Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
Факультет інформаційних технологій
Кафедра безпеки інформації та телекомунікацій

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня бакалавра

студента *Масла Дмитра Григорійовича*

академічної групи *125-19-2*

спеціальності *125 Кібербезпека*

за освітньо-професійною програмою *Кібербезпека*

на тему *Засоби забезпечення цілісності програмного забезпечення десктопних Windows-додатків*

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
кваліфікаційної роботи	к.т.н., доц. Сафаров О.О.			
розділів:				
спеціальний	к.т.н., доц. Сафаров О.О.			
економічний	к.е.н., доц. Пілова Д.П.	95	відмінно	

Рецензент				

Нормоконтролер	ст. викл. Мєшков В.І.			

Дніпро
2023

ЗАТВЕРДЖЕНО:
завідувач кафедри
безпеки інформації та телекомунікацій
_____ д.т.н., проф. Корнієнко В.І.

«_____»_____20___року

ЗАВДАННЯ
на кваліфікаційну роботу ступеня бакалавра

студенту *Маслу Дмитру Григорійовичу* **академічної групи** *125-19-2*
(прізвище та ініціали) (шифр)

спеціальності *125 Кібербезпека*

за освітньо-професійною програмою *Кібербезпека*

на тему *Засоби забезпечення цілісності програмного забезпечення десктопних Windows-додатків*

Затверджену наказом ректора НТУ «Дніпровська політехніка» від 16.05.23 № 350-с

Розділ	**Зміст**	**Термін виконання**
Розділ 1	Огляд предметної області	02.04.23 – 26.04.23
Розділ 2	Аналіз відомих методів виконання стороннього виконуваного коду у адресному просторі іншої програми та реалізація захисту	27.04.23 – 26.05.23
Розділ 3	Економічна частина	26.05.23 – 09.06.23

Завдання видано _____ Сафаров О.О.
(підпис керівника) (прізвище, ініціали)

Дата видачі завдання: 02.04.23

Дата подання до екзаменаційної комісії: 09.06.23

Прийнято до виконання _____ Масло Д.Г.
(підпис студента) (прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 88 с., 4 рис., 6 табл., 7 додатків, 11 джерел.

Об’єкт розробки: програмне забезпечення десктопних Windows-додатків.

Предмет розробки: програмна реалізація засобів захисту цілісності програмного коду.

Мета кваліфікаційної роботи: вдосконалення якості програмного забезпечення та підвищення рівня захищеності цілісності програмного забезпечення шляхом програмної реалізації захисту від найпоширеніших загроз.

У першому розділі проаналізовано операційну систему Windows, проблеми, пов’язані з безпекою користування нею, вбудовані механізми захисту програмного забезпечення та його коду. Розглянуті програми та процеси як потенційні цілі атак. Проаналізовані загрози від реалізації атак, що порушують цілісність виконуваного коду.

У другому розділі проаналізовані найпоширеніші типи атак на процеси, які тим чи іншим чином інжектують сторонній код у процес та розроблено програмні засоби захисту від цих атак

В третьому розділі були розраховані витрати на програмну реалізацію засобів захисту цілісності виконуваного коду Windows-додатків від трьох найпоширеніших типів атак та проведена оцінка потенційних збитків у випадку відсутності цих засобів захисту.

Практична значимість роботи полягає прогнозуванні основного вектору найпоширеніших атак, які порушують цілісність програмного коду та реалізації засобів захисту від них.

ЗАХИСТ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ЦІЛІСНІСТЬ ВИКОНУВАНОГО КОДУ, ЗАХИСТ ІНФОРМАЦІЇ, РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ОПЕРАЦІЙНА СИСТЕМА WINDOWS, АТАКИ НА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

ABSTRACT

Explanatory note: 88 p., 4 fig., 6 tables., 7 applications, 11 sources.

Object of development: software for Windows desktop applications.

Subject of development: software implementation of methods of program code integrity protection.

Purpose of qualification work: improving the quality of the software and increasing the level of protection of the integrity of the software through software implementation of protection against the most common threats.

The first section analyses the operating system Windows, problems associated with security of its usage, built-in mechanisms of software and its code protection. Programs and processes as potential targets of attacks are considered. Threats of attacks which break the integrity of executable code are analyzed.

The second section analyses the most widespread types of attacks on processes that in one way or another inject foreign code into the process and develops software defenses against these attacks

The third section calculated the costs of software implementation of means to protect the integrity of executable code of Windows applications from the three most common types of attacks and estimated the potential losses in the absence of these protections.

The practical value of the work consists in predicting the main vector of the most widespread attacks which break the integrity of the program code and implementation of the ways of protection against them.

SOFTWARE PROTECTION, EXECUTABLE CODE INTEGRITY, INFORMATION PROTECTION, SOFTWARE DEVELOPMENT, WINDOWS OPERATING SYSTEM, SOFTWARE ATTACKS

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ

ПК – персональний комп’ютер;

DEP – механізм запобігання виконанню даних;

ASLR – рандомізація розміщеного адресного простору;

PID – ідентифікатор процесу;

PPID – ідентифікатор батьківського процесу;

DLL – бібліотека динамічної компоновки;

EXE – виконуваний файл;

ROP – зворотно-орієнтоване програмування;

PE – портативний виконуваний файл;

RPC – виклик віддалених процедур;

ALPC – просунутий виклик локальних процедур.

ЗМІСТ

ВСТУП

На сьогоднішній день кожна людина для тих чи інших цілей використовує персональний комп'ютер. Для виконання поставлених задач вона використовує програмне забезпечення що автоматизує робочий процес. Кожна програма представляє собою виконуваний файл, що запускається користувачем, коли йому потрібна дана програма. Завантаживши тим чи іншим чином програму на комп'ютер, людина перевіряє цілісність виконуваного файлу, звіривши його хеш або цифровий підпис. Переконавшись, що завантажений оригінальний виконуваний файл користувач впевнений, що при використані даної програми вона буде виконувати виключно ті інструкції, що в ній містяться та були закладені розробниками даної програми.

У операційній системі Windows реалізований механізм, що робить захищеною ділянку пам'яті, у якій знаходиться виконуваний процес. Це дозволяє захистити виконуваний код та дані від інших процесів, що виконуються на даному комп'ютері.

Але також існують механізми, що дозволяють втручатися тим чи іншим чином у захищений адресний простір інших програм, виконуючи у ньому будь-який сторонній код. Такі дії можуть бути використані зловмисниками для порушення конфіденційності, цілісності чи доступності інформації, що обробляється у програмному засобі чи інформації, до якої даний процес має доступ. І ці дії відбуватимуться ніби цим самим перевіреним програмним засобом, через що їх набагато складніше виявити та прийняти запобіжні заходи для захисту інформації.

Однак для потрапляння програми, яка буде здійснювати атаку користувач тим чи іншим чином має завантажити її на свій комп'ютер та деякі з них запускати з правами адміністратора. Це є розповсюдженим підходом, адже користувачі дуже часто завантажують ті чи інші програмні засоби з інтернету та встановлюють їх у свої системі.

При виявлення підозрілої активності користувач чи системний адміністратор комп'ютеру можуть скористатися вбудованими або сторонніми засобами моніторингу активності, щоб визначити причини, через які виконуються певні підозрілі дії, але усі вони будуть приводити лише до певного програмного засобу, що може користуватися повною довірою.

Є декілька найпоширеніших методик втручання у адресний простір процесу з метою виконання у ньому тих чи інших інструкцій. Найефективніше застосовувати заходи попередження атак такого типу на етапі розробки програмного засобу, щоб він мав можливість самостійно захистити цілісність виконуваного коду, не покладаючи повністю на вбудовані в операційну систему Windows механізми захисту. До найпоширеніших атак серед зловмисників відноситься інжекція динамічної бібліотеки, зворотно-орієнтоване програмування та пряма інжекція стороннього коду з подальшим виконанням. Для забезпечення кращого захисту рекомендовано використання їх у комплексі.

РОЗДІЛ 1. ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Операційна система Windows та вбудовані механізми захисту виконуваного коду

Windows – це одна з найпопулярніших операційних систем для персональних комп'ютерів (ПК). Розроблена компанією Microsoft Windows була випущена в 1985 році і з того часу стала стандартним вибором для мільйонів користувачів по всьому світу.

Однією з основних причин популярності Windows на ПК є широка сумісність з апаратним забезпеченням. Windows підтримує величезну кількість пристроїв та різні конфігурації ПК, що дозволяє користувачам вибирати обладнання, що відповідає їх потребам та бюджету.

Іншим важливим аспектом популярності Windows є його дружній інтерфейс користувача. Windows має інтуїтивно зрозумілий і легкий у використанні інтерфейс, що робить його доступним для користувачів з різним рівнем комп'ютерної грамотності. Це дозволяє як новачкам, так і досвідченим користувачам швидко освоїтися з операційною системою та виконувати різні завдання.

Одним із сильних моментів Windows є також його широкий вибір програмного забезпечення. Більшість програм та програм, розроблених для ПК, доступні для Windows. Це включає офісні програми, графічні редактори, відеоігри, браузери і багато іншого. Багато доступного програмного забезпечення робить Windows привабливим вибором для користувачів, які хочуть мати широкий спектр можливостей та функцій на своїх комп'ютерах.

Крім того, Windows має величезну спільноту користувачів та розробників, яка активно обмінюється досвідом, допомогою та програмними рішеннями. Це дозволяє користувачам швидко знайти відповіді на свої запитання та рішення для своїх проблем через онлайн форуми, спільноти та підтримку Microsoft.

Загалом популярність Windows на ПК пояснюється її сумісністю, зручністю використання, доступністю широкого спектру програмного забезпечення та наявністю активної спільноти користувачів. Ці фактори роблять Windows кращим вибором для багатьох людей по всьому світу, що підтверджує його статус однієї з найпопулярніших операційних систем на ПК.

Саме популярність даної операційної системи є основною причиною того, що вона є найчастішою мішенню для зловмисників, які мають на меті втручання у нормальну роботу системи та користувацьких програм для порушення однієї або одразу усіх властивостей інформації, яка оброблюється.

У минулому Windows мала велику кількість проблем, пов'язану з недостатньою захищеністю системи та можливістю без значних зусиль з боку зловмисників втручатися у процес виконання програм. Це призводило до того, що кіберзлочини відбувалися вкрай часто і призводили до значних витрат як і звичайних користувачів комп'ютерів, так і тих, які використовують дану операційну систему у професійних цілях.

Це стало причиною переорієнтації напряму розробки Windows на забезпечення надійної безпеки системи. Особлива увага приділялася реалізації механізмів захисту виконуваного коду, адже такі атаки несуть найбільшу загрозу з причини того, що їх вкрай важно виявити. Порушуючи цілісність програмного коду деякої програми зловмисник приховує свій сторонній код у деякій повсякденній програмі, яка ніколи не викликатиме особливої уваги в користувачів операційної системи.

Існуючі нині механізми захисту виконуваного коду значно ускладнюють процес атак на програмні засоби з метою порушення цілісності їх програмного коду, але вони не є абсолютно надійними.

Тому розробникам програмного забезпечення важливо самостійно реалізовувати додатковий захист програмних засобів, який буде доповнювати вбудовані в операційну систему Windows механізми.

1.1.1 Віртуальний адресний простір процесів як спосіб захисту виконуваного коду

Windows реалізує систему віртуальної адресної пам'яті на основі лінійного адресного простору. Вона дає кожному процесу можливість ставитися до виділеної йому ділянки пам'яті як до свого великого закритого адресного простору. Віртуальна пам'ять надає логічне відображення пам'яті, яке може не відповідати її фізичному знаходженню. Під час роботи диспетчер пам'яті, за допомогою блоку керування пам'яттю, відображає віртуальні адреси пам'яті на фізичні, за якими і знаходяться дані. Операційна система керує відображенням та захистом пам'яті, що забезпечує відсутність конфліктів процесів при використанні пам'яті[1][2].

Принцип відображення віртуальної пам'яті на фізичну наведено на рис. 1.1.

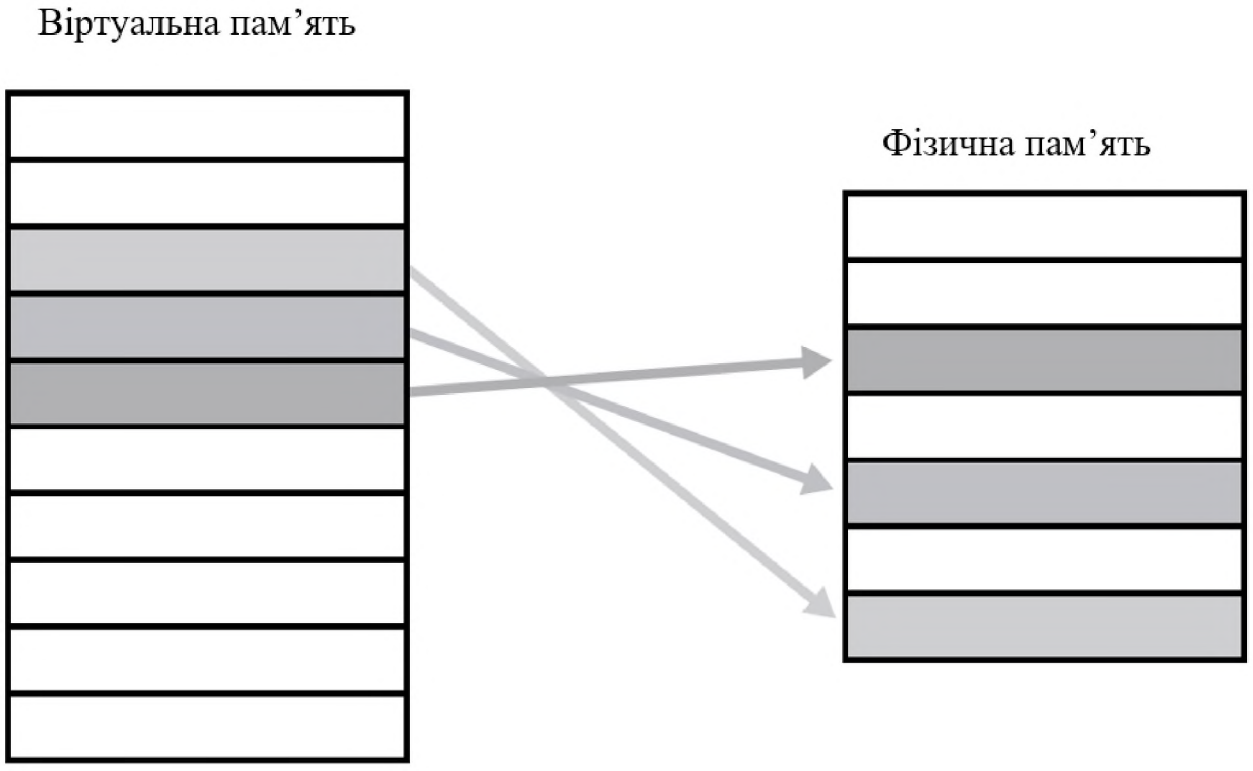

Рисунок 1.1 – Відображення віртуальної пам'яті на фізичну

Окрім розділення адресного простору процесу на окремі ділянки, що значно пришвидшує роботи операційної системи та полегшує знаходження ділянки пам'яті для процесу за рахунок того, що немає необхідності знаходити одне неперервне місце, механізм віртуальної пам'яті має велике значення і у реалізації захисту виконуваного коду програм.

Розподіляючи оперативну пам'ять між процесами, операційна система встановлює для кожного процесу права на кожну з цих ділянок. Таким чином процес має доступ до усього свого віртуального адресного простору і не має прав на інші ділянки пам'яті, які знаходяться поза межами його простору.

За допомогою цього механізму процесам зловмисників, які тим чи іншим чином потрапили на комп'ютер та були запущені на виконання, не вдасться просто взяти і переписати програмний код інших програм, порушивши їх цілісність.

Однак цей механізм захищає процеси лише від безпосереднього втручання у їх віртуальний адресний простір. Сучасні розробники шкідливого програмного забезпечення не використовують атаки такого роду. Натомість існують атаки, що здатні порушити цілісність програмного коду іншими методами. Частіше за все вони націлені на розширення адресного простору процесу, на який здійснюється атака з метою завантаження туди стороннього шкідливого коду, який потім виконується вже у віртуальному адресному просторі процесу.

1.1.2 Механізм запобігання виконанню даних

DEP (Data Execution Prevention) – це механізм захисту пам'яті, який використовується в операційній системі Windows для запобігання атакам, пов'язаним з виконанням коду з областей пам'яті, призначених для зберігання даних. DEP забезпечує захист від атак, таких як атаки на переповнення буфера, атаки з використанням шкідливого коду в пам'яті та інших типів атак, які намагаються використовувати область пам'яті, що виконується, для зловмисних цілей.

Захист пам'яті у операційній системі Windows реалізований таким чином, що кожна ділянка пам'яті у адресному просторі процесу має певні права, що забезпечують її захист від втручання ззовні та неконтрольованого доступу зсередини. До версії Windows NT у разі виникнення ситуації, за якої дані стеку помилково перезаписувалися, коли не вистачало місця для запису даних у певний виділений для них буфер, зловмисники могли навмисне підмінити тим чи іншим чином занадто великий потік даних, який програма може записати на стек. Цей потік даних міг виявитися програмним кодом, який у такому разі був би записаний у адресний простір програми. На той час усі ділянки пам'яті могли бути використані як для запису та збереження даних, так і для їх виконання. Тобто цей шкідливий код міг бути легко виконаний, адже усі ділянки пам'яті мати право на виконання програмного коду, який у них може бути записаний. У версії ж Windows NT був реалізований механізм DEP (Data execution prevention) – запобігання виконанню даних.

Механізм роботи DEP полягає у поділі віртуальної пам'яті на дві області: область виконання та область даних. Область виконання (також звана виконуваною) містить виконуваний код, включаючи операційну систему, драйвери та додатки. Область даних (також звана невиконуваною) містить дані, такі як стек викликів, купа, локальні змінні та інші дані, що використовуються програмою.

DEP позначає певні сторінки пам'яті в області даних як «не мають права на запуск». Це означає, що код, розташований у цих сторінках, не буде виконуватися, навіть якщо зловмисник спробує його виконати. Якщо атака спробує записати код, що виконується, в область даних і виконати його звідти, DEP виявить таку небажану поведінку і перерве виконання коду, що запобіжить успішному виконанню атаки[3].

1.1.3 Рандомізація розміщеного адресного простору

ASLR (Address Space Layout Randomization) – це механізм захисту цілісності програмного коду, що застосовується в операційній системі Windows для захисту від атак, які базуються на передбачуваності розташування в пам'яті різних компонентів програми.

ASLR особливо корисна для запобігання експлойтів, які намагаються використовувати відомі адреси функцій або вразливості, знаючи, де вони розташовані в пам'яті. Сутність цього механізму полягає у випадковому розподілі базових адрес, за якими розміщуються модулі, динамічні бібліотеки та інші компоненти програм.

Випадкову адресу мають наступні елементи програми:

– образ виконуваного файлу;
– динамічні і статичні бібліотеки;
– стек;
– куча.

В операційних системах Windows ASLR була вперше представлена в Windows Vista і Windows Server 2008. З того часу вона була постійно покращувалася та розширювалася.

Дана технологія створена для ускладнення використання вразливостей системи. Проте це лише механізм, який робить аналіз складнішим.

Існують кілька способів обходу ASLR, які можуть бути використані зловмисниками:

– не-ASLR модуль: не усі бібліотеки, які підключається до програми підтримують механізм рандомізації адресного простору. Такому випадку розміщення такого модулю буде відбуватися завжди за фіксованою адресою, яку може використати зловмисник;

– атаки перебору: зловмисники можуть здійснювати атаки методом перебору, генеруючи випадкові адреси, що передаються при переповненні буферу та намагаючись звернутися до них. При знаходженні дійсної адреси вони можуть

використовувати її для атаки. Хоча перебір є трудомістким та повільним, він досить поширений;

– заповнення адресного простору: у випадку, коли адресний простір системи майже повністю заповнений, вона може вимикати механізм ASLR для динамічних бібліотек, що підключається до процесів. Таким чином зловмисник може отримати фіксовану адресу модуля.

Методів обходу існує більше і не усі вони відомі розробникам операційної системи Windows та стороннього програмного забезпечення, але усі вони мають на меті одне – встановлення базової адреси того чи іншого модуля.

1.2 Програми, процеси та їх вразливості

На перший погляд програми та процеси схожі. У повсякденному використанні дані два терміни часто взаємозамінні навіть серед розробників програмного забезпечення. Але насправді вони докорінно відрізняються.

Програма – це статична послідовність інструкцій, яка зберігається у дисковому просторі комп'ютера та може бути запущена для виконання.

Процес – це екземпляр програми, яка виконується на комп'ютері. Він являє собою контейнер для набору ресурсів, що використовуються при виконанні екземпляру програми. На найвищому рівні абстракцій процес включає в себе наступне:

– закритий віртуальний адресний простір, що є набором адрес віртуальної пам'яті, якою процес може користуватися;

– виконувану програму, що містить програмний код та дані, які відображаються на віртуальний адресний простір процесу;

– набір відкритих дескрипторів різноманітних системних ресурсів (файлів, портів, ключів реєстру і т.д.);

– пов’язане з процесом середовище безпеки, що називається маркером доступу, який ідентифікує користувача, групу безпеки, права доступу, сесію та стан облікового запису користувача;

– унікальний ідентифікатор, що є ідентифікатором процесу (process identifier, або pid) та ідентифікатор батьківського процесу (Parent process identifier, або ppid);

– як мінімум один потік виконання.

Атаки з метою виконання стороннього коду всередині адресного простору програми частіше за все виконуються за допомогою втручання у віртуальний адресний простір процесу. Тобто вже після того як програма завантажена у оперативну пам’ять і створене умовно захищене середовище її виконання. Останнім часом це стало значно актуальніше, адже атаки на програми призводять до руйнування цифрового підпису програм і такі атаки вкрай легко виявляються[1][2].

1.3 Загрози виконання стороннього коду у адресному просторі процесу

Багато років розробники операційної системи Windows працюють над вдосконаленням системи захисту виконуваного коду. Причиною є численні атаки на процеси, що виконуються на комп’ютері, що мають на меті виконання стороннього коду у адресному просторі іншої програми і, як наслідок, порушення цілісності виконуваного коду. Процес, виконуючи ті чи інші дії у системі користувача, користується його довірою, адже користувач розуміє, що запустив саме ту програму, що скачав з інтернету або встановив разом з системою. Виконання ж стороннього коду у адресному просторі такої програми і має на меті введення в оману людини, яка може не підозрювати, що її програма була атакована та стала виконувати невластиві їй речі.

Потрапивши у віртуальний адресний простір деякої програми шкідливий код має можливість отримати доступ до будь-якої інформації, яка оброблюється у

даному процесі. Це означає, що можуть бути порушені усі властивості інформації: конфіденційність, цілісність та доступність.

Часто інжекція шкідливого коду відбувається для отримання привілеїв та прав доступу до захищених ресурсів, до яких має доступ тільки цей процес. Таким чином зловмисник виконуватиме взаємодію з цими ресурсами, видаючи себе за довірену програму. В цілому, цілі можуть бути найрізноманітніші, але майже завжди вони будуть шкідливими.

Таким чином можна зробити висновок, що реалізація механізмів захисту від атак, що направлені на порушення цілісності коду шляхом інжекції та виконання стороннього коду в адресному просторі програми є критично небезпечними.

Усі подібні атаки об'єднує одне – для їх реалізації користувач тим чи іншим чином має завантажити та запустити шкідливу програму на своєму комп'ютері. Часто для цього їм необхідні права адміністратора, які вони можуть отримати, якщо користувач їх їм надасть при запуску. У цій ситуації також немає нічого дивного, адже користувачі операційної системи часто встановлюють різноманітне програмне забезпечення. Нерідко саме на етапі встановлення, яке вимагає надання прав адміністратора виконується атака.

Встановлення антивірусів є корисною практикою, але дуже часто вони не виявляють подібні атаки, адже вони часто реалізуються за допомогою стандартного програмного інтерфейсу операційної системи Windows та розцінюються як правомірні. Тому набагато ефективніше реалізовувати програмний захист саме у програмах, як його потребують, а не покладатися на сторонній захист з боку операційної системи або антивірусів[1][4].

1.4 Функції перехоплення системних викликів

Будь-яка програма тим чи іншим чином використовує базові функції програмного інтерфейсу операційної системи Windows. Усі ці функції виділені у

окремі системні бібліотеки динамічної компоновки. Для кожного виклику системної функції програма має завантажити певну DLL у свій адресний простір, знайти у ній функцію з певним ім'ям та зберегти вказівник на неї. Зазвичай це відбувається автоматично і не потребує втручання програміста. Але існує спосіб заміни значення цього вказівника на будь-який інший. Це може призвести до того, що при виклику, наприклад, функції WriteFile буде викликатися не системна функція а та, на яку вказує відповідний вказівник, який може бути змінений кодом, що виконується у віртуальному адресному просторі даної програми. Зазвичай оригінальну адресу системної функції зберігають у інший вказівник, щоб не втратити можливість її виклику.

Функції підміни або перехоплення зазвичай називаються хук-функціями або просто хуками. Для коректного їх виклику замість системних функцій вони повинні мати той самий прототип(тип значення, що повертається та параметрів), щоб була можливою зміна вказівника який вказуватиме не на оригінальну функцію, а на функцію перехоплення.

Після виклику деякої функції, на яку встановлений хук, управління переходить у цю хук-функцію, яка, знаходячись у адресному просторі виконуваного процесу, може робити будь-що з переданими їй параметрами[6].

Даний механізм як і багато інших може бути використаний у різних цілях. Наприклад, він часто використовується зловмисниками для отримання доступу до параметрів, які передаються для виклику системних функцій. Таким чином можливо легко порушити усі властивості інформації, яка передається.

Однак даний механізм доцільно використовувати і для захисту програм. Часто атаки, що які прагнуть порушити цілісність виконуваного коду прагнуть виконати деякі стандартні системні функції у віртуальному адресному просторі програми. Це може бути виконано і вбудованими цілком документованими методами програмного інтерфейсу операційної системи Windows. Але ж це означає, що дані операції будуть виконані без відома програми, на яку здійснюється атака і таким чином вона не

матиме інформації, чи дійсно їй потрібно викликати ту чи іншу функцію з певними параметрами і не має можливості на це вплинути.

Саме для таких ситуацій можна використовувати хуки. Розробник програми може самостійно встановити хук-функції на визначені системні функції, які будуть перехоплювати виклики оригінальної функції. Таким чином з'являтиметься можливість перевірки необхідності виклику даної функції та переданих параметрів. У випадку, коли у хук-функції усі перевірки виконуються, вона викликає оригінальну функцію. Якщо ні – не викликає і таким чином блокує операцію.

Існують декілька різних бібліотек для C++, які реалізують встановлення хуків на системні функції і усі вони безкоштовні. У даній роботі буде використана бібліотека MHook. Вона є найпростішою у використанні, найоптимальнішою у швидкості роботи та показниками стабільності. Але може бути використана будь-яка з наступних[5]:

– EasyHook;
– Microsoft Detour;
– Nekta Deviare.

1.5 Висновок

У першому розділі була розглянута операційна система Windows, вбудовані в неї механізми захисту цілісності виконуваного коду, до яких відносяться:

- віртуальний адресний простір;
- механізм запобігання виконанню даних;
- рандомізація адресного простору.

Розібрані поняття програми та процесів, різниці між ними та пов'язаних з ними вразливостей. Проаналізовані також загрози та можливі наслідки виконання стороннього коду у адресному просторі процесу та функції перехоплення системних

викликів як механізм, що може застосовуватися не лише у злочинних цілях, а й для захисту програмного продукту.

РОЗДІЛ 2. АНАЛІЗ ВІДОМИХ МЕТОДІВ ВИКОНАННЯ СТОРОННЬОГО ВИКОНУВАНОГО КОДУ У АДРЕСНОМУ ПРОСТОРІ ІНШОЇ ПРОГРАМИ ТА РЕАЛІЗАЦІЯ ЗАХИСТУ

Не зважаючи на наявність в операційній системі Windows механізмів захисту виконуваного коду, дуже часто все ж таки відбуваються атаки на програми та процеси, які тим чи іншим чином порушують цілісність виконуваного коду програм. Частіше за все, неможливо назвати єдину причину, яка призвела до реалізації загрози. Саме комбінація недосконалості програмного продукту, операційної системи та людський фактор є основною причиною порушень безпеки при використанні програмних засобів.

Наприклад, якщо є програмний засіб, що має недоліки, пов'язані з його захищеністю, та користувач, який завантажив неперевірене програмне забезпечення, яке має втрутитися у нормальну роботу програмного засобу, який використовується людиною, але в операційній системі реалізовані захисні механізми, які попереджують усі спроби шкідливих програм порушити цілісність програмного коду інших програм та процесів, загроза, скоріш за все, не буде реалізована та буде блокована операційною системою.

Якщо ж деякий програмний додаток та операційна система мають недоліки, що дозволяють зловмиснику перешкоджати нормальній роботі програм та системи, але користувач не допускає їх потрапляння на комп'ютер, загроза також не буде реалізована.

І у випадку, коли неуважний користувач тим чи іншим чином завантажує шкідливий програмний засіб на комп'ютер, на якому встановлена операційна система, яка не має надійних механізмів захисту цілісності виконуваного коду, але користується він програмами, які мають свої механізми захисту коду, вірогідність реалізації загрози значно знижується.

Але ніколи не буде такої ситуації, за якої найобережніший у використанні комп'ютеру користувач запускатиме програмний засіб, що має вбудовані механізми захисту від усіх існуючих загроз цілісності виконуваного коду на комп'ютері зі встановленою операційною системою, яка реалізує абсолютно надійні механізми захисту.

З цієї причини єдиним можливим варіантом захисту цілісності програмного коду, який залежить від розробників програмного забезпечення є реалізація механізмів захисту від найпоширеніших атак.

До таких атак можна віднести:

1) інжекція динамічної бібліотеки;
2) обернено орієнтоване програмування;
3) пряма інжекція стороннього коду з подальшим виконанням (PE Injection).

2.1 Інжекція DLL

2.1.1 Поняття DLL та загрози, пов'язані з їх використанням

DLL розшифровується як Dynamic Link Library – бібліотека динамічної компоновки або динамічна бібліотека. Файли цих бібліотек мають відповідне розширення «.dll». Вони можуть мати ту саму структуру(секції коду, даних та ресурсів), що і виконувані «.exe» файли. Але вони можуть бути запущені для виконання. Найчастіше вони використовуються для зберігання спільного для багатьох програм коду, який, з точки мінімізації розміру програм, доцільніше винести у окремий файл, який, за потреби, можуть використовувати програми, що його потребують[1].

Бібліотеки динамічної компоновки несуть потенційні загрози цілісності виконуваного коду і, як наслідок, інформації, яка оброблюється з декількох причин:

– можливість підміни файлу динамічної бібліотеки;

– наявність точки входу при підключенні до процесу, у якій виконується програмний код.

Перша загроза вже давно не є актуальною, адже завантажуючи бібліотеку за власним бажанням процес має можливість усіляко перевірити даний файл. До заходів перевірки можна віднести перевірку цифрового підпису, імені, шляху до файлу тощо. Після виконання усіх перевірок приймається рішення завантажувати динамічну бібліотеку чи ні залежно від того чи дійсно це той самий файл, використання якого замислювали розробники програмного засобу.

Проте існує вразливість, пов'язана з використанням динамічних бібліотек, яка експлуатується зловмисниками до сих пір. Вона базується на тому, що в операційній системі Windows є можливість створення потоку у іншому процесі, який виконуватиме функцію, яка знаходиться в адресному просторі того процесу. Таким чином зловмисники викликають функцію завантаження бібліотеки у адресний простір процесу, а динамічна бібліотека, підключаючись до процесу, виконує програмний код вже у адресному просторі процесу, на який здійснюється атака.

Бібліотеки динамічної компоновки мають не лише функції для експорту, що можуть бути використані іншими процесами. Вони також мають свою точку входу у даний файл – функцію DllMain. Вона не є обов'язковою та майже завжди ігнорується у бібліотеках, що використовуються лише для зберігання та експорту коду.

Функція DllMain приймає три параметри:

1) HINSTANCE hinstDLL – дескриптор, що вказує на цей завантажений модуль (динамічну бібліотеку);

2) DWORD fdwReason – причина, з якої дана функція була викликана;

3) LPVOID lpvReserved – зарезервований поки що параметр, який не використовуються.

Повертає функція булеве значення 1 або 0, що сигналізує коду, який завантажує дану бібліотеку про успішність даної операції.

В одному процесі функція DllMain може бути викликана багато разів, але з різними цілями, які описуються другим параметром вхідної функції – fdwReason. Він може мати одне з чотирьох значень:

1) DLL_PROCESS_ATTACH – коли DLL завантажується у віртуальний адресний простір поточного процесу в результаті запуску процесу або в результаті завантаження під час виконання;

2) DLL_PROCESS_DETACH – коли DLL вивантажується з віртуального адресного простору процесу, оскільки його було завантажено невдало або кількість посилань досягла нуля (процеси або припинили роботу, або викликали FreeLibrary стільки ж разів, скільки завантажили бібліотеку);

3) DLL_THREAD_ATTACH – коли поточний процес створює новий потік. Коли це відбувається, система викликає функцію точки входу всіх бібліотек DLL, наразі підключених до процесу. Виклик здійснюється в контексті нового потоку;

4) DLL_THREAD_DETACH – коли завершується потік, викликається функція DllMain з даним параметром у контексті цього потоку.

Зазвичай процедура завантаження динамічної бібліотеки виконується самим процесом, що викликає функцію LoadLibrary або LoadLibraryEx.
Проте існує спосіб виклику у віртуальному адресному просторі деякого функцій, що вже знаходяться у його адресному просторі.

2.1.2 Створення віддаленого потоку у іншому процесі

В операційній системі Windows існує можливість створити новий потік у іншому процесі, який виконуватиме функцію, що вже знаходяться у його адресному просторі. Вона реалізується за допомогою функції CreateRemoteThread.

Частіше за все дана функція використовується програмами для відлагодження програмних засобів на етапі їх розробки. Але можливість створення потоку у іншому процесі робить можливим її використання і у протиправних діях.

Дана функція має наступний прототип:

```
HANDLE CreateRemoteThread(
  HANDLE                 hProcess,
  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  SIZE_T                 dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID                 lpParameter,
  DWORD                  dwCreationFlags,
  LPDWORD                lpThreadId
);
```

Вона приймає параметром дескриптор процесу, функцію, що має бути виконана у новому потоці та параметр, що передаватиметься у цю функцію

За допомогою цієї функції можна створити у певному процесі новий потік, що викликає функцію LoadLibrary, якій передається параметром ім'я динамічної бібліотеки.

Таким чином можливо завантажити динамічну бібліотеку у віртуальний адресний простір іншого процесу. Як зазначено у пункті 2.1.1, DLL при підключенні до процесу викликає функцію входу і виконує певний код. Саме цей код і буде кодом, виконаним у адресному просторі іншого процесу. Даний код може бути будь-яким (навіть шкідливим).

Тому процеси мають реалізовувати механізми захисту від даного типу атак, адже виконання у адресному просторі процесу стороннього коду може призвести до порушення усіх властивостей інформації, яка оброблюється у ньому.

Частіше за все даний тип атак має продовження. Звичайне виконання певного коду може дати не так багато інформації у порівнянні з встановленням функцій перехоплення (хуків).

Послідовність подій, які відбуваються при інжекції DLL наведена на рис.2.1.

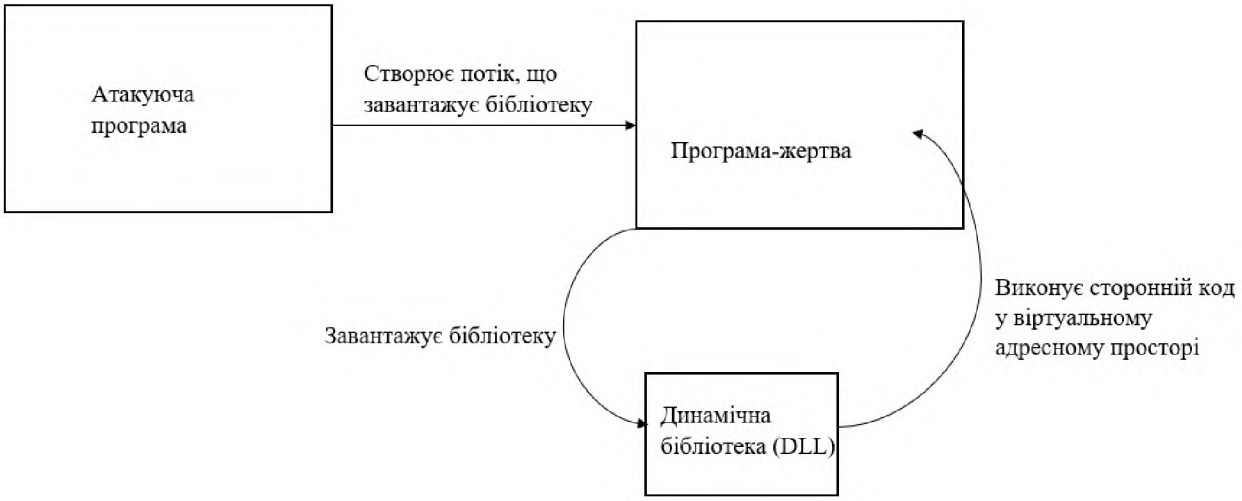

Рисунок 2.1 – Послідовність дій для інжекції DLL

2.1.3 Захист від інжекції DLL

Усі методи захисту програми від інжекції динамічних бібліотек, які, підключаючись, виконують сторонній код у адресному просторі програми базуються на механізмі, якій частіше за все використовується у парі з інжекцією DLL – це встановлення функцій перехоплення.

Як вже зазначалося, кожна програма для використання усіх системних функцій зберігає вказівники на ці функції для подальшого їх виклику. Та існує механізм підміни цієї адреси, що призводить до того, що програма, викликаючи звичайну, на перший погляд, системну функцію, яка знаходиться у системній бібліотеці динамічної компоновки, насправді викликає зовсім іншу функцію, код якої знаходиться і виконується у адресному просторі програми. До того ж, отримуючи усі дані параметрів, що передаються для виклику системній функції.

Але даний механізм виклику додаткової «функції-прокладки» може бути використаний не тільки зловмисниками, але й розробниками програмного забезпечення, які бажають дещо змінити стандартну поведінку програм при використанні системних функцій та зробити програмний засіб безпечнішим.

Оскільки для інжекції сторонньої динамічної бібліотеки у програму зловмисник викликає функцію CreateRemoteThread у своєму адресному просторі, розробники захищеної програми не мають можливості вплинути на неї, окрім самостійного втручання у її адресний простір та встановлення хуку на цю функцію для попередження її виклику. Але це перетворюватиме розробників на зловмисників, які самі пишуть код для втручання у виконання сторонньої програми.

Знаючи, що програма-зловмисник викликає функцію CreateRemoteThread з метою виклику у адресному просторі звичайної системної функції LoadLibrary, яка завантажує бібліотеку у адресний простір програми, що її викликає, можна зробити висновок, що необхідно тим чи іншим чином вплинути на виклик функції LoadLibrary.

Використовуючи можливість встановлення функцій перехоплення на виклики системних функцій, доцільно власноруч встановити функцію хук на LoadLibrary, щоб у разі її виклику в результаті створення віддаленого потоку у додатку шкідливою програмою, була можливість проведення дій, спрямованих на перевірку даної динамічної бібліотеки. Наприклад, можливо перевірити її цифровий підпис, ім'я, шлях та багато інших параметрів, які можуть вказати на безпечність або небезпечність завантаження даної бібліотеки.

У разі будь-яких порушень, можна просто не викликати оригінальну функцію. Таким чином блокуючи дану операцію. Доцільно також реалізувати систему збереження інформації про дану подію, щоб користувачі та розробники мали можливість дізнатися про спроби атаки на захищений застосунок.

Дані дії повинні бути виконані при старті програми, щоб не було такого проміжку часу, коли програма-зловмисник може втрутитися у адресний простір програми, що захищається. Механізм захисту не є дуже складним та може бути реалізований навіть розробником початкового рівня.

Таким чином можна встановити певну послідовність дій, які повинні бути виконані, щоб захиститися від даного типу атак:

1) обрати та під'єднати до проєкту будь-яку бібліотеку, що реалізує встановлення хуків на функції;

2) завантажити бібліотеку, у якій знаходиться функція, на яку необхідно встановити хук. Функція LoadLibrary знаходиться у бібліотеці kernel32.dll;

3) отримати вказівник на неї стандартним способом;

4) створити функцію з тим самим прототипом, що і у LoadLibrary, щоб було можливо підмінити їх адреси;

5) реалізувати у вищеназваній функції логіку, що перевірятиме безпечність динамічної бібліотеки тим чи іншим чином. У разі проходження перевірок викликається оригінальна функція, вказівник на яку власноруч зберігається. А у разі виявлення проблем з DLL оригінальна функція не викликається, тим самим блокуючи завантаження шкідливої динамічної бібліотеки.

За можливості, слід взагалі відмовитися від використання функції LoadLibrary, адже лише з її можливістю можна у режимі користувача інжектувати динамічну бібліотеку у адресний простір іншого процесу. Причиною цього є співпадіння кількості параметрів прототипу функції, яка може бути запущена у віддаленому потоці виконання за допомогою функції CreateRemoteThread та самої функції LoadLibrary. Вони обидві приймають один параметр.

Тому слід використовувати нову версію цієї функції, яка називається LoadLibraryEx. Вона має наступний прототип:

```
HMODULE LoadLibraryEx(
    LPTSTR lpLibFileName,
    HANDLE hFile,
    DWORD  dwFlags
);
```

Послідовність подій, які відбуваються при інжекції DLL у процес, що захищений від даної атаки наведено на рис. 2.2.

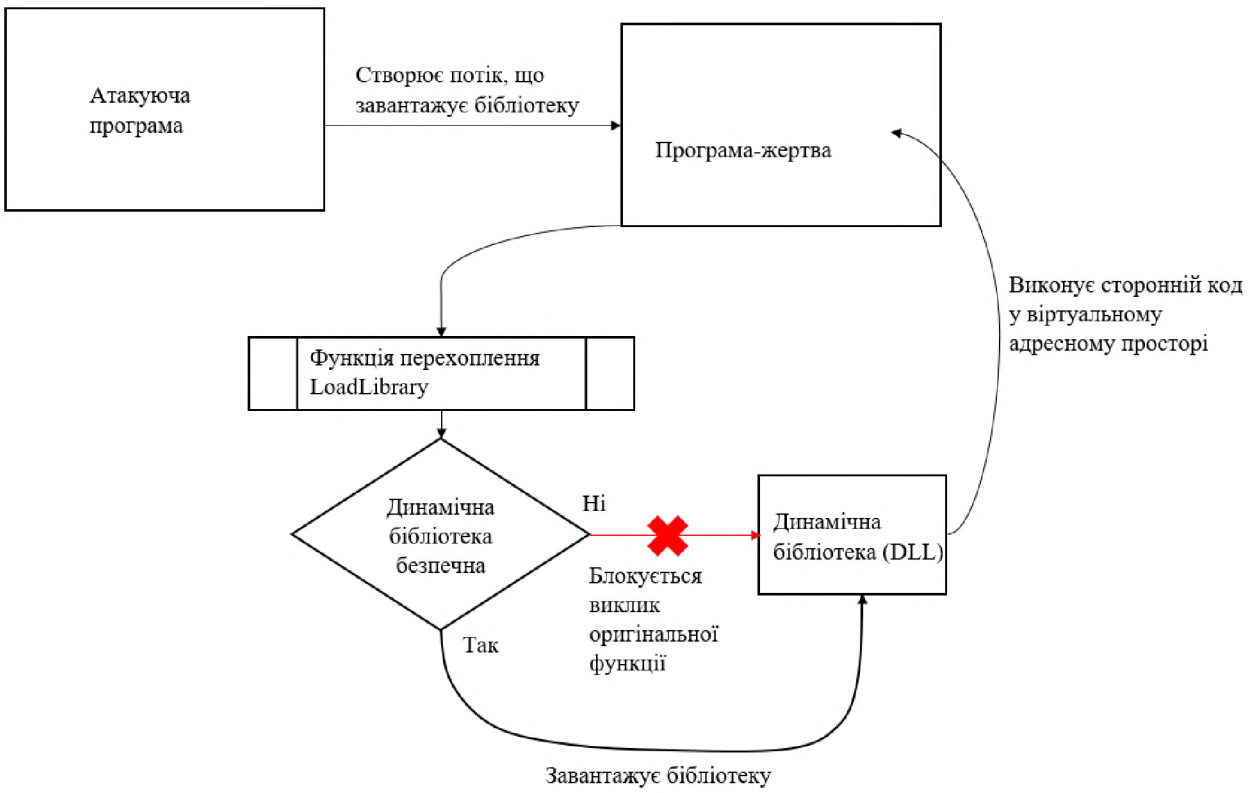

Рисунок 2.2 – Перехоплення функції LoadLibrary

Важливо помітити, що вона має вже 3 параметри та не може бути використана для створення віддаленого потоку. До того ж третім параметром вона приймає специфічні прапори, які можуть змінити процес завантаження динамічної бібліотеки. При встановленні прапору DONT_RESOLVE_DLL_REFERENCES функція входу DllMain, яка зазвичай викликається при підключенні динамічної бібліотеки, не буде викликана. Але на даний момент програмний інтерфейс операційної системи Windows не дає можливості виклику даної функції у віддаленому потоці іншого процесу і, скоріш за все, у найближчому майбутньому вона не з'явиться, бо функції програмного інтерфейсу Windows проектувалися на багато років вперед та вкрай рідко додаються нові.

Код програмної реалізації засобу захисту наведений у додатку Б.

2.2 Загроза зворотно-орієнтованого програмування (ROP-атака)

ROP – це абревіатура, що розшифровується як орієнтоване на повернення програмування або зворотно-орієнтоване програмування. Цей метод використовує вразливість переповнення буферу, що розташований на стеку та дозволяє обходити захисний механізм, який забороняє одночасне виконання та запис даних на ділянці пам'яті (DEP)[7].

2.2.1 Актуальність загрози

Атаки повторного використання коду, що базуються на зворотно-орієнтованому
програмування (ROP), набувають все більшої популярності з кожним роком і можуть бути застосовані навіть в умовах роботи захисних механізмів сучасних операційні системи, у тому числі Windows. Даний вид атаки до сих пір залишається одним з найнебезпечніших через те, що для його реалізації не потрібно ніяк змінювати параметри віртуального адресного простору програми. Необхідно лише знайти недосконаліть програмного коду у даній програмі і використати її, тим чи іншим чином переповнивши буфер даних на стеку програми. Для цього потрібно лише щоб програма самостійно зчитала занадто великий буфер даних і перезаписала таким чином дані на стеку у власному віртуальному адресному просторі.

До того ж досить важкою задачею є розробка механізму захисту від даного типу атак, адже неможливо точно спрогнозувати, який код буде записаний і виконаний в результаті переповнення буферу.

Однак виходячи з того, що зловмисник прагне виконати код, який записується у віртуальний адресний простір програми в результаті переповнення буферу, можна зробити висновок, що він має змінити параметри захисту ділянки коду,

завантаженого в результаті недосконалості програмного коду. Такі дії можливо попередити, адже відомо, на що вони спрямовані.

2.2.2 Реалізація ROP-атаки

Відомо, що механізм DEP запобігає виконанню коду, розташованого на ділянках пам'яті, які призначені для збереження даних програми. Тобто у разі переповнення буферу, коли зловмисник може перезаписати дані на стеку, він не матиме можливості виконати цей код.

Таким чином ключовою перепоною на шляху хакера, котрий прагне завантажити сторонній код у адресний простір програми та виконати його є механізм DEP.

Стек викликів – це структура, яка працює за принципом «останній прийшов – перший пішов». При виконанні процесу використовується для збереження даних, що використовуються у функції та адреси, за якою необхідно повернути керування після завершення функції. При кожному виклику функції в нього додаються три види даних:

1) локальні змінні функції;

2) адрес повернення до якої потрібно повернути виконання коду після завершення функції;

3) параметри, передані у функцію.

У разі переповнення буферу адреса повернення виконання вказуватиме на зовсім інше місце. У разі контрольованого перезапису дана адреса може дійсно вказувати на певний код і програма перейде до його виконання після завершення поточної функції. Виходячи з цього, можна припустити, що стек буде перезаписаний таким чином, що у ньому з'являться багато нових адрес, які використовуються для повернення керування до функції, яка викликала поточну функцію.

Це призводить до того, що при виклику ассемблерної інструкції ret поточною функцією керування переходить до коду, на який вказує перезаписана зловмисником адреса.

Структурний вигляд стеку та розміщених у ньому даних наведено на рис. 2.3.

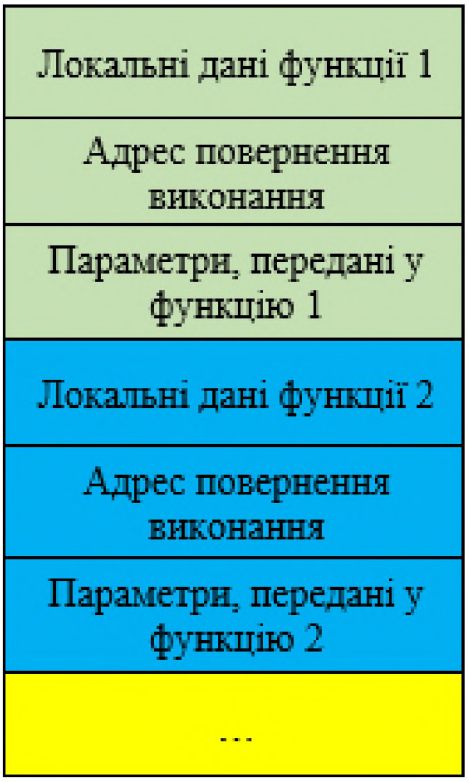

Рисунок 2.3 – Розміщення даних на стеку викликів

Адреса повернення виконання може вказувати лише на ділянку пам’яті, яка містить код та може бути виконана. В іншому випадку програма аварійно завершиться через механізм захисту DEP. Тому зазвичай нові адреси вказують на вже існуючий у віртуальному адресному просторі програми код, що виконує певні послідовності команд які завершуються інструкцією ret. Такі послідовності називаються гаджетами. Вони фактично є закінченнями будь-яких функцій. Це може бути або код програми, або код завантажених динамічних бібліотек.

Кожний гаджет виконує певні дії (наприклад, складає значення двох регістрів, змінює їх) та передає керування наступному гаджету. Вони зв'язуються в ланцюжок послідовно виконуваних шматочків коду. Таким чином, за допомогою ланцюжка гаджетів можна виконати певні шкідливі дії.

На відміну від звичайної програми, інструкції ROP-ланцюжка не розташовуються послідовно у пам'яті, а розбиваються на маленькі гаджети, зв'язані інструкціями, які отримують адресу наступного гаджету зі стеку.

Тобто ROP-ланцюжок – це послідовність вказівників на закінчення певних відомих функцій бібліотек динамічної компоновки, які при певній комбінації формують сторонню для програми логіку, яка виконується у адресному просторі програми.

Таким чином можливо виконати будь-який код, що вже знаходиться у програмі. Частіше за все, зловмиснику цього недостатньо, адже майже неможливо сформувати яку-небудь складну логіку, що виконувала б шкідливі дії[7].

Значно легше обійти механізм запобігання виконанню даних (DEP) та запустити записані дані як програмний код. Для цього зловмисники формують певну послідовність адрес, що вказують на код, послідовність виконання якого призводить до виклику функції VirtualProtect або VirtualProtectEx.

Перша функція має наступний прототип:

```
BOOL VirtualProtect(
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD  flNewProtect,
  PDWORD lpflOldProtect
);
```

Друга:

```
BOOL VirtualProtectEx(
  HANDLE hProcess,
```

```
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD  flNewProtect,
  PDWORD lpflOldProtect
);
```

За допомогою даних функцій можливо змінити параметри захисту будь-якої ділянки віртуального адресного простору програми. Першим параметром функції VirtualProtect передається адреса ділянки пам'яті, другим – розмір ділянки, третім – нове значення параметру захисту. Функція VirtualProtectEx має один додатковий параметр – дескриптор процесу, параметри ділянки віртуальної адреси якого змінюються. Але задля застосування її для іншого процесу, дескриптор повинен мати право доступу PROCESS_VM_OPERATION, що означає дозвіл на втручання у віртуальний простір. Дане право доступу завжди можна отримати, виконуючи операцію у віртуальному адресному просторі цього ж процесу.

Встановивши параметр захисту рівним PAGE_EXECUTE_READWRITE, ділянка пам'яті стає доступною на виконання і вимикається захист механізму DEP для цієї ділянки пам'яті. Таким чином стає можливим виконання стороннього програмного коду, розташованого на стеку у адресному просторі програми. Як зазначалося раніше, цей програмний код може бути записаний туди у разі виявлення зловмисником вразливостей програмного продукту, що може призвести до переповнення буферу даних та перезапису інформації у стеку.

2.2.3 Захист від ROP-атак

2.2.3.1 Усунення можливості переповнення буферу

Для реалізації атаки за допомогою зворотно-орієнтованого програмування зловмиснику спершу потрібно провести ретельний аналіз виконуваного файлу, щоб

виявити місця у коді, які потенційно можуть призводити до переповнення буферів тим чи іншим чином.

Досить часто переповнення буферу даних може відбуватися навіть без явних помилок зі сторони розробника програмного коду. Наприклад, можливе використання застарілих функцій та бібліотек, які були створені ще у часи, коли проблема атак через переповнення буферу ще не була настільки критичною. Такий код часто не виконує ніяких перевірок довжин даних і не перевіряє, чи дійсно розмір буферу достатній для їх розміщення.

Найвідомішими функціями мов C/C++, що не виконують перевірок розмірів даних є наступні:

- printf – виведення даних у консоль;
- sprintf – запис даних у буфер символів;
- strcat – поєднання символьних рядків;
- strcpy – копіювання рядку;
- gets – зчитування символьного рядка з консолі в буфер.

Варто відмовитися від їх використання та віддати перевагу новішим версіям цих функцій. Майже усі вони мають безпечні аналоги. Якщо таких немає, завжди є альтернативи використанню тих чи інших небезпечних з точки зору переповнення буферу функцій.

2.2.3.2 Програмна реалізація захисту від ROP-атак

Відомо, що для виконання стороннього коду, який розміщується на стеку у адресному просторі програми, необхідно обійти захисний механізм DEP, який робить неможливим використання даних на деякій ділянці у якості програмного коду. Зробити це можна єдиним чином – викликати функцію VirtualProtect або VirtualProtectEx, які змінюють параметри захисту певної ділянки віртуальної пам'яті. Таким чином при встановленні параметру захисту ділянки пам'яті, на якій

розташований сторонній код, записаний у результаті переповнення буферу, стає можливим виконання цього коду, адже вбудований механізм захисту даних від виконання тепер має інформацію, що дана ділянка пам'яті має права на виконання.

Зловмисник прагне викликати функцію VirtualProtect або VirtualProtectEx у віртуальному адресному просторі програми, для якої реалізується захист, використовуючи шматочки коду, розміщені у адресному просторі програми. Однак є можливість попередити її виклик, встановивши функцію перехоплення, за допомогою якої можливо перевірити, чи безпечне використання даної функції з переданими їй параметрами.

Розробнику програмної реалізації захисту від даної атаки слід перевіряти параметр функції flNewProtect на наявність у ньому біта PAGE_EXECUTE_READWRITE, який сигналізував би про спробу перетворити деяку ділянку віртуального адресного простору програми таку, що може містити код, який може бути виконаний.

У разі перехоплення виклику цієї функції, яка має даний біл, слід не викликати оригінальну функції з функції перехоплення для запобігання реальної зміни параметрів захисту даних, що могли бути перезаписані зловмисником, шляхом переповнення буферу. Для сигналізування про невдале завершення функції VirtualProtect та VirtualProtectEx повертають значення 0[8]. У випадку відсутності прапору PAGE_EXECUTE_READWRITE у параметрі flNewProtect необхідно викликати оригінальну функції, пропустивши даний виклик, адже він не несе потенційної загрози цілісності програмного коду.

Код програмної реалізації засобу захисту наведений у додатку В.

2.2.3.3 Використання найновіших версій механізмів захисту виконуваного коду

Механізми рандомізації розміщеного адресного простору (ASLR) та запобігання виконання даних (DEP) є серйозними перепонами на шляху зловмисників, що прагнуть виконати сторонній код у адресному просторі програм. Тому розробники операційної системи Windows працюють над створенням більш досконалих систем захисту програм. Оновлюються також компілятори програм, додаючи також нові можливості забезпечення захисту виконуваного коду. Тому вкрай важливо встановлювати оновлення операційної системи на стороні користувача та оновлювати програму, використовуючи нові версії мови програмування, за допомогою якої вона була написана.

2.3 Загроза прямої інжекції стороннього коду з подальшим виконанням (PE Injection)

Віртуальний адресний простір будь-якої програми, у якому зберігаються різноманітні дані для її виконання захищений багатьма механізмами, які мають на меті запобігання зміні даних у ньому з боку сторонніх процесів. Після завантаження портативного виконуваного файлу у оперативну пам'ять та запуску процесу неможливо просто переписати його дані. Проте операційна система Windows надає можливість розробникам програмного забезпечення створювати нові ділянки віртуальної пам'яті, що будуть відноситися до певного процесу. У цю нову ділянку пам'яті можуть бути записані будь-які дані. У тому числі і програмний код. Залишається тільки почати виконання даної ділянки коду. І це також можливо зробити стандартними документованими засобами операційної системи Windows[4].

2.3.1 Актуальність загрози

Загроза реалізації даного типу атаки є однією з найсерйозніших. Причиною цього є те, що для її виконання не потрібно викликати ніякі стандартні функції програмного інтерфейсу операційної системи Windows у віртуальному адресному

просторі іншої програми. Усі дії аж до моменту запуску шкідливого коду у іншій програмі виконуються виключно у програмі, яка здійснює атаку. З цієї причини неможливо встановити функцію перехоплення виклику у програмі, яку необхідно захистити для фільтрації та блокування підозрілих викликів функцій.

Також не є завадою для даної атаки вбудовані в операційну систему Windows механізми захисту від виконання даних (DEP) та рандомізації адресного простору (ASLR), адже не має значення точність розміщення стороннього коду у адресному просторі програми-жертви.

Єдиною завадою для реалізації даної атаки за відсутності спеціальних механізмів захисту самої програми, на яку здійснюється атака, є отримання прав адміністратора. Але при повсякденному використанні різноманітного програмного забезпечення користувач регулярно стикається з ситуацією, коли для роботи тієї чи іншої програми необхідно запустити її від імені адміністратора.

Виходячи з цього, можна зробити висновок, що неможливо розробити механізм захисту від даного типу загрози, використовуючи виключно код у режимі користувача, тобто у програмі, на яку здійснюється атака.

Тому стає необхідним створення додаткового програмного засобу, який мав би доступ до усіх процесів та міг би модифікувати їх поведінку таким чином, щоб вони не змогли отримати доступ до процесу, який необхідно захистити. Для реалізації даної задачі можливе лише використання драйверу, код якого виконується в режимі ядра.

Розробка такого типу програмного забезпечення є вкрай складною та потребує високого рівня досвіду розробника програмного забезпечення. Також для використання драйверу в операційній системі Windows обов'язковою є наявність в цього драйверу цифрового підпису від компанії Microsoft.

Усі ці складнощі призводять до того, що переважна більшість розробників програмних засобів відмовляються від використання драйверів.

2.3.2 Послідовність дій і опис функцій, що використовуються для атаки

Послідовність виклику функцій програмного інтерфейсу операційної системи Windows для реалізації наступна:

1) OpenProcess – дана функція призначена для отримання дескриптору процесу. Отримавши дескриптор, необхідно додати ділянку віртуальної пам'яті, записати туди код та виконати його.

Дана функція має наступний прототип:

```
HANDLE OpenProcess(
    DWORD dwDesiredAccess,
    BOOL  bInheritHandle,
    DWORD dwProcessId
);
```

Значення параметрів:

– dwDesiredAccess визначає права доступу до процесу, що мають бути отримані. Для реалізації даної атаки він повинен дорівнювати PROCESS_ALL_ACCESS, що означає повний доступ. Для отримання такого доступу процес, що інжектує сторонній код повинен мати права адміністратора;

– bInheritHandle визначає чи будуть дочірні процеси, створені інжектором наслідувати даний дескриптор. У даному випадку цей параметр не має значення;

– dwProcessId визначає ідентифікатор процесу, дескриптор якого намагається отримати зловмисник.

2) VirtualAllocEx – дана функція використовується для створення нової ділянки пам'яті, що буде відноситися до віртуального адресного простору зазначеного процесу.

Її прототип виглядає наступним чином:

```
LPVOID VirtualAllocEx(
      HANDLE hProcess,
      LPVOID lpAddress,
```

```
    SIZE_T dwSize,
    DWORD  flAllocationType,
    DWORD  flProtect
);
```

Значення параметрів:

– hProcess приймає дескриптор процесу, до віртуального адресного простору якого необхідно додати нову ділянку пам'яті. У даному випадку це має бути дескриптор, отриманий на попередньому кроці за допомогою функції OpenProcess;

– lpAddress визначає бажану початкову адресу ділянки пам'яті. Цей параметр може бути рівним нулю. У такому разі адреса буде обрана операційною системою самостійно;

– dwSize визначає розмір нової ділянки пам'яті;

– flAllocationType визначає тип виділення пам'яті. Для даної атаки використовується одразу два бітові значення MEM_COMMIT та MEM_RESERVE;

– flProtect визначає тип захисту нової ділянки пам'яті. Для подальшого виконання стороннього коду зловмисники встановлюють даний параметр рівним PAGE_EXECUTE_READWRITE. Таке значення параметру означає, що дані, що знаходяться у цій ділянці можуть бути прочитані, переписані та виконані.

Дана функція є базою для даного типу атаки, адже дозволяє виділити нову ділянку пам'яті, що відноситиметься до адресного простору програми, завантажити туди певний код та виконати його.

3) WriteProcessMemory – дана функція призначена для запису даних у ділянки пам'яті, що є частиною віртуального простору іншого процесу. Для її використання область даних повинна мати параметр захисту, що передбачає запис. Оскільки на попередньому кроці такий параметр був рівним PAGE_EXECUTE_READWRITE, виконання даної функції можливе.

Дана функція має наступний прототип:

```
BOOL WriteProcessMemory(
```

```
    HANDLE  hProcess,
    LPVOID  lpBaseAddress,
    LPCVOID lpBuffer,
    SIZE_T  nSize,
    SIZE_T  *lpNumberOfBytesWritten
);
```

Значення параметрів:

– hProcess вказує дескриптор процесу, у віртуальну пам'ять якого будуть записані дані. Це має бути дескриптор, отриманий за допомогою функції OpenProcess з пункту номер 1;

– lpBaseAddress вказує на адресу нової ділянки віртуальної пам'яті. Має бути передана адреса, отримана за допомогою функції VirtualAllocEx з пункту номер 2;

– lpBuffer вказує на буфер даних зі стороннім кодом. Цей сторонній код має являти собою деяку функцію, яку можна викликати та виконати у окремому потоці. Прототип даної функції має співпадати з типом LPTHREAD_START_ROUTINE, тобто повертати значення та приймати один параметр-вказівник;

– nSize визначає розмір у байтах буферу даних, які будуть записані;

– lpNumberOfBytesWritten повертає кількість записаних байтів даних і може бути проігнорований, адже ніяк не впливає на інжекцію.

4) CreateRemoteThread – дана функція вже використовувалася і була описана у атаці, за якої відбувалася інжекція DLL. Вона дозволяє створити новий потік у іншому процесі. У цьому потоці буде виконуватися функція, вказівник на яку буде переданий одним з параметрів функції. Значенням адреси цієї функції має бути адреса нових даних, які були додані за допомогою функції VirtualAllocEx та WriteProcessMemory.

Прототип даної функції виглядає наступним чином:

```
HANDLE CreateRemoteThread(
  HANDLE                 hProcess,
```

```
  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  SIZE_T                 dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID                 lpParameter,
  DWORD                  dwCreationFlags,
  LPDWORD                lpThreadId
);
```

Значення параметрів:

– hProcess визначає дескриптор процесу;

– LpThreadAttributes визначає атрибути безпеки створеного потоку. Можна використати атрибути за замовчанням, передавши 0;

– dwStackSize визначає розмір стеку. Може бути рівним 0 для використання значення за замовчанням;

– lpStartAddress – це вказівник на функцію, яка буде виконуватися у цьому потоці. Це має бути адреса, отримана за допомогою VirtualAllocEx, у яку записується стороння функція за допомогою WriteProcessMemory;

– lpParameter – це вказівник на параметр, що буде переданий у функцію. Може бути 0;

– dwCreationFlags визначає прапори, що змінюють поведінку на старті потоку. Не потрібні для даної атаки. Параметр має бути рівним нулю.

– lpThreadId повертає значення ідентифікатору нового потоку.

Таким чином за допомогою усього чотирьох системних функцій процес, що має права адміністратора може інжектувати сторонній код у програму, розширивши її віртуальний адресний простір, записавши у нього код і створивши новий потік, що виконає цю функцію.

2.3.3 Захист від прямої інжекції стороннього коду з подальшим виконанням

2.3.3.1 Огляд особливостей даної атаки

Даний тип атаки відрізняється від інших тим, що для інжекції стороннього коду у віртуальний адресний простір деякої програми не потрібно викликати будь-які стандартні функції програмного інтерфейсу операційної системи Windows у адресному просторі процесу, що атакується. У даному випадку виконується лише виклик функції, код якої з'являється у адресному просторі програми лише під час атаки та може бути будь-яким. Це робить неможливим встановлення хука на таку функцію на початку роботи програми, щоб перехопити даний виклик та блокувати його.

У даному випадку розробнику програмного забезпечення, яке слід захистити від даного типу атак залишається лише тим чи іншим чином зробити неможливим виконання однієї з чотирьох функцій, які створюють нову віртуальну пам'ять у процесі, записують туди дані та запускають їх виконання.

Тобто необхідно тим чи іншим чином контролювати виклик хоча б однієї з наступних функцій:

- OpenProcess;
- VirtualAllocEx;
- WriteProcessMemory;
- CreateRemoteThread.

Зробити це, використовуючи лише користувацький процес не є можливим, адже усі ці функції викликаються у адресному просторі іншого процесу. Теоретично, можливо самостійно інжектувати свій програмний код, який встановить хуки на усі ці функції всередині програми зловмисника, але такий підхід є вкрай неефективним, адже атаки на процеси є доволі витратними і у плані часу не програмну реалізацію такого методі, і, що набагато важливіше, у плані навантаження процесору, адже атакувати таким чином прийдеться одразу усі процеси, тому що не відомо, який саме може виявитися злочинним. До того ж такі дії переводять захищене програмне забезпечення у ранг шкідливого, адже воно саме втручатиметься у роботу інших процесів.

Таким чином можна зробити висновок, що за допомогою користувацьких процесів неможливо вирішити проблему реалізації захисту від даного типу атак.

Але слід враховувати, що існують не лише користувацькі програми, а й програми та процеси, що виконуються у режимі ядра. Вони мають можливість впливати майже на усі процеси, що відбуваються у операційній системі.

До того ж такі дії складно назвати злочинними по відношенню до сторонніх програм, доступ до яких можна отримати, використовуючи код, що виконується у режимі ядра, адже встановлення драйверів можливе лише за наявності у нього цифрового підпису, встановленого спеціалізованими компаніями сертифікації програмного забезпечення для операційної системи Windows. Також код драйверів виконується набагато швидше, адже виконує набагато менше різноманітних перевірок, маючи доступ до усіх ресурсів системи.

2.3.3.2 Використання драйверу для фільтрації викликів функцій

Використовуючи драйвер, код якого виконується у режимі ядра, можливо встановити функцію оберненого виклику на будь-яку функцію обробки дескрипторів користувацького режиму.

Функція оберненого виклику – це функція, яка викликається тоді, коли викликається функція, з якою її з'єднали. Вона може викликатися як до оригінальної функції, так і після. У випадку, коли вона викликається до, у розробника є можливість проаналізувати передані параметри та у разі необхідності змінити їх до виклику оригінальної функції. Даний механізм схожий на встановлення функцій перехоплення (хуків) у коді режиму користувача, але не потребує втручання у віртуальний адресний простір процесу, поведінка якого має бути змінена.

Для встановлення функцій оберненого виклику на функції, які працюють з дескрипторами у драйвері потрібно використовувати функцію ObRegisterCallbacks. Ця функція реєструє список процедур зворотного виклику для потоків, процесів і

операцій обробки робочого столу. Це робиться в режимі ядра і надає авторам драйверів можливість отримувати сповіщення, коли здійснюється виклик OpenProcess[9] для будь-якого процесу, що виконується у системі.

Дана функція має наступний прототип:

```
NTSTATUS ObRegisterCallbacks(
  POB_CALLBACK_REGISTRATION CallbackRegistration,
  PVOID                     *RegistrationHandle
);
```

Перший параметр функції являє собою вказівник на структуру OB_CALLBACK_REGISTRATION. Дана структура містить інформацію про розмір, та функції оберненого виклику іншої структури, яка є її полем та має бути записана. Ім'я внутрішньої структури – OB_OPERATION_REGISTRATION. У цій структурі вже зберігається адреса функції оберненого виклику, яка буде викликатися при спробі отримати дескриптор процесу та дані, що вказують на те, коли вона має бути викликана.

Другий параметр дозволяє зберегти дескриптор встановленого набору функцій оберненого виклику для подальшої їх деактивації за допомогою функції ObUnRegisterCallbacks.

Для коректної реєстрації функції оберненого виклику необхідно перш за все правильно заповнити дані структури OB_OPERATION_REGISTRATION, яка передається першим параметром у вищеназвану функцію ObRegisterCallbacks.

Структура виглядає наступним чином:

```
struct _OB_OPERATION_REGISTRATION {
    POBJECT_TYPE            *ObjectType;
    OB_OPERATION             Operations;
    POB_PRE_OPERATION_CALLBACK  PreOperation;
    POB_POST_OPERATION_CALLBACK PostOperation;
}
```

У першому полі вказується для операцій з яким типом дескрипторів буде викликатися функція, що реєструється. У даному випадку даний параметр має бути

рівним PsProcessType, що означає, що робота відбуватиметься з дескрипторами процесів.

Друге поле визначає для видів операцій створення дескриптору буде застосована функція оберненого виклику. Усього їх два – OB_OPERATION_HANDLE_CREATE та OB_OPERATION_HANDLE_DUPLICATE. Перший з них вказує на те, що виклик функції відбуватиметься у разі створення нового дескриптору. Другий – у разі створення копії існуючого дескриптору. Для реалізації даного механізму захисту необхідно контролювати лише створення дескрипторів, тому потрібне перше значення. Але можна вказати додатково вказати і другий, розширивши спектр виконуваних захисних дій.

У третє поле записується функція оберненого виклику, яка буде викликана при спробі будь-яким процесом отримати дескриптор іншого процесу. Прототип даної функції має співпадати з типом функцій POB_PRE_OPERATION_CALLBACK. Тобто приймати два параметри.

Перший – вказівник на контекст, що може передаватися у дану функцію. Цей контекст є полем RegistrationContext OB_CALLBACK_REGISTRATION, яка використовується для реєстрації функцій оберненого виклику. У даному випадку немає необхідності у передачі цього параметру і поле структури RegistrationContext може бути рівним нулю.

Другий – вказівник на структуру OB_PRE_OPERATION_INFORMATION, яка містить усю інформацію про операцію створення дескриптору, яка виконується одним з процесів. Також вона містить поле Object, що є вказівником на структуру EPROCESS, яка представляє процес у режимі ядра. Далі воно знадобиться для отримання ідентифікатору процесу, дескриптор якого створюється, щоб з'ясувати, чи це процес, захист для якого реалізується, чи ні. Для отримання інформації про дескриптор, створення якого виконується необхідно використати поле даної структури під назвою OB_PRE_OPERATION_PARAMETERS. Ця структура у свою чергу містить дві структури:

– OB_PRE_CREATE_HANDLE_INFORMATION;

– OB_PRE_DUPLICATE_HANDLE_INFORMATION.

Для збереження інформації про права доступу до дескриптору, який створюється, або копіюється відповідно.

Для фільтрації створення дескриптору необхідно використовувати першу структуру.

Структура OB_PRE_CREATE_HANDLE_INFORMATION виглядає наступним чином:

```
struct OB_PRE_CREATE_HANDLE_INFORMATION {
        ACCESS_MASK        DesiredAccess;
        ACCESS_MASK            OriginalDesiredAccess;
}
```

Доцільніше спершу розглянути друге поле даної структури.

Друге поле OriginalDesiredAccess містить права доступу, які бажає отримати процес, що створює дескриптор процесу.

Перше поле спершу дорівнює другому полю, але пізніше його значення заміниться на те, яке не буде містити прапорів, які можуть дозволити зловмиснику інжектувати код.

Для реалізації атаки необхідно отримати дескриптор процесу з наступними правами:

– PROCESS_VM_OPERATION для функції VirtualAllocEx, WriteProcessMemory та CreateRemoteThread;

– PROCESS_VM_WRITE для функції WriteProcessMemory та CreateRemoteThread;

– PROCESS_CREATE_THREAD для функції CreateRemoteThread;

– PROCESS_VM_READ для функції CreateRemoteThread.

Таким чином можна зробити висновок, що для попередження отримання стороннім процесом дескриптору процесу, для якого розроблюється захист, за допомогою якого можна розширити віртуальний адресний простір, записати туди

шкідливий код та виконати його, необхідно не допускати створення дескриптору процесу з вищеперерахованими правами. Найкращим варіантом є заборона отримання усіх цих прав з метою запобігання можливим іншим атакам, які будуть розроблені у майбутньому та, можливо, не будуть вимагати деяких з цих прав.

Також необхідно реалізувати розпізнавання процесу, для якого реалізується даний механізм захисту з-поміж інших, щоб не допустити впливу функції оберненого виклику на інші процеси. Для цього необхідно з'ясувати ідентифікатор даного процесу. Зробити це за допомогою стандартної функції PsGetCurrentProcessId не можна, адже виконання функції оберненого виклику виконується у контексті процесу, який використовує функцію OpenProcess, а цей процес і є потенційно шкідливою програмою. Тому необхідно використати інший спосіб. Як зазначалося раніше, структура OB_PRE_OPERATION_INFORMATION, що є другим параметром функції оберненого виклику, має поле Object. Це поле, у випадку роботи з дескрипторами процесів, має тип EPROCESS. Дана структура режиму ядра призначена для збереження системних даних про процес. У тому числі і його ідентифікатор. Отримати його з цієї структури можна за допомогою функції PsGetProcessId.

Отримавши ідентифікатор процесу, можливо з'ясувати, чи не сторонній процес намагається отримати дескриптор процесу, який необхідно захистити. Якщо це так, просто змінюється бітова маска доступу, що має бути отримана на ту саму, але без прав доступу, які можуть бути використані для реалізації атаки.

Реалізувати передачу ідентифікатора процесу, який необхідно захищати можна багатьма способами(канали, фільтр порт, через файл, за допомогою RPC/ALPC тощо).

В результаті усіх цих дій програма зловмисник все ще може отримати дескриптор процесу, але він буде не придатним для реалізації атаки через відсутність необхідних прав доступу.

Код програмної реалізації засобу захисту наведений у додатку Г.

2.4. Висновок

В другому розділі було проаналізовані наступні види атак:

1) інжекція динамічної бібліотеки;
2) зворотно-орієнтоване програмування;
3) пряма інжекція стороннього коду з подальшим виконанням.

Дані атаки є найпоширенішими серед зловмисників, які мають на меті порушення цілісності програмного коду десктопних Windows-додатків.

Детально спрогнозувавши вектори та особливості атак, до кожної з них було запропоновано, розглянуто та програмно реалізовано мовою C++ ефективні засоби захисту, які роблять неможливими втручання у роботу процесу та виконання стороннього коду у його віртуальному адресному просторі. Дані засоби захисту можуть бути застосовані для будь-якої програми, що призначена для використання на комп'ютері з встановленою операційною системою Windows.

Для захисту від інжекції динамічної бібліотеки необхідно встановити функцію перехоплення системного виклику на функцію LoadLibrary, щоб мати можливість перевіряти переданий шлях до бібліотеки вже після виклику цієї функції, який може бути ініційованим іншим процесом.

Для програмної реалізації захисту від атаки зворотно-орієнтованого програмування було вирішено встановити хук-функцію на VirtualProtect та VirtualProtectEx, адже було з'ясовано, що саме ці функції необхідно викликати зловмиснику, щоб зробити дані на стеку, які були перезаписані у результаті переповнення буферу, виконуваними. Це дозволяє перевірити наявність біта, що вказує на відповідну зміну прав до ділянки пам'яті та блокувати виклик за необхідності.

Реалізація програмного засобу захисту програмного забезпечення від прямої інжекції стороннього коду з подальшим виконанням вимагає створення драйверу,

який виконуватиме свій код у режимі ядра та матиме доступ до усіх операцій усіх процесів. Використовуючи цю можливість, можливо перехоплювати усі спроби отримання дескрипторів процесів у системі та модифікувати ті, які стосуються процесу, який необхідно захистити. Для недопущення атаки змінюються права доступу, які прагне отримати програма зловмисника таким чином, що отриманий дескриптор не може бути використаний для інжекції коду. Однак даний метод є найскладнішим та вимагає сертифікації драйверу у корпорації Microsoft.

РОЗДІЛ 3. ЕКОНОМІЧНА ЧАСТИНА

В ході аналізу існуючих загроз були обрані 3 найпоширеніші з них:

1) інжекція динамічної бібліотеки;
2) зворотно-орієнтоване програмування;
3) пряма інжекція стороннього коду з подальшим виконанням (PE Injection).

До кожної з них були запропоновані та реалізовані методи захисту цілісності програмного коду додатків.

Метою даного розділу є обґрунтування економічної доцільності застосування програмних методів захисту цілісності виконуваного коду десктопних програмних засобів для операційної системи Windows.

Розрахунки виконуються окремо для кожного програмного методу. Також розглядається варіант застосування даних методів як для нових продуктів, розробники яких можуть одразу реалізувати запропоновані програмні методи захисту цілісності програмного коду, так і для вже існуючих продуктів, які не мають такого захисту.

3.1 Економічна доцільність реалізації захисту від інжекції динамічної бібліотеки

3.1.1 Розрахунок капітальних витрат на розробку механізму захисту від даного типу атак

3.1.1.1 Визначення трудомісткості розробки програмної реалізації захисту від атаки

Трудомісткість розробки програмної реалізації.

$$t = t_{\text{тз}} + t_{\text{в}} + t_{\text{а}} + t_{\text{вз}} + t_{\text{озб}} + t_{\text{овр}} + t_{\text{д}}, \text{годин} \quad , \tag{3.1}$$

де $t_{\text{тз}}$ – тривалість складання технічного завдання на розробку програмного механізму захисту цілісності програмного коду;

$t_{\text{в}}$ – тривалість аналізу даної загрози, вивчення ТЗ, літератури про захист програмного коду;

$t_{\text{а}}$ – тривалість аналізу ризиків, пов'язаних з даною загрозою;

$t_{\text{вз}}$ – тривалість визначення вимог до реалізації захисту від даного типу атаки;

$t_{\text{озб}}$ – тривалість вибору основного рішення з забезпечення безпеки інформації;

$t_{\text{овр}}$ – тривалість організації виконання відновлювальних робіт і забезпечення неперервного функціонування програмної реалізації захисту від атаки;

$t_{\text{д}}$ – тривалість документального оформлення результату реалізації програмного механізму захисту від даного типу атак.

Вихідні дані для визначення трудомісткості реалізації програмного механізму захисту від атаки з метою порушення цілісності виконуваного коду шляхом інжекції динамічної бібліотеки приведені в таблиці 3.1.

Таблиця 3.1 – Тривалість розробки програмного механізму захисту

$t_{\text{тз}}$, год	$t_{\text{в}}$, год	$t_{\text{а}}$, год	$t_{\text{вз}}$, год	$t_{\text{озб}}$, год	$t_{\text{овр}}$, год	$t_{\text{д}}$, год
10	20	10	15	10	0	10

Розрахуємо трудомісткість розробки програмної реалізації за формулою (3.1):

$$t = 10 + 20 + 10 + 15 + 10 + 0 + 10 = 75 \text{ годин.}$$

3.1.1.2 Розрахунок витрат на розробку програмного механізму захисту

Витрати на розробку програмного механізму захисту цілісності коду від інжекції динамічної бібліотеки $К_{пм}$ складаються з витрат на заробітну плату спеціаліста з розробки програмного забезпечення $З_{зп}$ і вартості витрат машинного часу, що необхідний для розробки механізму захисту програмного забезпечення $З_{мч}$:

$$К_{пм} = З_{зп} + З_{мч} . \quad (3.2)$$

Заробітна плата виконавця враховує основну і додаткову заробітну плату, а також відрахування на соціальні потреби (пенсійне страхування, страхування на випадок безробіття, соціальне страхування тощо) і визначається за формулою:

$$З_{зп} = t \cdot З_{рпз}, \text{ грн}, \quad (3.3)$$

де t – загальна тривалість розробки політики безпеки, годин;

$З_{рпз}$ – середньогодинна заробітна плата спеціаліста з розробки програмного забезпечення з нарахуванням, грн/годину.

Даний механізм захисту може бути реалізований розробником програмного забезпечення початкового рівня.

$З_{рпз}$ дорівнює 100 грн/год[10].

Розрахуємо заробітну плату розробника за формулою (3.3):

$$З_{зп} = 75 \cdot 100 = 7500 \text{ грн.}$$

Вартість машинного часу для реалізації програмного захисту від інжекції динамічної бібліотеки визначається за формулою:

$$З_{мч} = t \cdot С_{мч}, \text{ грн}, \quad (3.4)$$

де t – трудомісткість реалізації програмного захисту від інжекції динамічної бібліотеки;

$C_{мч}$ – вартість 1 години машинного часу ПК, грн./година.

Вартість 1 години машинного часу ПК визначається за формулою:

$$\text{С}_{\text{мч}} = P \cdot t_{\text{нал}} \cdot \text{С}_{\text{е}} + \frac{\Ф_{\text{зал}} \cdot \text{Н}_{\text{а}}}{F_p} + \frac{\text{К}_{\text{лпз}} \cdot \text{Н}_{\text{апз}}}{F_p}, \text{грн}, \quad (3.5)$$

де Р – встановлена потужність ПК, кВт;

$\text{С}_{\text{е}}$ – тариф на електричну енергію, грн/кВт·година;

$\Ф_{\text{зал}}$ – залишкова вартість ПК на поточний рік, грн.;

$\text{Н}_{\text{а}}$ – річна норма амортизації на ПК, частки одиниці;

$\text{Н}_{\text{апз}}$ – річна норма амортизації на ліцензійне програмне забезпечення, частки одиниці;

$\text{К}_{\text{лпз}}$ – вартість ліцензійного програмного забезпечення, грн.;

$\text{F}_{\text{р}}$ – річний фонд робочого часу (за 40-годинного робочого тижня $\text{F}_{\text{р}} = 1920$).

Залишкова вартість ПК визначається виходячи з фактичного терміну його експлуатації як різниця між первісною вартістю та зносом за час використання.

Знос за час використання ПК для реалізації програмного захисту від інжекції динамічної бібліотеки настільки несуттєвим, що їм можна знехтувати.

Таким чином залишкова вартість ПК є рівною первісній вартості.

Для реалізації даного програмного механізму захисту може бути використаний будь-який комп'ютер, на якому можливо запустити компілятор. Станом на травень 2023-го року мінімальна ціна такого комп'ютеру складає 15 000 грн.

Таким чином залишкова вартість ПК $\Ф_{\text{зал}}$ дорівнює 15 000 грн.

Вихідні дані для розрахунку вартості години машинного часу ПК приведені в таблиці 3.2.

Таблиця 3.2 – Розрахунок вартості машинного часу ПК

P, кВт	C_e, грн/кВт·год	$Ф_{зал}$, грн	$H_а$, частка одиниці	$H_{апз}$, частка одиниці	$K_{лпз}$, грн	$F_р$, год
0,1	2,64	15 000	0,3	0	0	1920

Визначимо вартість 1 години машинного часу ПК за формулою (3.5):

$$C_{мч} = 0{,}1 \cdot 1 \cdot 2{,}64 + \frac{15\,000 \cdot 0{,}3}{1920} = 2{,}6\text{ , грн/год,}$$

Розрахуємо вартість машинного часу за формулою (3.4):

$$З_{мч} = 75 \cdot 2{,}6 = 195\text{ , грн.}$$

Таким чином, підставивши отримані результати у формулу (3.2), отримаємо величину витрат на програмну реалізацію захисту від інжекції динамічної бібліотеки

$$К_{пм} = 7500 + 195 = 7695\text{ , грн.}$$

Капітальні (фіксовані) витрати на програмну реалізацію захисту від інжекції динамічної бібліотеки складають:

$$К = К_{пр} + К_{зпз} + К_{пм} + К_{аз} + К_{навч} + К_{н}\text{, грн,} \qquad (3.6)$$

де $К_{пр}$ – вартість розробки проєкту інформаційної безпеки та залучення для цього зовнішніх консультантів, тис. грн;

$К_{зпз}$ – вартість закупівель ліцензійного основного й додаткового програмного забезпечення (ПЗ), тис. грн;

$К_{пм}$ – вартість програмної реалізації захисту від інжекції динамічної бібліотеки;

$К_{аз}$ – вартість закупівлі апаратного забезпечення та допоміжних матеріалів, тис. грн;

$К_{навч}$ – вартість на навчання технічних фахівців і обслуговуючого персоналу, тис. грн;

$К_{н}$ – витрати на встановлення обладнання та налагодження системи інформаційної безпеки, тис. грн.

Для реалізації програмного захисту від інжекції динамічної бібліотеки немає необхідності у розробці проєкту інформаційної безпеки та залучення зовнішніх консультантів, тому $К_{пр} = 0$ грн.

Програмна реалізацій захисту не потребує закупівлі ніякого ліцензійного програмного забезпечення, тому $К_{зпз} = 0$ грн.

Закупівлі апаратного забезпечення та допоміжних матеріалів також не є необхідними, тому $К_{аз} = 0$ грн.

Для даної реалізації захисту немає необхідності у навчанні технічних фахівців і обслуговуючого персоналу, який також не потребується, адже необхідна лише одноразова його реалізація у програмному засобі, яка і надалі буде актуальна та не потребуватиме підтримки. Тому $К_{навч} = 0$ грн.

Встановлення обладнання та налагодження системи інформаційної безпеки також не потрібне, адже уся реалізація є лише у програмному коді програми. Тому $К_{навч} = 0$ грн.

Таким чином капітальні (фіксовані) витрати на проєктування та впровадження проєктного варіанта системи інформаційної безпеки К дорівнюють вартості програмної реалізації захисту від інжекції динамічної бібліотеки $К_{пм}$ за формулою (3.6):

$$К = К_{пм} = 7695 \text{ грн.}$$

3.1.2 Розрахунок експлуатаційних витрат

Експлуатаційні витрати – це поточні витрати на експлуатацію та обслуговування об'єкта проєктування за визначений період, що виражені у грошовій формі.

За методикою Gather Group до поточних витрат варто відносити наступні:

– вартість Upgrade-відновлення й модернізації системи ($С_в$);

– витрати на керування системою в цілому ($С_к$);

– витрати, викликані активністю користувачів системи інформаційної безпеки ($С_{ак}$ – «активність користувача»).

Отже, річні поточні (експлуатаційні) витрати на функціонування програмного захисту цілісності коду від атаки шляхом інжекції динамічної бібліотеки складають:

$$С = С_в + С_к + С_{ак}, тис.грн. \quad (3.7)$$

Вартість Upgrade-відновлення й модернізації системи ($С_в$) для реалізації захисту цілісності виконуваного коду від інжекції динамічної бібліотеки становить 0 грн, адже вона не потребує оновлення чи модернізації. Причиною цього є те, що для реалізації атаки та захисту від неї використовуються стандартний програмний інтерфейс операційної системи Windows, який не змінюється. $С_в = 0$ грн.

Витрати на керування також не виникають, адже захист реалізується звичайним додаванням коду у програму, який виконується при її старті. Таким чином втручання у даний процес та керування ним не потрібне. $С_к = 0$ грн.

Витрати, викликані активністю користувачів системи інформаційної системи безпеки також дорівнюють нулю, адже таку користувачі відсутні. $С_{ак} = 0$ грн.

Вартість електроенергії, що споживається апаратурою протягом року також дорівнює нулю, адже програмна реалізація захисту коду являє собою лише певну кількість рядків коду, яка зберігається на постійному носії інформації та не потребує електроживлення. $С_{ел} = 0$ грн.

3.1.3 Оцінка можливого збитку від атаки

Атака з метою порушення цілісності виконуваного коду шляхом інжекції динамічної бібліотеки є досить небезпечною, адже вона достатньо проста у реалізації і не вимагає спеціальної підготовки зловмисника. Це є причиною того, що дана атака є найпоширенішою серед усіх атак подібної направленості.

Потрапивши у захищений віртуальний адресний простір програми, виконуваний код динамічної бібліотеки має можливість порушити усі властивості інформації, що обробляється у програмному засобі, адже, фактично, стає його частиною.

У випадку порушення конфіденційності інформації користувачів програмного засобу, наприклад, витік бази даних або історії переписки з іншим користувачем передбачені штрафи згідно чинного законодавства України.

Для програмного продукту, що розробляється, у випадку реалізації атаки на процес з метою порушення цілісності його виконуваного коду шляхом інжекції динамічної бібліотеки, яка може призвести до порушення властивостей інформації, що в ньому оброблюється, величина збитку буде визначатися розміром середнім штрафу, помноженим на імовірність реалізації даної загрози.

Дана загроза є вкрай розповсюдженою, адже вона є простою у реалізації, але залишається вкрай небезпечною і часто призводить до порушення усіх властивостей інформації, з якою взаємодіє програмне забезпечення. Таким чином імовірність реалізації загрози можна оцінити у 90 відсотків.

Величина штрафу за витік інформації визначається статтею 188-39 «Порушення законодавства у сфері захисту персональних даних» кодексу України про адміністративні правопорушення[11]. Вона визначає, що у разі недодержання порядку захисту персональних даних, яке призвело до витоку персональних даних накладається штраф, сума якого варіюється від ста до п’ятисот неоподаткованих мінімумів доходів громадян.

Таким чином величина штрафу на травень 2023-го року складає від 1700 грн. до 8500 грн. Середня величина штрафу складає 5100 грн.

У випадку реалізації даної загрози величина можливого збитку складатиме 4590 грн.

При повторному порушенні протягом року розмір штрафу матиме величину від однієї тисячі до двох тисяч неоподаткованих мінімумів. У такому разі фактична сумі штрафу складатиме від 17000 до 34000 гривень. Величина можливого збитку, як добуток середньої величини штрафу та імовірності реалізації загрози складатиме 22950 грн.

Атаки такого типу відбуваються в середньому 5 рази на рік. Таким чином величина можливого збитку, яка являє собою суму штрафів за порушення захисту персональних даних, дорівнює 96390 грн. При розрахунку врахована різниця сум штрафів за перший на рік випадок витоку даних та усі наступні.

3.2 Економічна доцільність реалізації захисту від зворотно-орієнтованого програмування

3.2.1 Розрахунок капітальних витрат на розробку механізму захисту від даного типу атак

3.2.1.1 Визначення трудомісткості розробки програмної реалізації захисту від атаки

Вихідні дані для визначення трудомісткості реалізації програмного механізму захисту від атаки з метою порушення цілісності виконуваного коду шляхом атаки зворотно-орієнтованого програмування приведені в таблиці 3.3.

Таблиця 3.3 – Тривалість розробки програмного механізму захисту

$t_{тз}$, год	$t_{в}$, год	$t_{а}$, год	$t_{вз}$, год	$t_{озб}$, год	$t_{овр}$, год	$t_{д}$, год

15	50	20	15	10	0	15

Розрахуємо трудомісткість розробки програмної реалізації за формулою (3.1):

$$t = 15 + 50 + 20 + 15 + 10 + 0 + 15 = 125, \text{годин}$$

3.2.1.2 Розрахунок витрат на розробку програмного механізму захисту

Реалізація програмного захисту від даної атаки може бути виконана розробником середнього рівня.

Зрпз дорівнює 150 грн/год[10].

Розрахуємо заробітну плату розробника за формулою (3.3):

$$З_{зп} = 125 \cdot 150 = 18750 \text{ грн.}$$

Залишкова вартість ПК визначається виходячи з фактичного терміну його експлуатації як різниця між первісною вартістю та зносом за час використання.

Знос за час використання ПК для реалізації програмного захисту від атаки зворотно-орієнтованого програмування настільки несуттєвим, що їм можна знехтувати.

Таким чином залишкова вартість ПК є рівною первісній вартості.

Для реалізації даного програмного механізму захисту може бути використаний будь-який комп'ютер, на якому можливо запустити компілятор. Станом на травень 2023-го року мінімальна ціна такого комп'ютеру складає 15 000 грн.

Таким чином залишкова вартість ПК $Ф_{зал}$ дорівнює 15 000 грн.

Вихідні дані для розрахунку вартості години машинного часу ПК приведені в таблиці 3.4.

Таблиця 3.4 – Розрахунок вартості машинного часу ПК

P, кВт	C_e, грн/кВт·год	$Ф_{зал}$, грн	$Н_а$, частка одиниці	$Н_{апз}$, частка одиниці	$К_{лпз}$, грн	$F_р$, год

0,1	2,64	15 000	0,3	0	0	1920

Визначимо вартість 1 години машинного часу ПК за формулою (3.5):

$$С_{мч} = 0,1 \cdot 1 \cdot 2,64 + \frac{15\,000 \cdot 0,3}{1920} = 2,6\text{ , грн/год,}$$

Розрахуємо вартість машинного часу за формулою (3.4):

$$З_{мч} = 125 \cdot 2,6 = 325\text{, грн.}$$

Таким чином, підставивши отримані результати у формулу (3.2), отримаємо величину витрат на програмну реалізацію захисту від атаки зворотно-орієнтованого програмування:

$$К_{пм} = 18750 + 325 = 19075\text{ , грн.}$$

Для реалізації програмного захисту від зворотно-орієнтованого програмування немає необхідності у розробці проєкту інформаційної безпеки та залучення зовнішніх консультантів, тому $К_{пр} = 0$ грн.

Програмна реалізацій захисту не потребує закупівлі ніякого ліцензійного програмного забезпечення, тому $К_{зпз} = 0$ грн.

Закупівлі апаратного забезпечення та допоміжних матеріалів також не є необхідними, тому $К_{аз} = 0$ грн.

Для даної реалізації захисту немає необхідності у навчанні технічних фахівців і обслуговуючого персоналу, який також не потребується, адже необхідна лише одноразова його реалізація у програмному засобі, яка і надалі буде актуальна та не потребуватиме підтримки. Тому $К_{навч} = 0$ грн.

Встановлення обладнання та налагодження системи інформаційної безпеки також не потрібне, адже уся реалізація є лише у програмному коді програми. Тому $К_{навч} = 0$ грн.

Таким чином капітальні (фіксовані) витрати на проєктування та впровадження проєктного варіанта системи інформаційної безпеки К дорівнюють вартості

програмної реалізації захисту від атаки зворотно-орієнтованого програмування $К_{пм}$ за формулою (3.6):

$$К = К_{пм} = 19075 \text{ грн.}$$

3.2.2 Розрахунок експлуатаційних витрат

Вартість Upgrade-відновлення й модернізації системи ($С_в$) для реалізації захисту цілісності виконуваного коду від зворотно-орієнтованого програмування становить 0 грн, адже вона не потребує оновлення чи модернізації. Причиною цього є те, що для реалізації атаки та захисту від неї використовуються стандартний програмний інтерфейс операційної системи Windows, який не змінюється. $С_в = 0$ грн.

Витрати на керування також не виникають, адже захист реалізується звичайним додаванням коду у програму, який виконується при її старті. Таким чином втручання у даний процес та керування ним не потрібне. $С_к = 0$ грн.

Витрати, викликані активністю користувачів системи інформаційної системи безпеки також дорівнюють нулю, адже таку користувачі відсутні. $С_{ак} = 0$ грн.

Вартість електроенергії, що споживається апаратурою протягом року також дорівнює нулю, адже програмна реалізація захисту коду являє собою лише певну кількість рядків коду, яка зберігається на постійному носії інформації та не потребує електроживлення. $С_{ел} = 0$ грн.

3.2.3 Оцінка можливого збитку від атаки

Атака з метою порушення цілісності виконуваного коду шляхом зворотно-орієнтованого програмування є досить небезпечною, адже вона не потребує отримання зловмисником прав адміністратора. Це є причиною того, що дана атака досі є вкрай поширеною серед зловмисників.

Потрапивши у захищений віртуальний адресний простір програми, виконуваний код динамічної бібліотеки має можливість порушити усі властивості

інформації, що обробляється у програмному засобі, адже, фактично, стає його частиною.

У випадку порушення конфіденційності інформації користувачів програмного засобу, наприклад, витік бази даних або історії переписки з іншим користувачем передбачені штрафи згідно чинного законодавства України.

Для програмного продукту, що розробляється, у випадку реалізації атаки на процес з метою порушення цілісності його виконуваного коду, яка може призвести до порушення властивостей інформації, що в ньому оброблюється, величина збитку буде визначатися розміром середнім штрафу, помноженим на імовірність реалізації даної загрози.

Дана загроза є розповсюдженою, адже вона є простою у реалізації, але залишається вкрай небезпечною і часто призводить до порушення усіх властивостей інформації, з якою взаємодіє програмне забезпечення. Таким чином імовірність реалізації загрози можна оцінити у 90 відсотків.

Величина штрафу за витік інформації визначається статтею 188-39 «Порушення законодавства у сфері захисту персональних даних» кодексу України про адміністративні правопорушення[11]. Вона визначає, що у разі недодержання порядку захисту персональних даних, яке призвело до витоку персональних даних накладається штраф, сума якого варіюється від ста до п’ятисот неоподаткованих мінімумів доходів громадян.

Таким чином величина штрафу на травень 2023-го року складає від 1700 грн. до 8500 грн. Середня величина штрафу складає 5100 грн.

У випадку реалізації даної загрози величина можливого збитку складатиме 4590 грн.

При повторному порушенні протягом року розмір штрафу матиме величину від однієї тисячі до двох тисяч неоподаткованих мінімумів. У такому разі фактична сумі штрафу складатиме від 17000 до 34000 гривень. Величина можливого збитку, як

добуток середньої величини штрафу та імовірності реалізації загрози складатиме 22950 грн.

Атаки такого типу відбуваються в середньому 2 рази на рік. Таким чином величина можливого збитку, яка являє собою суму штрафів за порушення захисту персональних даних, дорівнює 27540 грн. При розрахунку врахована різниця сум штрафів за перший на рік випадок витоку даних та усі наступні.

3.3 Економічна доцільність реалізації захисту від прямої інжекції стороннього коду з подальшим виконанням (PE Injection)

3.3.1 Розрахунок капітальних витрат на розробку механізму захисту від даного типу атак

3.3.1.1 Визначення трудомісткості розробки програмної реалізації захисту від атаки

Вихідні дані для визначення трудомісткості реалізації програмного механізму захисту від атаки з метою порушення цілісності виконуваного коду шляхом прямої інжекції стороннього коду з подальшим виконанням в таблиці 3.5.

Таблиця 3.5 – Тривалість розробки програмного механізму захисту

$t_{тз}$, год	$t_{в}$, год	$t_{а}$, год	$t_{вз}$, год	$t_{озб}$, год	$t_{овр}$, год	$t_{д}$, год
20	75	20	15	10	0	20

Розрахуємо трудомісткість розробки програмної реалізації за формулою (3.1):

$$t = 20 + 75 + 20 + 15 + 10 + 0 + 20 = 160, \text{годин}$$

3.3.1.2 Розрахунок витрат на розробку програмного механізму захисту

Реалізація програмного захисту від даної атаки може бути виконана розробником високого рівня.

Зрпз дорівнює 250 грн/год[10].

Розрахуємо заробітну плату розробника за формулою (3.3):

$$З_{зп} = 160 \cdot 250 = 40000 \text{ грн.}$$

Залишкова вартість ПК визначається виходячи з фактичного терміну його експлуатації як різниця між первісною вартістю та зносом за час використання.

Знос за час використання ПК для реалізації програмного захисту від прямої інжекції стороннього коду з подальшим виконанням настільки несуттєвим, що їм можна знехтувати.

Таким чином залишкова вартість ПК є рівною первісній вартості.

Для реалізації даного програмного механізму захисту може бути використаний будь-який комп'ютер, на якому можливо запустити компілятор. Станом на травень 2023-го року мінімальна ціна такого комп'ютеру складає 15 000 грн.

Таким чином залишкова вартість ПК $Ф_{зал}$ дорівнює 15 000 грн.

Вихідні дані для розрахунку вартості години машинного часу ПК приведені в таблиці 3.6

Таблиця 3.6 – Розрахунок вартості машинного часу ПК

P, кВт	$C_е$, грн/кВт·год	$Ф_{зал}$, грн	$Н_а$, частка одиниці	$Н_{апз}$, частка одиниці	$К_{лпз}$, грн	$F_р$, год
0,1	2,64	15 000	0,3	0	0	1920

Визначимо вартість 1 години машинного часу ПК за формулою (3.5):

$$С_{мч} = 0{,}1 \cdot 1 \cdot 2{,}64 + \frac{15\,000 \cdot 0{,}3}{1920} = 2{,}6 \text{ , грн/год,}$$

Розрахуємо вартість машинного часу за формулою (3.4):

$$З_{мч} = 160 \cdot 2{,}6 = 416 \text{ , грн.}$$

Таким чином, підставивши отримані результати у формулу (3.2), отримаємо величину витрат на програмну реалізацію захисту від прямої інжекції стороннього коду з подальшим виконанням:

$$К_{пм} = 40\,000 + 416 = 40\,416 \text{ , грн.}$$

Для реалізації програмного захисту від прямої інжекції стороннього коду з подальшим виконанням немає необхідності у розробці проєкту інформаційної безпеки та залучення зовнішніх консультантів, тому $К_{пр} = 0$ грн.

Програмна реалізація захисту потребує додавання цифрового підпису до драйверу від одного з центрів сертифікації програмного коду. Вартість такої операції станом на травень 2023-го року складає 99 доларів. Згідно поточному курсу національного банку України, на 1 долар США коштує 36,56 грн., тому $К_{зпз} = 99 \cdot 36{,}56 = 3\,619{,}44$ грн.

Закупівлі апаратного забезпечення та допоміжних матеріалів також не є необхідними, тому $К_{аз} = 0$ грн.

Для даної реалізації захисту немає необхідності у навчанні технічних фахівців і обслуговуючого персоналу, який також не потребується, адже необхідна лише одноразова його реалізація у програмному засобі, яка і надалі буде актуальна та не потребуватиме підтримки. Тому $К_{навч} = 0$ грн.

Встановлення обладнання та налагодження системи інформаційної безпеки також не потрібне, адже уся реалізація є лише у програмному коді програми. Тому $К_{навч} = 0$ грн.

Таким чином капітальні (фіксовані) витрати на проєктування та впровадження проєктного варіанта системи інформаційної безпеки К дорівнюють сумі вартості програмної реалізації захисту від прямої інжекції стороннього коду з подальшим виконанням $К_{пм}$ та вартості $К_{зпз}$ за формулою (3.6):

$$К = К_{пм} + К_{зпз} = 40\,416 + 3\,619{,}44 = 44\,035{,}44 \text{ грн.}$$

3.3.2 Розрахунок експлуатаційних витрат

Вартість Upgrade-відновлення й модернізації системи ($С_в$) для реалізації захисту цілісності виконуваного коду від прямої інжекції стороннього коду з подальшим виконанням становить 0 грн, адже вона не потребує оновлення чи модернізації. Причиною цього є те, що для реалізації атаки та захисту від неї використовуються стандартний програмний інтерфейс операційної системи Windows, який не змінюється. $С_в = 0$ грн.

Витрати на керування також не виникають, адже захист реалізується звичайним додаванням коду у програму та використанням драйверу, який запускається при її старті. Таким чином втручання у даний процес та керування ним не потрібне. $С_к = 0$ грн.

Витрати, викликані активністю користувачів системи інформаційної системи безпеки також дорівнюють нулю, адже таку користувачі відсутні. $С_{ак} = 0$ грн.

Вартість електроенергії, що споживається апаратурою протягом року також дорівнює нулю, адже програмна реалізація захисту коду являє собою лише певну кількість рядків коду, яка зберігається на постійному носії інформації та не потребує електроживлення. $С_{ел} = 0$ грн.

3.3.3 Оцінка можливого збитку від атаки

Атака з метою порушення цілісності виконуваного коду шляхом прямої інжекції стороннього коду з подальшим виконанням є небезпечною, адже вона достатньо проста у реалізації і не вимагає спеціальної підготовки зловмисника.

Потрапивши у захищений віртуальний адресний простір програми, виконуваний код динамічної бібліотеки має можливість порушити усі властивості інформації, що обробляється у програмному засобі, адже, фактично, стає його частиною.

У випадку порушення конфіденційності інформації користувачів програмного засобу, наприклад, витік бази даних або історії переписки з іншим користувачем передбачені штрафи згідно чинного законодавства України.

Для програмного продукту, що розробляється, у випадку реалізації атаки на процес з метою порушення цілісності його виконуваного коду, яка може призвести до порушення властивостей інформації, що в ньому оброблюється, величина збитку буде визначатися розміром середнім штрафу, помноженим на імовірність реалізації даної загрози.

Дана загроза є вкрай розповсюдженою, адже вона є простою у реалізації, але залишається вкрай небезпечною і часто призводить до порушення усіх властивостей інформації, з якою взаємодіє програмне забезпечення. Таким чином імовірність реалізації загрози можна оцінити у 90 відсотків.

Величина штрафу за витік інформації визначається статтею 188-39 «Порушення законодавства у сфері захисту персональних даних» кодексу України про адміністративні правопорушення[11]. Вона визначає, що у разі недодержання порядку захисту персональних даних, яке призвело до витоку персональних даних накладається штраф, сума якого варіюється від ста до п’ятисот неоподаткованих мінімумів доходів громадян.

Таким чином величина штрафу на травень 2023-го року складає від 1700 грн. до 8500 грн. Середня величина штрафу складає 5100 грн.

У випадку реалізації даної загрози величина можливого збитку складатиме 4590 грн.

При повторному порушенні протягом року розмір штрафу матиме величину від однієї тисячі до двох тисяч неоподаткованих мінімумів. У такому разі фактична сумі штрафу складатиме від 17000 до 34000 гривень. Величина можливого збитку, як добуток середньої величини штрафу та імовірності реалізації загрози складатиме 22950 грн.

Атаки такого типу відбуваються в середньому 4 рази на рік. Таким чином величина можливого збитку, яка являє собою суму штрафів за порушення захисту персональних даних, дорівнює 73440 грн. При розрахунку врахована різниця сум штрафів за перший на рік випадок витоку даних та усі наступні.

3.4 Реалізація програмного захисту від атак для існуючих програм

Реалізація кожного механізму захисту є вкрай важливою для забезпечення цілісності виконуваного коду та, як наслідок, властивостей інформації, яка оброблюється у програмному засобі. У випадку створення нової програми можливо розрахувати, вартість реалізації кожного механізму захисту. Але у випадку програм, що розроблені сторонніми розробниками ситуація значно відрізняється. В такій ситуації немає можливості ані захистити програму, ані розрахувати вартість такого захисту, бо немає прямого доступу до вихідного коду програми.

Тому єдиним можливим варіантом залишається повідомлення розробниками певного програмного засобу про існування загроз, що можуть порушити усі властивості інформації, яка оброблюється у їх програмах. Також варто додати опис механізмів програмної реалізації захисту від даних атак з прикладами програмного коду.

Більшість програм містять інформацію про їх розробників та контакти зв'язку для з ними. Саме за ними і слід писати їм та описати усю важливість реалізації захисту від даного виду атак.

3.5 Висновок

В економічному розділі була розрахована тривалість та вартість програмної реалізації захисту від трьох найпоширеніших атак, направлених на порушення

цілісності виконуваного коду, враховуючи різні рівні зарплатні розробників програмного забезпечення.

Капітальні витрати на програмну реалізацію захисту від інжекції динамічної бібліотеки складають 7695 грн., від зворотно-орієнтованого програмування – 19075 грн., від прямої інжекції стороннього коду з подальшим виконанням – 44 035,44 грн.

Розраховані можливі збитки від реалізації атак. Вони визначаються сумою штрафів за витік персональних даних та мають наступні величини:

– реалізація загрози інжекції динамічної бібліотеки – 96390 грн.;
– реалізація загрози зворотно-орієнтованого програмування – 22950 грн.;
– реалізація загрози прямої інжекції коду з подальшим виконанням – 73440 грн.

Можна зробити висновок, що програмна реалізація захисту від даних атак є економічно доцільною, адже кожна реалізація захисту створюється лише на одноразово на етапі створення програми та не потребує подальшого втручання, зменшуючи імовірність реалізації атак та попереджуючи накладання штрафів за порушення законодавства у сфері персональних даних.

Також розглянута можливість пропозиції впровадження цих захисних механізмів розробникам вже існуючих програмних продуктів.

ВИСНОВКИ

У кваліфікаційній роботі було проаналізовано особливості реалізації програмних засобів захисту програмного забезпечення десктопних Windows-додатків, які є однією з найчастіших мішеней зловмисників через поширеність даної операційної системи.

Розглянуті основні фактори, що призводять до реалізації атак на програмне забезпечення та порушення цілісності виконуваного коду та зроблено висновок, що основним з них є необережність користувачів у користуванні комп'ютером, що призводить до потрапляння шкідливого програмного забезпечення у нього.

Проаналізовані вбудовані механізми захисту від даного типу атак, які попри усі свої можливості не здатні реалізувати стовідсотковий захист для сотень мільйонів користувачів. Тому було зроблено висновок, що програмні засоби мають самостійно реалізовувати додаткові засоби захисту цілісності свого виконуваного коду від найпоширеніших атак.

Були проаналізовані три найпоширеніші атаки такого типу:

1) інжекція динамічної бібліотеки;
2) зворотно-орієнтоване програмування;
3) пряма інжекція стороннього коду з подальшим виконанням.

Для кожної з них був спрогнозований вектор атаки та визначена послідовність дій та компонентів, що необхідні для реалізації даної загрози. Базуючись на отриманих даних, були запропоновані та програмно реалізовані засоби захисту виконуваного коду процесів мовою C++, які впроваджуються розробниками програмного забезпечення ще на етапі його проєктування. Вони роблять неможливим для зловмисника виконати його код у віртуальному адресному просторі процесу програми, у якій наявні ці механізми захисту.

Також була доведена економічна доцільність програмної реалізації засобів захисту цілісності програмного забезпечення десктопних Windows-додатків та їх

виконуваного коду. Для цього була розрахована вартість реалізації та величина можливих збитків у разі відмови від неї. Виходячи з того, що програмна реалізація виконується лише одноразово та не потребує постійної підтримки розробників програмного забезпечення, а атаки відбуваються усе частіше, враховуючи збільшення кількості користувачів операційної системи Windows, зроблено висновок, що доцільніше одноразово на етапі розробки програмного забезпечення реалізувати засоби захисту, щоб уникнути штрафів, передбачених статтею 188-39 «Порушення законодавства у сфері захисту персональних даних» кодексу України про адміністративні правопорушення.

ПЕРЕЛІК ПОСИЛАНЬ

1 М. Руссинович, Д. Соломон – Внутрішній устрій Microsoft Windows 6-е видання (2013)

2 Сучасні операційні системи. Таненбаум 5-е видання (2015)

3 DEP та ASLR: кватирки у вікнах вашого комп'ютера [Електронний ресурс] – https://vc.ru/u/399338-codeby-net/103681-dep-i-aslr-fortochki-v-oknah-vashego-kompyutera (Дата звернення: 09.06.23)

4 Process Injection Part 1: The Theory [Електронний ресурс] – https://secarma.com/process-injection-part-1-the-theory/ (Дата звернення: 09.06.23)

5 Practical Comparison of the Most Popular API Hooking Libraries: Microsoft Detours, EasyHook, Nektra Deviare, and Mhook [Електронний ресурс] – https://www.apriorit.com/dev-blog/win-comparison-of-api-hooking-libraries (Дата звернення: 09.06.23)

6 Basic Windows API Hooking [Електронний ресурс] – https://medium.com/geekculture/basic-windows-api-hooking-acb8d275e9b8 (Дата звернення: 09.06.23)

7 ROP-ланцюжки та гаджети: вчимося розробляти експлойти [Електронний ресурс] – https://vc.ru/u/399338-codeby-net/103531-rop-cepochki-i-gadzhety-uchimsya-razrabatyvat-eksployty (Дата звернення: 09.06.23)

8 ROP Chain. How to Defend from ROP Attacks (Basic Example) [Електронний ресурс] – https://www.apriorit.com/dev-blog/434-rop-exploit-protection (Дата звернення: 09.06.23)

9 Kernel Karnage – Part 6 (Last Call) [Електронний ресурс] – https://blog.nviso.eu/2021/12/09/kernel-karnage-part-6-last-call/ (Дата звернення: 09.06.23)

10 Зарплати українських розробників — зима 2023 [Електронний ресурс] – https://dou.ua/lenta/articles/salary-report-devs-winter-2023/ (Дата звернення: 09.06.23)

11 Кодекс України про адміністративні правопорушення [Електронний ресурс] – https://zakon.rada.gov.ua/laws/show/80731-10#Text (Дата звернення: 09.06.23)

ДОДАТОК А. Відомість матеріалів кваліфікаційної роботи

№	Формат	Найменування	Кількість листів	Примітка
1	А4	Реферат	2	
2	А4	Список умовних скорочень	1	
3	А4	Зміст	3	
4	А4	Вступ	2	
5	А4	1 Розділ	11	
6	А4	2 Розділ	30	
7	А4	3 Розділ	20	
8	А4	Висновки	2	
9	А4	Перелік посилань	2	
10	А4	Додаток А	1	
11	А4	Додаток Б	3	
12	А4	Додаток В	3	
13	А4	Додаток Г	3	
14	А4	Додаток Ґ	1	
15	А4	Додаток Д	1	
16	А4	Додаток Е	1	

ДОДАТОК Б. Лістинг програмної реалізації захисту від інжекції DLL

//Файл HookUtils.h:

```
#pragma once
#include <Windows.h>
#include <atlbase.h>
#include "plog/Log.h"
#include "mhook.h"


namespace hookUtils
{
   template<typename FnType>
   FnType HookFunc(LPCWSTR ModuleName, LPCSTR TargetFunctionName, FnType
HookFunction)
   {
      CHandle Module(GetModuleHandle(ModuleName));
      if (!Module)
      {
         PLOGE << "Module " << ModuleName << " not found. LE : " << GetLastError();
         std::cout << "Module not found : " << ModuleName << std::endl;
         return nullptr;
      }

      FnType realFunction =
reinterpret_cast<FnType>(GetProcAddress(reinterpret_cast<HMODULE>(Module.m_h),
TargetFunctionName));
      if (!realFunction)
      {
         PLOGE << "Function " << TargetFunctionName << " not found. LE : " <<
GetLastError();
         std::cout << "Function not found " << TargetFunctionName << std::endl;
         return nullptr;
      }

      BOOL hookSucceeded =
Mhook_SetHook(reinterpret_cast<PVOID*>(&realFunction), HookFunction);
      if (!hookSucceeded)
      {
```

```
            PLOGE << "Failed to hook function " << TargetFunctionName << ". LE : " <<
GetLastError();
            std::cout << "Hook failed " << TargetFunctionName << std::endl;
            return nullptr;
        }
        return realFunction;
    }
}
```

//Файл main.cpp:

```
#include <Windows.h>
#include <iostream>
#include "HookUtils.h"
#include <plog/Log.h>
#include <plog/Initializers/RollingFileInitializer.h>

static decltype(::LoadLibraryA)* g_loadLibraryA = nullptr;
static decltype(::LoadLibraryW)* g_loadLibraryW = nullptr;

bool isDigitalSignatureValid(const std::wstring& dllName)
{
    //Перевірка цифрового підпису виконується тут
    return true;
}

bool isNameAllowed(const std::wstring& dllName)
{
    //Перевірка імені виконується тут
    return false;
}

bool isPathAllowed(const std::wstring& dllName)
{
    //Перевірка шляху виконується тут
    return true;
}

bool dllUtils::isDllSafe(const std::wstring& dllName)
{
    PLOGI << L"Checking dll : " << dllName;
    return isNameAllowed(dllName) && isPathAllowed(dllName) &&
isDigitalSignatureValid(dllName);
```

```
}

//Ця функція викликатиметься, коли програма викликає функцію LoadLibraryA
//Ми маємо можливість проаналізувати ім'я бібліотеки перед її завантаженням
//щоб прийняти рішення, чи дійсно вона безпечна
HMODULE WINAPI LoadLibraryA_Hook(_In_ LPCSTR lpLibFileName)
{
   if (!dllUtils::isDllSafe(std::wstring(lpLibFileName, lpLibFileName +
std::strlen(lpLibFileName))))
   {
      PLOGW << "Attempt to load unknown Dll. Name : " << lpLibFileName;
      return nullptr;
   }

   return g_loadLibraryA(lpLibFileName);
}

//Ця функція викликатиметься, коли програма викликає функцію LoadLibraryW
//Ми маємо можливість проаналізувати ім'я бібліотеки перед її завантаженням
//щоб прийняти рішення, чи дійсно вона безпечна
HMODULE WINAPI LoadLibraryW_Hook(_In_ LPCWSTR lpLibFileName)
{
   if (!dllUtils::isDllSafe(lpLibFileName))
   {
      PLOGW << "Attempt to load unknown Dll. Name : " << lpLibFileName;
      return nullptr;
   }

   return g_loadLibraryW(lpLibFileName);
}

int wmain(int argc, wchar_t** argv)
{
   plog::init(plog::debug, L"C:\\Users\\User\\Desktop\\HookProtection.log");

   g_loadLibraryA = hookUtils::HookFunc(L"kernel32.dll", "LoadLibraryA",
LoadLibraryA_Hook);
   g_loadLibraryW = hookUtils::HookFunc(L"kernel32.dll", "LoadLibraryW",
LoadLibraryW_Hook);

   return 0;
}
```

ДОДАТОК В. Лістинг програмної реалізації захисту від зворотно-орієнтованого програмування

//Файл hookUtils.h:

```
#pragma once
#pragma once
#include <Windows.h>
#include <atlbase.h>
#include "plog/Log.h"
#include "mhook.h"

namespace hookUtils
{
   template<typename FnType>
   FnType HookFunc(LPCWSTR ModuleName, LPCSTR TargetFunctionName, FnType
HookFunction)
   {
      CHandle Module(GetModuleHandle(ModuleName));
      if (!Module)
      {
         PLOGE << "Module " << ModuleName << " not found. LE : " << GetLastError();
         std::wcout << L"Module not found : " << ModuleName << std::endl;
         return nullptr;
      }

      FnType realFunction =
reinterpret_cast<FnType>(GetProcAddress(reinterpret_cast<HMODULE>(Module.m_h),
TargetFunctionName));
      if (!realFunction)
      {
         PLOGE << "Function " << TargetFunctionName << " not found. LE : " <<
GetLastError();
         std::cout << "Function not found " << TargetFunctionName << std::endl;
         return nullptr;
      }

      BOOL hookSucceeded =
Mhook_SetHook(reinterpret_cast<PVOID*>(&realFunction), HookFunction);
      if (!hookSucceeded)
      {
```

```
            PLOGE << "Failed to hook function " << TargetFunctionName << ". LE : " <<
GetLastError();
            std::cout << "Hook failed " << TargetFunctionName << std::endl;
            return nullptr;
        }
        return realFunction;
    }
}


//Файл main.cpp:

#include <Windows.h>
#include <iostream>
#include <plog/Log.h>
#include <plog/Initializers/RollingFileInitializer.h>
#include "hookUtils.h"

static decltype(::VirtualProtect)* g_virtualProtectOriginal = nullptr;
static decltype(::VirtualProtectEx)* g_virtualProtectExOriginal = nullptr;

bool g_setHook = false;

BOOL WINAPI VirtualProtect_Hook(
    _In_  LPVOID lpAddress,
    _In_  SIZE_T dwSize,
    _In_  DWORD flNewProtect,
    _Out_ PDWORD lpflOldProtect
)
{
    //Функція встановлення хук-функції MHook_SetHook сама використовує функцію
VirtualProtect
    //Це призводить до того, що на етапі встановлення хуку виникає безкінечна
рекурсія
    //Для подолання цієї проблеми вирішено за допомогою прапору g_setHook
сигналізувати про закінчення встановлення хуків
    //Під час їх встановлення замість оригінальної функції викликається розширена -
VirtualProtectEx
    if (!g_setHook)
    {
        CHandle currentProcess(::GetCurrentProcess());
```

```
        return g_virtualProtectExOriginal(currentProcess.m_h, lpAddress, dwSize,
flNewProtect, lpflOldProtect);
    }
    std::cout << "VirtualProtect called" << std::endl;
    if (flNewProtect & PAGE_EXECUTE_READWRITE)
    {
        PLOGI << "Probably someone is trying to use ROP attack. Block this call" <<
std::endl;
        return false;
    }
    return g_virtualProtectOriginal(lpAddress, dwSize, flNewProtect, lpflOldProtect);
}

BOOL WINAPI VirtualProtectEx_Hook(
    _In_ HANDLE hProcess,
    _In_ LPVOID lpAddress,
    _In_ SIZE_T dwSize,
    _In_ DWORD flNewProtect,
    _Out_ PDWORD lpflOldProtect
)
{
    PLOGI << "VirtualProtectEx called" << std::endl;
    if (flNewProtect & PAGE_EXECUTE_READWRITE)
    {
        PLOGI << "Probably someone is trying to use ROP attack. Block this call" <<
std::endl;
        return false;
    }
    return g_virtualProtectExOriginal(hProcess, lpAddress, dwSize, flNewProtect,
lpflOldProtect);
}

int wmain(int argc, wchar_t** argv)
{
    plog::init(plog::debug, L"C:\\Users\\User\\Desktop\\HookProtection.log");
    g_virtualProtectExOriginal = hookUtils::HookFunc(L"Kernel32.dll",
"VirtualProtectEx", VirtualProtectEx_Hook);
    g_virtualProtectOriginal = hookUtils::HookFunc(L"Kernel32.dll", "VirtualProtect",
VirtualProtect_Hook);
    g_setHook = true;
    return 0;
}
```

ДОДАТОК Г. Лістинг програмної реалізації захисту від прямої інжекції стороннього коду з подальшим виконанням

//Файл Driver.cpp:

```
#include <fltKernel.h>
#include <dontuse.h>

#define PROCESS_CREATE_THREAD (0x0002)
#define PROCESS_VM_OPERATION (0x0008)
#define PROCESS_VM_READ (0x0010)
#define PROCESS_VM_WRITE (0x0020)

HANDLE g_callbackHandle = nullptr;
ULONG g_processToProtect = 0;

constexpr ACCESS_MASK g_unsafeProcessAccess = PROCESS_VM_OPERATION |
PROCESS_VM_WRITE | PROCESS_CREATE_THREAD | PROCESS_VM_READ;

VOID UnloadDriver(IN PDRIVER_OBJECT DriverObject)
{
   UNREFERENCED_PARAMETER(DriverObject);
   ::ObUnRegisterCallbacks(g_callbackHandle);
   return;
}

OB_PREOP_CALLBACK_STATUS preProccessDescriptorCreateCallback(
   _In_ PVOID                        registrationContext,
   _Inout_ POB_PRE_OPERATION_INFORMATION      operationInformation
)
{
   UNREFERENCED_PARAMETER(registrationContext);

   ACCESS_MASK& desiredAccess = operationInformation->Parameters-
>CreateHandleInformation.DesiredAccess;

   if (!operationInformation->Object)
   {
      return OB_PREOP_SUCCESS;
   }
```

```
    auto pid =
HandleToUlong(::PsGetProcessId(static_cast<PEPROCESS>(operationInformation-
>Object)));

    if (pid != g_processToProtect)
    {
        return OB_PREOP_SUCCESS;
    }

    if (FlagOn(desiredAccess, g_unsafeProcessAccess))
    {
        ClearFlag(desiredAccess,g_unsafeProcessAccess);
    }

    return OB_PREOP_SUCCESS;
}

NTSTATUS registerCallbacks()
{
    OB_OPERATION_REGISTRATION regOperation{};

    regOperation.ObjectType = PsProcessType;
    regOperation.Operations = OB_OPERATION_HANDLE_CREATE |
OB_OPERATION_HANDLE_DUPLICATE;
    regOperation.PreOperation = preProccessDescriptorCreateCallback;
    regOperation.PostOperation = nullptr;

    OB_CALLBACK_REGISTRATION regCallback{};
    regCallback.Version = OB_FLT_REGISTRATION_VERSION;
    regCallback.OperationRegistrationCount = 1;
    regCallback.RegistrationContext = nullptr;
    regCallback.OperationRegistration = &regOperation;
    ::RtlInitUnicodeString(&regCallback.Altitude, L"400400");

    auto status = ::ObRegisterCallbacks(&regCallback, &g_callbackHandle);
    return status;
}

extern "C"
NTSTATUS
DriverEntry (
    _In_ PDRIVER_OBJECT DriverObject,
```

```
    _In_ PUNICODE_STRING RegistryPath
    )
{
    UNREFERENCED_PARAMETER(RegistryPath);
    DriverObject->DriverUnload = UnloadDriver;

    NTSTATUS status = registerCallbacks();

    return status;
}
```

ДОДАТОК Ґ. Відгук керівника економічного розділу

Економічний розділ виконаний відповідно до вимог, які ставляться до кваліфікаційних робіт, та заслуговує на оцінку 95 б. («відмінно»).

Керівник розділу _____ доц. Пілова Д.П.

(підпис) (ініціали, прізвище)

ДОДАТОК Д. Відгук керівника кваліфікаційної роботи

ВІДГУК
на кваліфікаційну роботу студента групи 125-19-2
Масло Дмитра Григорійовича
на тему: «Засоби забезпечення цілісності програмного забезпечення десктопних Windows-додатків»

Пояснювальна записка складається зі вступу, трьох розділів і висновків, викладених на 88 сторінках.

Метою кваліфікаційної роботи є аналіз існуючих загроз цілісності виконуваного коду десктопних додатків для операційної системи Windows та програмна реалізація засобів захисту від них.

Тема кваліфікаційної роботи безпосередньо пов'язана з об'єктом діяльності бакалавра спеціальності 125 "Кібербезпека". Для досягнення поставленої мети в кваліфікаційній роботі вирішуються наступні задачі: аналіз вразливостей операційної системи Windows, процесів і програм, що в ній виконуються і є потенційними цілями для атак зловмисників, аналіз існуючих атак, що порушують цілісність виконуваного коду, розробка програмних засобів захисту від цього типу атак.

Практичне значення результатів кваліфікаційної роботи полягає у розроблених засобах захисту Windows-додатків та обґрунтуванні економічного ефекту від їх впровадження.

За час дипломування Масло Д.Г. проявив себе фахівцем, здатним самостійно вирішувати поставлені задачі та заслуговує присвоєння кваліфікації бакалавра за спеціальністю 125 Кібербезпека, освітньо-професійна програма «Кібербезпека».

Рівень запозичень у кваліфікаційній роботі не перевищує вимог "Положення про систему виявлення та запобігання плагіату".

Кваліфікаційна робота заслуговує оцінки «відмінно».

Керівник
кваліфікаційної роботи доц. каф. БІТ, к.т.н. Сафаров О.О.

Керівник спец. розділу доц. каф. БІТ, к.т.н. Сафаров О.О.

ДОДАТОК Е. Перелік документів на оптичному носії

1. Диплом_Масло.pdf
2. Презентація_Масло.pptx
3. Програми.zip