рисунок 1.33 – Обхід перевірки CheckRemoteDebuggerPresent

Насправді сам метод з використанням WINAPI функції NTQueryInformationProcess є дієвим, оскільки ця функція звертається до ядра задля перевірки, а це означає якщо емулювати дану функцію, замість прямого виклику WINAPI можна отримати достатньо дієвий спосіб детектування налагодження, що змусить атакуючого достатньо довго аналізувати код, щоб зрозуміти що відбувається, а якщо до цього ще застосувати алгоритми заплутування, мутації та віртуалізації – це дасть змогу дуже сильно заплутати програмний код, що унеможливить подальший аналіз.[15]

Точки зупину.

Точки зупину є основним інструментом, що надається налагоджувачам. Точки зупинки дозволяють переривати виконання програми в заданому місці.

Існує два типи контрольних точок:

1. Програмні точки зупину (Software breakpoints)
2. Апаратні точки зупину (Hardware breakpoints)

Для програмних точок зупину існує спеціальна інструкція int 3h із кодом операції 0xCC – яка використовується для виклику дескриптора налагодження. Коли центральний процесор виконує цю інструкцію, генерується переривання, і керування передається налагоджувачу. Щоб отримати контроль, налагоджувач повинен додати в код інструкцію int 3h. Щоб виявити точку зупину, ми можемо обчислити контрольну суму функції.

Ось приклад:

```
DWORD CalcFuncCrc(PUCHAR funcBegin, PUCHAR funcEnd)
{
    DWORD crc = 0;
    for (; funcBegin < funcEnd; ++funcBegin)
    {
        crc += *funcBegin;
    }
    return crc;
}
#pragma auto_inline(off)
VOID DebuggeeFunction()
{
    int calc = 0;
    calc += 2;
    calc <<= 8;
    calc -= 3;
}
VOID DebuggeeFunctionEnd()
{
};
#pragma auto_inline(on)
DWORD g_origCrc = 0x2bd0;
int main()
{
    DWORD crc = CalcFuncCrc((PUCHAR)DebuggeeFunction, (PUCHAR)DebuggeeFunctionEnd);
    if (g_origCrc != crc)
    {
        std::cout << "Stop debugging program!" << std::endl;
        exit(-1);
    }
    return 0;
}
```

Рисунок 1.34 – Приклад перевірки CRC

Глобальна змінна g_origCrc містить crc, уже обчислений функцією CalcFuncCrc. Щоб виявити кінець функції, ми використовуємо прийом функції-заглушки. Оскільки код функції розташований послідовно, кінець DebuggeeFunction є початком функції DebuggeeFunctionEnd. Ми також використали директиву #pragma auto_inline(off), щоб запобігти компілятору робити функції вбудованими.[16]

Немає універсального підходу для обходу перевірки програмної точки зупину. Щоб обійти цей захист, вам слід знайти код, що обчислює контрольну суму, і замінити повернуте значення константою, а також значення всіх змінних, що зберігають контрольні суми функції.

Щодо апаратних точок зупину в архітектурі x86 існує набір регістрів налагодження, які використовуються розробниками під час перевірки та налагодження коду. Ці регістри дозволяють переривати виконання програми та передавати керування налагоджувачу під час звернення до пам’яті для читання чи запису. Регістри налагодження є привілейованим ресурсом і можуть використовуватися програмою лише в реальному режимі або безпечному режимі з рівнем привілеїв CPL=0.

Існує вісім регістрів налагодження DR0-DR7:

1. DR0-DR3 – регістри точки зупину;
2. DR4-DR5 – зарезервовані;
3. DR6 – стан налагодження;
4. DR7 – контроль налагодження.

DR0-DR3 містять лінійні адреси точок зупину. Порівняння цих адрес виконується перед фізичною трансляцією адреси. Кожна з цих точок зупину окремо описана в регістрі DR7. Регістр DR6 вказує, яка точка зупину активована. DR7 визначає режим активації точки зупину за режимом доступу: читання, запис або виконання.

Ось приклад перевірки апаратної точки зупину:

```
CONTEXT ctx = {};
ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
if (GetThreadContext(GetCurrentThread(), &ctx))
{
    if (ctx.Dr0 != 0 || ctx.Dr1 != 0 || ctx.Dr2 != 0 || ctx.Dr3 != 0)
    {
        std::cout << "Stop debugging program!" << std::endl;
        exit(-1);
    }
}
```

Рисунок 1.35 – Приклад перевірки апаратних точок зупину

Також можна скинути апаратні точки зупину за допомогою функції SetThreadContext.

Ось приклад апаратного скидання точки зупину:

```
CONTEXT ctx = {};
ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
SetThreadContext(GetCurrentThread(), &ctx);
```

Рисунок 1.36 – Обхід перевірки апаратних точок зупину

Якщо ми заглянемо всередину функції GetThreadContext, то побачимо, що вона викликає функцію NtGetContextThread:

```
0:000> u KERNELBASE!GetThreadContext L6
KERNELBASE!GetThreadContext:
7538d580 8bff            mov     edi,edi
7538d582 55              push    ebp
7538d583 8bec            mov     ebp,esp
7538d585 ff750c          push    dword ptr [ebp+0Ch]
7538d588 ff7508          push    dword ptr [ebp+8]
7538d58b ff1504683975    call    dword ptr [KERNELBASE!_imp__NtGetContextThread (75396804)]
```

Рисунок 1.37 – Дизасемблювання апаратних точок зупину

Щоб скинути значення в Dr0-Dr7, необхідно скинути прапор CONTEXT_DEBUG_REGISTERS у полі ContextFlags структури CONTEXT, а потім відновити його значення після вихідного виклику функції NtGetContextThread. Що стосується функції SetThreadContext, вона викликає NtSetContextThread.[17]

У наступному прикладі показано, як обійти апаратну перевірку точки зупину та скинути її:

```
typedef NTSTATUS(NTAPI* pfnNtGetContextThread)(
    _In_  HANDLE              ThreadHandle,
    _Out_ PCONTEXT            pContext
    );
typedef NTSTATUS(NTAPI* pfnNtSetContextThread)(
    _In_ HANDLE               ThreadHandle,
    _In_ PCONTEXT             pContext
    );
pfnNtGetContextThread g_origNtGetContextThread = NULL;
pfnNtSetContextThread g_origNtSetContextThread = NULL;
NTSTATUS NTAPI HookNtGetContextThread(
    _In_  HANDLE               ThreadHandle,
    _Out_ PCONTEXT             pContext)
{
    DWORD backupContextFlags = pContext->ContextFlags;
    pContext->ContextFlags &= ~CONTEXT_DEBUG_REGISTERS;
    NTSTATUS status = g_origNtGetContextThread(ThreadHandle, pContext);
    pContext->ContextFlags = backupContextFlags;
    return status;
}
NTSTATUS NTAPI HookNtSetContextThread(
    _In_ HANDLE               ThreadHandle,
    _In_ PCONTEXT             pContext)
{
    DWORD backupContextFlags = pContext->ContextFlags;
    pContext->ContextFlags &= ~CONTEXT_DEBUG_REGISTERS;
    NTSTATUS status = g_origNtSetContextThread(ThreadHandle, pContext);
    pContext->ContextFlags = backupContextFlags;
    return status;
}
void HookThreadContext()
{
    HMODULE hNtDll = LoadLibrary(TEXT("ntdll.dll"));
    g_origNtGetContextThread = (pfnNtGetContextThread)GetProcAddress(hNtDll, "NtGetContextThread");
    g_origNtSetContextThread = (pfnNtSetContextThread)GetProcAddress(hNtDll, "NtSetContextThread");
    Mhook_SetHook((PVOID*)&g_origNtGetContextThread, HookNtGetContextThread);
    Mhook_SetHook((PVOID*)&g_origNtSetContextThread, HookNtSetContextThread);
}
```

Рисунок 1.38 – Обхід апаратних точок зупину

У сучасному світі захист від зловмисних атак на програмне забезпечення є важливою проблемою. Для запобігання таким атакам, розробники ПЗ використовують різноманітні методи та техніки обфускації, компресії та мутації коду, а також методи протидії налагодженню. Методи та техніки протидії налагодженню включають в себе захист від налагодження, захист від відладки, захист від перехоплення сигналу та захист від зчитування пам'яті. Кожен з цих методів та технік забезпечує захист від певних видів атак та може бути ефективним в різних ситуаціях. Алгоритми обфускації, компресії та мутації коду дозволяють розробникам зменшити ризики використання програмного забезпечення зловмисниками. Кожен з цих алгоритмів має свої плюси та мінуси, і їх вибір залежить від конкретної задачі та вимог до програмного забезпечення. У загальному, методи та

техніки протидії налагодженню, а також алгоритми обфускації, компресії та мутації коду є важливими інструментами для захисту від зловмисних атак на програмне забезпечення. Комбінація цих методів та технік може допомогти забезпечити більш високий рівень безпеки та захисту від зловмисних атак.[18]

1.6 Порівняння технік та методів за критеріями

Для порівнянна технік та методів протидії зворотної розробки були обрані дані критерії, оцінка була сформована за попередніми результатами досліджень та статей попередників[19] [1] [2] [3] [4] [5]:

1. Складність реалізації;
2. Ефективність захисту;
3. Вплив на продуктивність;
4. Рівень розуміння;
5. Можливість обходу;
6. Витрати часу;
7. Підтримка та оновлення;
8. Складність використання;
9. Ефективність на великих проектах.

Оцінка може мати одне із значень: Висока, середня, низька.

Таблиця 1.1 – Порівняння методів та технік за критерієм

Критерії	Алгоритми заплутування	Алгоритми мутації	Алгоритми віртуалізації
Складність реалізації	Висока	Середня	Висока
Ефективність захисту	Висока	Середня	Висока

Вплив на продуктивність	Середній	Низький	Висока
Рівень розуміння	Високий	Середній	Низький
Можливість обходу	Низька	Середня	Низька
Витрати часу	Високі	Середні	Високі
Підтримка та оновлення	Висока	Низька	Висока
Складність використання	Низька	Низька	Низька
Ефективність на великих проектах	Висока	Середня	Висока

Критерії	Алгоритми компресії	Методи та техніки протидії налагодженню	Техніка емуляції викликів WinAPI
Складність реалізації	Низька	Висока	Висока
Ефективність захисту	Низька	Висока	Висока
Вплив на продуктивність	Низький	Низька	Низька
Рівень розуміння	Низький	Високий	Високий
Можливість обходу	Висока	Середня	Низька
Витрати часу	Низька	Високі	Високі
Підтримка та оновлення	Висока	Висока	Середня
Складність використання	Низька	Низька	Середня
Ефективність на великих проектах	Висока	Середня	Висока

Як можна побачити найкращим та найдоцільнішим серед алгоритмів буде використання алгоритмів віртуалізації, однак потрібно пам’ятати що використання тільки одного методу захисту недостатнє у сьогодняшніх реаліях, потрібно також подумати та запровадити захист від налагоджувача,

застосувати компресію коду та інше. Щодо захисту від налагоджувача існує головна проблема – пряме використання викликів WinAPI для перевірки налагоджувача. Для вирішення цієї проблеми було запропоновано розробити та імплементувати емуляцію викликів WinAPI.

Висновки до розділу 1

У першому розділі дипломної роботи було проаналізовано різноманітні методи та техніки протидії зворотній розробці програмного забезпечення. Було описано та проаналізовано такі методи, як обфускація коду, віртуалізація коду, компресія коду, мутація коду та протидія налагодженню.

Кожен з цих методів має свої переваги та недоліки залежно від того, які критерії є важливими для конкретної ситуації. Наприклад, методи обфускації та віртуалізації коду мають високу ефективність захисту від зворотної розробки, але можуть знизити швидкодію програми. З іншого боку, методи компресії та мутації коду можуть допомогти зменшити розмір програми та зберегти її функціональність, але не є надійними для захисту від зворотної розробки.

Техніка емуляції викликів WinAPI була включена до таблиці порівняння з іншими методами та техніками протидії зворотній розробці. За цими критеріями, техніка емуляції викликів WinAPI має дуже позитивний вплив на ефективність захисту та дуже низький вплив на швидкодію програми.

Отже, вибір конкретного методу чи техніки протидії залежить від потреб захисту конкретної програми та компромісів між ефективністю захисту та продуктивністю програми. Для досягнення найкращого результату рекомендується комбінувати різні методи та техніки протидії зворотній розробці.

2 МОДЕЛЬ ЕМУЛЯЦІЇ ВИКЛИКІВ WINAPI

У цьому розділі буде розглянуто теоретичні відомості про WinAPI, PEB та TEB, перехід виклику WinAPI з user-mode до kernel-mode, перехоплення викликів WinAPI на прикладі бібліотеки MinHook. Також буде запропонована архітектура бібліотеки для емуляції викликів WinAPI та вирішення проблем, які виникли при аналізі.

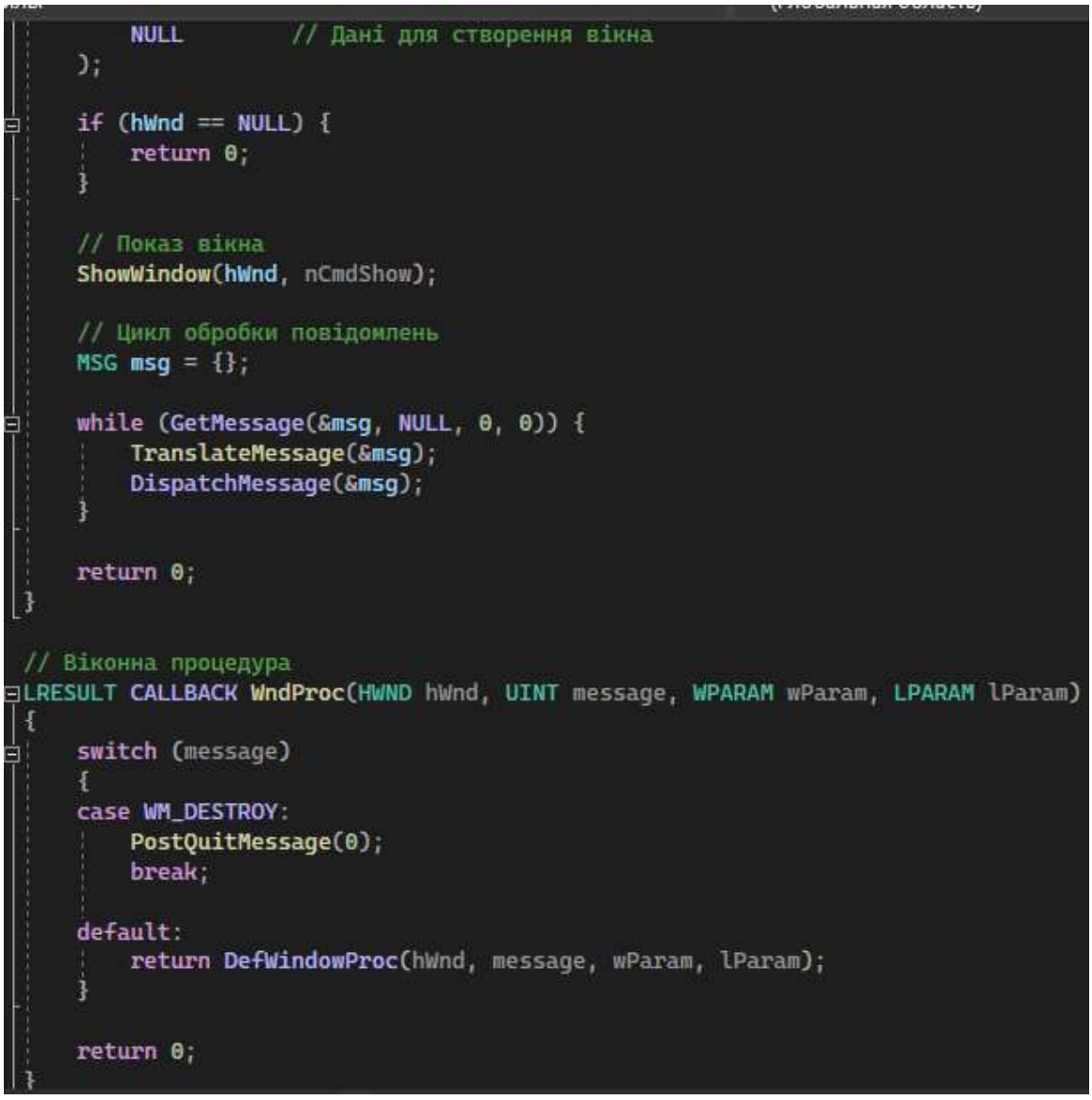
2.1 WinAPI

WinAPI - це набір функцій та інструментів програмування для операційної системи Windows. Він включає в себе набір готових функцій, які розробники можуть використовувати для розробки програм для Windows. WinAPI містить багато різних функцій, які дають можливість працювати з різними складовими операційної системи, такими як вікна, файлова система, мережа, процеси, потоки і багато іншого. Функції WinAPI можуть бути викликані з різних мов програмування, таких як C++, C#, Visual Basic та інші. За допомогою WinAPI розробники можуть створювати додатки, які можуть взаємодіяти з операційною системою та з іншими додатками. Також WinAPI дозволяє розробникам звертатися до різних системних ресурсів, таких як бази даних, сервери, пристрої вводу-виводу та інші. WinAPI є дуже потужним інструментом для розробників, але його використання може бути складним і вимагати багато знань. Однак, завдяки WinAPI розробники можуть створювати більш складні програми для операційної системи Windows з більшою функціональністю та можливостями. У програмуванні на мові C/C++, WinAPI можна викликати через бібліотеку Windows.h. Ця бібліотека містить прототипи всіх функцій WinAPI та константи, які використовуються в програмуванні на Windows.[20]

Ось приклад коду на мові C++, де використовуються функції WinAPI для створення вікна та його обробки подій:

```cpp
#include <windows.h>

// Прототип віконної процедури
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

// Головна функція програми
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    // Реєстрація класу вікна
    const wchar_t CLASS_NAME[] = L"Sample Window Class";

    WNDCLASS wc = {};

    wc.lpfnWndProc = WndProc;
    wc.hInstance = hInstance;
    wc.lpszClassName = CLASS_NAME;

    RegisterClass(&wc);

    // Створення вікна
    HWND hWnd = CreateWindowEx(
        0,
        CLASS_NAME,
        L"Sample Window",
        WS_OVERLAPPEDWINDOW,

        // Розташування та розмір вікна
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,

        NULL,       // Дескриптор батьківського вікна
        NULL,       // Дескриптор меню
```

Рисунок 2.1 – Приклад застосунку з використанням WinAPI



```
        NULL        // Дані для створення вікна
    );

    if (hWnd == NULL) {
        return 0;
    }

    // Показ вікна
    ShowWindow(hWnd, nCmdShow);

    // Цикл обробки повідомлень
    MSG msg = {};

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return 0;
}

// Віконна процедура
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }

    return 0;
}
```

Рисунок 2.1 – Приклад застосунку з використанням WinAPI

Цей код визначає функцію вікна WindowProc, яка обробляє повідомлення, що надходять до вікна. Функція WinMain реєструє клас вікна, створює вікно і запускає головний цикл повідомлень для обробки повідомлень, що надходять до вікна. Функція CreateWindowEx створює вікно з використанням заданих параметрів. У цьому прикладі вікно створюється з назвою "Hello, World!" і розміром 800x600 пікселів. Функція ShowWindow відображає вікно на екрані. Функція GetMessage отримує наступне повідомлення з черги повідомлень для вікна, після чого функції TranslateMessage та DispatchMessage

перекладають і обробляють повідомлення. Коли функція GetMessage повертає 0, програма завершується.[21]

WinAPI - це дуже потужний інструмент для розробника, який дозволяє не тільки створити застосунок, а також розробити його захист.

2.2 PEB та TEB

PEB

PEB (Process Environment Block) - це структура даних в операційній системі Windows, яка містить інформацію про процес, що виконується, та його середовище. PEB зберігає важливу інформацію про процес, таку як шлях до виконуваного файлу, список завантажених бібліотек, параметри командного рядка та інші налаштування, необхідні для правильної роботи програми. PEB знаходиться в пам'яті процесу та містить декілька полів, які містять важливу інформацію. Наприклад, поле ProcessParameters містить вказівники на блоки пам'яті, які містять інформацію про параметри командного рядка та середовище процесу. Поле Ldr містить вказівник на структуру, яка містить інформацію про завантажені модулі (бібліотеки) та їх області пам'яті. PEB може бути використаний для отримання інформації про процес та його середовище від іншого процесу. Це може бути корисно для відлагодження або для зловмисних цілей, якщо зловмисник може отримати доступ до PEB іншого процесу. Однак, доступ до PEB вимагає високих привілеїв, тому зазвичай він недоступний для звичайних користувачів.[22]

Ось вміст структури PEB:

```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  PVOID                         Reserved4[3];
  PVOID                         AtlThunkSListPtr;
  PVOID                         Reserved5;
  ULONG                         Reserved6;
  PVOID                         Reserved7;
  ULONG                         Reserved8;
  ULONG                         AtlThunkSListPtr32;
  PVOID                         Reserved9[45];
  BYTE                          Reserved10[96];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved11[128];
  PVOID                         Reserved12[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

Рисунок 2.2 – Структура PEB

Отримати доступ до PEB процесу можна отримати за допомогою зчитування зміщення 0x30 для x86, та 0x60 для x64 від початку TEB.

TEB

TEB (Thread Environment Block) - це блок даних, який містить інформацію про поточний потік в процесі. Кожен потік має свій власний TEB, який знаходиться в області пам'яті, відведеній для потока. TEB містить важливу інформацію про потік, таку як:

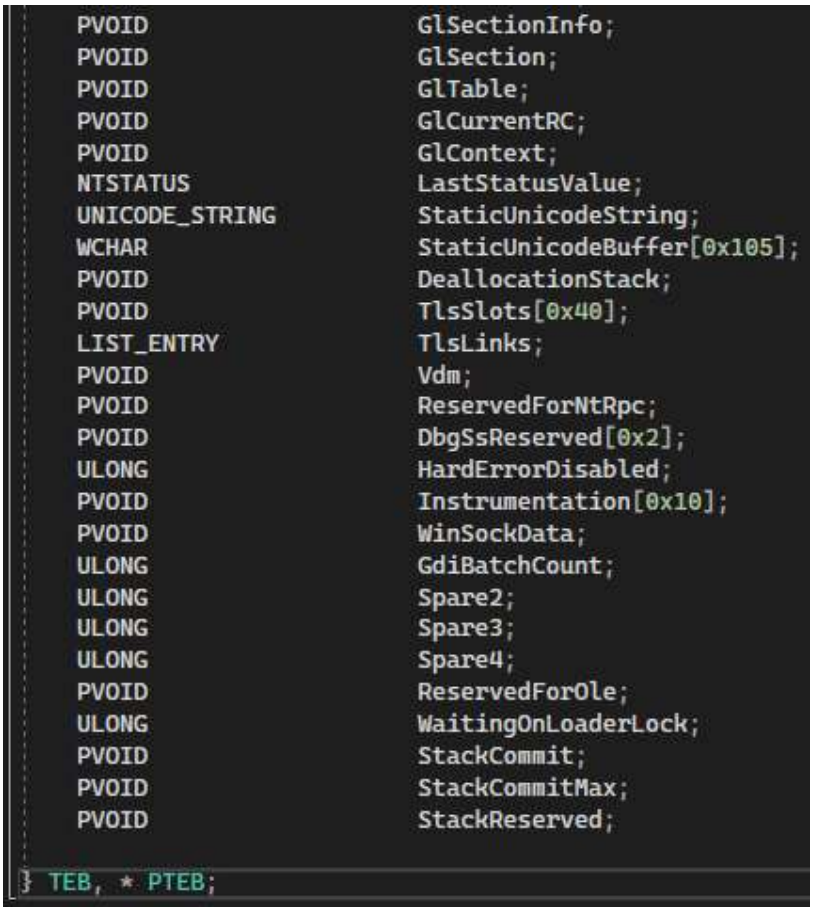
1. Адреса стеку потока;
2. Адреса TEB блоку потока;
3. Адреса блоку PEB процесу;
4. Покажчик на останню оброблену структуру винятків (SEH);
5. Таблиця структур TIB (Thread Information Block) та інші.

TEB також містить важливі поля для обробки винятків та багатьох інших механізмів, що дозволяють потоку працювати з контекстом виконання та системними ресурсами.[23], [24], [25].

Структура TEB має такий вигляд:

```
typedef struct _TEB {

    NT_TIB                  Tib;
    PVOID                   EnvironmentPointer;
    CLIENT_ID               Cid;
    PVOID                   ActiveRpcInfo;
    PVOID                   ThreadLocalStoragePointer;
    PPEB                    Peb;
    ULONG                   LastErrorValue;
    ULONG                   CountOfOwnedCriticalSections;
    PVOID                   CsrClientThread;
    PVOID                   Win32ThreadInfo;
    ULONG                   Win32ClientInfo[0x1F];
    PVOID                   WOW32Reserved;
    ULONG                   CurrentLocale;
    ULONG                   FpSoftwareStatusRegister;
    PVOID                   SystemReserved1[0x36];
    PVOID                   Spare1;
    ULONG                   ExceptionCode;
    ULONG                   SpareBytes1[0x28];
    PVOID                   SystemReserved2[0xA];
    ULONG                   GdiRgn;
    ULONG                   GdiPen;
    ULONG                   GdiBrush;
    CLIENT_ID               RealClientId;
    PVOID                   GdiCachedProcessHandle;
    ULONG                   GdiClientPID;
    ULONG                   GdiClientTID;
    PVOID                   GdiThreadLocaleInfo;
    PVOID                   UserReserved[5];
    PVOID                   GlDispatchTable[0x118];
    ULONG                   GlReserved1[0x1A];
    PVOID                   GlReserved2;
    PVOID                   GlSectionInfo;
```

Рисунок 2.3 – Структура TEB



```
    PVOID                   GlSectionInfo;
    PVOID                   GlSection;
    PVOID                   GlTable;
    PVOID                   GlCurrentRC;
    PVOID                   GlContext;
    NTSTATUS                LastStatusValue;
    UNICODE_STRING          StaticUnicodeString;
    WCHAR                   StaticUnicodeBuffer[0x105];
    PVOID                   DeallocationStack;
    PVOID                   TlsSlots[0x40];
    LIST_ENTRY              TlsLinks;
    PVOID                   Vdm;
    PVOID                   ReservedForNtRpc;
    PVOID                   DbgSsReserved[0x2];
    ULONG                   HardErrorDisabled;
    PVOID                   Instrumentation[0x10];
    PVOID                   WinSockData;
    ULONG                   GdiBatchCount;
    ULONG                   Spare2;
    ULONG                   Spare3;
    ULONG                   Spare4;
    PVOID                   ReservedForOle;
    ULONG                   WaitingOnLoaderLock;
    PVOID                   StackCommit;
    PVOID                   StackCommitMax;
    PVOID                   StackReserved;

} TEB, * PTEB;
```

Рисунок 2.4 – Структура TEB

За допомогою структури TEB також можна отримати адрес PEB та інше.

2.3 Перехоплення WinAPI

Перехоплення WinAPI - це процес перехоплення викликів функцій WinAPI, що виконуються у процесі. Цей процес може бути використаний для різноманітних цілей, таких як моніторинг системи, введення фільтрів для певних функцій, підмена функцій і т.д. Один з найбільш поширених способів перехоплення WinAPI - використання функції detours бібліотеки Microsoft Detours або ж бібліотеки MinHook. Ці бібліотеки надають функціональність для перехоплення функцій, які використовуються в програмі, і зміни їх поведінки. За допомогою перехоплення WinAPI можна відслідковувати виклики певних функцій, збирати статистику або замінювати функції на власні, що дозволяє робити додаткову обробку даних, отриманих з цих функцій. Проте, використання перехоплення WinAPI може мати й свої

недоліки, такі як погіршення продуктивності програми через додаткові операції перехоплення і обробки даних. Також, це може створювати додаткові точки вразливості у програмі, що можуть бути використані для злому. Тому, при використанні перехоплення WinAPI необхідно ретельно вивчати всі можливі наслідки і використовувати цей метод обережно та з обґрунтованими причинами.

Detours та MinHook є двома популярними бібліотеками для перехоплення функцій в Windows. Обидві бібліотеки мають подібний функціонал, а саме, дозволяють перехоплювати функції, що імпортуються з інших DLL-файлів, та замінювати їх на власні реалізації. Одним з головних відмінностей між Detours та MinHook є ліцензування. Detours є комерційним продуктом, а MinHook є відкритою бібліотекою з вільною ліцензією MIT. Щодо функціональності, Detours має більше можливостей, підтримка перехоплення функцій віртуальних таблиць функцій та інші. Однак, ці можливості можуть бути зайвими для більшості проектів. MinHook є більш простою та легшою в використанні бібліотекою, що може бути важливим фактором для швидкої розробки та впровадження проектів. Вона також має менший розмір, що може бути важливим для проектів з обмеженим обсягом.

Узагальнюючи, обидві бібліотеки є потужними інструментами для перехоплення функцій в Windows і мають свої переваги та недоліки. Вибір між Detours та MinHook залежить від конкретних вимог проекту, умінь розробників та ліцензійних умов.

Розглянемо процес перехоплення функції MessageBox з використанням MinHook.

Для перехоплення будь-якої функції WinAPI необхідно і достатньо знати три речі:

1. Назву функції;
2. Назву модуля, де знаходиться ця функція;
3. Адресу функції у необхідному процесі.

Для отримання першого та другого пунктів достатньо звернутися до Windows MSDN та знайти функцію MessageBox:

Syntax

C++ Copy

```
int MessageBox(
  [in, optional] HWND    hWnd,
  [in, optional] LPCTSTR lpText,
  [in, optional] LPCTSTR lpCaption,
  [in]           UINT    uType
);
```

Рисунок 2.5 – Опис функції MessageBox

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-dialogbox-l1-1-0 (introduced in Windows 8)

Рисунок 2.6 – Опис функції MessageBox

Як можна побачити, необхідна функція знаходиться у User32.dll.

Для отримання адреси функції достатньо використати такі WinAPI функції: GetProcAddress та GetModuleHandle.

Syntax

C++ Copy

```
FARPROC GetProcAddress(
  [in] HMODULE hModule,
  [in] LPCSTR  lpProcName
);
```

Рисунок 2.7 – Опис функції GetProcessAddress

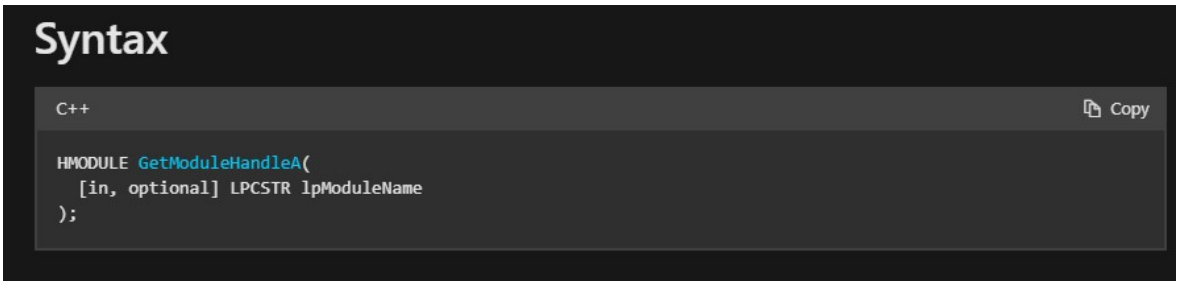


Рисунок 2.8 – Опис функції GetModuleHandleA

Перший аргумент функції GetProcAddress - це адреса необхідної DLL, другий аргумент – це назва шуканої функції, тобто отримаємо такий виклик – GetProcAddress(GetModuleHandleA(“user32.dll”), “MessageBoxA”). Результатом такого виклику буде адреса необхідної функції. Все що залишилось це використати необхідну функцію в бібліотеці MinHook для її перехоплення.

Так як ми маємо всі необхідні дані можна написати демонстративний код з використанням бібліотеки MinHook:

```cpp
#include <iostream>

#include <Windows.h>

#include "MinHook.h"
#pragma comment (lib, "libMinHook.x86.lib")

typedef int (__stdcall* MyMessageBoxA)(HWND  hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType)
MyMessageBoxA msg_box_addr_org;

int MessageBoxA_hooked(HWND  hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType)
{
    return msg_box_addr_org(hWnd, "I do not Love KPI!", "Error", MB_ICONERROR);
}

int main()
{
    // not hooked
    MessageBoxA(NULL, "I Love KPI!", "Information", MB_ICONINFORMATION);
    //

    // getting msg box address
    auto msg_box_addr = GetProcAddress(GetModuleHandleA("user32.dll"), "MessageBoxA");
    //

    // initializing minhook
    MH_Initialize();
    //

    // hooking
    MH_CreateHook(msg_box_addr, MessageBoxA_hooked, (LPVOID*)&msg_box_addr_org);
    //
```

Рисунок 2.9 – Приклад коду з використанням бібліотеки MinHook

```
    //

    // enabling hooks
    MH_EnableHook(MH_ALL_HOOKS);
    //

    // calling the same function
    MessageBoxA(NULL, "I Love KPI!", "Information", MB_ICONINFORMATION);
    //

    getchar();
    return true;
}
```

Рисунок 2.10 – Приклад коду з використанням бібліотеки MinHook

Скомпілювавши та запустивши отримаємо результат:

Рисунок 2.8 – Результат виконання застосунку з бібліотекою MinHook

При першому виклику функції MessageBoxA викликається оригінальна функція, коли при другому виклику викликається власна функція MessageBoxA яка змінює зовнішній вигляд віконця а також текст. Але як це працює і що саме робить MinHook? Відкривши скомпільований застосунок у відлагоджувачі x64dbg можна побачити результат:

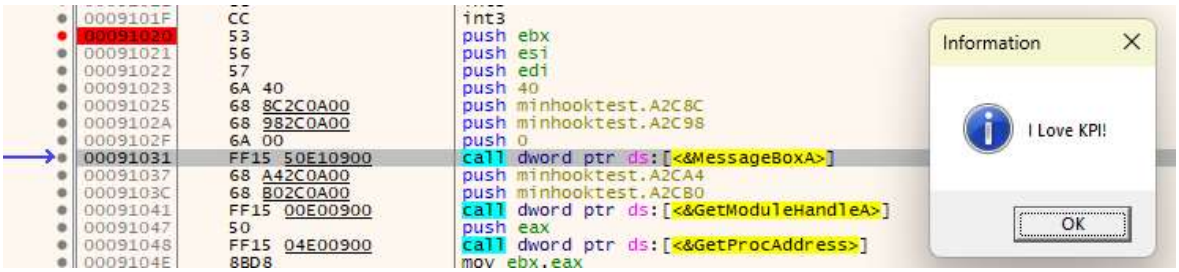


Рисунок 2.11 – Дизасемблювання застосунку з MinHook

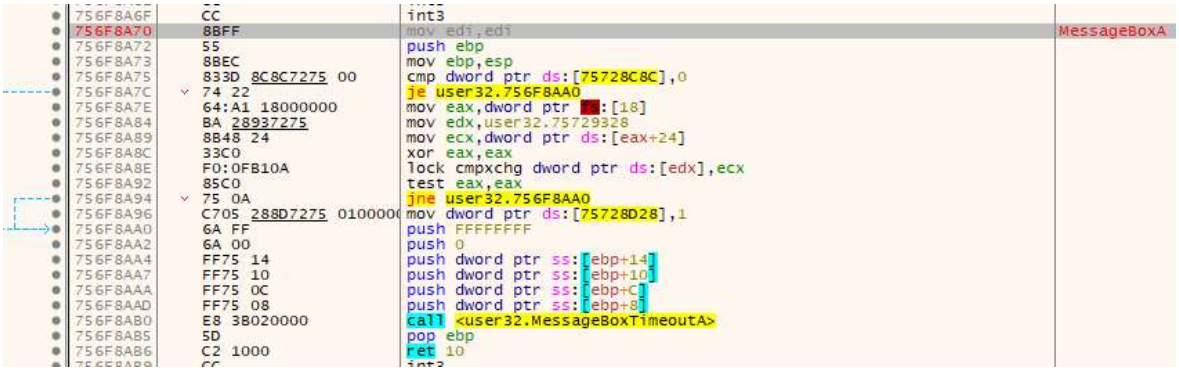


Рисунок 2.12 – Дизасемблювання застосунку з MinHook

При першому виклику функції ми бачимо що MessageBoxA не модифікована.

При другому ж виклику отримаємо результат:

Рисунок 2.13 – Дизасемблювання застосунку з MinHook

Рисунок 2.14 – Дизасемблювання застосунку з MinHook

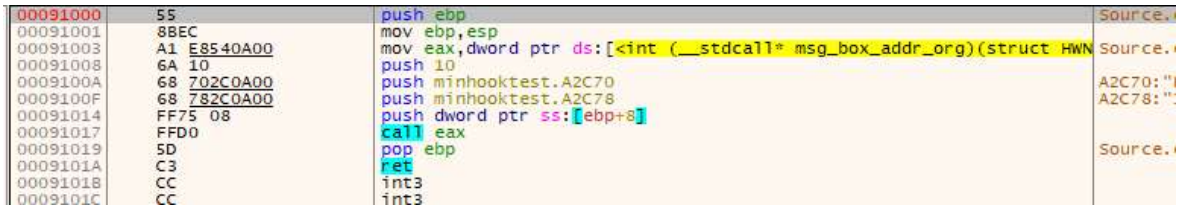


Рисунок 2.15 – Дизасемблювання застосунку з MinHook

Як бачимо у початок функції MessageBoxA була записана інструкція jmp MessageBoxA_hooked, тобто все що MinHook робить це перезаписує початок функції, який виконує стрибок до власної функції. Але не все так просто, не

потрібно забувати про те, що не перехоплена функція MessageBoxA мала на своєму початку пролог функції у вигляді інструкції: mov edi, edi; push ebp; mov ebp, esp. Що ж цими інструкціями трапилось? Справа в тому що бібліотека MinHook має у собі інтегрований дизасамблер який допомагає проаналізувати фукнцію для перехоплення та спроектувати коректний трамплін. Простими словами розмір інструкції jmp у нашому випадку дорівнює 5 байт, а це означає що MinHook повинен скопіювати оригінальні 5 байт у свій трамплін і тільки потім викликати орігінальну функцію.

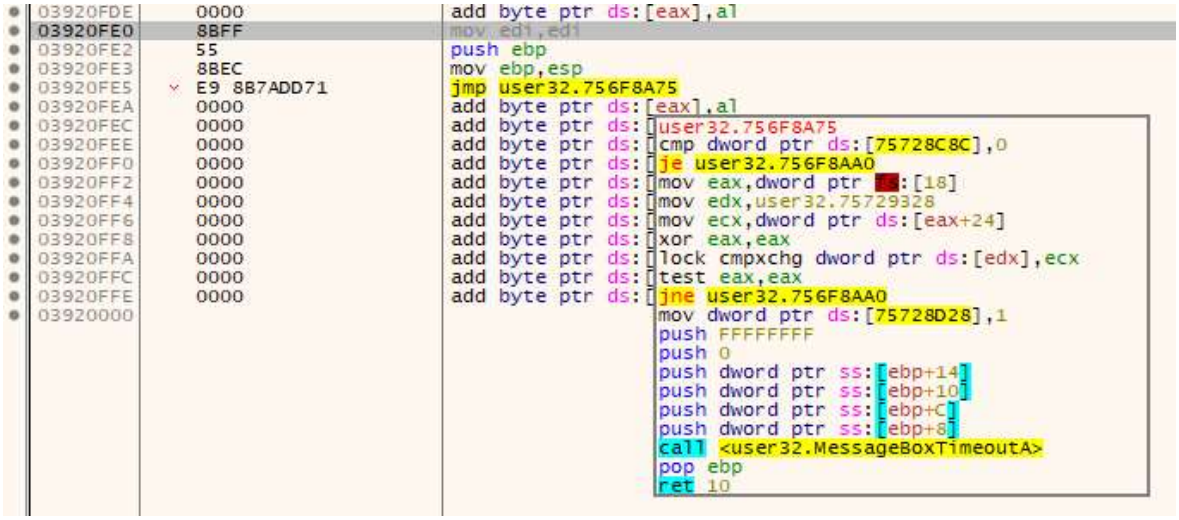


Рисунок 2.16 – Дизасемблювання застосунку з MinHook

Як можна побачити вище, MinHook саме це й зробив, тобто при виклику msg_box_addr_org ми потрапляємо до згенерованого трампліну MinHook, який має у собі оригінальні інструкції MessageBoxA, а потім робимо виклик оригінальної функції MessageBoxA.

Як висновок, використання прямих викликів WinAPI є дуже простою мішенню для зловмисника, так як перехоплення WinAPI не є дуже складним.

2.4 Перехід виклику WinAPI з user-mode до kernel-mode

Для аналізу був написаний код, який викликає WinAPI функцію MessageBoxA, яка створює невелике вікно для інформування користувача.

```cpp
#include <Windows.h>
#include <iostream>

int main()
{
    // calling direct WinAPI function
    MessageBoxA(NULL, "KPI", "I love KPI!", MB_ICONINFORMATION);
    //

    getchar();

    return true;
}
```

Рисунок 2.17 – Приклад коду з використанням функції MessageBoxA

Рисунок 2.18 – Результат виконання

Використовуючи x64dbg було проаналізовано як MessageBoxA переходить з простору користувача до надання сигналу ядру системи Windows: функція MessageBoxA робить стрибок до функції MessageBoxTimeoutA, яка конвертує вхідні дані с ASCII у UNICODE та викликає MessageBoxTimeoutW, потім функція нарешті викликає функцію ZwRaiseHardError з ntdll.dll, яка містить у собі інструкції з використанням syscall для запиту до простору ядра системи Windows.

Рисунок 2.19 – Дизасемблювання застосунку

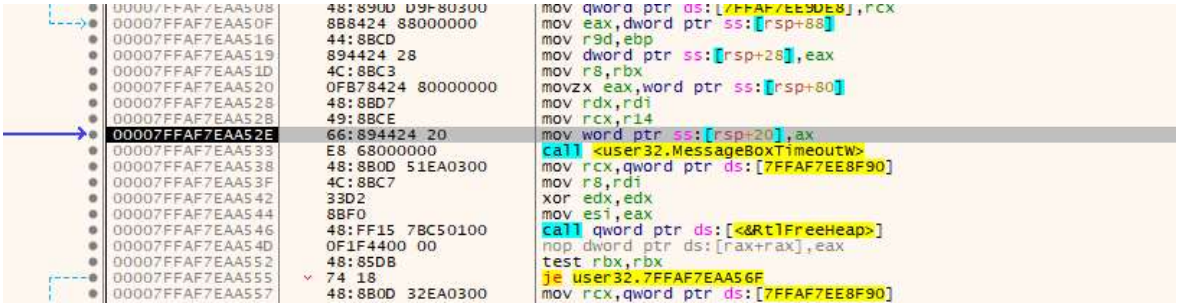


Рисунок 2.20 – Дизасемблювання застосунку

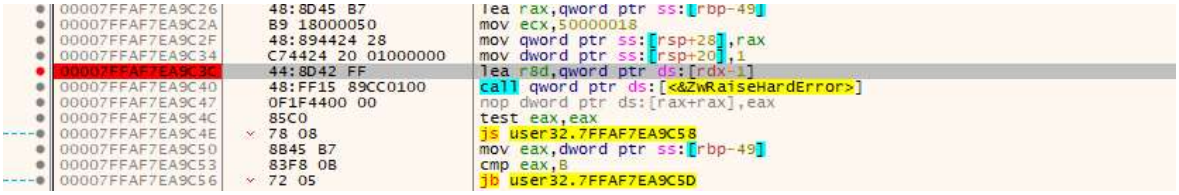


Рисунок 2.21 – Дизасемблювання застосунку

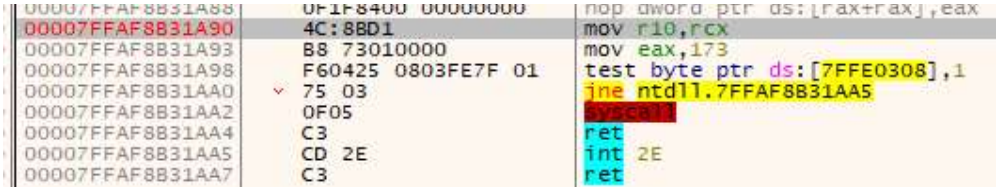


Рисунок 2.22 – Дизасемблювання застосунку

Отже стає зрозуміло що фінальною функцією у цьому ланцюжку є функція ZwRaiseHardError, яка подає сигнал ядру системи використовуючи інструкцію syscall.

2.5 Вирішення проблем емуляції викликів WinAPI

Якщо говорити про емуляцію викликів WinAPI, то з попереднього пункту стає зрозуміло що існують деякі проблеми та нюанси:

1. Індекс функцій ntdll.dll змінюється з кожною версією Windows;
2. Інструкція syscall працює лише на 64-бітній системі та 64-бітному застосунку;

3. Необхідність зробити алокацію початкового буфера, де будуть функції ZwAllocateVirtualMemory та ZwProtectVirtualMemory, які відповідають за алокацію в пам’яті та зміну захисту адрес відповідно.

NtQueueApcThreadEx												0x012e	0x012e	0x012e	0x012e	0x0142	0x0145	0x0142	0x0145	
NtRaiseException	0x00e1	0x00e1	0x00e1	0x00e1	0x00e1	0x00e1	0x012b	0x0126	0x0126	0x0126	0x0126	0x012f	0x012f	0x012f	0x012f	0x0143	0x0146	0x0143	0x0146	
NtRaiseHardError	0x00e2	0x00e2	0x00e2	0x00e2	0x00e2	0x00e2	0x012c	0x0127	0x0127	0x0127	0x0127	0x0130	0x0130	0x0130	0x0130	0x0144	0x0147	0x0144	0x0147	
NtReadFile	0x0003	0x0003	0x0003	0x0003	0x0003	0x0003	0x0003	0x0003	0x0003	0x0003	0x0003	0x0003	0x0003	0x0003	0x0003	0x0004	0x0005	0x0004	0x0005	
NtReadFileScatter	0x002b	0x002b	0x002b	0x002b	0x002b	0x002b	0x002b	0x002b	0x002b	0x002b	0x002b	0x002b	0x002b	0x002b	0x002b	0x002c	0x002d	0x002c	0x002d	
NtReadOnlyEnlistment							0x012d	0x0128	0x0128	0x0128	0x0128	0x0131	0x0131	0x0131	0x0131	0x0145	0x0148	0x0145	0x0148	
NtReadRequestData	0x0051	0x0051	0x0051	0x0051	0x0051	0x0051	0x0051	0x0051	0x0051	0x0051	0x0051	0x0051	0x0051	0x0051	0x0051	0x0052	0x0053	0x0052	0x0053	
NtReadVirtualMemory	0x003c	0x003c	0x003c	0x003c	0x003c	0x003c	0x003c	0x003c	0x003c	0x003c	0x003c	0x003c	0x003c	0x003c	0x003c	0x003d	0x003e	0x003d	0x003e	
NtRecoverEnlistment							0x012e	0x0129	0x0129	0x0129	0x0129	0x0132	0x0132	0x0132	0x0132	0x0146	0x0149	0x0146	0x0149	
NtRecoverResourceManager							0x012f	0x012a	0x012a	0x012a	0x012a	0x0133	0x0133	0x0133	0x0133	0x0147	0x014a	0x0147	0x014a	
NtRecoverTransactionManager							0x0130	0x012b	0x012b	0x012b	0x012b	0x0134	0x0134	0x0134	0x0134	0x0148	0x014b	0x0148	0x014b	
NtRegisterProtocolAddressInformation							0x0131	0x012c	0x012c	0x012c	0x012c	0x0135	0x0135	0x0135	0x0135	0x0149	0x014c	0x0149	0x014c	
NtRegisterThreadTerminatePort	0x00e3	0x00e3	0x00e3	0x00e3	0x00e3	0x00e3	0x0132	0x012d	0x012d	0x012d	0x012d	0x0136	0x0136	0x0136	0x0136	0x014a	0x014d	0x014a	0x014d	
NtReleaseCMFViewOwnership							0x0133	0x012e	0x012e	0x012e	0x012e									
NtReleaseKeyedEvent	0x00e4	0x00e4	0x00e4	0x00e4	0x00e4	0x00e4	0x0134	0x012f	0x012f	0x012f	0x012f	0x0137	0x0137	0x0137	0x0137	0x014b	0x014e	0x014b	0x014e	
NtReleaseMutant	0x001d	0x001d	0x001d	0x001d	0x001d	0x001d	0x001d	0x001d	0x001d	0x001d	0x001d	0x001d	0x001d	0x001d	0x001d	0x001e	0x001f	0x001e	0x001f	
NtReleaseSemaphore	0x0007	0x0007	0x0007	0x0007	0x0007	0x0007	0x0007	0x0007	0x0007	0x0007	0x0007	0x0007	0x0007	0x0007	0x0007	0x0008	0x0009	0x0008	0x0009	
NtReleaseWorkerFactoryWorker							0x0135	0x0130	0x0130	0x0130	0x0130	0x0138	0x0138	0x0138	0x0138	0x014c	0x014f	0x014c	0x014f	
NtRemoveIoCompletion	0x0006	0x0006	0x0006	0x0006	0x0006	0x0006	0x0006	0x0006	0x0006	0x0006	0x0006	0x0006	0x0006	0x0006	0x0006	0x0007	0x0008	0x0007	0x0008	
NtRemoveIoCompletionEx							0x0136	0x0131	0x0131	0x0131	0x0131	0x0139	0x0139	0x0139	0x0139	0x014d	0x0150	0x014d	0x0150	
NtRemoveProcessDebug	0x00e5	0x00e5	0x00e5	0x00e5	0x00e5	0x00e5	0x00e5	0x0137	0x0132	0x0132	0x0132	0x013a	0x013a	0x013a	0x013a	0x014e	0x0151	0x014e	0x0151	
NtRenameKey	0x00e6	0x00e6	0x00e6	0x00e6	0x00e6	0x00e6	0x0138	0x0133	0x0133	0x0133	0x0133	0x013b	0x013b	0x013b	0x013b	0x014f	0x0152	0x014f	0x0152	
NtRenameTransactionManager								0x0134	0x0134	0x0134	0x0134	0x013c	0x013c	0x013c	0x013c	0x0150	0x0153	0x0150	0x0153	
NtReplaceKey	0x00e7	0x00e7	0x00e7	0x00e7	0x00e7	0x00e7	0x0139	0x0135	0x0135	0x0135	0x0135	0x013d	0x013d	0x013d	0x013d	0x0151	0x0154	0x0151	0x0154	
NtReplacePartitionUnit								0x0136	0x0136	0x0136	0x0136	0x013e	0x013e	0x013e	0x013e	0x0152	0x0155	0x0152	0x0155	
NtReplyPort	0x0009	0x0009	0x0009	0x0009	0x0009	0x0009	0x0009	0x0009	0x0009	0x0009	0x0009	0x0009	0x0009	0x0009	0x0009	0x000a	0x000b	0x000a	0x000b	
NtReplyWaitReceivePort	0x0008	0x0008	0x0008	0x0008	0x0008	0x0008	0x0008	0x0008	0x0008	0x0008	0x0008	0x0008	0x0008	0x0008	0x0008	0x0009	0x000a	0x0009	0x000a	
NtReplyWaitReceivePortEx	0x0028	0x0028	0x0028	0x0028	0x0028	0x0028	0x0028	0x0028	0x0028	0x0028	0x0028	0x0028	0x0028	0x0028	0x0028	0x0029	0x002a	0x0029	0x002a	
NtReplyWaitReplyPort	0x00e8	0x00e8	0x00e8	0x00e8	0x00e8	0x00e8	0x013a	0x0137	0x0137	0x0137	0x0137	0x013f	0x013f	0x013f	0x013f	0x0153	0x0156	0x0153	0x0156	
NtRequestDeviceWakeup	0x00e9	0x00e9	0x00e9	0x00e9	0x00e9	0x00e9	0x013b	0x0138	0x0138	0x0086	0x0138									
NtRequestPort	0x00ea	0x00ea	0x00ea	0x00ea	0x00ea	0x00ea	0x013c	0x0139	0x0139	0x0139	0x0139	0x0140	0x0140	0x0140	0x0140	0x0154	0x0157	0x0154	0x0157	

Рисунок 2.23 – Список індексів системних викликів на різних системах

З рисунку вище стає зрозуміло що наприклад для тієї ж функції ZwRaiseHardError індекс для syscall змінюється з кожною версією Windows, а це означає що необхідно придумати універсальний метод по отримуванню індексів необхідних функцій.

Після аналізу декількох функцій в ntdll.dll, було помічено що кожна функція має ідентичну інструкцію яка розміщує необхідний індекс функції у регістр eax: mov eax, syscall_index.

```
00007FFF2C391A88   0F1F8400 00000000   nop dword ptr ds:[rax+rax],eax
00007FFF2C391A90   4C:8BD1             mov r10,rcx                          ZwRaiseHardError
00007FFF2C391A93   B8 73010000         mov eax,173
00007FFF2C391A98   F60425 0803FE7F 01  test byte ptr ds:[7FFE0308],1
00007FFF2C391AA0   75 03               jne ntdll.7FFF2C391AA5
00007FFF2C391AA2   0F05                syscall
00007FFF2C391AA4   C3                  ret
00007FFF2C391AA5   CD 2E               int 2E
00007FFF2C391AA7   C3                  ret
```

Рисунок 2.24 – Дизасемблювання ntdll.dll

```
00007FFF2C38ED40   4C:8BD1             mov r10,rcx                          NtWriteFile
00007FFF2C38ED43   B8 08000000         mov eax,8
00007FFF2C38ED48   F60425 0803FE7F 01  test byte ptr ds:[7FFE0308],1
00007FFF2C38ED50   75 03               jne ntdll.7FFF2C38ED55
00007FFF2C38ED52   0F05                syscall
00007FFF2C38ED54   C3                  ret
00007FFF2C38ED55   CD 2E               int 2E
00007FFF2C38ED57   C3                  ret
```

Рисунок 2.25 – Дизасемблювання ntdll.dll

```
00007FFF2C38F640   4C:8BD1             mov r10,rcx                          ZwProtectVirtualMemory
00007FFF2C38F643   B8 50000000         mov eax,50                           50:'P'
00007FFF2C38F648   F60425 0803FE7F 01  test byte ptr ds:[7FFE0308],1
00007FFF2C38F650   75 03               jne ntdll.7FFF2C38F655
00007FFF2C38F652   0F05                syscall
00007FFF2C38F654   C3                  ret
00007FFF2C38F655   CD 2E               int 2E
00007FFF2C38F657   C3                  ret
```

Рисунок 2.26 – Дизасемблювання ntdll.dll

Було прийнято рішення отримувати адресу ntdll.dll через структуру PEB, аналізувати таблицю експорту даного модуля для отримання адрес необхідних функцій та пошук у даних функціях інструкції mov eax, syscall_index та зчитування syscall_index.

Щодо другої проблеми – необхідно розглянути три випадки:

1. 32-бітний застосунок на 32-бітній системі;
2. 32-бітний застосунок на 64-бітній системі;
3. 64-бітний застосунок на 64-бітній системі.

У першому випадку проаналізувавши як функція WinAPI переходить з простору користувача до простору ядра можна зробити висновок що замість інструкції syscall у 32-бітній системі використовується інструкція sysenter.

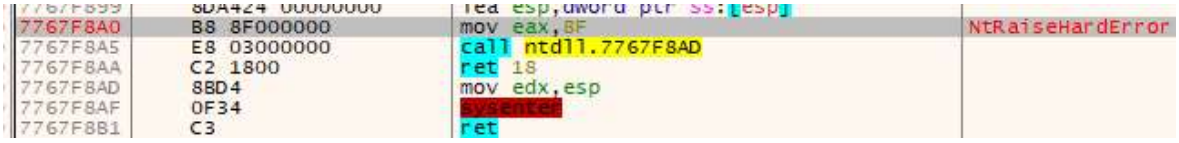


Рисунок 2.27 – Дизасемблювання ntdll.dll x86

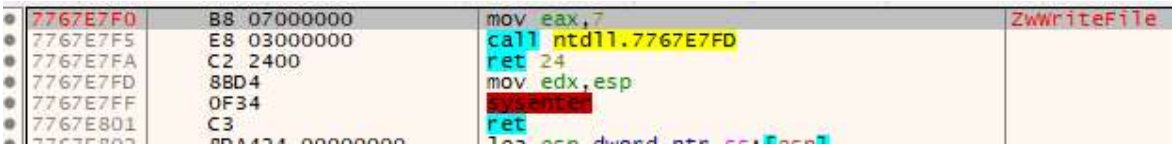


Рисунок 2.28 – Дизасемблювання ntdll.dll x86

У другому випадку ситуація дещо цікавіша – замість використання інструкцій syscall або sysenter використовуються функція Wow64Transition у модулі ntdll.dll.

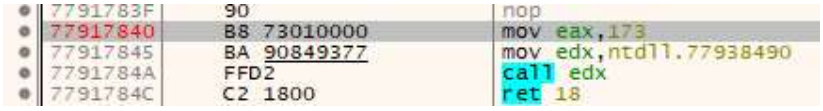


Рисунок 2.29 – Дизасемблювання ntdll.dll x86-x64

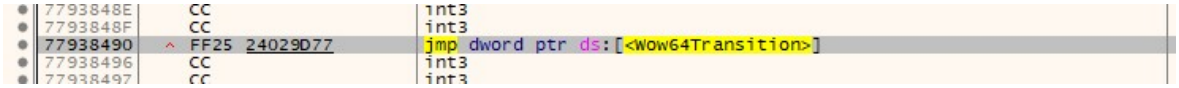


Рисунок 2.30 – Дизасемблювання ntdll.dll x86-x64

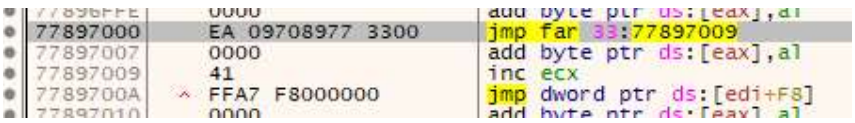


Рисунок 2.31 – Дизасемблювання ntdll.dll x86-x64

Інструкція jmp far 33:xxxxxx змінює режим процесора з 32-бітного у 64-бітний, тобто простими словами здійснюється стрибок у 64-бітний сегмент де викликається 64-бітна інструкція syscall, а потім використовуючи інструкцію ret far 23 повертається до 32-бітного сегменту коду.

Для вирішення цієї проблеми було використано структуру TEB, а саме поле WOW32Reserved яке містить адресу до jmp far 33:xxxxxx.

Задля розрізнення першого та другого випадків був доданий додатковий маркер, який шукає функцію Wow64Transition у модулі ntdll.dll, якщо функція присутня – це означає що це 32-бітний застосунок запущений на 64-бітній системі, у інакшому випадку – 32-бітний застосунок запущений на 32-бітній системі.

У третьому та останньому випадку можна просто використовувати інструкцію syscall, так як вона підтримується 64-бітним процесором та системою.

Фінальну проблему було вирішено використанням функцій malloc та VirtualProtect. Перша створює буфер з правами запису та читання, остання додає право на виконання коду.

Вирішивши всі проблеми, які було виявлено при аналізі емуляції викликів функцій WinAPI можна приступити до розробки архітектури бібліотеки.

2.6 Архітектура бібліотеки для емуляції викликів WinAPI

Ініціалізацію бібліотеки можна поділити на такі етапи:

1. Отримання адреси PEB процесу;
2. Отримання адреси ntdll.dll через PEB;
3. Отримання таблиці експорту ntdll.dll через парсинг PE заголовку;
4. Пошук адрес необхідних функцій використовуючи таблицю експорту;
5. Пошук індексів функцій для системного виклику;

6. Алокація двох початкових буферів для функцій ZwAllocateVirtualMemory (для алокації буферу) та ZwProtectVirtualMemory (для зміни захисту відповідного буферу);
7. Алокація всіх функцій у пам’яті;
8. Використання алокованих функцій.

Візуально бібліотека виглядає таким чином:

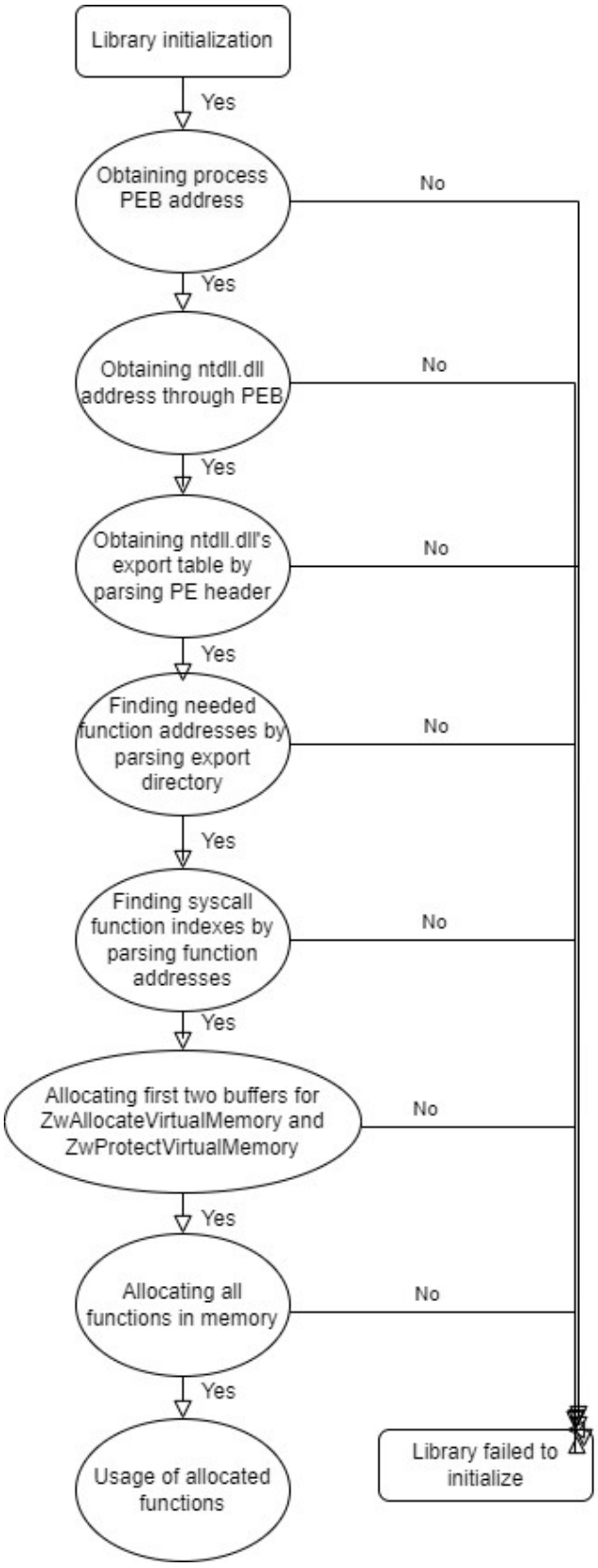


Рисунок 2.32 – Блок-схема бібліотеки для емуляції викликів WinAPI

Висновки до розділу 2

У даному розділі було розглянуто основні аспекти пов'язані з WinAPI та його взаємодією з процесами, а також методи перехоплення WinAPI за допомогою MinHook та перехід виклику WinAPI з user-mode до kernel-mode. Для ефективної роботи з WinAPI важливо розуміти як взаємодіяти з об'єктами ядра, такими як PEB та TEB, для отримання необхідної інформації про процес.

Було розглянуто проблеми емуляції викликів WinAPI, які були пов’язані з постійною зміною індексів функцій від версії до версії Windows, також було розглянуті різні методи системного виклику в залежності від бітності системи та бітності застосунку.

Окрема увага була приділена архітектурі бібліотеки для емуляції викликів WinAPI, яка повинна бути оптимізованою та ефективною. Бібліотека повинна підтримувати широкий спектр функцій WinAPI та забезпечувати швидке та точне відтворення їх поведінки. Також бібліотека повинна бути простою для редагування та оновлення іншими розробниками.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ЕМУЛЯЦІЇ ВИКЛИКІВ WINAPI

У цьому розділі буде розглянута технічна частина бібліотеки та її імплементація, а також будуть протестовані додатки з використанням прямих викликів WinAPI та земулюваних. Буде проаналізована ступінь захищеності на кожному етапі захисту та зроблені висновки.

3.1 Розробка бібліотеки

Як мова програмування було обрано C++, подальша розробка була продовжена у середовищі розробки Visual Studio Community 2022. Проект не потребує сторонніх модулів, все необхідне береться з Windows SDK, який йде у комплекті з інсталяцією Visual Studio Community 2022.

Проект містить у собі такі файли:

1. Source.cpp – файл з описом головної функції, використовується для тестування бібліотеки;
2. AntiDebugLib.h – опис самої бібліотеки;
3. Crc32.h – імплементація алгоритму crc32, яка використовується бібліотекою;
4. ntapi_typedefs.h – опис NT функцій у ntdll.dll та їх аргументів.

Для використання бібліотеки розробнику необхідно скопіювати файли; AntiDebugLib.h, Crc32.h, ntapi_typedefs.h та підключити бібліотеку до проекту за допомоги директиви #include “AntiDebugLib.h”.

Бібліотека вміщає у собі 1 публічну функцію:

1. get_syscall_addr – повертає адресу алокованої функції з використанням системного виклику за індексом.

```
}

inline void* get_syscall_addr(SUPPORTED_FUNCTIONS index)
{
    return syscall_addresses[index];
}
```

Рисунок 3.1 – Опис коду бібліотеки

Бібліотека вміщає у собі 2 приватні функції, які використовуються бібліотекою при ініціалізації:

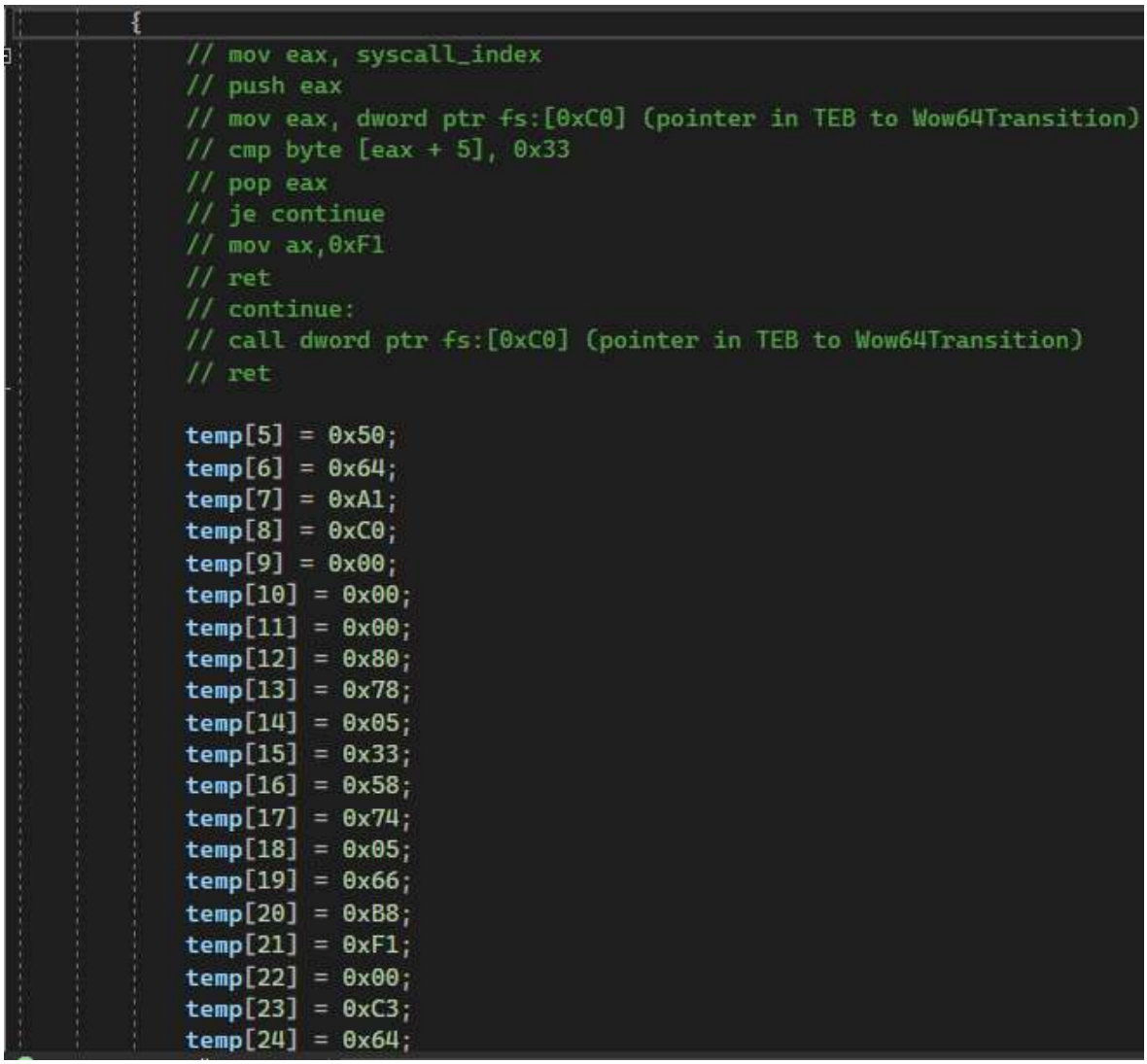
1. allocate_syscall – вхідними параметрами для цієї функції є адреса та індекс функції, функція записує потрібний системний виклик до адреси за індексом функції;
2. find_syscall_index – вхідними параметрами для цієї функції є адреса функції в ntdll.dll та власний індекс функції у типі enum для запису, функція виконує невеличке дизасемблювання за адресою та шукає інструкцію mov eax, syscall_index, потім зчитує syscall_index та записує його у приватний буфер бібліотеки за індексом.

```
inline void find_syscall_index(void* func_ptr, int index)
{
    auto temp = reinterpret_cast<BYTE*>(func_ptr);

    while (*temp != 0xC3)
    {
        if (*temp == 0xB8)
        {
            syscall_table[index] = *reinterpret_cast<WORD*>(++temp);
            break;
        }

        temp++;
    }
}
```

Рисунок 3.2 – Опис коду бібліотеки

```
inline void allocate_syscall(void* buffer_addr, SUPPORTED_FUNCTIONS index)
{
    auto temp = reinterpret_cast<char*>(buffer_addr);
    auto temp_syscall_index = reinterpret_cast<char*>(&syscall_table[index]);

    temp[0] = 0xB8;
    temp[1] = temp_syscall_index[0];
    temp[2] = temp_syscall_index[1];
    temp[3] = 0x0;
    temp[4] = 0x0;

#ifdef _WIN64
    // mov eax, syscall_index
    // mov r10, rcx
    // syscall
    // ret
    temp[5] = 0x4C;
    temp[6] = 0x8B;
    temp[7] = 0xD1;
    temp[8] = 0x0F;
    temp[9] = 0x05;
    temp[10] = 0xC3;
#else
    if (is_wow64)
```

Рисунок 3.3 – Опис коду бібліотеки



```
{
    // mov eax, syscall_index
    // push eax
    // mov eax, dword ptr fs:[0xC0] (pointer in TEB to Wow64Transition)
    // cmp byte [eax + 5], 0x33
    // pop eax
    // je continue
    // mov ax,0xF1
    // ret
    // continue:
    // call dword ptr fs:[0xC0] (pointer in TEB to Wow64Transition)
    // ret

    temp[5] = 0x50;
    temp[6] = 0x64;
    temp[7] = 0xA1;
    temp[8] = 0xC0;
    temp[9] = 0x00;
    temp[10] = 0x00;
    temp[11] = 0x00;
    temp[12] = 0x80;
    temp[13] = 0x78;
    temp[14] = 0x05;
    temp[15] = 0x33;
    temp[16] = 0x58;
    temp[17] = 0x74;
    temp[18] = 0x05;
    temp[19] = 0x66;
    temp[20] = 0xB8;
    temp[21] = 0xF1;
    temp[22] = 0x00;
    temp[23] = 0xC3;
    temp[24] = 0x64;
```

Рисунок 3.4 – Опис коду бібліотеки

```
            temp[24] = 0x64;
            temp[25] = 0xFF;
            temp[26] = 0x15;
            temp[27] = 0xC0;
            temp[28] = 0x00;
            temp[29] = 0x00;
            temp[30] = 0x00;
            temp[31] = 0xC3;
        }
        else
        {
            // mov eax, syscall_index
            // mov ebx, esp
            // sysenter
            // ret
            temp[5] = 0x8B;
            temp[6] = 0xD4;
            temp[7] = 0x0F;
            temp[8] = 0x34;
            temp[9] = 0xC3;
        }
#endif
    }
```

Рисунок 3.5 – Опис коду бібліотеки

```
#pragma once

#include <winternl.h>
#include "ntapi_typedefs.h"
#include "Crc32.h"

#define MAX_SYSCALL_CALL_SIZE 32
#define SUPPORTED_FUNCTIONS_AMOUNT 6
enum SUPPORTED_FUNCTIONS
{
    ZwAllocateVirtualMemory,
    ZwProtectVirtualMemory,
    ZwFreeVirtualMemory,
    ZwRaiseHardError,
    ZwTerminateProcess,
    ZwQueryInformationProcess
};

const int function_hashes[SUPPORTED_FUNCTIONS_AMOUNT] =
{
    HASH("ZwAllocateVirtualMemory"),
    HASH("ZwProtectVirtualMemory"),
    HASH("ZwFreeVirtualMemory"),
    HASH("ZwRaiseHardError"),
    HASH("ZwTerminateProcess"),
    HASH("ZwQueryInformationProcess")
};
const int wow64transition = HASH("Wow64Transition");
```

Рисунок 3.6 – Опис коду бібліотеки

Бібліотека імпортує лише один файл із Windows SDK під назвою winternl.h. MAX_SYSCALL_SIZE – це максимальний розмір в байтах пейлоаду з використанням системного виклику. SUPPORTED_FUNCTIONS_AMOUNT – кількість функцій які підтримуються.

Структура enum SUPPORTED_FUNCTIONS – власний перелік функцій за індексом.

function_hashes – перелік хешів функцій які підтримуються. Варто помітити що порядок функцій повинен бути точно таким як у структурі enum SUPPORTED_FUNCTIONS.

wow64transition – хеш функції Wow64Transition.

Для хешування було імплементовано функції для вирахування crc32 у двух видах:

1. HASH – хеш вираховується при компіляції, таким чином у скомпільованому вигляді ми не побачимо ніяких текстових даних, а лише їх хеші. Це було зроблено задля того щоб зловмисник не зміг так просто знайти які самі функції імплементовані. Розробник може відредагувати калькуляцію crc32, тим самим змінить значення хешів.
2. HASH_RUNTIME – хеш вираховується під час виконання програми. Функція потрібна для того щоб власне вирахувати хеш певних даних під час виконання програми та порівняти зі значеннями які були вираховані за допомогою першого типу, тобто певні константні данні.

Бібліотека також має певні приватні змінні:

```
class AntiDebugLib
{
private:
    bool is_wow64 = false;

    WORD syscall_table[SUPPORTED_FUNCTIONS_AMOUNT];
    void* syscall_addresses[SUPPORTED_FUNCTIONS_AMOUNT];
```

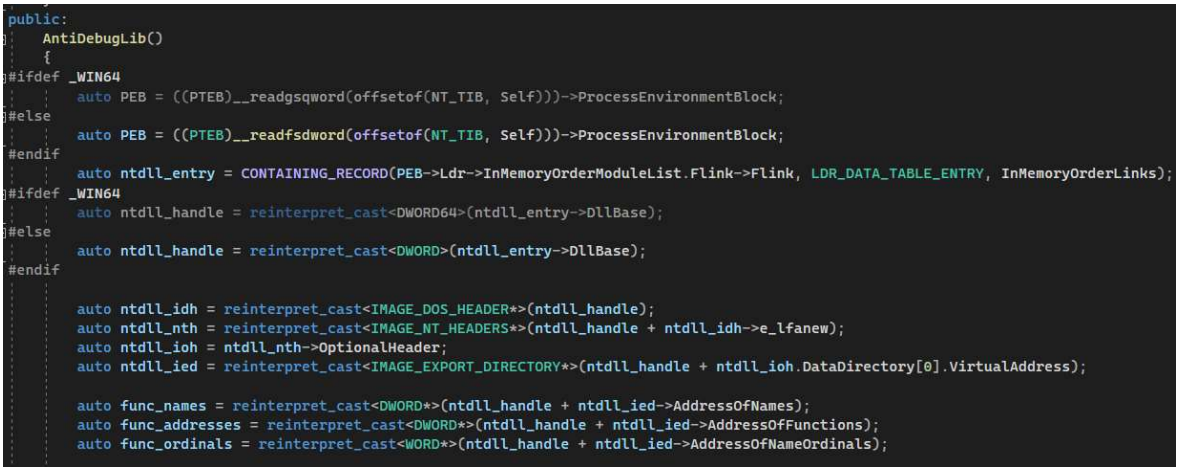
Рисунок 3.7 – Опис коду бібліотеки

is_wow64 – маркер який детектує чи 32-бітний застосунок запущений на 64-бітній системи. False – запущений на 32-бітній системі, True – запущений на 64-бітній системі. Маркер потрібний для функції allocate_syscall, він дозволяє зрозуміти який саме пейлоад треба записувати: чи зі звичайним sysetner (для 32-бітних систем), чи використати Wow64Transition (для 64-бітних систем).

syscall_table – масив розміром SUPPORTED_FUNCTIONS_AMOUNT з індексами відповідних функцій ntdll.dll. Запис до масиву виконується у функції find_syscall_index.

syscall_addresses – масив розміром SUPPORTED_FUNCTIONS_AMOUNT який вміщає у собі адреси алокованих функцій для використання. Запис до масиву виконується у функції allocate_syscall, читання у функції get_syscall_addr.

Бібліотека має конструктор власне який відповідає за її ініціалізацію, пошук індексу функцій в ntdll.dll та алокації функцій які підтримуються:



```
public:
    AntiDebugLib()
    {
#ifdef _WIN64
        auto PEB = ((PTEB)__readgsqword(offsetof(NT_TIB, Self)))->ProcessEnvironmentBlock;
#else
        auto PEB = ((PTEB)__readfsdword(offsetof(NT_TIB, Self)))->ProcessEnvironmentBlock;
#endif
        auto ntdll_entry = CONTAINING_RECORD(PEB->Ldr->InMemoryOrderModuleList.Flink->Flink, LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks);
#ifdef _WIN64
        auto ntdll_handle = reinterpret_cast<DWORD64>(ntdll_entry->DllBase);
#else
        auto ntdll_handle = reinterpret_cast<DWORD>(ntdll_entry->DllBase);
#endif


        auto ntdll_idh = reinterpret_cast<IMAGE_DOS_HEADER*>(ntdll_handle);
        auto ntdll_nth = reinterpret_cast<IMAGE_NT_HEADERS*>(ntdll_handle + ntdll_idh->e_lfanew);
        auto ntdll_ioh = ntdll_nth->OptionalHeader;
        auto ntdll_ied = reinterpret_cast<IMAGE_EXPORT_DIRECTORY*>(ntdll_handle + ntdll_ioh.DataDirectory[0].VirtualAddress);

        auto func_names = reinterpret_cast<DWORD*>(ntdll_handle + ntdll_ied->AddressOfNames);
        auto func_addresses = reinterpret_cast<DWORD*>(ntdll_handle + ntdll_ied->AddressOfFunctions);
        auto func_ordinals = reinterpret_cast<WORD*>(ntdll_handle + ntdll_ied->AddressOfNameOrdinals);
```

Рисунок 3.8 – Опис коду бібліотеки

Змінна PEB приймає значення адреси PEB у процесі. Для 32-бітних та 64-бітних процесів отримання адреси PEB дещо відрізняється, тому була використана директива #define _WIN64.

Змінна ntdll_entry примає інформацію про модуль ntdll.dll. У будь-якому процесі ntdll.dll є завжди за рахунком другою у таблиці LDR, на першому місці завжди інформація про сам процес. Використовуючи структуру PEB та поле Ldr ми отримали таблицю Ldr яка містить у собі інформацію про всі завантажені модулі у процесі та звернулися до другого за рахунком модуля, тобто отримали інформацію про ntdll.dll.

Змінна ntdll_handle приймає хендл (адресу) модуля ntdll.dll з використанням інформації отриманої в змінній ntdll_entry.

Змінні ntdll_idh, ntdll_nth, ntdll_ioh, ntdll_ied приймають значення, які знаходяться у PE хедері ntdll, для нас є важлива змінна ntdll_ied (для отримання таблиці експорту модуля ntdll.dll).

Змінна func_names – показчик на масив назв функцій експорту у модулі ntdll.dll.

Змінна func_addresses – показчик на масив адрес функцій експорту у модулі ntdll.dll.

Змінна func_ordinals – показчик на масив ординат функцій експорту у модулі ntdll.dll.

```
for (auto i = 0; i < ntdll_ied->NumberOfFunctions; i++)
{
    auto func_hash = HASH_RUNTIME(reinterpret_cast<const char*>(ntdll_handle + func_names[i]));

    if (func_hash == wow64transition)
    {
        is_wow64 = true;
    }

    for (auto j = 0; j < SUPPORTED_FUNCTIONS_AMOUNT; j++)
    {
        if (func_hash == function_hashes[j])
        {
            auto func_addr = reinterpret_cast<void*>(ntdll_handle + func_addresses[func_ordinals[i]]);
            find_syscall_index(func_addr, j);
        }
    }
}
```

Рисунок 3.9 – Опис коду бібліотеки

Головний цикл, який відповідає за обробку експорту та пошук індексів функцій у модулі ntdll.dll. Як можна побачити саме тут використовується хеш-функція HASH_RUNTIME для калькуляції crc32 під час виконання застосунку.

Логіка цього циклу дуже проста:

1. Робиться хеш назви функції у ntdll.dll;
2. Порівнюється з константими хешами, які є у застосунку;
3. Якщо хеші співпадають, виконує пошук індексу функції у модулі ntdll.dll.

Фінальною стадією бібліотеки є алокація всіх знайдених функцій, які підтримуються у пам’ять:

```
auto syscall_call = (char*)malloc(MAX_SYSCALL_CALL_SIZE);
auto syscall_call_protection = (char*)malloc(MAX_SYSCALL_CALL_SIZE);

this->allocate_syscall(syscall_call, ZwAllocateVirtualMemory);
this->allocate_syscall(syscall_call_protection, ZwProtectVirtualMemory);

DWORD old_protection = NULL;
VirtualProtect(syscall_call, MAX_SYSCALL_CALL_SIZE, PAGE_EXECUTE_READWRITE, &old_protection);
VirtualProtect(syscall_call_protection, MAX_SYSCALL_CALL_SIZE, PAGE_EXECUTE_READWRITE, &old_protection);

for (auto i = 0; i < SUPPORTED_FUNCTIONS_AMOUNT; i++)
{
    SIZE_T mem_size = MAX_SYSCALL_CALL_SIZE;
    void* new_addr = NULL;
    reinterpret_cast<ZwAllocateVirtualMemoryWrapper>(syscall_call)(CURRENT_PROCESS, &new_addr, NULL, &mem_size,
        MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    this->allocate_syscall(new_addr, SUPPORTED_FUNCTIONS(i));
    reinterpret_cast<ZwProtectVirtualMemoryWrapper>(syscall_call_protection)(CURRENT_PROCESS, &new_addr, &mem_size, PAGE_EXECUTE_READ,
        &old_protection);
    syscall_addresses[i] = new_addr;
}

free(syscall_call);
free(syscall_call_protection);
```

Рисунок 3.10 – Опис коду бібліотеки

syscall_call – буфер в який буде записано системний виклик для функції ZwAllocateVirtualMemory (алокація буферу в пам’яті).

syscall_call_protection – буфер в який буде записано системний виклик для функції ZwProtectVirtualMemory (зміна захисту для відповідного алокованого буферу).

Після цього відбувається виклик функцій allocate_syscall для алокації заданих функцій, потім завдяки функції VirtualProtect змінюється захист буферів та додається право на виконання.

Головний цикл працює таким чином:

1. Алокується буфер з розміром MAX_SYSCALL_CALL_SIZE через виклик syscall_call;
2. Алокується системний виклик і-тої функції до попередньо алокованого буферу;
3. Змінюється захист буфера та залишаються лише права на читання та виконання через виклик syscall_call_protection;
4. Адреса алокованої функції записується у приватний масив syscall_addresses.

Після цього викликається функція free для буферів syscall_call та syscall_call_protection.

Ініціалізацію бібліотеки можна вважати завершеною та можливою для використання.

Для того щоб використовувати бібліотеку необхідно створити її об’єкт:

```
#include <Windows.h>
#include <string>
#include <iostream>
#include "AntiDebugLib.h"

int main()
{
    AntiDebugLib* adbg_instance = new AntiDebugLib();

    auto func = reinterpret_cast<ZwQueryInformationProcessWrapper>(adbg_instance->get_syscall_addr(ZwQueryInformationProcess));

    DWORD is_debugger_present = 0x1337;
    auto process_debug_port = 7;

    if (func(CURRENT_PROCESS, process_debug_port, &is_debugger_present, sizeof(DWORD), NULL) != 0x0 || is_debugger_present != 0)
    {
        std::cout << "Debugger detected" << std::endl;
    }
    else
    {
        std::cout << "Debugger is not detected" << std::endl;
    }

    getchar();
    return true;
}
```

Рисунок 3.11 – Опис коду бібліотеки

adbg_instance – екземпляр бібліотеки.

func – адреса алокованої функції ZwQueryInformationProcess, отримана завдяки виклику adbg_instance->get_syscall_addr(ZwQueryInformationProcess).

Як можна побачити використання самої бібліотеки є дуже простим і зрозумілим. Для того щоб додати підтримку для будь-якої функції ntdll.dll розробнику достатньо змінити у бібліотеці такі значення: SUPPORTED_FUNCTIONS_AMOUNT, відредагувати структуру enum SUPPORTED_FUNCTIONS та масив хешів function_hashes. Бібліотека є дуже простою для розуміння та її експлуатації.

3.2 Тестування бібліотеки

Для тестування були написані застосунки, з прямим використанням викликів WinAPI та з використанням емуляції викликів WinAPI. Список функцій які використовувались: CheckRemoteDebuggerPresent, CreateFileA, ReadFile, CloseHandle, MessageBoxA, ExitProcess.

Відповідно емулювані функції мають такі назви: ZwQueryInformationProcess, ZwCreateFile, ZwReadFile, ZwClose, ZwRaiseHardError, ZwTerminateProcess.

Обидва застосунки були поступово захищені алгоритмами мутації, віртуалізації, заплутування та компресії з використанням комерційного рішення під назвою VMProtect.

Код застосунку з використанням прямих викликів WinAPI:

```
#include <Windows.h>
#include <iostream>
#include "VMProtectSDK.h"
#pragma comment (lib, "VMProtectSDK64.lib")

char data[10];

int main()
{
    VMProtectBegin("entrypoint");
    BOOL debugger_present = FALSE;

    if (CheckRemoteDebuggerPresent(GetCurrentProcess(), &debugger_present) && !debugger_present)
    {
        auto file_handle = CreateFileA("cdkey.key", GENERIC_READ, NULL, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

        if (file_handle)
        {
            DWORD bytes_read = NULL;
            if (ReadFile(file_handle, data, 10, &bytes_read, NULL))
            {
                CloseHandle(file_handle);

                if (strstr(data, "encrypted"))
                {
                    MessageBoxA(NULL, "License check passed.", "Success", MB_ICONINFORMATION);
                }
                else
                {
                    MessageBoxA(NULL, "License check is not passed.", "Error", MB_ICONERROR);
```

Рисунок 3.12 – Приклад коду без використання бібліотеки

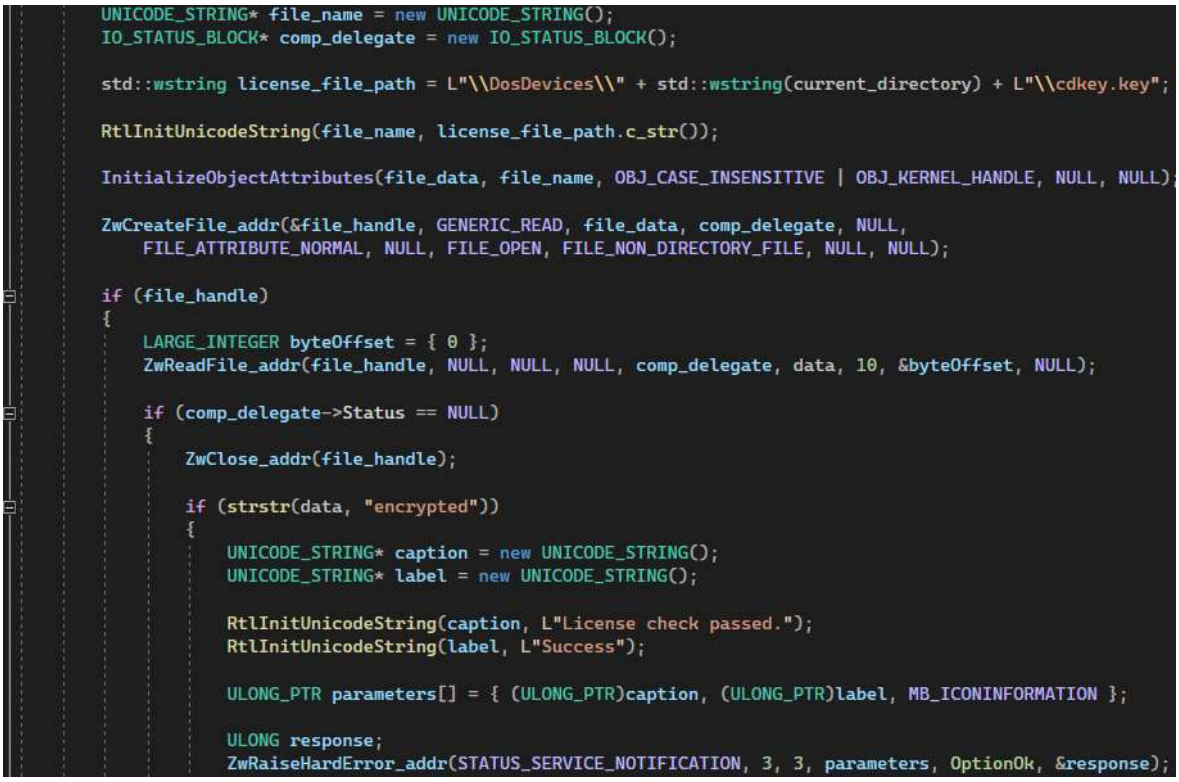
```
                ExitProcess(4);
            }
        }
        else
        {
            MessageBoxA(NULL, "Failed to read license file.", "Error", MB_ICONERROR);
            ExitProcess(3);
        }
    }
    else
    {
        MessageBoxA(NULL, "Failed to open license file.", "Error", MB_ICONERROR);
        ExitProcess(2);
    }
}
else
{
    MessageBoxA(NULL, "Debugger present.", "Error", MB_ICONERROR);
    ExitProcess(1);
}
VMProtectEnd();
getchar();
return true;
}
```

Рисунок 3.13 – Приклад коду без використання бібліотеки

Код застосунку з використанням емуляції викликів WinAPI:

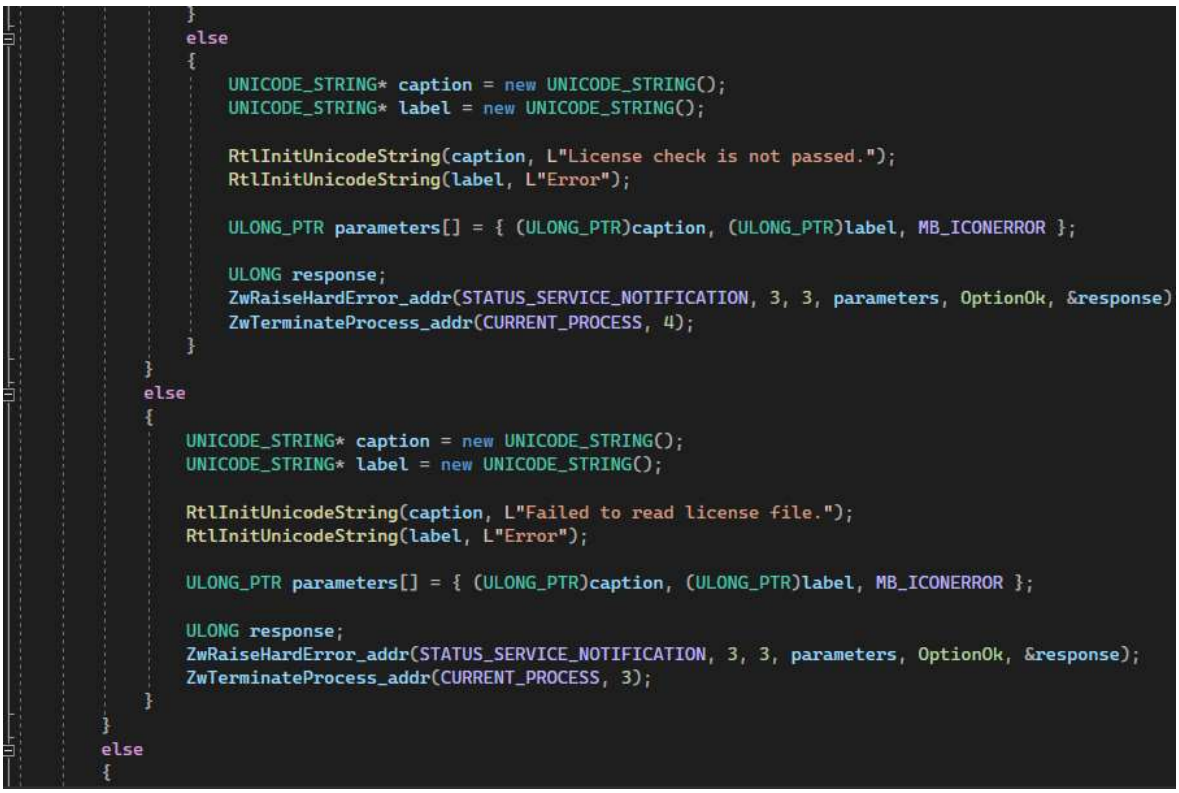
```
#include <Windows.h>
#include <iostream>
#include "VMProtectSDK.h"
#include "AntiDebugLib.h"
#pragma comment (lib, "VMProtectSDK64.lib")

#pragma comment (lib, "ntdll")

char data[10];
wchar_t current_directory[1024];

int main()
{
    VMProtectBegin("entrypoint");
    GetCurrentDirectoryW(1024, current_directory);

    AntiDebugLib* adbg_instance = new AntiDebugLib();

    auto ZwQueryInformationProcess_addr = reinterpret_cast<ZwQueryInformationProcessWrapper>(adbg_instance->get_syscall_addr(ZwQueryInformationProcess));
    auto ZwCreateFile_addr = reinterpret_cast<ZwCreateFileWrapper>(adbg_instance->get_syscall_addr(ZwCreateFile));
    auto ZwReadFile_addr = reinterpret_cast<ZwReadFileWrapper>(adbg_instance->get_syscall_addr(ZwReadFile));
    auto ZwClose_addr = reinterpret_cast<ZwCloseWrapper>(adbg_instance->get_syscall_addr(ZwClose));
    auto ZwRaiseHardError_addr = reinterpret_cast<ZwRaiseHardErrorWrapper>(adbg_instance->get_syscall_addr(ZwRaiseHardError));
    auto ZwTerminateProcess_addr = reinterpret_cast<ZwTerminateProcessWrapper>(adbg_instance->get_syscall_addr(ZwTerminateProcess));

    DWORD64 is_debugger_present = 0x1337;
    auto process_debug_port = 7;
    BOOL debugger_present = FALSE;

    if (ZwQueryInformationProcess_addr(CURRENT_PROCESS, process_debug_port, &is_debugger_present, sizeof(DWORD64),NULL) == 0 && is_debugger_present == 0)
    {
        HANDLE file_handle = NULL;
        OBJECT_ATTRIBUTES* file_data = new OBJECT_ATTRIBUTES();
```

Рисунок 3.14 – Приклад коду з використання бібліотеки



```
UNICODE_STRING* file_name = new UNICODE_STRING();
IO_STATUS_BLOCK* comp_delegate = new IO_STATUS_BLOCK();

std::wstring license_file_path = L"\\DosDevices\\" + std::wstring(current_directory) + L"\\cdkey.key";

RtlInitUnicodeString(file_name, license_file_path.c_str());

InitializeObjectAttributes(file_data, file_name, OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);

ZwCreateFile_addr(&file_handle, GENERIC_READ, file_data, comp_delegate, NULL,
    FILE_ATTRIBUTE_NORMAL, NULL, FILE_OPEN, FILE_NON_DIRECTORY_FILE, NULL, NULL);

if (file_handle)
{
    LARGE_INTEGER byteOffset = { 0 };
    ZwReadFile_addr(file_handle, NULL, NULL, NULL, comp_delegate, data, 10, &byteOffset, NULL);

    if (comp_delegate->Status == NULL)
    {
        ZwClose_addr(file_handle);

        if (strstr(data, "encrypted"))
        {
            UNICODE_STRING* caption = new UNICODE_STRING();
            UNICODE_STRING* label = new UNICODE_STRING();

            RtlInitUnicodeString(caption, L"License check passed.");
            RtlInitUnicodeString(label, L"Success");

            ULONG_PTR parameters[] = { (ULONG_PTR)caption, (ULONG_PTR)label, MB_ICONINFORMATION };

            ULONG response;
            ZwRaiseHardError_addr(STATUS_SERVICE_NOTIFICATION, 3, 3, parameters, OptionOk, &response);
```

Рисунок 3.15 – Приклад коду з використання бібліотеки



```
            }
            else
            {
                UNICODE_STRING* caption = new UNICODE_STRING();
                UNICODE_STRING* label = new UNICODE_STRING();

                RtlInitUnicodeString(caption, L"License check is not passed.");
                RtlInitUnicodeString(label, L"Error");

                ULONG_PTR parameters[] = { (ULONG_PTR)caption, (ULONG_PTR)label, MB_ICONERROR };

                ULONG response;
                ZwRaiseHardError_addr(STATUS_SERVICE_NOTIFICATION, 3, 3, parameters, OptionOk, &response)
                ZwTerminateProcess_addr(CURRENT_PROCESS, 4);
            }
        }
        else
        {
            UNICODE_STRING* caption = new UNICODE_STRING();
            UNICODE_STRING* label = new UNICODE_STRING();

            RtlInitUnicodeString(caption, L"Failed to read license file.");
            RtlInitUnicodeString(label, L"Error");

            ULONG_PTR parameters[] = { (ULONG_PTR)caption, (ULONG_PTR)label, MB_ICONERROR };

            ULONG response;
            ZwRaiseHardError_addr(STATUS_SERVICE_NOTIFICATION, 3, 3, parameters, OptionOk, &response);
            ZwTerminateProcess_addr(CURRENT_PROCESS, 3);
        }
    }
    else
    {
```

Рисунок 3.16 – Приклад коду з використання бібліотеки

```
        }
        else
        {
            UNICODE_STRING* caption = new UNICODE_STRING();
            UNICODE_STRING* label = new UNICODE_STRING();

            RtlInitUnicodeString(caption, L"Failed to open license file.");
            RtlInitUnicodeString(label, L"Error");

            ULONG_PTR parameters[] = { (ULONG_PTR)caption, (ULONG_PTR)label, MB_ICONERROR };

            ULONG response;
            ZwRaiseHardError_addr(STATUS_SERVICE_NOTIFICATION, 3, 3, parameters, OptionOk, &response);
            ZwTerminateProcess_addr(CURRENT_PROCESS, 2);
        }
    }
    else
    {
        UNICODE_STRING* caption = new UNICODE_STRING();
        UNICODE_STRING* label = new UNICODE_STRING();

        RtlInitUnicodeString(caption, L"Debugger present.");
        RtlInitUnicodeString(label, L"Error");

        ULONG_PTR parameters[] = { (ULONG_PTR)caption, (ULONG_PTR)label, MB_ICONERROR };

        ULONG response;
        ZwRaiseHardError_addr(STATUS_SERVICE_NOTIFICATION, 3, 3, parameters, OptionOk, &response);
        ZwTerminateProcess_addr(CURRENT_PROCESS, 1);
    }
    VMProtectEnd();
    getchar();
    return true;
```

Рисунок 3.17 – Приклад коду з використання бібліотеки

Обидва коди виконують одну й ту саму функцію, роботу застосунків можна розділити на такі етапи:

1. Перевірка на наявність налагоджувача (використовуючи функції CheckRemoteDebuggerPresent/ZwQueryInformationProcess);
2. Якщо налагоджувач не був детектований, створюється хендл файлу cdkey.key, який лежить у теці з додатком (використовуючи функції CreateFileA/ZwCreateFile);
3. Зчитується контент файлу (використовуючи функції ReadFile/ZwReadFile);
4. Закривається хендл файлу (використовуючи функції CloseHandle/ZwClose);
5. Перевіряється наявність слова encrypted у зчитаних даних;
6. Якщо слово наявне у зчитаних даних, користувач отримує сповіщення що перевірку ліцензії пройдено.

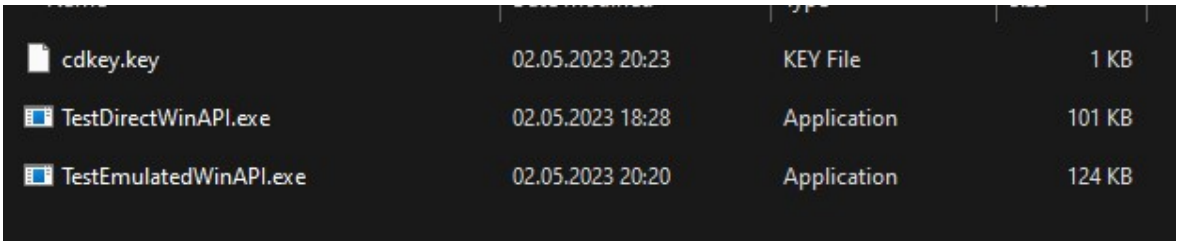


Рисунок 3.18 – Скомпільовані застосунки

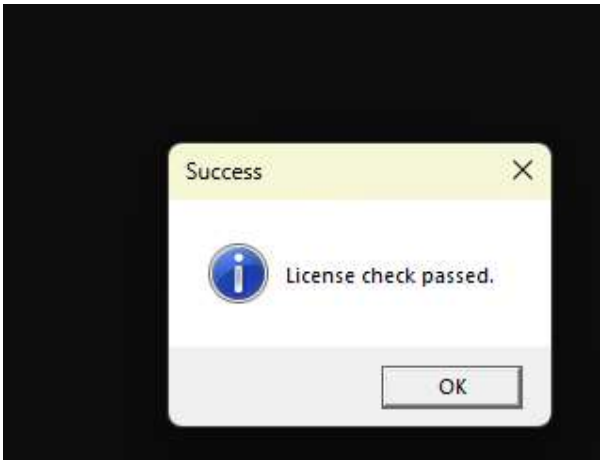


Рисунок 3.19 – Результат виконання застосунку

Якщо запустити застосунок під налагоджувачем отримаємо данне сповіщення:

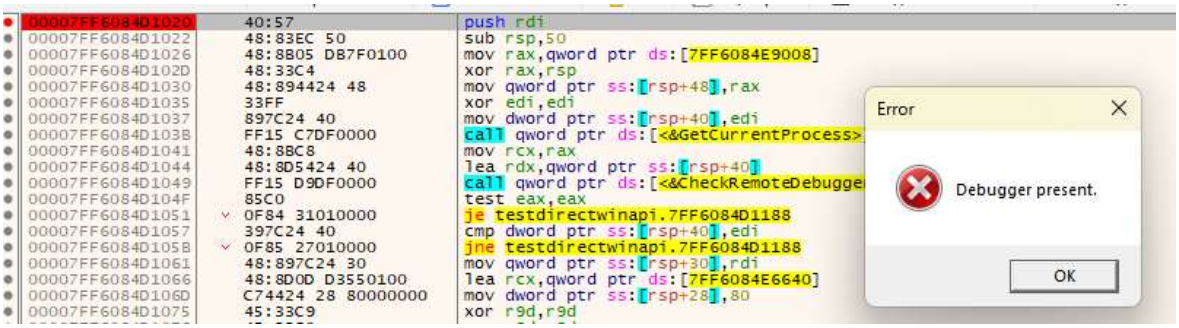


Рисунок 3.20 – Дизасемблювання застосунку

Застосунки були поступово захищенні з використанням комерційного рішення під назвою VMProtect, умовно можна поділити на такі етапи:

1. Чистий код – додатки були скомпільовані та не захищені;
2. Алгоритми мутації – додатки були скомпільовані та захищені з використанням алгоритму мутації;
3. Алгоритми мутації та віртуалізації – додатки були скомпільовані та захищені з використанням алгоритмів мутації та віртуалізації;

4. Алгоритми мутації, віртуалізації та компресії – додатки були скомпільовані та захищені з використанням алгоритмів мутації, віртуалізації та компресії.

Кожен етап був проаналізований на можливість зламу.

Рисунок 3.21 – Дизасемблювання застосунку

Як можна побачити у додатку який використовує прямі виклики WinAPI, достатньо просто та швидко знайти функцію main та побачити як і що саме перевіряється, перевірка на налагоджувач можна обійти змінивши інструкції на NOP після виклику CheckRemoteDebuggerPresent.

Рисунок 3.22 – Дизасемблювання застосунку

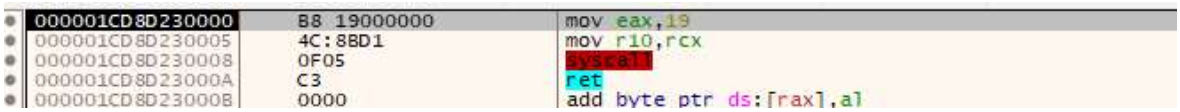


Рисунок 3.23 – Дизасемблювання застосунку

У випадку з емуляцією викликів WinAPI також можна знайти функцію main, але як можна побачити, немає використання прямих викликів WinAPI, тому умовно якщо би були використані алгоритми шифрування текстових даних, не можна було би знайти функцію main так швидко.

Як можна побачити емуляцію викликів WinAPI недоцільно використовувати без застосування алгоритмів мутації, віртуалізації, компресії, тощо.

Після використання алгоритму мутації на додатках маємо такий результат:

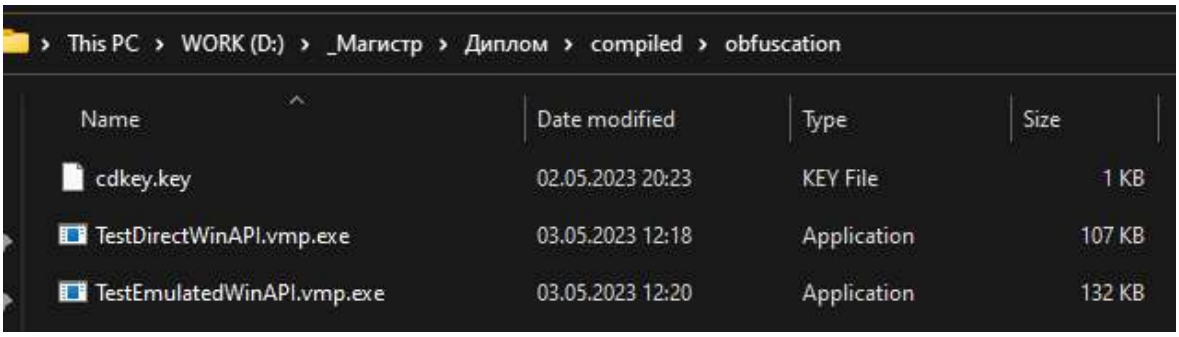


Рисунок 3.24 – Результат захисту VMProtect

Розмір файлів дещо зріс, та дизасемблерний код виглядає дещо іншим:

```
00007FF6AA2DD675    DCA2 31A4D790           fsub st(0),qword ptr ds:[rdx-6F285BCF]
00007FF6AA2DD67B    91                      xchg ecx,eax
00007FF6AA2DD67C    06                      ???
00007FF6AA2DD67D    D258 F4                 rcr byte ptr ds:[rax-C],cl
00007FF6AA2DD680    4D:A8 94                test al,94
00007FF6AA2DD683    4C:5C                   pop rsp
00007FF6AA2DD685    74 87                   je testdirectwinapi.vmp.7FF6AA2DD60E
00007FF6AA2DD687    A8 E5                   test al,E5
00007FF6AA2DD689    41                      ???
00007FF6AA2DD68A    27                      ???
00007FF6AA2DD68B    F8                      clc
00007FF6AA2DD68C    A6                      cmpsb
00007FF6AA2DD68D    2834CA                  sub byte ptr ds:[rdx+rcx*8],dh
00007FF6AA2DD690    89D9                    mov ecx,ebx
00007FF6AA2DD692    90                      nop
00007FF6AA2DD693    66:A9 1F24              test ax,241F
00007FF6AA2DD697    33FF                    xor edi,edi
00007FF6AA2DD699    E9 00000000             jmp testdirectwinapi.vmp.7FF6AA2DD69E
00007FF6AA2DD69E    897C24 40               mov dword ptr ss:[rsp+40],edi
00007FF6AA2DD6A2    E9 00000000             jmp testdirectwinapi.vmp.7FF6AA2DD6A7
00007FF6AA2DD6A7    FF15 5B19FFFF           call qword ptr ds:[<&GetCurrentProcess>]
00007FF6AA2DD6AD    48:8BC8                 mov rcx,rax
00007FF6AA2DD6B0    49:0FBFD7               movsx rdx,r15w
00007FF6AA2DD6B4    48:8D5424 40            lea rdx,qword ptr ss:[rsp+40]
00007FF6AA2DD6B9    E9 00000000             jmp testdirectwinapi.vmp.7FF6AA2DD6BE
00007FF6AA2DD6BE    FF15 6419FFFF           call qword ptr ds:[<&CheckRemoteDebuggerPresent>]
00007FF6AA2DD6C4    85C0                    test eax,eax
00007FF6AA2DD6C6    0F84 E6010000           je testdirectwinapi.vmp.7FF6AA2DD8B2
00007FF6AA2DD6CC    397C24 40               cmp dword ptr ss:[rsp+40],edi
00007FF6AA2DD6D0    E9 00000000             jmp testdirectwinapi.vmp.7FF6AA2DD6D5
00007FF6AA2DD6D5    0F85 D7010000           jne testdirectwinapi.vmp.7FF6AA2DD8B2
00007FF6AA2DD6DB    48:897C24 30            mov qword ptr ss:[rsp+30],rdi
00007FF6AA2DD6E0    C0D6 19                 rcl dh,19
00007FF6AA2DD6E3    87D2                    xchg edx,edx
00007FF6AA2DD6E5    48:8D0D 848FFFFF        lea rcx,qword ptr ds:[7FF6AA2D6670]
00007FF6AA2DD6EC    C74424 28 80000000      mov dword ptr ss:[rsp+28],80
00007FF6AA2DD6F4    2BD6                    sub edx,esi
00007FF6AA2DD6F6    0F9BC2                  setnp dl
00007FF6AA2DD6F9    45:2BC9                 sub r9d,r9d
00007FF6AA2DD6FC    45:33C0                 xor r8d,r8d
00007FF6AA2DD6FF    6641:0FB6D3             movzx dx,r11b
00007FF6AA2DD704    8AD5                    mov dl,ch
00007FF6AA2DD706    C74424 20 03000000      mov dword ptr ss:[rsp+20],3
00007FF6AA2DD70E    0FB7D5                  movzx edx,bp
00007FF6AA2DD711    BA BDC61E8B             mov edx,8B1EC6BD
00007FF6AA2DD716    E9 00000000             jmp testdirectwinapi.vmp.7FF6AA2DD71B
00007FF6AA2DD71B    BA 00000080             mov edx,80000000
00007FF6AA2DD720    48:895C24 60            mov qword ptr ss:[rsp+60],rbx
00007FF6AA2DD725    E9 00000000             jmp testdirectwinapi.vmp.7FF6AA2DD72A
00007FF6AA2DD72A    FF15 E018FFFF           call qword ptr ds:[<&CreateFileA>]
00007FF6AA2DD730    48:8BD8                 mov rbx,rax
00007FF6AA2DD733    E9 00000000             jmp testdirectwinapi.vmp.7FF6AA2DD738
00007FF6AA2DD738    48:85C0                 test rax,rax
00007FF6AA2DD73B    E9 00000000             jmp testdirectwinapi.vmp.7FF6AA2DD740
```

Рисунок 3.25 – Дизасемблювання застосунку

Як можна побачити, інструкції дещо змінилися, функція на початку має “сміття”, але все-одно можна знайти функції main за текстовими данними та побачити як виконується код.

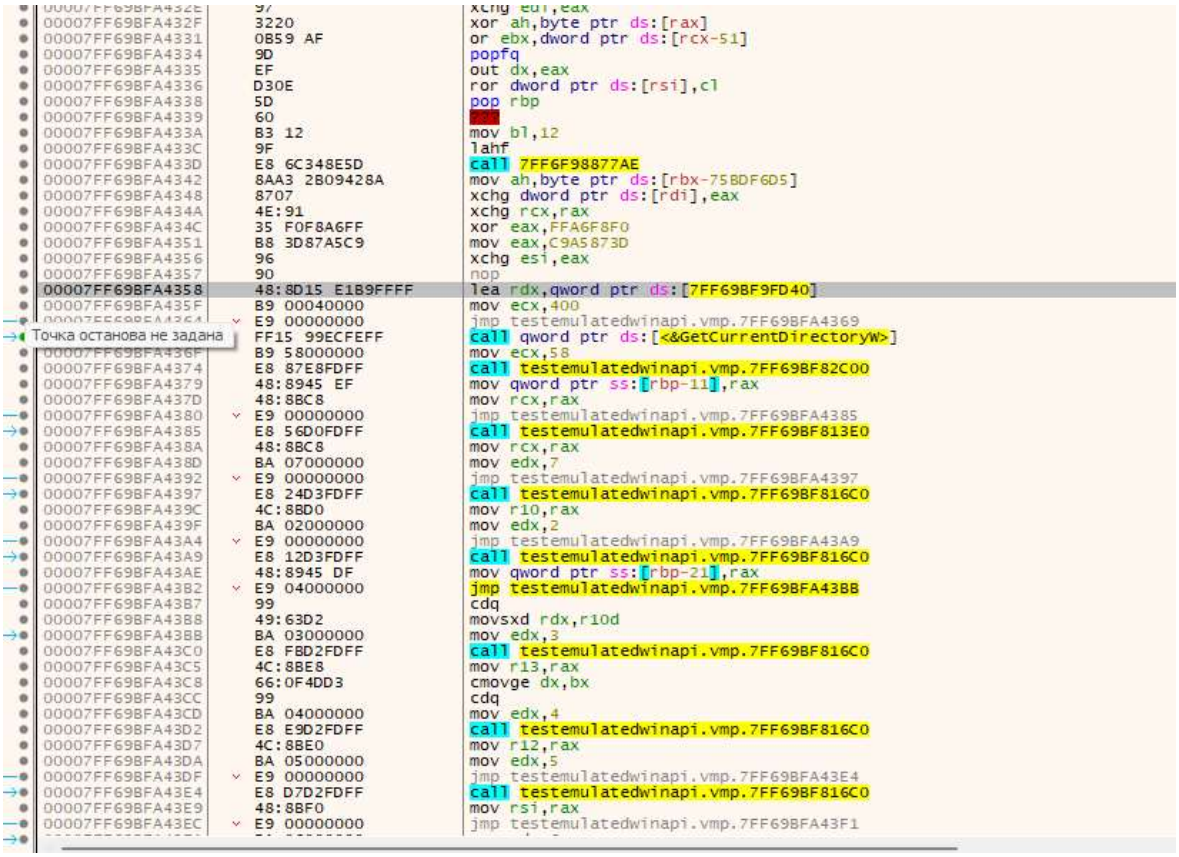


Рисунок 3.26 – Дизасемблювання застосунку

У версії додатка з емуляцією викликів WinAPI теж можна побачити “сміття” на початку функції та дещо змінені інструкції, однак все-одно можна зрозуміти як програма виконується.

Можна зробити висновок що використання алгоритму мутації також не є достатньою умовою для захисту.

На третьому етапі були застосовані алгоритми мутації та віртуалізації, після захисту додатки виглядають таким чином:

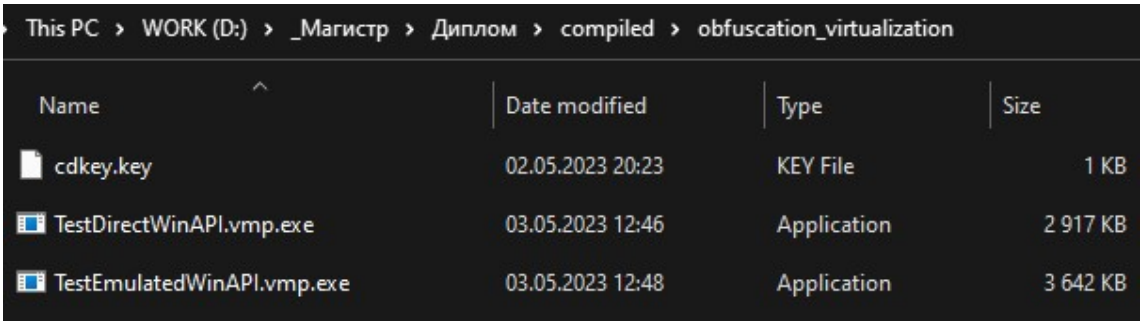


Рисунок 3.27 – Результат захисту VMProtect

Як можна побачити розмір файлів став у багато разів більшим.

Рисунок 3.28 – Пошук текстових даних у застосунку

Після захисту більше неможливо знайти текстові дані, а отже потрібно спробувати поставити точки зупину на такі функції як CheckRemoteDebuggerPresent, CreateFileA та інші.

Рисунок 3.29 – Дизасемблювання застосунку

Як можна побачити точка зупину спрацювала, якщо дійти до інструкції ret то контроль повернеться назад до нашого додатку і можна побачити таке:

Рисунок 3.30 – Дизасемблювання застосунку

Як можна побачити це “сміття” і є те що робить віртуалізація, більше не можна побачити код який був до цього, а отже обійти перевірки стає дещо складніше, однак ми маємо тут ключову вразливість – використання WinAPI, а це означає що знаючи функції, які використовує застосунок, можна змінити його поведінку, або взагалі зламати застосунок. Знаючи код, зрозуміло що застосунок використовує функцію CheckRemoteDebuggerPresent, а отже щоб обійти дану перевірку достатньо змінити результат даної функції, подивившись на MSDN вхідні параметри функції, можна зробити висновок що флаг який говорить, чи запущений процес під налагоджувачем є другий аргумент. Перевіримо цю гіпотезу та змінимо флаг після виконання функції. У 64-бітних додатках перші чотири аргументи йдуть у регістри: rcx, rdx, r8, r9, всі інші поміщаються у стек. Так як функція CheckRemoteDebuggerPresent має лише два аргументи, отже потрібно дивитися лише на регістри rcx, rdx. Так як нам потрібно модифікувати другий аргумент, потрібно дивитися на регістр rdx:

Рисунок 3.31 – Дизасемблювання застосунку

Можна побачити що у rcx поміщається значення FF, це означає хендлом процесу є сам процес, тобто перевірити даний процес на наявність відлагоджувача. Другим аргументом є адреса у rdx куди записується значення BOOL, тобто 1 байт – true або false.

Після виконання функції WinAPI бачимо що по даній адресі маємо значення 1, тобто true і як результат отримаємо віконце що знайдено налагоджувач:

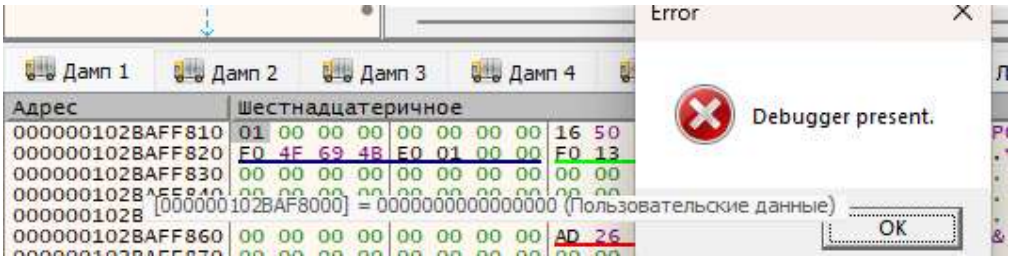


Рисунок 3.32 – Дизасемблювання застосунку

Якщо ж значення змінити на 0 – false, побачимо що викликається точка зупину на функції CreateFileA, що означає що перевірка на налагоджувач успішно була обійдена:

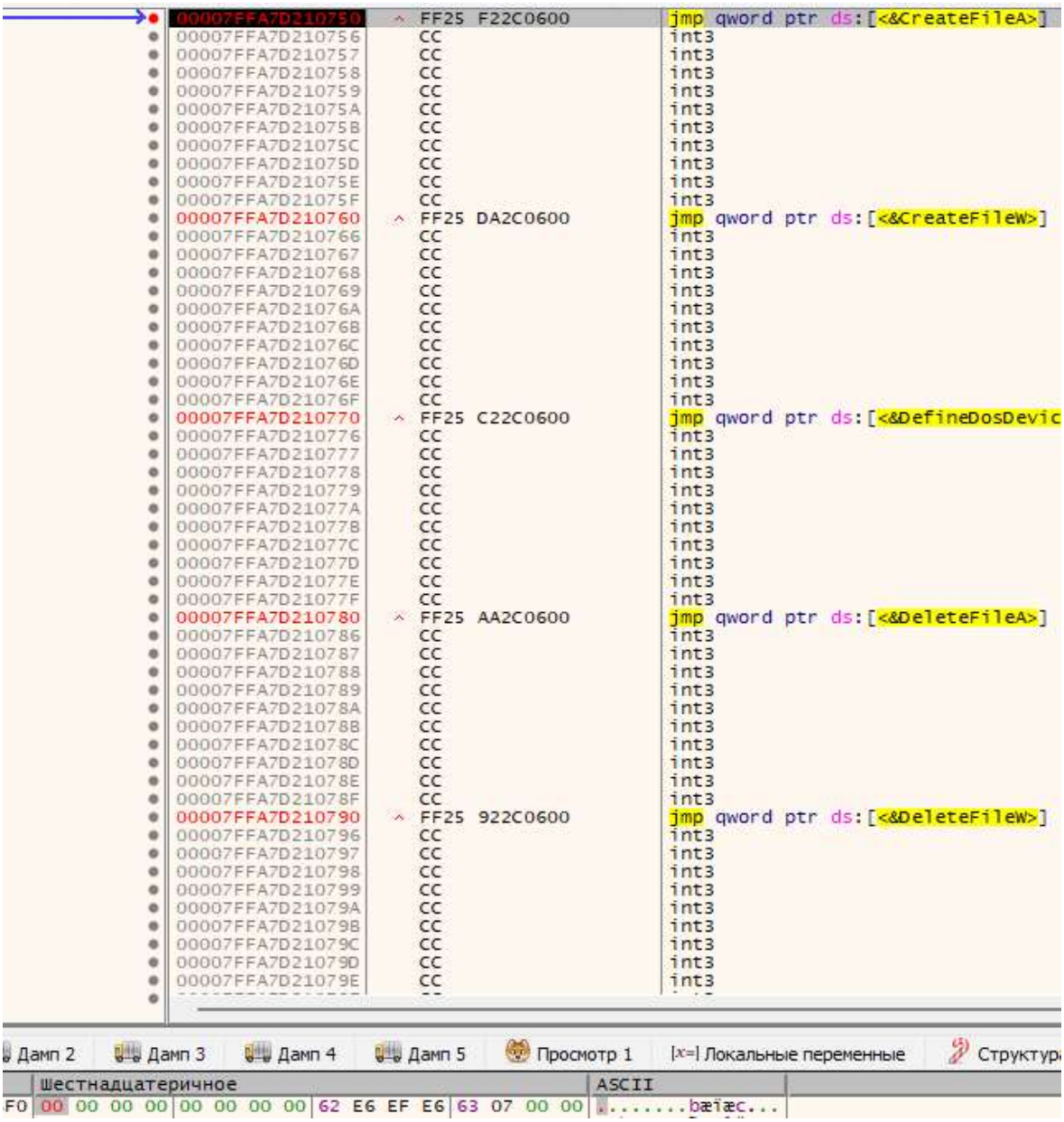


Рисунок 3.33 – Дизасемблювання застосунку

Подивившись на аргументи CreateFileA, бачимо назву файлу cdkey.key, отже можна одразу припустити що цей файл відкривається на читання або запис, результатом виконання CreateFileA буде хендл файлу поміщений у регістр rax:

Рисунок 3.34 – Дизасемблювання застосунку

Рисунок 3.35 – Дизасемблювання застосунку

Отже маємо хендл файлу 0xE0, його необхідно запам’ятати, адже саме цей хендл буде використовуватись у таких функціях як WriteFile або ReadFile. Поставивши точки зупину на WriteFile та ReadFile продовдуємо виконання програми та бачимо що програма зупиняється на функції ReadFile, отже робимо висновок що даний файл зчитується, а не записується:

Рисунок 3.36 – Дизасемблювання застосунку

rcx – містить у собі хендл файлу (0xE0)

rdx – містить у собі показчик на буфер куди дані зчитуються

Подивившись у дамп адреси rdx після виконання виклику ReadFile маємо такий результат:

Рисунок 3.37 – Дизасемблювання застосунку

Як бачимо зчиталося слово encrypted з файлу, логічним припущенням було б: якщо ці дані були зчитані, то напевно вони повинні у подальшій роботі програми бути перевірені, для цього ставимо апаратну точку зупину на читання данного буферу:

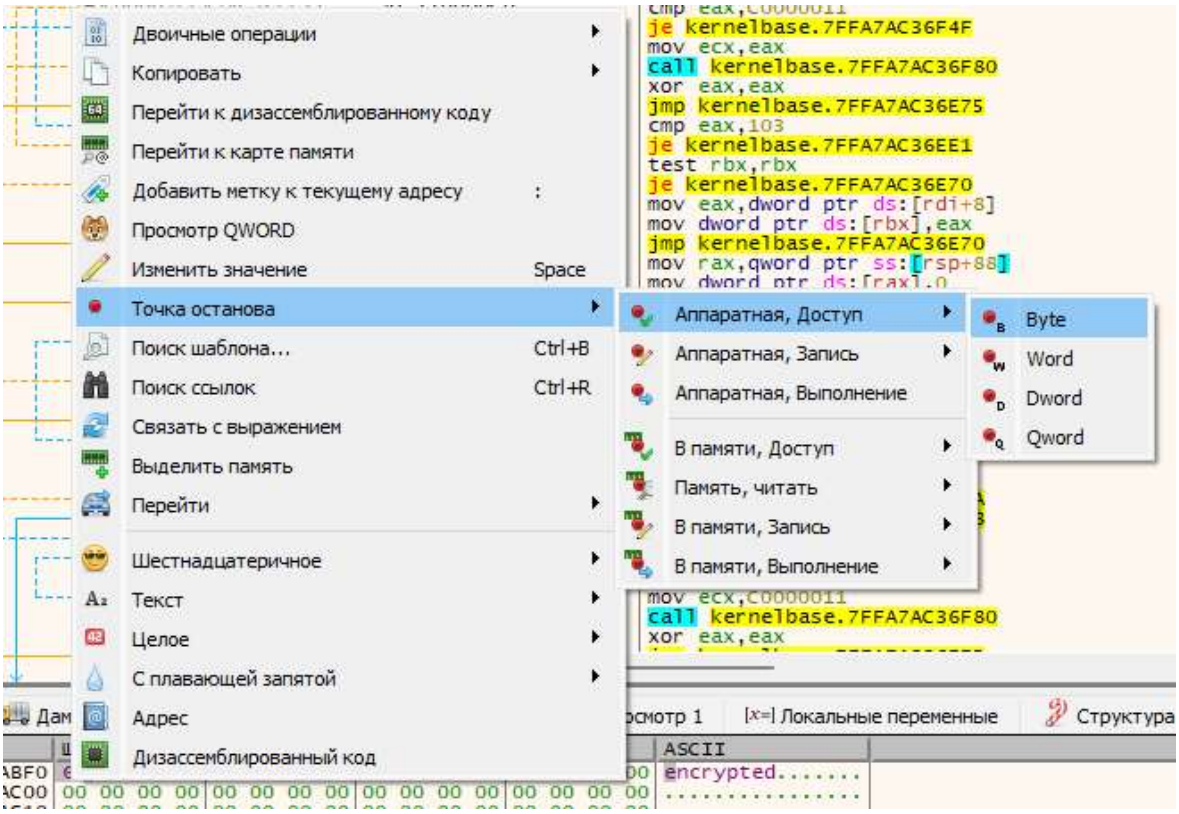


Рисунок 3.38 – Дизасемблювання застосунку

Потрапляємо у дане місце:

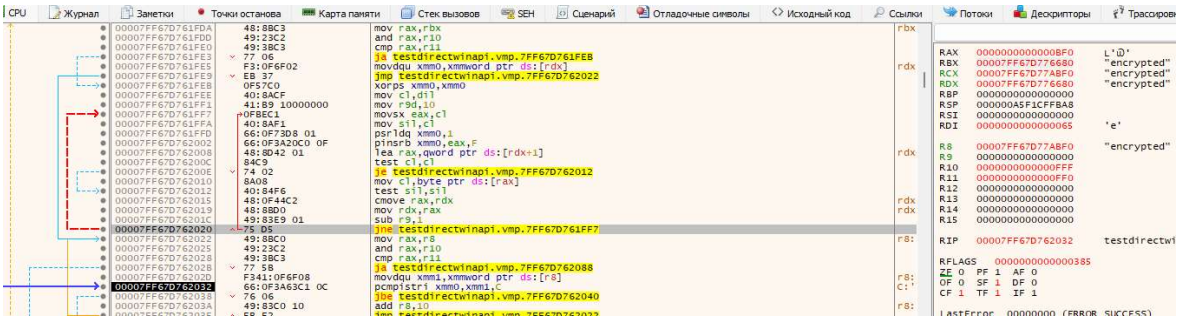


Рисунок 3.39 – Дизасемблювання застосунку

Що ж тут можна побачити, якщо придивитися уважно на регістри rcx та rbx то адреси різняться, але мають однакові данні. Якщо подивитися у дамп адреси регістра rcx отримаємо такий результат:

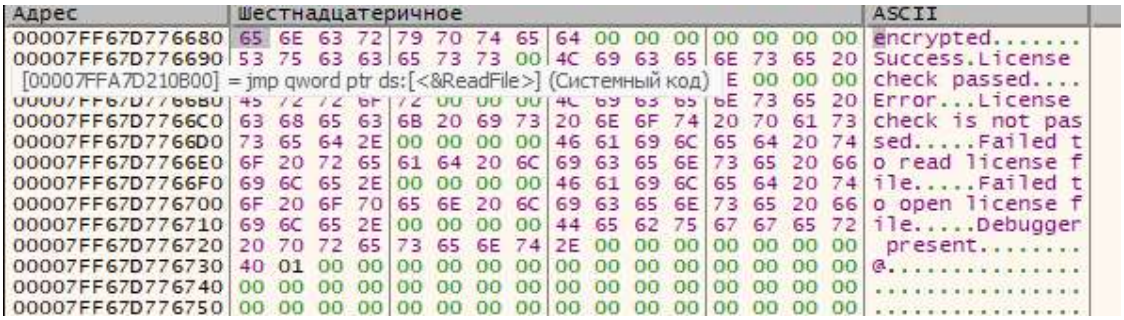


Рисунок 3.40 – Дизасемблювання застосунку

Що ж ми бачимо, текстові данні про ліцензію, невже це те що перевіряє програма? Саме так, для перевірки змінимо дані файлу cdkey.key на довільні та зробимо ті ж самі операції:

Рисунок 3.41 – Дизасемблювання застосунку

Як можна побачити що дані були змінені на invalid, але все одно ми бачимо дані encrypted, якщо продовжити роботу програми отримаємо що ліцензія невірна:

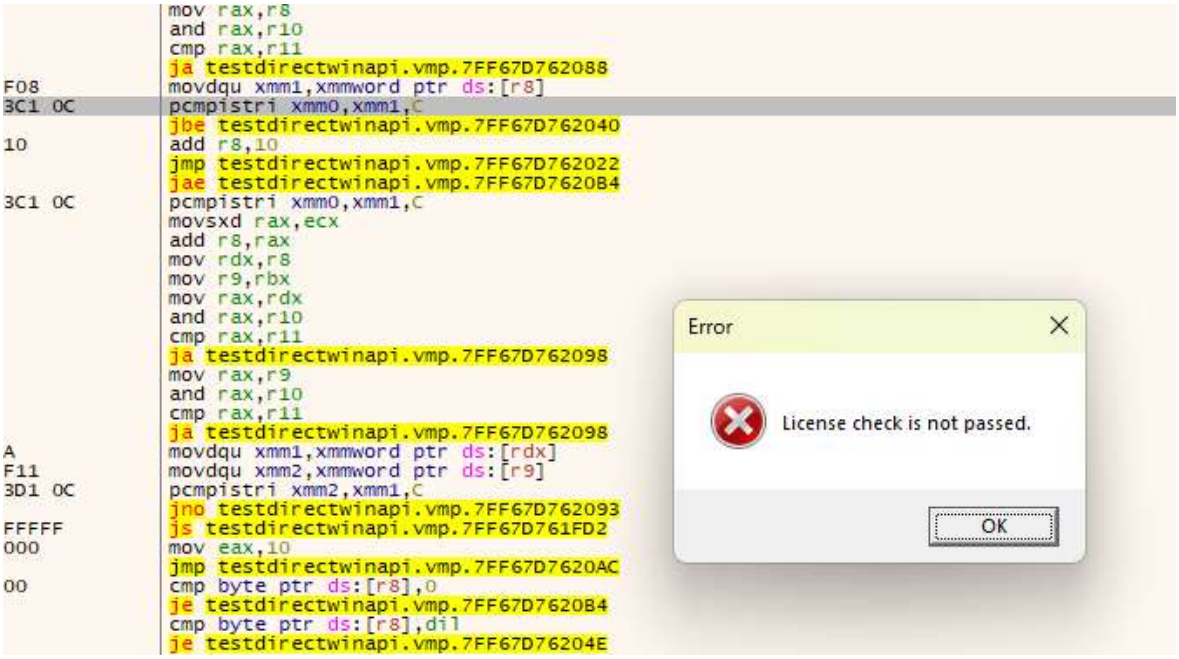


Рисунок 3.42 – Дизасемблювання застосунку

Якщо ж змінити дані на encrypted отримаємо що ліцензія вірна:

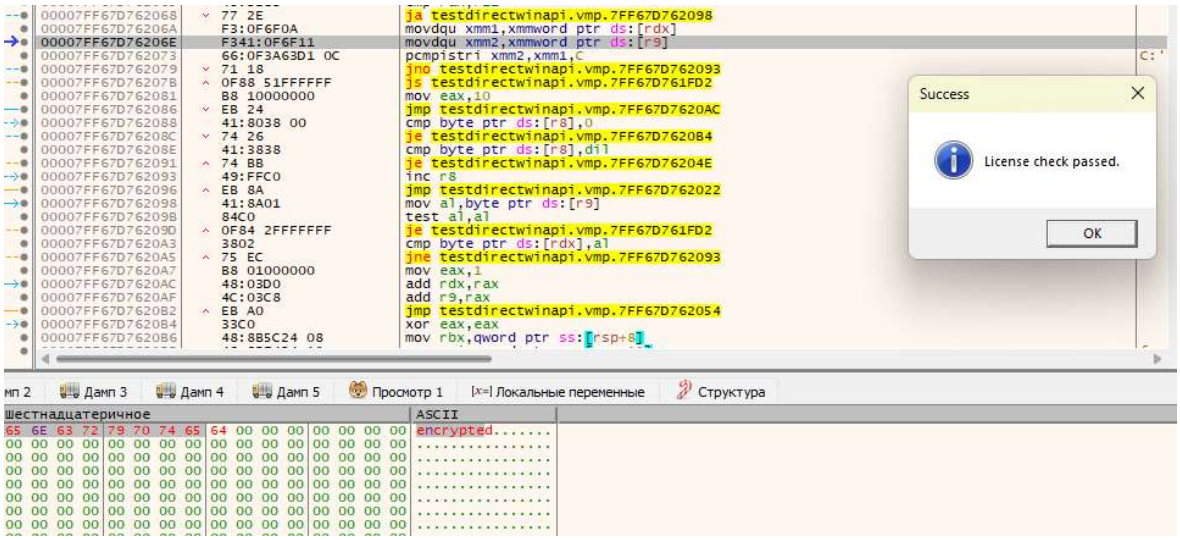


Рисунок 3.43 – Дизасемблювання застосунку

Все це доводить що використання прямих викликів WinAPI є вразливістю, принаймі для захисту, адже у даному прикладі навіть не потрібно було аналізувати віртуальну машину VMProtect, а достатньо було запропонувати гіпотези, які б функції могла використовувати програма і зробити злам самої програми.

Якщо ж проробити ті самі кроки на захищеному додатку з використанням емуляції викликів WinAPI, то нічого не вийде, адже точки зупину не будуть працювати, так як функції ніколи не будуть викликані, тому що вони земулювані:

Рисунок 3.44 – Дизасемблювання застосунку

Це доводить ефективність використання емуляції викликів WinAPI за умови додатково використання алгоритмів мутації та віртуалізації.

На четвертому та останньому етапі додатково було використано алгоритм компресії, насправді все що він робить це шифрує секцію коду додатку та розпаковує його під час виконанняґ, це ніяк не змінює проблему використання WinAPI:

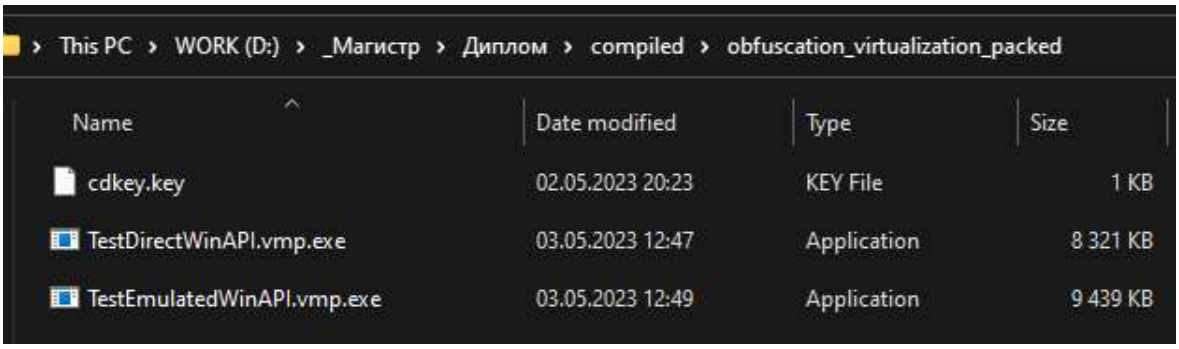


Рисунок 3.45 – Результат захисту VMProtect

Як бачимо розмір додатків збільшився у майже два рази від попереднього. При аналізі помітно що змінилася точка входу, та немає коду у секції .text:

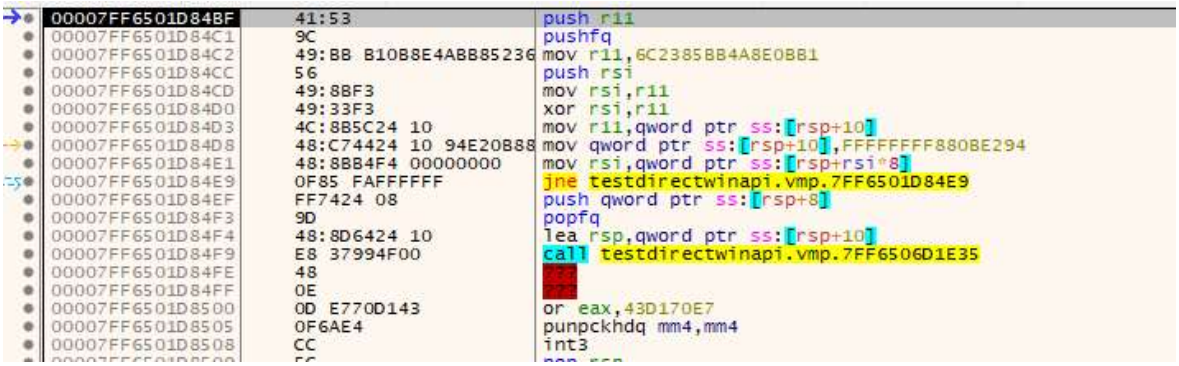


Рисунок 3.46 – Дизасемблювання застосунку

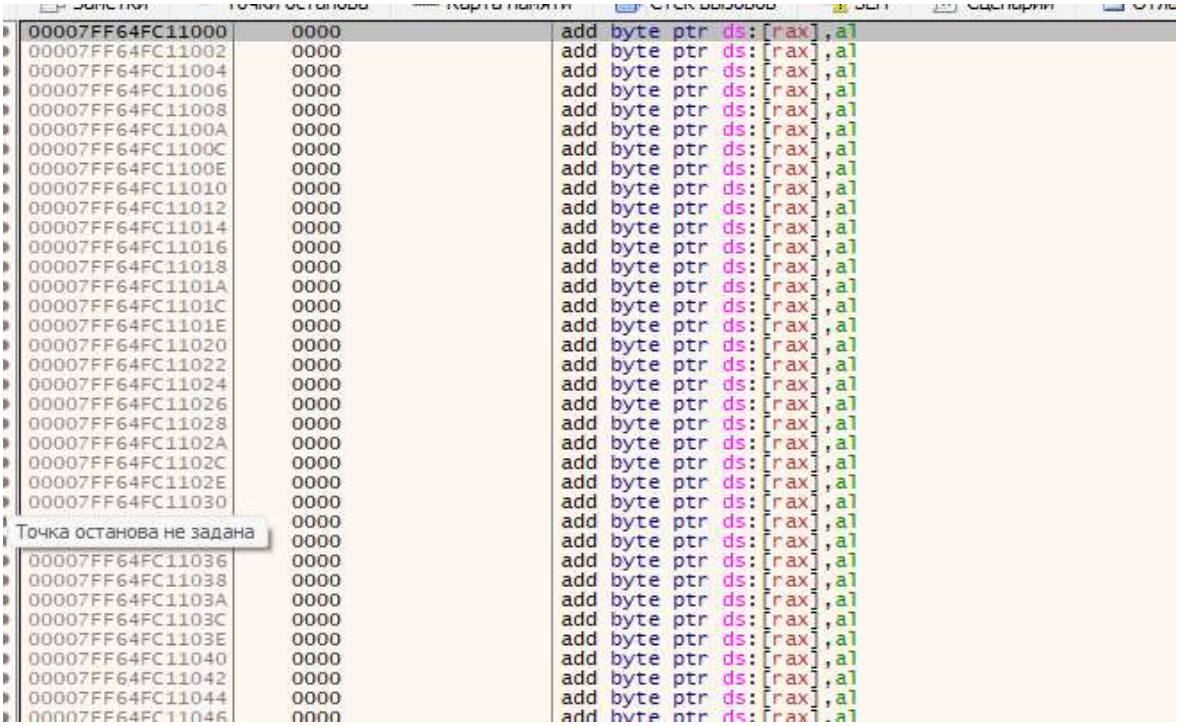


Рисунок 3.47 – Дизасемблювання застосунку

Але поставивши точки зупину на тих самих функціях, бачимо що вони все одно спрацьовують, більш того ми можемо побачити розпаковану секцію .text:

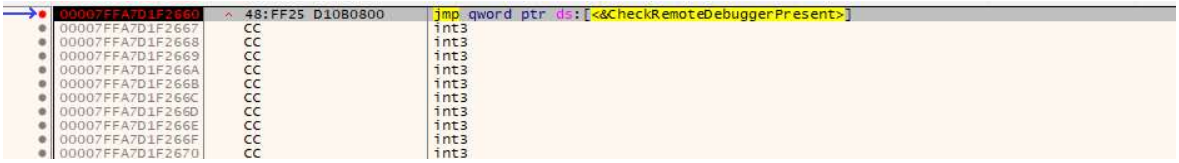


Рисунок 3.48 – Дизасемблювання застосунку

Рисунок 3.49 – Дизасемблювання застосунку

Це знову доводить що навіть при використанні усіх алгоритмів захисту можна все одно зламати застосунок знаючи функції WinAPI, які він використовує.

Якщо подивитися на захищенний застосунок з використанням емуляції викликів WinAPI, то аналіз стає ще складнішим, тому що потрібно знайти власноруч як відбувається розпаковка секції .text, тому аналіз стає ще складнішим, що доводить ефективність використання даної техніки:

Рисунок 3.50 – Дизасемблювання застосунку

Рисунок 3.51 – Дизасемблювання застосунку

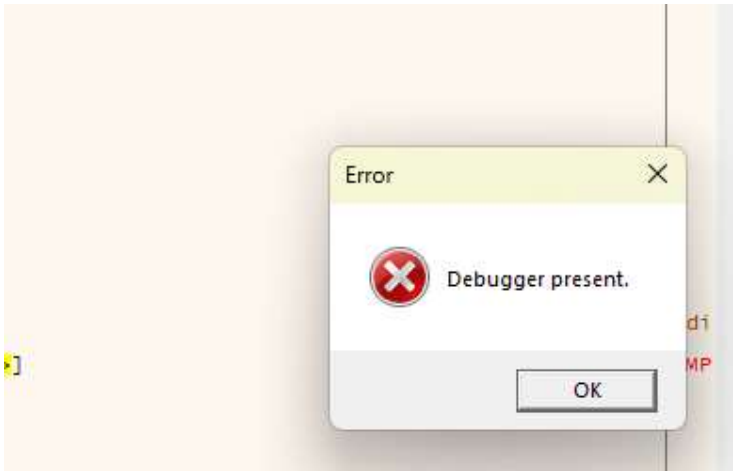


Рисунок 3.52 – Дизасемблювання застосунку

3.3 Аналіз результатів

Після аналізу чотирьох випадків захисту, а саме: не захищений, захищений з використанням алгоритму мутації, захищений з використанням алгоритмів мутації та віртуалізації, захищений з використанням алгоритмів мутації, віртуалізації та компресії можна зробити висновок – навіть при найзахищенішому випадку, коли використовуються усі алгоритми захисту, застосунок який використовував прямі виклики WinAPI все-одно був вразливий до зламу, в той час застосунок з технікою емуляції викликів WinAPI був край важким для аналізу. Це може свідчити про те що така техніка має право на існування і є досі ефективною та може і повинна використовуватися розробниками при проектуванні захисних механізмів таких як: перевірка ліцензії, детектування налагоджувача, тощо.

Сутність бібліотеки саме говорить про те що не треба відмовлятися від WinAPI взагалі, але якщо говорити про аспект саме захисту програмного забезпечення, то використання бібліотеки є ключовим аспектом.

Висновки до розділу 3

У цьому була розроблена бібліотека для емуляції викликів WinAPI на мові C++, яка дозволяє замінювати оригінальні виклики на емульовані виклики з відповідною поведінкою.

Було проведено порівняльний аналіз двох додатків з використанням прямих викликів WinAPI та з емульованими викликами WinAPI, де використання емульованих викликів не сильно вплинуло на продуктивність додатку. Бібліотека була також успішно застосована для захисту програм від зворотного проектування за допомогою комерційного рішення VMProtect.

Отже, бібліотека для емуляції викликів WinAPI може бути корисною у випадках, коли потрібно замінити оригінальні виклики на емульовані виклики з метою захисту програм від зворотного проектування та підвищення безпеки програмного забезпечення.

4 РОЗРОБЛЕННЯ СТАРТАП ПРОЕКТУ

4.1 Опис ідеї стартап-проекту

Опис ідеї, напряму та вигоди стартап-проекту.

Таблиця 4.1 — Опис ідеї, напряму та вигоди стартап-проекту

Зміст ідеї	Напрям застосування	Вигоди для користувачів
Розробка емуляції викликів WinAPI	Компанії та підприємства	Гнучке налаштування
		Проста інтеграція

Таблиця 4.2 - Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№ п/п	Техніко-економічні характеристики ідеї	Потенційні товари/концепції конкурентів		W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Мій проект	Конкурент			
1	Швидкодія	Значно менше використання інструкцій	Більше використання інструкцій			+
2	Інтеграція	Проста до інтеграції, можливість модифікувати бібліотеку	Закритий код, неможливо зробити так як хочеш			+

4.2 Технологічний аудит ідеї проєкту

Технології для створення проекту, визначення присутності необхідних технологій та аналіз цих технологій чи є потрібність у розробці.

Таблиця 4.3 — Технологічна здійсненність ідеї проекту

№ п/п	Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Емуляція викликів WinAPI	Visual Studio	Доробити	Доступні

4.3 Аналіз ринкових можливостей запуску стартап-проєкту

В даному підрозділі проведено аналіз попиту, включаючи наявність, динаміку розвитку ринку та обсяг попиту. Також були визначені ринкові можливості, які можна використати під час впровадження проекту, а також ринкові загрози, які можуть перешкодити реалізації проекту. Результати цього аналізу зібрані в Таблиці 4.4.

Таблиця 4.4 - Характеристика потенційного ринку для проекту

№ п/п	Показники стану ринку	Характеристика
1	Кількість основних гравців	Не відомо
2	Загальний обсяг продажів грн/ум.од.	12 ум. Од.
3	Динаміка ринку	Зростає
4	Наявність обмежень для виходу	Немає
5	Специфічні умови для сертифікації чи дотримання стандартів	Немає
6	Середня норма рентабельності на ринку, %	>125%

З цього аналізу можна зробити висновок про відкритість ринку, який зараз зростає. Додатково, в даній сфері не було знайдено конкурентів, що робить ринок інвестиційно привабливим.

Таблиця 4.5 – Потенційні клієнти продукту

No п/п	Потреба, що формує ринок	Цільова аудиторія	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
1	Захист програмного забезпечення від зламу	Розробники програмного забезпечення	-	Швидкодія, проста імплементація

Таблиця 4.6 - Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1	Неправильна робота програмного забезпечення	Використання бібліотеки може призвести до зміни поведінки програмного забезпечення	Оптимізація функцій, додавання підтримки інших функцій
2	Відсутність попиту	Використання комерційних рішень	Участь в спеціалізованих конференціях, запуск рекламної компанії

Таблиця 4.7 – Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії

1	Удосконалення техніки	Підтримка інших систем	Впровадження нового функціоналу

Таблиця 4.8 – Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив діяльності підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
Вказати тип конкуренції - олігополія	Відсутня конкуренція	Вузька спеціалізація забезпезпечує якість і актуальністю
За рівнем конкурентної боротьби -національна	Послуги надаватимуться також за межами території України	Вдосконалення техніки
За галуззю - міжгалузева	В послугах зацікавлені компанії із різних галузей	Пошук нових клієнтів
За видами товарів - товарно-видова	Надана послуга є дуже специфічною, що робить її унікальною	Надання послуг у сфері кібербезпеки
За характером конкурентних переваг - цінова	Основна мотивація для замовників – забезпечення безпеки власної інфраструктури, швидке реагування на можливі атаки	Надання послуг

За інтенсивностю - марочна	Привабливість послуги	Вдосконалення техніки

Таблиця 4.9 - Аналіз галузевої конкуренції за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти в галузі	Постачальники	Клієнти	Товари замінники
	Немає	Компанії з надання послуг кібербезпеки	Компанії з надання послуг кібербезпеки	Підприємства, стурбовані станом власної кібербезпеки	Немає
Висновки	Низька інтенсивність боротьби за ринок	Висока можливість успішного виходу на ринок	Не диктують умови роботи ринку	Стурбовані станом власної кібербезпеки	Немає обмежень на ринку

Таблиця 4.10 - Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування
1	Якість продукту	Максимально оптимізована і має високу швидкодію
2	Наявність прямих конкурентів	Аналогічних спеціалізованих послуг немає в вільному продажі

Таблиця 4.11 - Порівняльний аналіз сторін проекту

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з … (назва підприємства)						
			3	2	1	0	1	2	3
1	Наявність прямих конкурентів – унікальний продукт	15				+			
2	Низька ціна	15				+			

Таблиця 4.12 — SWOT- аналіз стартап-проекту

Сильні сторони: завдяки вузькій спеціалізації є проєкт є унікальним. Також перевагами є ціна , відсутність конкуренції та можливість застосування у різних галузях	Слабкі сторони: занадто вузька спеціалізація
Можливості: можливості розширення спектру послуг.	Загрози: поява конкурентів

Таблиця 4.13 — Альтернативи ринкового впровадження стартап-проекту

№ п/п	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Оформлення патенту	Висока	До 6 місяців

2	Реклама	Середня	До 4 місяців
3	Підписання договору з клієнтами	Висока	До 1 року
4	Вдосконалення методики і реалізації	Висока	До 1 року

4.4 Розроблення ринкової стратегії проєкту

Таблиця 4.14 — Вибір цільових груп потенційних споживачів

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах груп	Інтенсивність конкуренції в секторі	Простота входу
1	Розробники програмного забезпечення	Є попит	Середній	Низька	Низька

Таблиця 4.15 - Визначення базової стратегії розвитку

Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
Популяризація власних послуг завдяки переконання клієнту у необхідності наших послуг	Участь у конференціяї, реклама	Великі перспективи та визначні якості наявного рішення	Підвищувати якість надаваних послуг, активна рекламна кампанія

Таблиця 4.16 — Визначення базової стратегії конкурентної поведінки

Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки
Ні	Нових і старих	Ні	Стратегія лідера

Таблиця 4.17 — Визначення стратегії позиціонування

Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)
Швидкість, інтеграція	Стратегія спеціалізації	-	-

4.5 Розроблення маркетингової програми проекту

Таблиця 4.18 — Визначення ключових переваг концепції потенційного товару

Потреби	Вигоди, що пропонує товар	Ключові переваги над конкурентами
Максимальний захист програмного забезпечення від зламу	Швидкість та інтеграція	Вузька спеціалізація

Таблиця 4.19 - Опис 3 рівнів моделі товару

Рівні	Сутність та складові		
Товар за задумом	Бібліотека для емуляції викликів WinAPI		
Товар у реальному виконанні	Властивості		
	Емулює виклики WinAPI		
Товар із підкріпленням	До продажу: активне рекламування власних послуг		
	Після продажу: участь у спеціалізованиє конференціх, реклама власного продукту		
За рахунок чого потенційний товар буде захищено від копіювання: патент			

Таблиця 4.20 — Визначення меж встановлення ціни

Рівень цін на товари - замінники	Рівень цін на товари - аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі
-	-	Високий	10000-20000 ум. Од.

Таблиця 4.21 - Формування системи збуту

Специфіка закупівельної поведінки цільових груп	Функції збуту, які має виконувати постачальник	Глибина каналів збуту	Оптимальна система збуту
Власна система збуту	Консультації	Продаж користувачам	Продаж без посередників

Таблиця 4.22 — Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	Канали комінукацій якими користуються клієнти	Ключові пропозиції для позиціонування	Завдання рекламного повідомлення
Стримані у виборі нових послуг	Корпоративна пошта	Ефективна емуляція викликів WinAPI, швидкодія	Масова розсилка

Висновки до розділу 4

У четвертому розділі дипломної роботи було проаналізовано перспективи запуску стартап-проекту, який надає послуги з захисту програмного забезпечення. В ході дослідження було встановлено, що проект можливий, але може стикнутися з певними труднощами.

Конкуренція в даній галузі наразі не є високою, що дає стартапу можливість мати успіх при виході на ринок.

Отже, аналіз перспектив для запуску стартап-проекту показав, що він може бути успішним, хоча потребує уважного урахування певних труднощів, які можуть виникнути під час реалізації проекту.

ВИСНОВКИ

В результаті проведеної роботи були детально проаналізовані алгоритми захисту програмного забезпечення від зворотної розробки та протидії налагодженню, зокрема такі як: алгоритми заплутування, алгоритми мутації, алгоритми віртуалізації, алгоритми компресії та техніки і методи протидії налагодженню. Після аналізу всіх технік і методів їх було порівняно за критерієм.

Після порівняння було запропоновано розробити та імплементувати техніку емуляції викликів WinAPI для поліпшення існуючих технік та методів протидії налагодження. Під час розробки бібліотеки виникли труднощі пов’язані із специфікою системи, а також різна розрядність як і системи Windows так і додатків.

Після розробки та імплементації бібліотеки було порівняно два додатки, які мають однакову функцію, однак перший використовував прямі виклики WinAPI, в той час як другий земулюванні. Було продемонстровано вразливість у першому навіть при максимально захищеному рівні, в той час як для зламу другого потрібно було аналізувати віртуалізацію, мутацію, компресію у рішені VMProtect, щоб зачепитись хоча-б за щось. Згідно результатів бібліотека показала себе дуже ефективно та є дуже простою у використанні.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Zhang, B. "Research Summary of Anti-debugging Technology" / B. Zhang. In Journal of Physics Conference Series 1744(4):042186.
2. Collberg, C.S., Thomborson, C.D., & Low, D. "A Taxonomy of Obfuscating Transformations" / C.S. Collberg, C.D. Thomborson, D. Low.
3. You, I., & Yim, K. "Malware Obfuscation Techniques: A Brief Survey" / I. You, K. Yim. In 2010 International Conference on Broadband, Wireless Computing, Communication and Applications, 297-300.
4. Jesse, C., Rabek, R., Khazan, , Lewandowski, S., Cunningham, R. "Detection of Injected Dynamically Generated, and Obfuscated Malicious Code" / C. Jesse, R. Rabek, S. Khazan, S. Lewandowski, R. Cunningham. In Proceedings of the 2003 ACM Workshop on Rapid Malcode.
5. Hewardt, M., Pravat, D. "Advanced Windows Debugging" / M. Hewardt, D. Pravat.
6. Richter, J., Nasarre, C. "Windows via C/C++" / J. Richter, C. Nasarre.
7. Russinovich, M., Solomon, D., Ionescu, A. "Windows Internals" / M. Russinovich, D. Solomon, A. Ionescu.
8. Hart, J. M. "Windows System Programming" / J. M. Hart.
9. Eilam, E. "Reversing: Secrets of Reverse Engineering" / E. Eilam.
10. Sikorski, M., Honig, A. "Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software" / M. Sikorski, A. Honig.
11. Blunden, B. "The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System" / B. Blunden.
12. Seitz, J. "Gray Hat Python: Python Programming for Hackers and Reverse Engineers" / J. Seitz.
13. Eagle, C. "The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler" / C. Eagle.

14. Dang, B., Gazet, A., Bachaalany, E. "Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation" / B. Dang, A. Gazet, E. Bachaalany.
15. Petzold, C. "Windows API Guide" / C. Petzold.
16. Simon, R. J., Hahn, E. S. "The Windows API Bible: A Comprehensive Reference for Programmers" / R. J. Simon, E. S. Hahn.
17. Marak, V., Josevski, B. "Windows Malware Analysis Essentials" / V. Marak, B. Josevski.
18. Dang, B., Gazet, A. "Practical Reverse Engineering: Reversing Obfuscation" / B. Dang, A. Gazet.
19. Yurichev, D. "Reversing: Secrets of Reverse Engineering for Beginners" / D. Yurichev.
20. Hyde, R. "The Art of Assembly Language" / R. Hyde.
21. Carvey, H. "Windows Forensic Analysis Toolkit: Advanced Analysis Techniques for Windows 10" / H. Carvey.
22. Bazhaniuk, O. "Reversing: Secrets of Reverse Engineering for Beginners and Experts" / O. Bazhaniuk.
23. Sikorski, M., Honig, A. "Practical Malware Analysis: A Hands-On Guide to Dissecting Malicious Software" / M. Sikorski, A. Honig.
24. Рейценштейн, К., & Гальчинський, Л. (2023). ТЕХНІКА ЕМУЛЯЦІЇ ВИКЛИКІВ WINAPI У ЗАСТОСУНКАХ WINDOWS ЯК МЕТОД ПРОТИДІЇ ЗВОРОТНОЇ РОЗРОБКИ. Grail of Science, (27), 361-366.
25. Hoglund, G., Butler, J. "Rootkits: Subverting the Windows Kernel" / G. Hoglund, J. Butler.

ДОДАТОК А

```
#include <Windows.h>
#include <string>
#include <iostream>
#include "AntiDebugLib.h"

int main()
{
    AntiDebugLib* adbg_instance = new AntiDebugLib();

    auto func =
reinterpret_cast<ZwQueryInformationProcessWrapper>(adbg_instance-
>get_syscall_addr(ZwQueryInformationProcess));

    DWORD is_debugger_present = 0x1337;
    auto process_debug_port = 7;

    if (func(CURRENT_PROCESS, process_debug_port,
&is_debugger_present, sizeof(DWORD), NULL) != 0x0 || is_debugger_present !=
0)
    {
        std::cout << "Debugger detected" << std::endl;
    }
    else
    {
        std::cout << "Debugger is not detected" << std::endl;
    }

    getchar();
```

```
    return true;
}
#pragma once

#include <winternl.h>
#include "ntapi_typedefs.h"
#include "Crc32.h"

#define MAX_SYSCALL_CALL_SIZE 32
#define SUPPORTED_FUNCTIONS_AMOUNT 6
enum SUPPORTED_FUNCTIONS
{
    ZwAllocateVirtualMemory,
    ZwProtectVirtualMemory,
    ZwFreeVirtualMemory,
    ZwRaiseHardError,
    ZwTerminateProcess,
    ZwQueryInformationProcess
};

const int function_hashes[SUPPORTED_FUNCTIONS_AMOUNT] =
{
    HASH("ZwAllocateVirtualMemory"),
    HASH("ZwProtectVirtualMemory"),
    HASH("ZwFreeVirtualMemory"),
    HASH("ZwRaiseHardError"),
    HASH("ZwTerminateProcess"),
    HASH("ZwQueryInformationProcess")
};
const int wow64transition = HASH("Wow64Transition");
```

```
class AntiDebugLib
{
private:
    bool is_wow64 = false;

    WORD syscall_table[SUPPORTED_FUNCTIONS_AMOUNT];
    void* syscall_addresses[SUPPORTED_FUNCTIONS_AMOUNT];

    inline void allocate_syscall(void* buffer_addr, SUPPORTED_FUNCTIONS
index)
    {
        auto temp = reinterpret_cast<char*>(buffer_addr);
        auto temp_syscall_index =
reinterpret_cast<char*>(&syscall_table[index]);

        temp[0] = 0xB8;
        temp[1] = temp_syscall_index[0];
        temp[2] = temp_syscall_index[1];
        temp[3] = 0x0;
        temp[4] = 0x0;

#ifdef _WIN64
        // mov eax, syscall_index
        // mov r10, rcx
        // syscall
        // ret
        temp[5] = 0x4C;
```

```
            temp[6] = 0x8B;
            temp[7] = 0xD1;
            temp[8] = 0x0F;
            temp[9] = 0x05;
            temp[10] = 0xC3;
    #else
            if (is_wow64)
            {
                    // mov eax, syscall_index
                    // push eax
                    // mov eax, dword ptr fs:[0xC0] (pointer in TEB to
Wow64Transition)
                    // cmp byte [eax + 5], 0x33
                    // pop eax
                    // je continue
                    // mov ax,0xF1
                    // ret
                    // continue:
                    // call dword ptr fs:[0xC0] (pointer in TEB to
Wow64Transition)
                    // ret

                    temp[5] = 0x50;
                    temp[6] = 0x64;
                    temp[7] = 0xA1;
                    temp[8] = 0xC0;
                    temp[9] = 0x00;
                    temp[10] = 0x00;
                    temp[11] = 0x00;
                    temp[12] = 0x80;
```

```
        temp[13] = 0x78;
        temp[14] = 0x05;
        temp[15] = 0x33;
        temp[16] = 0x58;
        temp[17] = 0x74;
        temp[18] = 0x05;
        temp[19] = 0x66;
        temp[20] = 0xB8;
        temp[21] = 0xF1;
        temp[22] = 0x00;
        temp[23] = 0xC3;
        temp[24] = 0x64;
        temp[25] = 0xFF;
        temp[26] = 0x15;
        temp[27] = 0xC0;
        temp[28] = 0x00;
        temp[29] = 0x00;
        temp[30] = 0x00;
        temp[31] = 0xC3;
}
else
{
        // mov eax, syscall_index
        // mov ebx, esp
        // sysenter
        // ret
        temp[5] = 0x8B;
        temp[6] = 0xD4;
        temp[7] = 0x0F;
        temp[8] = 0x34;
```

```
                temp[9] = 0xC3;
            }
    #endif
        }

        inline void find_syscall_index(void* func_ptr, int index)
        {
            auto temp = reinterpret_cast<BYTE*>(func_ptr);

            while (*temp != 0xC3)
            {
                if (*temp == 0xB8)
                {
                    syscall_table[index] =
*reinterpret_cast<WORD*>(++temp);
                    break;
                }

                temp++;
            }
        }
    public:
        AntiDebugLib()
        {
    #ifdef _WIN64
            auto PEB = ((PTEB)__readgsqword(offsetof(NT_TIB, Self)))-
>ProcessEnvironmentBlock;
    #else
            auto PEB = ((PTEB)__readfsdword(offsetof(NT_TIB, Self)))-
>ProcessEnvironmentBlock;
```

```
    #endif
            auto ntdll_entry = CONTAINING_RECORD(PEB->Ldr->InMemoryOrderModuleList.Flink->Flink, LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks);
    #ifdef _WIN64
            auto ntdll_handle = reinterpret_cast<DWORD64>(ntdll_entry->DllBase);
    #else
            auto ntdll_handle = reinterpret_cast<DWORD>(ntdll_entry->DllBase);
    #endif

            auto ntdll_idh = reinterpret_cast<IMAGE_DOS_HEADER*>(ntdll_handle);
            auto ntdll_nth = reinterpret_cast<IMAGE_NT_HEADERS*>(ntdll_handle + ntdll_idh->e_lfanew);
            auto ntdll_ioh = ntdll_nth->OptionalHeader;
            auto ntdll_ied = reinterpret_cast<IMAGE_EXPORT_DIRECTORY*>(ntdll_handle + ntdll_ioh.DataDirectory[0].VirtualAddress);

            auto func_names = reinterpret_cast<DWORD*>(ntdll_handle + ntdll_ied->AddressOfNames);
            auto func_addresses = reinterpret_cast<DWORD*>(ntdll_handle + ntdll_ied->AddressOfFunctions);
            auto func_ordinals = reinterpret_cast<WORD*>(ntdll_handle + ntdll_ied->AddressOfNameOrdinals);

            for (auto i = 0; i < ntdll_ied->NumberOfFunctions; i++)
            {
```

```
                auto func_hash = HASH_RUNTIME(reinterpret_cast<const
char*>(ntdll_handle + func_names[i]));

                if (func_hash == wow64transition)
                {
                        is_wow64 = true;
                }

                for (auto j = 0; j < SUPPORTED_FUNCTIONS_AMOUNT;
j++)
                {
                        if (func_hash == function_hashes[j])
                        {
                                auto func_addr =
reinterpret_cast<void*>(ntdll_handle + func_addresses[func_ordinals[i]]);
                                find_syscall_index(func_addr, j);
                        }
                }
        }

        auto syscall_call = (char*)malloc(MAX_SYSCALL_CALL_SIZE);
        auto syscall_call_protection =
(char*)malloc(MAX_SYSCALL_CALL_SIZE);

        this->allocate_syscall(syscall_call, ZwAllocateVirtualMemory);
        this->allocate_syscall(syscall_call_protection,
ZwProtectVirtualMemory);

        DWORD old_protection = NULL;
```

```
        VirtualProtect(syscall_call, MAX_SYSCALL_CALL_SIZE,
PAGE_EXECUTE_READWRITE, &old_protection);
        VirtualProtect(syscall_call_protection,
MAX_SYSCALL_CALL_SIZE, PAGE_EXECUTE_READWRITE,
&old_protection);

        for (auto i = 0; i < SUPPORTED_FUNCTIONS_AMOUNT; i++)
        {
            SIZE_T mem_size = MAX_SYSCALL_CALL_SIZE;
            void* new_addr = NULL;

    reinterpret_cast<ZwAllocateVirtualMemoryWrapper>(syscall_call)(CURRE
NT_PROCESS, &new_addr, NULL, &mem_size,
                    MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);
                this->allocate_syscall(new_addr,
SUPPORTED_FUNCTIONS(i));

    reinterpret_cast<ZwProtectVirtualMemoryWrapper>(syscall_call_protection
)(CURRENT_PROCESS, &new_addr, &mem_size, PAGE_EXECUTE_READ,
                    &old_protection);
                syscall_addresses[i] = new_addr;
        }

        free(syscall_call);
        free(syscall_call_protection);
    }

    inline void* get_syscall_addr(SUPPORTED_FUNCTIONS index)
    {
```

```
            return syscall_addresses[index];
        }
    };

    #ifndef COMPILE_TIME_CRC_H
    #define COMPILE_TIME_CRC_H

    #include <cstdint>
    #include <type_traits>


    namespace crcdetail
    {
       static constexpr const uint32_t table[256] =
       {
          0x00000000U, 0x77073096U, 0xEE0E612CU, 0x990951BAU,
0x076DC419U,
          0x706AF48FU, 0xE963A535U, 0x9E6495A3U, 0x0EDB8832U,
0x79DCB8A4U,
          0xE0D5E91EU, 0x97D2D988U, 0x09B64C2BU, 0x7EB17CBDU,
0xE7B82D07U,
          0x90BF1D91U, 0x1DB71064U, 0x6AB020F2U, 0xF3B97148U,
0x84BE41DEU,
          0x1ADAD47DU, 0x6DDDE4EBU, 0xF4D4B551U, 0x83D385C7U,
0x136C9856U,
          0x646BA8C0U, 0xFD62F97AU, 0x8A65C9ECU, 0x14015C4FU,
0x63066CD9U,
          0xFA0F3D63U, 0x8D080DF5U, 0x3B6E20C8U, 0x4C69105EU,
0xD56041E4U,
```

0xA2677172U, 0x3C03E4D1U, 0x4B04D447U, 0xD20D85FDU, 0xA50AB56BU,
0x35B5A8FAU, 0x42B2986CU, 0xDBBBC9D6U, 0xACBCF940U, 0x32D86CE3U,
0x45DF5C75U, 0xDCD60DCFU, 0xABD13D59U, 0x26D930ACU, 0x51DE003AU,
0xC8D75180U, 0xBFD06116U, 0x21B4F4B5U, 0x56B3C423U, 0xCFBA9599U,
0xB8BDA50FU, 0x2802B89EU, 0x5F058808U, 0xC60CD9B2U, 0xB10BE924U,
0x2F6F7C87U, 0x58684C11U, 0xC1611DABU, 0xB6662D3DU, 0x76DC4190U,
0x01DB7106U, 0x98D220BCU, 0xEFD5102AU, 0x71B18589U, 0x06B6B51FU,
0x9FBFE4A5U, 0xE8B8D433U, 0x7807C9A2U, 0x0F00F934U, 0x9609A88EU,
0xE10E9818U, 0x7F6A0DBBU, 0x086D3D2DU, 0x91646C97U, 0xE6635C01U,
0x6B6B51F4U, 0x1C6C6162U, 0x856530D8U, 0xF262004EU, 0x6C0695EDU,
0x1B01A57BU, 0x8208F4C1U, 0xF50FC457U, 0x65B0D9C6U, 0x12B7E950U,
0x8BBEB8EAU, 0xFCB9887CU, 0x62DD1DDFU, 0x15DA2D49U, 0x8CD37CF3U,
0xFBD44C65U, 0x4DB26158U, 0x3AB551CEU, 0xA3BC0074U, 0xD4BB30E2U,
0x4ADFA541U, 0x3DD895D7U, 0xA4D1C46DU, 0xD3D6F4FBU, 0x4369E96AU,
0x346ED9FCU, 0xAD678846U, 0xDA60B8D0U, 0x44042D73U, 0x33031DE5U,

0xAA0A4C5FU, 0xDD0D7CC9U, 0x5005713CU, 0x270241AAU, 0xBE0B1010U,

0xC90C2086U, 0x5768B525U, 0x206F85B3U, 0xB966D409U, 0xCE61E49FU,

0x5EDEF90EU, 0x29D9C998U, 0xB0D09822U, 0xC7D7A8B4U, 0x59B33D17U,

0x2EB40D81U, 0xB7BD5C3BU, 0xC0BA6CADU, 0xEDB88320U, 0x9ABFB3B6U,

0x03B6E20CU, 0x74B1D29AU, 0xEAD54739U, 0x9DD277AFU, 0x04DB2615U,

0x73DC1683U, 0xE3630B12U, 0x94643B84U, 0x0D6D6A3EU, 0x7A6A5AA8U,

0xE40ECF0BU, 0x9309FF9DU, 0x0A00AE27U, 0x7D079EB1U, 0xF00F9344U,

0x8708A3D2U, 0x1E01F268U, 0x6906C2FEU, 0xF762575DU, 0x806567CBU,

0x196C3671U, 0x6E6B06E7U, 0xFED41B76U, 0x89D32BE0U, 0x10DA7A5AU,

0x67DD4ACCU, 0xF9B9DF6FU, 0x8EBEEFF9U, 0x17B7BE43U, 0x60B08ED5U,

0xD6D6A3E8U, 0xA1D1937EU, 0x38D8C2C4U, 0x4FDFF252U, 0xD1BB67F1U,

0xA6BC5767U, 0x3FB506DDU, 0x48B2364BU, 0xD80D2BDAU, 0xAF0A1B4CU,

0x36034AF6U, 0x41047A60U, 0xDF60EFC3U, 0xA867DF55U, 0x316E8EEFU,

0x4669BE79U, 0xCB61B38CU, 0xBC66831AU, 0x256FD2A0U, 0x5268E236U,

0xCC0C7795U, 0xBB0B4703U, 0x220216B9U, 0x5505262FU, 0xC5BA3BBEU,

```
        0xB2BD0B28U, 0x2BB45A92U, 0x5CB36A04U, 0xC2D7FFA7U,
0xB5D0CF31U,
        0x2CD99E8BU, 0x5BDEAE1DU, 0x9B64C2B0U, 0xEC63F226U,
0x756AA39CU,
        0x026D930AU, 0x9C0906A9U, 0xEB0E363FU, 0x72076785U,
0x05005713U,
        0x95BF4A82U, 0xE2B87A14U, 0x7BB12BAEU, 0x0CB61B38U,
0x92D28E9BU,
        0xE5D5BE0DU, 0x7CDCEFB7U, 0x0BDBDF21U, 0x86D3D2D4U,
0xF1D4E242U,
        0x68DDB3F8U, 0x1FDA836EU, 0x81BE16CDU, 0xF6B9265BU,
0x6FB077E1U,
        0x18B74777U, 0x88085AE6U, 0xFF0F6A70U, 0x66063BCAU,
0x11010B5CU,
        0x8F659EFFU, 0xF862AE69U, 0x616BFFD3U, 0x166CCF45U,
0xA00AE278U,
        0xD70DD2EEU, 0x4E048354U, 0x3903B3C2U, 0xA7672661U,
0xD06016F7U,
        0x4969474DU, 0x3E6E77DBU, 0xAED16A4AU, 0xD9D65ADCU,
0x40DF0B66U,
        0x37D83BF0U, 0xA9BCAE53U, 0xDEBB9EC5U, 0x47B2CF7FU,
0x30B5FFE9U,
        0xBDBDF21CU, 0xCABAC28AU, 0x53B39330U, 0x24B4A3A6U,
0xBAD03605U,
        0xCDD70693U, 0x54DE5729U, 0x23D967BFU, 0xB3667A2EU,
0xC4614AB8U,
        0x5D681B02U, 0x2A6F2B94U, 0xB40BBE37U, 0xC30C8EA1U,
0x5A05DF1BU,
        0x2D02EF8DU
    };
```

```
    constexpr uint32_t compute(const char* data, uint32_t len, uint32_t crc = 0)
    {
        crc = crc ^ 0xFFFFFFFFU;
        for (uint32_t i = 0; i < len; i++)
        {
            crc = table[(BYTE)*data ^ (crc & 0xFF)] ^ (crc >> 8);
            data++;
        }
        crc = crc ^ 0xFFFFFFFFU;
        return crc;
    }

    uint32_t compute_runtime(const char* data, uint32_t len, uint32_t crc = 0)
    {
        crc = crc ^ 0xFFFFFFFFU;
        for (uint32_t i = 0; i < len; i++)
        {
            crc = table[(BYTE)*data ^ (crc & 0xFF)] ^ (crc >> 8);
            data++;
        }
        crc = crc ^ 0xFFFFFFFFU;
        return crc;
    }
}  // namespace crcdetail

// Macro to be used, guarantees hash is computed at compile time.
#define HASH(A)                                                 \
    std::integral_constant<uint32_t, crcdetail::compute(A, sizeof(A)-1)>::value
```

```
    #define HASH_RUNTIME(A)                                    \
        crcdetail::compute_runtime(A, strlen(A))

    #endif  // COMPILE_TIME_CRC_H
    #pragma once

    #ifdef _WIN64
    #define CALL_CONVENTION __stdcall
    #define CURRENT_PROCESS (HANDLE)0xFFFFFFFFFFFFFFFF
    #else
    #define CALL_CONVENTION __cdecl
    #define CURRENT_PROCESS (HANDLE)0xFFFFFFFF
    #endif

    typedef NTSTATUS(CALL_CONVENTION*
ZwAllocateVirtualMemoryWrapper)(HANDLE ProcessHandle, PVOID*
BaseAddress, ULONG_PTR ZeroBits, PSIZE_T RegionSize,
        ULONG AllocationType,
        ULONG Protect);
    typedef NTSTATUS(CALL_CONVENTION*
ZwProtectVirtualMemoryWrapper)(HANDLE ProcessHandle, PVOID*
BaseAddress, PSIZE_T NumberOfBytesToProtect,
        ULONG NewAccessProtection,
        PULONG OldAccessProtection);
    typedef NTSTATUS(CALL_CONVENTION*
ZwFreeVirtualMemoryWrapper)(HANDLE ProcessHandle, PVOID*
BaseAddress, PSIZE_T RegionSize, ULONG FreeType);
    typedef NTSTATUS(CALL_CONVENTION*
ZwTerminateProcessWrapper)(HANDLE ProcessHandle, NTSTATUS
ExitStatus);
```

```
    typedef NTSTATUS(CALL_CONVENTION*
ZwQueryInformationProcessWrapper)(HANDLE ProcessHandle, UINT
ProcessInformationClass, PVOID ProcessInformation,
        ULONG ProcessInformationLength,
        PULONG ReturnLength);
```

ДОДАТОК Б

```
#include <iostream>

#include <Windows.h>

#include "MinHook.h"
#pragma comment (lib, "libMinHook.x86.lib")

typedef int (__stdcall* MyMessageBoxA)(HWND  hWnd, LPCSTR lpText,
LPCSTR lpCaption, UINT uType);
MyMessageBoxA msg_box_addr_org;

int MessageBoxA_hooked(HWND  hWnd, LPCSTR lpText, LPCSTR
lpCaption, UINT uType)
{
    return msg_box_addr_org(hWnd, "I do not Love KPI!", "Error",
MB_ICONERROR);
}

int main()
{
    // not hooked
    MessageBoxA(NULL, "I Love KPI!", "Information",
MB_ICONINFORMATION);
    //

    // getting msg box address
    auto msg_box_addr = GetProcAddress(GetModuleHandleA("user32.dll"),
"MessageBoxA");
```

```
        //

        // initializing minhook
        MH_Initialize();
        //

        // hooking
        MH_CreateHook(msg_box_addr, MessageBoxA_hooked,
(LPVOID*)&msg_box_addr_org);
        //

        // enabling hooks
        MH_EnableHook(MH_ALL_HOOKS);
        //

        // calling the same function
        MessageBoxA(NULL, "I Love KPI!", "Information",
MB_ICONINFORMATION);
        //

        getchar();
        return true;
    }
```

ДОДАТОК В

```
#include <Windows.h>
#include <iostream>
#include "VMProtectSDK.h"
#pragma comment (lib, "VMProtectSDK64.lib")

char data[10];

int main()
{
    VMProtectBegin("entrypoint");
    BOOL debugger_present = FALSE;

    if (CheckRemoteDebuggerPresent(GetCurrentProcess(),
&debugger_present) && !debugger_present)
    {
        auto file_handle = CreateFileA("cdkey.key", GENERIC_READ,
NULL, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

        if (file_handle)
        {
            DWORD bytes_read = NULL;
            if (ReadFile(file_handle, data, 10, &bytes_read, NULL))
            {
                CloseHandle(file_handle);

                if (strstr(data, "encrypted"))
                {
```

```
                                MessageBoxA(NULL, "License check passed.",
"Success", MB_ICONINFORMATION);
                        }
                        else
                        {
                                MessageBoxA(NULL, "License check is not
passed.", "Error", MB_ICONERROR);
                                ExitProcess(4);
                        }
                }
                else
                {
                        MessageBoxA(NULL, "Failed to read license file.",
"Error", MB_ICONERROR);
                        ExitProcess(3);
                }
        }
        else
        {
                MessageBoxA(NULL, "Failed to open license file.", "Error",
MB_ICONERROR);
                ExitProcess(2);
        }
    }
    else
    {
        MessageBoxA(NULL, "Debugger present.", "Error",
MB_ICONERROR);
        ExitProcess(1);
    }
```

```
    VMProtectEnd();
    getchar();
    return true;
}
```

ДОДАТОК Г

```
#include <Windows.h>
#include <iostream>
#include "VMProtectSDK.h"
#include "AntiDebugLib.h"
#pragma comment (lib, "VMProtectSDK64.lib")

#pragma comment (lib, "ntdll")

char data[10];
wchar_t current_directory[1024];

int main()
{
    VMProtectBegin("entrypoint");
    GetCurrentDirectoryW(1024, current_directory);

    AntiDebugLib* adbg_instance = new AntiDebugLib();

    auto ZwQueryInformationProcess_addr =
reinterpret_cast<ZwQueryInformationProcessWrapper>(adbg_instance-
>get_syscall_addr(ZwQueryInformationProcess));
    auto ZwCreateFile_addr =
reinterpret_cast<ZwCreateFileWrapper>(adbg_instance-
>get_syscall_addr(ZwCreateFile));
    auto ZwReadFile_addr =
reinterpret_cast<ZwReadFileWrapper>(adbg_instance-
>get_syscall_addr(ZwReadFile));
```

```
	auto ZwClose_addr = reinterpret_cast<ZwCloseWrapper>(adbg_instance->get_syscall_addr(ZwClose));
	auto ZwRaiseHardError_addr = reinterpret_cast<ZwRaiseHardErrorWrapper>(adbg_instance->get_syscall_addr(ZwRaiseHardError));
	auto ZwTerminateProcess_addr = reinterpret_cast<ZwTerminateProcessWrapper>(adbg_instance->get_syscall_addr(ZwTerminateProcess));

	DWORD64 is_debugger_present = 0x1337;
	auto process_debug_port = 7;
	BOOL debugger_present = FALSE;

	if (ZwQueryInformationProcess_addr(CURRENT_PROCESS, process_debug_port, &is_debugger_present, sizeof(DWORD64),NULL) == 0 && is_debugger_present == 0)
	{
		HANDLE file_handle = NULL;
		OBJECT_ATTRIBUTES* file_data = new OBJECT_ATTRIBUTES();
		UNICODE_STRING* file_name = new UNICODE_STRING();
		IO_STATUS_BLOCK* comp_delegate = new IO_STATUS_BLOCK();

		std::wstring license_file_path = L"\\DosDevices\\" + std::wstring(current_directory) + L"\\cdkey.key";

		RtlInitUnicodeString(file_name, license_file_path.c_str());
```

```
        InitializeObjectAttributes(file_data, file_name,
OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);

        ZwCreateFile_addr(&file_handle, GENERIC_READ, file_data,
comp_delegate, NULL,
                FILE_ATTRIBUTE_NORMAL, NULL, FILE_OPEN,
FILE_NON_DIRECTORY_FILE, NULL, NULL);

        if (file_handle)
        {
                LARGE_INTEGER byteOffset = { 0 };
                ZwReadFile_addr(file_handle, NULL, NULL, NULL,
comp_delegate, data, 10, &byteOffset, NULL);

                if (comp_delegate->Status == NULL)
                {
                        ZwClose_addr(file_handle);

                        if (strstr(data, "encrypted"))
                        {
                                UNICODE_STRING* caption = new
UNICODE_STRING();
                                UNICODE_STRING* label = new
UNICODE_STRING();

                                RtlInitUnicodeString(caption, L"License check
passed.");
                                RtlInitUnicodeString(label, L"Success");
```

```
                    ULONG_PTR parameters[] = {
(ULONG_PTR)caption, (ULONG_PTR)label, MB_ICONINFORMATION };

                    ULONG response;

    ZwRaiseHardError_addr(STATUS_SERVICE_NOTIFICATION, 3, 3,
parameters, OptionOk, &response);
                }
                else
                {
                    UNICODE_STRING* caption = new
UNICODE_STRING();
                    UNICODE_STRING* label = new
UNICODE_STRING();

                    RtlInitUnicodeString(caption, L"License check is
not passed.");
                    RtlInitUnicodeString(label, L"Error");

                    ULONG_PTR parameters[] = {
(ULONG_PTR)caption, (ULONG_PTR)label, MB_ICONERROR };

                    ULONG response;

    ZwRaiseHardError_addr(STATUS_SERVICE_NOTIFICATION, 3, 3,
parameters, OptionOk, &response);

    ZwTerminateProcess_addr(CURRENT_PROCESS, 4);
                }
            }
```

```
                else
                {
                        UNICODE_STRING* caption = new
UNICODE_STRING();
                        UNICODE_STRING* label = new
UNICODE_STRING();

                        RtlInitUnicodeString(caption, L"Failed to read license
file.");
                        RtlInitUnicodeString(label, L"Error");

                        ULONG_PTR parameters[] = { (ULONG_PTR)caption,
(ULONG_PTR)label, MB_ICONERROR };

                        ULONG response;

        ZwRaiseHardError_addr(STATUS_SERVICE_NOTIFICATION, 3, 3,
parameters, OptionOk, &response);
                        ZwTerminateProcess_addr(CURRENT_PROCESS, 3);
                }
        }
        else
        {
                UNICODE_STRING* caption = new UNICODE_STRING();
                UNICODE_STRING* label = new UNICODE_STRING();

                RtlInitUnicodeString(caption, L"Failed to open license file.");
                RtlInitUnicodeString(label, L"Error");
```

```
                ULONG_PTR parameters[] = { (ULONG_PTR)caption,
(ULONG_PTR)label, MB_ICONERROR };

                ULONG response;

        ZwRaiseHardError_addr(STATUS_SERVICE_NOTIFICATION, 3, 3,
parameters, OptionOk, &response);
                ZwTerminateProcess_addr(CURRENT_PROCESS, 2);
            }
        }
        else
        {
            UNICODE_STRING* caption = new UNICODE_STRING();
            UNICODE_STRING* label = new UNICODE_STRING();

            RtlInitUnicodeString(caption, L"Debugger present.");
            RtlInitUnicodeString(label, L"Error");

            ULONG_PTR parameters[] = { (ULONG_PTR)caption,
(ULONG_PTR)label, MB_ICONERROR };

            ULONG response;
            ZwRaiseHardError_addr(STATUS_SERVICE_NOTIFICATION, 3,
3, parameters, OptionOk, &response);
            ZwTerminateProcess_addr(CURRENT_PROCESS, 1);
        }
        VMProtectEnd();
        getchar();
        return true;
    }
```