Issue 17, 2025

https://doi.org/10.23939/sisn2025.17.017

УДК 004.4

ЗАХИСТ ПРОГРАМНОГО КОДУ НА ПЛАТФОРМІ .NET

Андрій Фечан[1], Олексій Ховерко[2]

[1, 2] Національний університет "Львівська політехніка"
Кафедра програмного забезпечення, Львів, Україна
[1] E-mail: andrii.v.fechan@lpnu.ua, ORCID:0000-0001-9970-5497
[2] E-mail: oleksii.y.khoverko@lpnu.ua, ORCID:0009-0007-1831-6627

В роботі проведено дослідження функціонування складних інформаційних систем, що спрямоване на аналіз існуючих методів захисту, розробку концепції механізму трансформації коду, яка забезпечить високий рівень захисту .NET застосунків. Розглянуто архітектуру .NET Framework із сучасними універсальними вдосконаленнями. Один з важливих елементів є Base Class Library (BCL), що є набором базових класів і бібліотек, які забезпечують стандартні функціональні можливості, такі як робота з файлами, мережею, базами даних, опрацювання тексту, а також базові структури даних. Визначено, що обфускація є одним із найпоширеніших методів захисту програмного забезпечення, що полягає у модифікації вихідного або проміжного коду таким чином, щоб зберегти його функціональність, але зробити його важким для розуміння. Встановлено основні особливості вдосконалення коду, що полягає у заплутуванні потоків виконання, шифруванні рядкових літералів, використання пакерів і криптерів. Характерною особливістю для пакерів і криптерів є пряма інжекція в пам'ять, що відображає свою функціональність у розпакуванні або розшифруванні коду програми безпосередньо в оперативну пам'ять, яке ускладнює зняття дампу процесу. Окрім того, заплутування потоків виконання до-зволяє модифікувати логіку роботи програмного забезпечення завдяки вставлення зайвих умов, циклів або переходів, що створює складні для розуміння структури виконання. Запропоновано процес антивідлагодження, що є одним із ключових компонентів захисту програмного забезпечення від реверс-інжинірингу. Сформовано концептуальний підхід до реалізації механізмів антивідлагодження програмного забезпечення, що полягає у багаторівневому захисті програмного забезпечення і змінює інструкції та поведінку віртуальної машини. Одні з важливих елементів таких алгоритмів є перевірка батьківських процесів та виявлення відлагодження за допомогою апаратних точок зупинки. Це, своєю чергою, відкриває можливість під час проектування віртуальної машини у поєднанні з іншими техніками створювати значні перешкоди для аналізу програмного коду.

Ключові слова – алгоритм, програмний код, віртуальна машина, обфускація, .NET Framework.

Постановка проблеми

З розвитком інформаційних технологій усе більшої актуальності набуває проблема захисту програмного забезпечення від несанкціонованого доступу, декомпіляції та реверс-інжинірингу. Особливо вразливими до подібних загроз є застосунки, розроблені на платформі .NET, оскільки їхній проміжний MSIL-код легко піддається аналізу та зворотному дослідженню за допомогою спеціалізованих інструментів (Kienle,2010, Francisca, 2012). Це створює серйозні ризики для розробників, оскільки зловмисники можуть отримати доступ до вихідного коду, викрасти інтелектуальну власність або модифікувати програму для шкідливих цілей.

Сучасні методи захисту .NET застосунків, такі як обфускація коду (Tang, 2018), використання криптерів і пакерів, мають певні обмеження. Обфускація ускладнює розуміння коду, але не гарантує його повний захист (Golovko, 2024). Пакери та криптери тимчасово приховують логіку роботи

застосунка, проте досвідчені фахівці з реверс-інжинірингу можуть обійти ці механізми. Віртуальні машини (Fang, 2011), як засіб захисту, пропонують більш складний і надійний підхід шляхом перетворення коду в нестандартний формат, який важко піддається аналізу (Anckaert,2006).

Аналіз безпеки роботи програмного забезпечення, шляхом виявлення критичних причин незахищеності (вразливостей) вбудованих мікропрограм або, іншими словами, інженерія програмного забезпечення, є основою дослідження останніх років. Одним із шляхів такого дослідження є зворотне дослідження програмного забезпечення, що є процесом аналізу готового продукту або системи для виявлення його внутрішньої структури, принципів роботи чи способу виготовлення (Ghosh, 2013). У програмуванні це зазвичай означає аналіз програмного коду або виконуваних файлів для розуміння їхньої логіки, виявлення вразливостей або відтворення функціоналу без вихідного коду.

На сьогодні "цифровізація" сягає бурхливого розвитку, оскільки чи не кожний стикається з роботою електронних пристроїв таких як комп'ютер, смартфон тощо. Під час роботи таких пристроїв вирішується прості повсякденні задачі, невід'ємною частиною рішень яких є користування Інтернетом у вигляді перегляду Веб-сторінок, надсилання та отримання пошти, доступ до соціальних мереж та ін. За останні десятиліття розвиток екосистеми Інтернету речей (IoT), яка ввійшла в повсякденне життя, створює проблеми, пов'язані з безпечною роботою широкої номенклатури пристроїв за певними алгоритмами (Nadir,2022). В основі роботи таких пристроїв лежить цифрове програмне забезпечення. Загалом програма – це спосіб "навчити" комп'ютери (і подібні пристрої), як слід виконувати поставлене завдання. Своєю чергою послідовність операцій, призначених для вирішення певної проблеми або виконання даного завдання, називається алгоритмом. Коли алгоритм перекладається мовою, зрозумілою комп'ютеру, він стає програмою. Порушення виконання закладених певних алгоритмів роботи приладів, призводить до виходу з ладу цілої системи загалом. Тому дослідження спрямоване на аналіз існуючих методів захисту, розробку концепції механізму трансформації коду, що забезпечить високий рівень захисту .NET застосунків. Результати цієї роботи можуть бути використані для підвищення безпеки програмного забезпечення та зменшення ризиків несанкціонованого доступу до вихідного коду програм.

Аналіз останніх досліджень та публікацій

Існуючі методи захисту на сьогоднішній день досить широко використовуються в комерційних програмних рішеннях для захисту програмного коду, а дослідження алгоритмів переходу виконання програм у машинний код є невід'ємною частиною дослідження архітектури та інженерії програмного забезпечення (Kienle,2010, Francisca, 2012).

Архітектура .NET Framework. NET Framework є потужною платформою для розробки програмного забезпечення, яка забезпечує широкий набір інструментів і бібліотек для створення застосунків різного типу, десктопних, веб-додатків, мобільних програм і сервісів. Архітектура .NET Framework складається з кількох ключових компонентів, які забезпечують її функціональність (Рис. 1). Перші три компоненти знизу вважаються базовою архітектурою .Net Framework, яка з'явилася в 2005 році, і після цього Microsoft додавала до .Net Framework нові компоненти і можливості.

На нижньому рівні архітектури знаходиться Common Language Runtime (CLR) – основний виконуючий механізм .NET Framework. CLR забезпечує управління пам'яттю, виконання коду, опрацювання винятків, збір сміття та безпеку виконання. Завдяки цьому розробники можуть фокусуватися на логіці застосунків, не турбуючись про низькорівневі деталі. CLR також дозволяє запускати код, написаний різними мовами програмування, завдяки підтримці Common Language Specification (CLS).

Важливим елементом є Base Class Library (BCL) – набір базових класів і бібліотек, які забезпечують стандартні функціональні можливості, такі як робота з файлами, мережею, базами даних, опрацювання тексту, а також базові структури даних. BCL є універсальним інструментом для розробників і використовується в усіх типах застосунків .NET.

Поверх цих основних компонентів розташовані ADO.NET, ASP.NET, Windows Forms та інші фреймворки, які спеціалізуються на розробленні застосунків для різних середовищ. ADO.NET забезпечує доступ до даних і управління базами даних, ASP.NET – створення веб-додатків, а Windows Forms – розроблення настільних застосунків з графічним інтерфейсом користувача. Ці фреймворки спрощують процес розроблення, надаючи високорівневі інструменти для вирішення завдань.

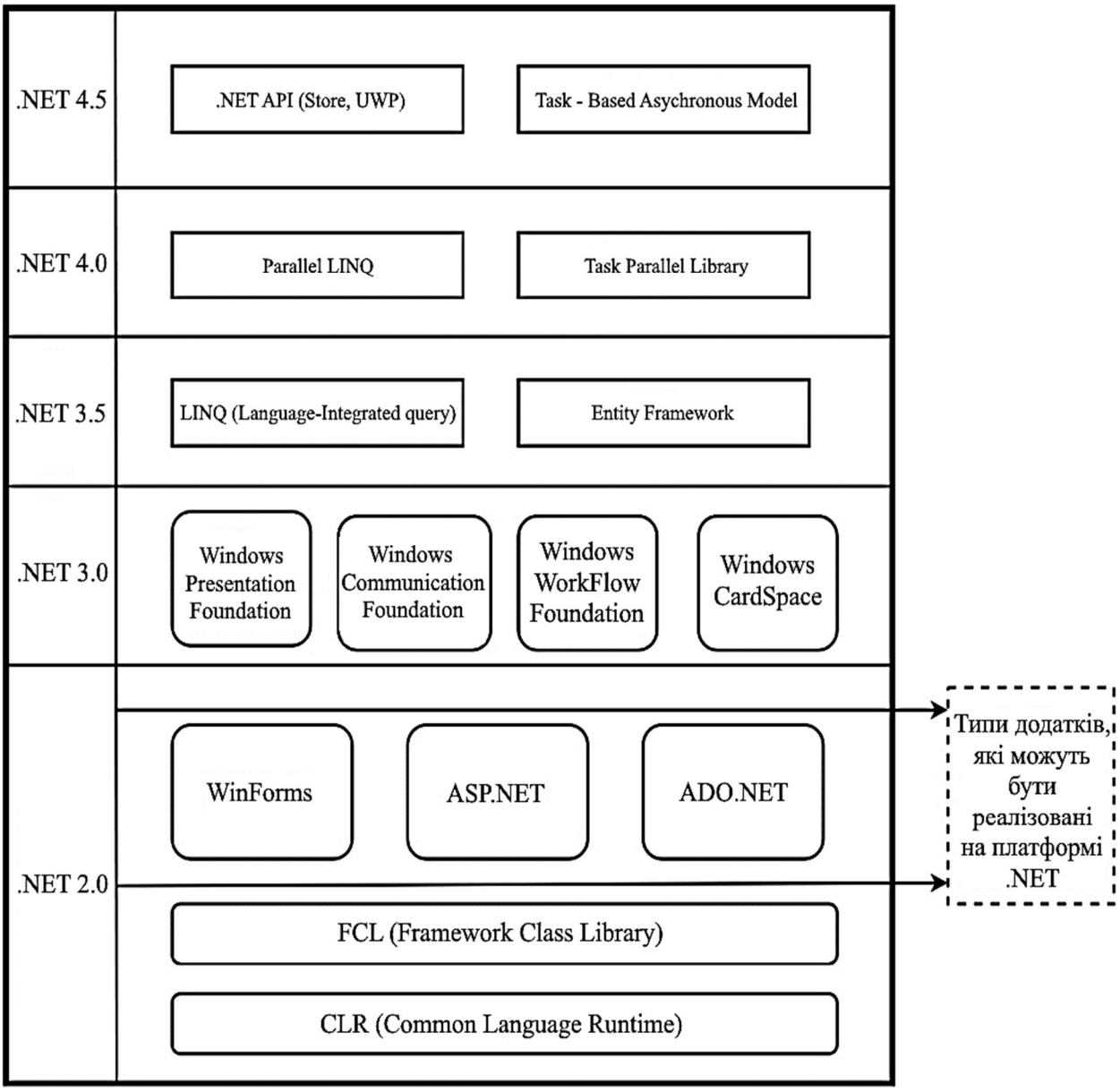

Рис. 1. Початкова архітектура .NET Framework і її зміни з новими версіями

Обфускація (заплутування) коду. Обфускація (заплутування коду) є одним із найпоширеніших методів захисту програмного забезпечення, зокрема для .NET застосунків (Tang, 2018). Цей метод полягає у модифікації вихідного або проміжного коду таким чином, щоб зберегти його функціональність, але зробити його важким для розуміння людиною та автоматизованими інструментами (Golovko, 2024, Fang,2011). Припустімо, що ми написали програмний код мовою C#, або Visual Basic.NET і хочемо виконати його обфускацію. Схема нижче описує цей процес (Рис. 2).

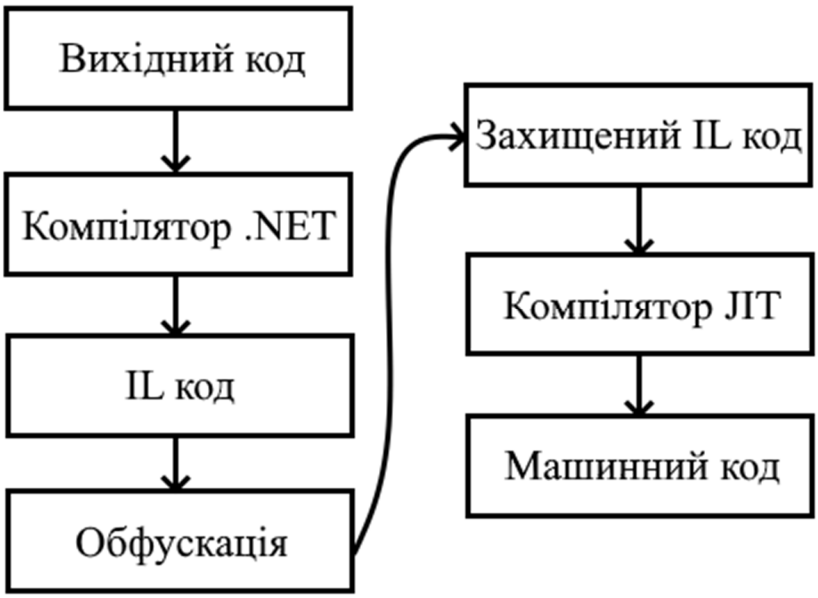

Рис.2. Схема компіляції і обфускації коду

Основні техніки обфускації включають:

- *Перейменування ідентифікаторів* полягає у заміні зрозумілих назв класів, методів і змінних на випадкові або беззмістовні символи. Це ускладнює аналіз коду і знижує ефективність автоматизованих інструментів декомпіляції.
- *Заплутування потоків виконання* сприяє модифікації логіки програми через вставку зайвих умов, циклів або переходів, що створює складні для розуміння структури виконання.
- *Шифрування рядкових літералів* сприяє приховуванню текстових даних у коді шляхом їх шифрування та дешифрування під час виконання. Це дозволяє приховати конфіденційну інформацію або важливі ресурси програми.
- *Інжекція фрагментів коду* полягає у додаванні неважливих або зайвих фрагментів, які ускладнюють аналіз коду, не впливаючи на його логіку.

Існують різні інструменти обфускації для .NET, серед яких популярними є Dotfuscator, ConfuserEx та SmartAssembly (Anckaert,2006). Кожен із них пропонує власний набір функцій для заплутування коду, однак навіть найкращі обфускатори мають певні обмеження.

Перейменування ідентифікаторів як метод обфускації коду. Перейменування ідентифікаторів є одним із найпоширеніших і дієвих способів обфускації програмного коду. Цей метод передбачає зміну назв змінних, функцій, класів та інших ідентифікаторів на такі, що не несуть жодного смислового навантаження. Основна мета цього підходу – ускладнити розуміння коду або ввести в оману тих, хто намагається здійснити реверс-інжиніринг чи незаконно використовувати програму.

Суть методу полягає в заміні зрозумілих і логічних назв на випадкові або безглузді. Наприклад, змінна, що зберігає проміжний результат обчислень і спочатку має назву Program, може бути перейменована на будь яку літеру латиниці або хеш від випадкового значення. Така зміна ускладнює розуміння логіки роботи програми та зменшує ймовірність її інтуїтивного аналізу.

Ручне та автоматичне перейменування:

Процес перейменування ідентифікаторів може виконуватися вручну або за допомогою спеціалізованих програм для обфускації. Автоматизовані інструменти аналізують код і замінюють ідентифікатори випадковими послідовностями символів або заздалегідь визначеними наборами неінформативних назв.

Статичне перейменування:

При статичному підході кожному унікальному ідентифікатору призначається нова назва, яка залишається незмінною у всьому проєкті. Це простіший, але менш гнучкий метод.

$$S \rightarrow O \rightarrow T,$$

де S – Множина початкових ідентифікаторів коду,
O – Множина ідентифікаторів після обфускації,
T – Функція перетворення.

Для кожного елемента x із множини S існує унікальний елемент у множині O, що задовольняє умову T(x) = y.

Динамічне перейменування:

Динамічний підхід (Ghosh, 2013) передбачає зміну ідентифікаторів залежно від контексту їх використання (C – контекстна функція), що значно ускладнює аналіз логіки програми.

$$S \rightarrow O \rightarrow C \rightarrow T, C\ T(x) = y' \rightarrow C(y) = y'$$

Перейменування ідентифікаторів є важливим інструментом обфускації, що дозволяє захистити програмний код від несанкціонованого доступу й аналізу. Незважаючи на певні обмеження, цей метод є одним із найпопулярніших у програмній індустрії завдяки своїй ефективності та простоті впровадження.

Заплутування потоків виконання. Заплутування потоків виконання є однією з ключових технік, яка використовується для ускладнення аналізу програми. Цей метод полягає у зміні логіки виконання коду таким чином, щоб зробити її заплутаною для реверс-інженера або автоматизованих інструментів аналізу. Незважаючи на те, що програма продовжує працювати коректно, її структура стає складною для розуміння та відтворення.

Основні техніки заплутування потоків виконання:

- Вставка зайвих умов шляхом додавання логічних перевірок або умовних операторів, які не впливають на кінцевий результат виконання програми, але створюють враження складної логіки.

```
public int CalculateSum(int a, int b)
{
    if (a > 1000) // Умова, яка ніколи не виконується
    {
        Console.WriteLine("Ця частина коду недоступна.");
    }
    return a + b;
}
```

- Розгалуження з помилковими шляхами через додавання фрагментів коду, які не виконуються за жодних умов, але ускладнюють аналіз коду

```
public int GetValue(int x)
{
    if (x < 0)
    {
        return -1; // Невикористовувана гілка
    }
```

```
    else if (x == 0)
    {
        return 0;
    }
    else
    {
        return x;
    }
}
```

- Перестановка блоків коду забезпечується, коли логічні блоки програми змінюють свій порядок виконання, використовуючи переходи (наприклад, GOTO) для підтримки правильної послідовності виконання.

```
public void ProcessData()
{
    GotoLabel2:
        Console.WriteLine("Крок 2");
        goto GotoLabel3;
    GotoLabel1:
        Console.WriteLine("Крок 1");
        goto GotoLabel2;
    GotoLabel3:
        Console.WriteLine("Крок 3");
}
```

- Циклічні залежності забезпечуються створенням циклів, які виконують зайві ітерації, або циклів, які включаються умовно.

```
public void ExecuteWithLoops()
{
    for (int i = 0; i < 10; i++)
    {
        if (i == 5)
        {
            break; // Беззмістовний цикл
        }
    }
    Console.WriteLine("Робота завершена.");
}
```

Шифрування рядкових літералів. Шифрування рядкових літералів є одним із ключових методів захисту текстових даних, які використовуються у програмному забезпеченні. Більшість застосунків містять рядкові літерали, що включають конфіденційну інформацію, таку як ключі доступу, ідентифікаційні дані, повідомлення для користувачів або налаштування. Ці дані можуть бути легко витягнуті з виконуваних файлів або бібліотек за допомогою статичних аналізаторів. Шифрування дозволяє приховати зміст таких даних, унеможливлюючи їх пряме використання зловмисниками.

Шифрування реалізується шляхом перетворення рядків у зашифрований формат, який зберігається у виконуваному файлі, ресурсах програми або базах даних. Під час виконання програми ці рядки дешифруються динамічно, що дозволяє зберігати їх у захищеному вигляді, доки вони не потрібні для використання. Для шифрування можуть використовуватися різноманітні алгоритми, від простих перетворень (наприклад, XOR) до більш складних (AES, RC2, Blowfish).

На етапі шифрування всі чутливі рядки програми конвертуються у зашифрований формат. Збереження зашифрованих даних може здійснюватися у виконуваному файлі, бібліотеках або окремих файлах ресурсів. Під час виконання програми, коли рядок необхідний для роботи, він дешифрується в оперативній пам’яті, що забезпечує захист від статичного аналізу.

```
public void ProcessData()
{
    string password = strdecr(3782587946); // літерал "12345"
    bool flag = DoJob(password);
    if(flag)
        Console.WriteLine(strdecr(5792308547); // літерал "Виконано
неуспішно"
    else
        Console.WriteLine(strdecr(1960553409)); // літерал "Виконано
неуспішно"
}
```

В даному прикладі всі рядкові літерали були замінені викликом методу strdecr з числовим параметром, який в подальшому буде використовуватися для дешифрування оригінального значення літералу.

Однією з основних переваг шифрування рядкових літералів є захист конфіденційних даних, таких як ключі API, паролі та інша важлива інформація, що стає недоступною для аналізу. Крім того, це ускладнює реверс-інжиніринг. Навіть при отриманні доступу до виконуваного файлу зловмисники не можуть відновити зашифровані дані без дешифрування. Також шифрування легко поєднується з іншими методами захисту, наприклад, з обфускацією.

Використання пакерів і криптерів. Пакери та криптери є ще одним поширеним методом захисту програмного забезпечення від аналізу і декомпіляції. Основна мета цих інструментів – приховати вихідний або проміжний код програми, упакувавши його в зашифрований або стиснутий формат, який розпаковується або дешифрується безпосередньо під час виконання програми.

Пакери здійснюють стиснення або упаковку виконуваних файлів. Після упаковки програма містить спеціальний завантажувач, який розпаковує основний код у пам'яті під час запуску. Це ускладнює аналіз файлу за допомогою стандартних інструментів, оскільки вихідний код недоступний у статичному вигляді.

Криптери виконують шифрування виконуваного коду або його окремих частин. При запуску програми зашифровані ділянки коду розшифровуються динамічно. Це дозволяє приховати важливі частини програми, наприклад, алгоритми шифрування, ліцензійні перевірки чи ключові функції.

Основні функції пакерів і криптерів:

Стиснення коду відбувається шляхом зменшення розміру виконуваних файлів за рахунок упаковки.

Шифрування вмісту передбачає захист важливих ділянок коду від статичного аналізу.

Антидебаг механізми полягають у виявленні спроб відлагодження та блокування роботи в середовищах аналізу.

Інжекція в пам'ять забезпечується розпакуванням або розшифруванням коду безпосередньо в оперативну пам'ять, що ускладнює зняття дампу процесу.

Формулювання цілі статті

Метою роботи є аналіз існуючих та розроблення концепції нових підходів до захисту програмного коду на платформі .NET. Ці підходи можуть бути використані при проектуванні віртуальної машини, що буде трансформувати проміжний MSIL-код у спеціальний набір байт-коду, що виконується лише в межах цієї машини. Об'єднання цих підходів значно ускладнює процес реверс-інжинірингу та робить використання стандартних інструментів декомпіляції надскладною задачею.

Для досягнення поставленої мети були визначені наступні задачі:

1. Провести огляд існуючих методів захисту .NET застосунків, включаючи існуючі методи обфускації (заплутування) коду таких як перейменування ідентифікаторів, заплутування потоків виконання, шифрування рядкових літералів, використання пакерів та криптерів, особливістю останніх яких є безпосередня інжекція коду в оперативну пам'ять.
2. Проаналізувати сучасні архітектури на основі платформ .NET Framework.
3. Окреслити переваги запровадження антивідлагодження як способу додаткового захисту програмного коду.
4. Провести метод перевірки батьківського процесу, що є одним із способів виявлення відлагоджувачів, які можуть бути підключені до програмних модулів.
5. Розглянути працездатність різного роду функцій, які забезпечують базові алгоритми захисту .NET застосунків, що є системним індикатором підключення відлагоджувачів до процесів виконання програмного коду.
6. Забезпечити інженерію програмного коду для виявлення відлагодження за допомогою апаратних точок зупинки шляхом запровадження нових підходів до ін'єктованих потоків в пам'яті процесів.

Виклад основного матеріалу

Антивідлагодження (anti-debugging) є одним із ключових компонентів захисту програмного забезпечення від реверс-інжинірингу та аналізу. Ця техніка полягає у впровадженні механізмів, які ускладнюють або унеможливлюють використання інструментів відлагодження для аналізу виконання програми.

Впровадження антивідлагодження як способу додаткового захисту коду. Антивідлагоджувальні техніки поділяються на кілька категорій залежно від способу їх реалізації:

Функція IsDebuggerPresent

Один із найпростіших способів захисту від відлагодження – це використання функції IsDebuggerPresent із бібліотеки WinAPI. Вона дозволяє визначити, чи підключено до процесу відлагоджувач. Якщо підключено – функція повертає TRUE, якщо ні – FALSE. Нижче наведено приклад коду, що демонструє, як можна застосувати цю функцію для виявлення відлагоджувача.

```
[DllImport("kernel32.dll")]
private static extern bool IsDebuggerPresent();
public static bool CheckDebugger()
{
    return IsDebuggerPresent();
}
```

Однак сам факт виклику функції *IsDebuggerPresent* (Рис.3.) може викликати підозри, навіть якщо її виклик приховано. Цей метод вважається базовим і його можна обійти за допомогою таких інструментів, як ScyllaHide – плагін для x64dbg – популярної програми для відлагодження, спеціально створений для обходу антивідлагоджувальних механізмів.

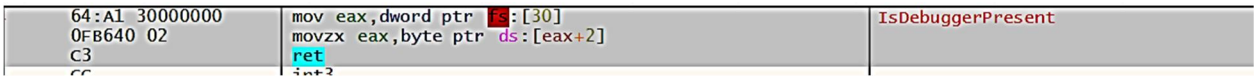

Рис. 3 Функція IsDebuggerPresent

Ефективнішим рішенням є створення власної версії функції IsDebuggerPresent, яка використовує структуру PEB (Process Environment Block). Ця структура містить поле BeingDebugged, значення якого встановлюється в 1, якщо процес працює в режимі відлагодження.

Альтернативний підхід до функції IsDebuggerPresent полягає у перевірці значення BeingDebugged. Наприклад, у власній реалізації функція IsDebuggerPresentUser повертає TRUE, якщо поле BeingDebugged дорівнює 1.

```
BOOL IsDebuggerPresentUser() {
#ifdef _WIN64
    PPEB uPeb = (PEB*)(__readgsqword(0x60));
#elif _WIN32
    PPEB uPeb = (PEB*)(__readfsdword(0x30));
#endif
  if (uPeb->BeingDebugged == 1)
    return TRUE;

   return FALSE;
}
```

Ще один підхід до створення власної версії функції IsDebuggerPresent передбачає використання недокументованого поля NtGlobalFlag, яке також знаходиться у структурі PEB (Process Environment Block). Це поле може слугувати додатковим індикатором того, чи працює процес у режимі відлагодження.

Елемент NtGlobalFlag встановлюється в шістнадцяткове значення 0x70, якщо процес запускається за участі відлагоджувача. У звичайному режимі роботи, без відлагодження, значення цього прапора залишається рівним 0. Такий підхід дозволяє визначати запуск програми під управлінням відлагоджувальних інструментів.

Втім, важливо зазначити, що цей метод має свої обмеження. Поле NtGlobalFlag отримує значення 0x70 лише на момент створення процесу під відлагоджувачем. Це значення виходить завдяки трьом прапорцям – FLG_HEAP_ENABLE_TAIL_CHECK (0x10), FLG_HEAP_ENABLE_FREE_CHECK (0x20), FLG_HEAP_VALIDATE_PARAMETERS (0x40). Недоліком цього методу є те, що він не дозволяє виявити відлагоджувачі, які підключаються до процесу вже після його запуску. Таким чином, використання цього підходу доречно лише для виявлення відлагодження на ранніх етапах роботи програми, наприклад, у момент її ініціалізації.

```
#define FLG_HEAP_ENABLE_TAIL_CHECK   0x10
#define FLG_HEAP_ENABLE_FREE_CHECK   0x20
#define FLG_HEAP_VALIDATE_PARAMETERS 0x40

BOOL IsDebuggerPresentUser() {

#ifdef _WIN64
    PPEB uPeb = (PEB*)(__readgsqword(0x60));
```

```
#elif _WIN32
    PPEB uPeb = (PEB*)(__readfsdword(0x30));
#endif
  if (uPeb->NtGlobalFlag == (FLG_HEAP_ENABLE_TAIL_CHECK |
FLG_HEAP_ENABLE_FREE_CHECK | FLG_HEAP_VALIDATE_PARAMETERS))
    return TRUE;


  return FALSE;
}
```

Перевірка батьківського процесу. Метод перевірки батьківського процесу є одним із способів виявлення відлагоджувачів, які можуть бути підключені до програми. Його суть полягає в тому, щоб з'ясувати, який процес є батьківським для поточного, і перевірити, чи не належить цей процес до відомих відлагоджувальних інструментів, таких як x64dbg, Ollydbg, або Cheat Engine. Цей підхід реалізується за допомогою функції CreateToolhelp32Snapshot, яка дозволяє отримати знімок усіх процесів у системі.

Після створення знімка програма аналізує структуру PROCESSENTRY32, яка містить інформацію про процеси, включаючи ідентифікатор батьківського процесу (th32ParentProcessID). Далі виконується пошук імені батьківського процесу та його порівняння зі списком відомих відлагоджувачів. Якщо ім'я збігається з одним із підозрілих, можна зробити висновок, що програму запущено в середовищі, яке намагається її налагодити.

Цей метод має кілька переваг. Він дозволяє не тільки виявляти відлагоджувачі, а й інші потенційно небажані програми, які можуть маніпулювати процесами. Крім того, список підозрілих процесів можна налаштовувати та оновлювати залежно від нових загроз. Проте він вразливий до обходу, якщо відлагоджувач приховує своє ім'я або працює під іншим іменем. Також створення та аналіз знімка процесів може бути ресурсоємним, особливо у системах із великою кількістю активних процесів.

Функція CheckRemoteDebuggerPresent. Функція CheckRemoteDebuggerPresent(), що входить до бібліотеки kernel32.dll, використовується для виявлення підключеного відлагоджувача. Вона працює, викликаючи низькорівневу функцію ZwQueryInformationProcess із параметром ProcessInformation-Class. Цей параметр дозволяє отримати значення ProcessDebugPort, яке вказує на стан відлагодження. Якщо значення порту не дорівнює нулю, це означає, що до процесу підключений відлагоджувач.

Перевагою цього методу є його простота у використанні, оскільки він абстрагує складнощі викликів ядра та надає зручний інструмент для перевірки як поточного процесу, так і інших. Це робить функцію універсальною для виявлення відлагоджувачів у реальному часі. Також вона працює ефективно й швидко, оскільки отримує інформацію безпосередньо від ядра операційної системи

```
bool IsDebuggerAttached(HANDLE process) {
    BOOL isDebuggerPresent = FALSE;
    if (CheckRemoteDebuggerPresent(process, &isDebuggerPresent)) {
        return isDebuggerPresent;
    }
    return false;
}
```

По аналогії з IsDebuggerPresent ми можемо реалізувати цю функцію самостійно

```
bool IsDebuggerAttached() {
    HMODULE ntdll = GetModuleHandleA("ntdll.dll");
    if (!ntdll) return false;
```

```
        pZwQueryInformationProcess ZwQueryInformationProcess =
            (pZwQueryInformationProcess)GetProcAddress(ntdll,
"ZwQueryInformationProcess");
        if (!ZwQueryInformationProcess) {
            return false;
        }
        HANDLE currentProcess = GetCurrentProcess();
        ULONG debugPort = 0;
        ULONG returnLength = 0;
        NTSTATUS status = ZwQueryInformationProcess(
            currentProcess,
            0x1E, // ProcessDebugPort
            &debugPort,
            sizeof(debugPort),
            &returnLength
        );
    if (status == STATUS_SUCCESS) {
            return debugPort != 0;
        } else {
            return false;
        }
    }
```

Функція NtQuerySystemInformation. Функція NtQuerySystemInformation() із бібліотеки ntdll.dll використовується для отримання системної інформації. Один із її класів – SystemKernelDebuggerInformation (0x23) — дозволяє визначити, чи підключено до системи відлагоджувач ядра. Цей клас повертає два елементи. KdDebuggerEnabled вказує на підтримку відлагоджувача в системі (міститься в молодшому байті al), і KdDebuggerNotPresent сигналізує про його підключення (значення знаходиться у старшому байті ah). Якщо значення KdDebuggerNotPresent дорівнює нулю, це означає, що відлагоджувач ядра підключений до системи.

Цей метод дозволяє отримувати інформацію безпосередньо від операційної системи, що ускладнює його обходження. Крім того, він сумісний із багатьма версіями Windows, оскільки клас SystemKernelDebuggerInformation існує ще з часів Windows NT. Така перевірка є особливо корисною для виявлення налагоджувачів, які працюють на рівні ядра, що робить її ефективною на етапі запуску програми.

```
    bool IsKernelDebuggerPresent() {
        ULONG info[2] = { 0 };
        ULONG returnLength = 0;
        NTSTATUS status = NtQuerySystemInformation(0x23, &info, sizeof(info),
&returnLength);
        if (status == STATUS_SUCCESS) {
            return (info[0] & 0xFF00) == 0; // AH == 0
        }
        return false;
    }
```

Виявлення відлагодження за допомогою апаратних точок зупинки. Цей підхід застосовний лише тоді, коли під час відлагодження використовуються апаратні точки зупинки. Апаратні точки зупинки, які ще називають регістрами апаратного відлагодження, є функцією сучасних процесорів, що дозволяє зупиняти виконання програми при досягненні певної адреси в пам'яті або при виникненні події. Ця функція вбудована в апаратну частину процесора, що робить її швидшою та ефективнішою порівняно з програмними точками зупинки, які потребують участі операційної системи чи налагоджувача для контролю виконання програми (Рис.4,5).

Коли апаратні точки зупинки активні, значення деяких регістрів змінюється. Це дозволяє виявити підключення налагоджувача до процесу. Якщо регістри Dr0, Dr1, Dr2 або Dr3 мають ненульові значення, це свідчить про наявність активних апаратних точок зупинки.

Наприклад, при встановленні апаратної точки зупинки на виклик системної функції NtAllocateVirtualMemory за допомогою налагоджувача x64dbg, значення регістра Dr0 змінюється з 0 на адресу, відповідну функції NtAllocateVirtualMemory. Ця зміна може бути використана для визначення, що процес знаходиться під налагодженням.

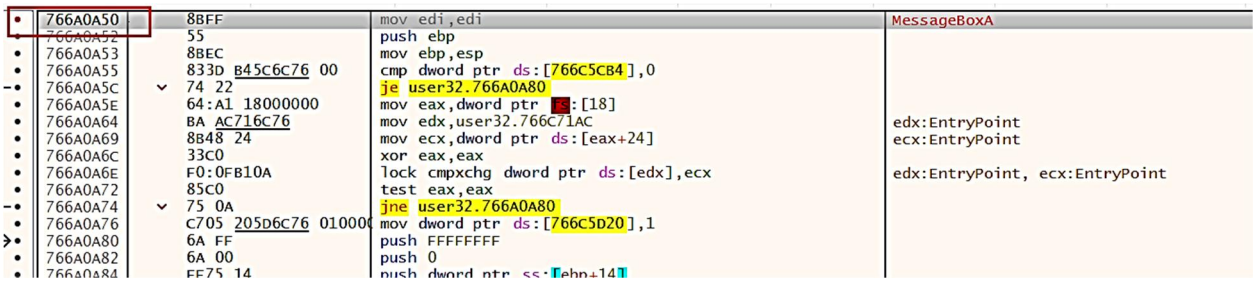

Рис.4. Встановлена апаратна точка зупинки на функції MessageBoxA

Рис.5. Значення регістру Dr0

```
BOOL DebuggerPresent() {

    CONTEXT Ctx = { .ContextFlags = CONTEXT_DEBUG_REGISTERS };

    if (!GetThreadContext(GetCurrentThread(), &Ctx)) {
        printf("GetThreadContext failed : %d \n", GetLastError());
        return FALSE;
    }

    if (Ctx.Dr0 != NULL || Ctx.Dr1 != NULL || Ctx.Dr2 != NULL || Ctx.Dr3
!= NULL)
        return TRUE;
    return FALSE;
}
```

Виявлення ін'єктованих потоків в пам'яті процесу. У сучасному програмному забезпеченні безпека коду є критично важливим аспектом, особливо для комерційних продуктів, що потребують захисту від несанкціонованого аналізу та модифікації. Однією з технік, яку використовують дослідники та зловмисники для реверс-інжинірингу програм, є ін'єкція стороннього коду. Цей метод дозволяє перехоплювати та змінювати виклики функцій, що дає змогу контролювати виконання коду, змінювати його логіку та навіть обходити механізми захисту. Впровадження механізмів захисту від цих технік є одним із нових та ефективних підходів до захисту програмного коду, що ускладнює його аналіз та модифікацію.

В статті пропонується підхід до виявлення серед потоків поточного процесу ін'єктованих (сторонніх) потоків. Він може використовуватися в різних сценаріях, пов'язаних із безпекою, антивірусним захистом, виявленням шкідливого коду та запобіганням несанкціонованого втручання в процес. Також це важливо, оскільки цей підхід може бути використаний з іншими елементами захисту, в тому числі і при проектуванні віртуальної машини.

Інжектовані потоки – це потоки, які не були створені вихідним процесом, а додані стороннім програмним забезпеченням. Це може відбуватися через використання технік.

DLL Injection (ін'єкція динамічних бібліотек)
Code Injection (безпосереднє впровадження коду в пам'ять процесу)
Thread Hijacking (захоплення існуючого потоку)
Process Hollowing (переписування пам'яті процесу)
Reflective DLL Injection (завантаження DLL без використання LoadLibrary)

```
public static bool HasInjectedThreads(bool validateModuleRange)
{
    const uint MEMORY_IMAGE = 16777216u;
    const uint MEMORY_COMMITTED = 4096u;
    const int THREAD_QUERY_INFORMATION = 9;
    const uint THREAD_ACCESS_QUERY = 64u;

    int currentProcessId = Process.GetCurrentProcess().Id;

    foreach (ProcessThread thread in Process.GetCurrentProcess().Threads)
    {
        var threadIdentity = new Structs.CLIENT_ID
        {
            UniqueProcess = (IntPtr)currentProcessId,
            UniqueThread = (IntPtr)thread.Id
        };

        var threadAttributes = new Structs.OBJECT_ATTRIBUTES
        {
            Length = Marshal.SizeOf(typeof(Structs.OBJECT_ATTRIBUTES)),
            RootDirectory = IntPtr.Zero,
            ObjectName = IntPtr.Zero,
            Attributes = 0u,
            SecurityDescriptor = IntPtr.Zero,
```

```
                SecurityQualityOfService = IntPtr.Zero
            };
            if (NtOpenThread(out IntPtr threadHandle, THREAD_ACCESS_QUERY,
ref threadAttributes, ref threadIdentity) != 0 || threadHandle == IntPtr.Zero)
            {
                continue;
            }
            IntPtr threadStartAddress = IntPtr.Zero;
            int queryStatus = NtQueryInformationThread(threadHandle,
THREAD_QUERY_INFORMATION, ref threadStartAddress, (uint)IntPtr.Size,
IntPtr.Zero);


            Utils.CloseHandle(threadHandle);


            if (queryStatus != 0)
            {
                continue;
            }

            var memoryInfo = new Structs.MEMORY_BASIC_INFORMATION();


            if (GetVirtualMemoryQuery(threadStartAddress, ref memoryInfo, out
_))
            {
                if (memoryInfo.Type != MEMORY_IMAGE || memoryInfo.State !=
MEMORY_COMMITTED)
                {
                    return true;
                }


                if (!IsAddressInModuleRange(threadStartAddress))
                {
                    return true;
                }
            }
        }
        return false;
    }


    public static bool GetVirtualMemoryQuery(IntPtr BaseAddress, ref
Structs.MEMORY_BASIC_INFORMATION MemoryInformation, out uint ReturnLength)
    {
      uint memoryInformationLength =
(uint)Marshal.SizeOf(typeof(Structs.MEMORY_BASIC_INFORMATION));
```

```
      if ((NtQueryVirtualMemory(new IntPtr(-1), BaseAddress, 0u, ref
MemoryInformation, memoryInformationLength, out ReturnLength)) == 0)
      {
            return true;
      }
      return false;
     }

     private static bool IsAddressInRange(IntPtr Address)
            {
                  foreach (ProcessModule processModule in
Process.GetCurrentProcess().Modules)
                  {
                        IntPtr baseAddress = processModule.BaseAddress;
                        IntPtr intPtr = IntPtr.Add(baseAddress,
processModule.ModuleMemorySize);
                        if (Address.ToInt64() >= baseAddress.ToInt64() &&
Address.ToInt64() < intPtr.ToInt64())
                        {
                              return true;
                        }
                  }
                  return false;
            }
```

Цей метод виконує перевірку всіх потоків поточного процесу, щоб визначити, чи є серед них ін’єктовані потоки, які можуть вказувати на зловмисне втручання. Вона перебирає всі потоки, відкриває їх дескриптори та отримує інформацію про їхню пам’ять. Головне завдання функції — виявити потоки, які працюють із підозрілих ділянок пам’яті або не належать до відомих модулів програми.

На початку функція ініціалізує кілька важливих змінних, таких як MEM_IMAGE (16777216), що позначає пам’ять, у якій зберігається виконуваний код, і MEM_COMMIT (4096), що вказує на виділену пам’ять. Вона отримує список потоків поточного процесу, після чого для кожного потоку створює структури CLIENT_ID (з ID поточного процесу та потоку) і OBJECT_ATTRIBUTES (що містить мета-інформацію про потік). Далі функція викликає NtOpenThread, щоб отримати дескриптор потоку, що дозволяє зчитувати його інформацію.

Якщо дескриптор потоку успішно отримано, функція викликає NtQueryInformationThread, щоб дізнатися адресу початку виконання потоку. Потім вона використовує NtQueryVirtualMemory, щоб отримати атрибути пам’яті за цією адресою. Якщо пам'ять потоку не має типу MEM_IMAGE або її стан не є MEM_COMMIT, це вказує на потенційне стороннє втручання. Крім того, викликається функція IsAddressInRange(ThreadInformation), яка перевіряє, чи знаходиться потік у межах дозволених модулів програми. Якщо знайдено хоча б один підозрілий потік, функція негайно повертає true, вказуючи на наявність потенційної загрози. Якщо ж усі потоки проходять перевірку, функція повертає false, що означає, що ін’єкцій не виявлено.

Висновки

У ході проведеного дослідження було проаналізовано наявні методи захисту застосунків, розроблених для платформи .NET. Зокрема, розглянуто обфускацію коду, використання пакерів і криптерів, шифрування рядкових літералів, а також техніки заплутування потоків виконання. Ці методи забезпечують базовий рівень захисту програмного забезпечення, але мають певні обмеження, які можуть бути використані зловмисниками для обходу захисту.

Проведений аналіз дозволив сформувати концептуальний підхід до реалізації механізмів антивідлагодження програмного забезпечення. Запропоновано процес антивідлагодження, що є важливою складовою багаторівневого захисту програмного забезпечення і змінює інструкції та поведінку віртуальної машини. Окрім того, запропоновано метод виявлення ін'єктованих потоків, впровадження якого є новим підходом до захисту програмного коду. Це, своєю чергою, відкриває можливість під час проектування віртуальної машини у поєднанні з іншими техніками створювати значні перешкоди для реверс-інженірингу. Розроблені та запропоновані методи можуть бути інтегровані у процес розробки комерційного програмного забезпечення для зменшення ризиків несанкціонованого доступу до вихідного коду, захисту інтелектуальної власності та підвищення загальної безпеки продуктів на платформі .NET.

СПИСОК ЛІТЕРАТУРИ

1. Anckaert, B., Jakubowski, M., & Venkatesan, R. (2006). Proteus: Virtualization for Diversified Tamper-Resistance. In DRM '06: Proceedings of the ACM workshop on Digital rights management, 30, 47 –58. doi:10.1145/1179509.1179521.
2. Fang, H., Wu, Y., Wang, S., & Huang, Y. (2011). Multi-stage Binary Code Obfuscation Using Improved Virtual Machine. Lecture Notes in Computer Science, 7001, 168–181. doi:10.1007/978-3-642-24861-0_12.
3. Francisca, O., Onyemaechi, O., & Okechukwu, O. (2012). Exploring the two faces of Software Reverse Engineering. International Journal of Advanced Research in Computer Science and Software Engineering, 2(4), 366– 370. doi:10.23956/ijarcsse.
4. Ghosh, S., Hiser, J., & Davidson, J. (2013). Software protection for dynamically-generated code. In Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, 1, 1–12. doi:10.1145/2430553.2430554.
5. Golovko, I., Medzatyi, D., & Ivanchenko, O. (2024). Methods of obfuscation of program code using artificial intelligence. Informatics, computing and automation. Bulletin of the Vernadskyi University of Technology, 35(74), 115 – 123. doi:10.32782/2663-5941/2024.5.1/18.
6. Kienle, H.M., & Müller, H.A. (2010). The Tools Perspective on Software Reverse Engineering: Requirements, Construction and Evaluation. Advances in Computers, 79, 189–290. doi:10.1016/S0065-2458(10)79005-7.
7. Nadir, I., Mahmood, H., & Asadullah, G. (2022). A taxonomy of IoT firmware security and principal firmware analysis techniques. International Journal of Critical Infrastructure Protection, 38, 100552–100585. doi:j.ijcip.2022.100552.
8. Tang, Z., Li, M., Ye, G., Cao, S., Chen, M., Gong, X., Fang, D., & Wang, Z. (2018). VMGuards: A Novel Virtual Machine Based Code Protection System with VM Security as the First Class Design Concern. Applied Sciences, 8(5), 771–794. doi:10.3390/app8050771.

SOFTWARE CODE PROTECTION ON THE .NET PLATFORM

Andriy Fechan, Oleksii Khoverko

[1, 2] Lviv Polytechnic National University
Department of Software, Lviv, Ukraine
[1] E-mail: andrii.v.fechan@lpnu.ua, ORCID:0000-0001-9970-5497
[2] E-mail: oleksii.y.khoverko@lpnu.ua, ORCID:0009-0007-1831-6627

In the work, a study of the functioning of complex information systems is carried out, which is aimed at the analysis of existing protection methods, the development of the concept of the code transformation mechanism, which will ensure a high level of protection of .NET applications. The

architecture of the .NET Framework with modern universal improvements is considered. One of the important elements is the Base Class Library (BCL), which is a set of base classes and libraries that provide standard functionality, such as working with files, networks, databases, text processing, and basic data structures. It was determined that obfuscation is one of the most common methods of software protection, which consists in modifying the source or intermediate code in such a way as to preserve its functionality, but make it difficult to understand. The main features of code improvement are established, which consist in obfuscating execution flows, encrypting string literals, and using packers and crypters. A characteristic feature of packers and crypters is direct injection into memory, which reflects its functionality in unpacking or decrypting program code directly into random access memory (RAM), which makes it difficult to remove a dump of the process. In addition, the entanglement of execution threads allows you to modify the logic of the software operation by inserting unnecessary conditions, loops or transitions, which creates complex execution structures. The anti-debugging process is proposed, which is one of the key components of software protection against reverse engineering. A conceptual approach to the implementation of software anti-debugging mechanisms is formed, which consists in multi-level protection of software and changes the instructions and behavior of the virtual machine. One of the important elements of such algorithms is checking parent processes and detecting debugging using hardware breakpoints. This, in its turn, opens up the possibility during the design of a virtual machine in combination with other techniques to create significant obstacles for the analysis of the software code.

Keywords – algorithm, program code, virtual machine, obfuscation, .NET Framework.