В. А. Каплун, О. В. Дмитришин, Ю. В. Баришев

ЗАХИСТ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ЧАСТИНА 2

Міністерство освіти і науки України
Вінницький національний технічний університет

В. А. Каплун, О. В. Дмитришин, Ю. В. Баришев

ЗАХИСТ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Частина 2

Навчальний посібник

Вінниця
ВНТУ
2014

УДК 681.3.07
ББК 32.973-018.2я73
К 20

Рекомендовано до друку Вченою радою Вінницького національного технічного університету Міністерства освіти і науки України (протокол № від)

Рецензенти:

А. М. Пєтух, доктор технічних наук, професор
В. М. Михалевич, доктор технічних наук, професор
Л. І. Тимченко, доктор технічних наук, професор

Каплун, В. А.

К 20 Захист програмного забезпечення. Частина 2 : навчальний посібник / В. А. Каплун, О. В. Дмитришин, Ю. В. Баришев – Вінниця : ВНТУ, 2014. – 105 с.

Посібник містить теоретичні відомості щодо методів захисту програмного забезпечення від несанкціонованого дослідження – як від статичного, так і від динамічного. Детально розглянуто такі методи захисту, як обфускація програмного коду, ускладнення логіки, емуляція, антидампінгові прийоми, маніпулювання заголовками виконуваних файлів, а також методи захисту від інтерактивних та автоматичних дизасемблерів, від налагоджувачів реального та захищеного режимів.

Навчальний посібник, призначений для студентів напряму підготовки 6.170101 «Безпека інформаційних і комунікаційних систем» для вивчення дисципліни «Захист програмного забезпечення», є логічним продовженням матеріалів першої частини однойменного навчального посібника.

УДК 681.3.07
ББК 32.973-018.2я73

ЗМІСТ

ВСТУП

Світові дослідження останніх років показали, що функціональні характеристики та характеристики надійності комп'ютерних систем (КС) визначаються якістю і надійністю програмного забезпечення (ПЗ), що входить в їх склад. Окрім проблем якості і надійності програмного забезпечення при створенні КС, фундаментальна проблема його безпеки набуває все більшої актуальності. При цьому в рамках даної проблеми на перший план висувається безпека технологій створення програмного забезпечення комп'ютерних систем. Даний аспект проблеми безпеки програмних комплексів є порівняно новим і пов'язаний з можливістю впровадження в тіло програмних засобів на етапі їх розробки (або модифікації в ході авторського супроводу) так званих «програмних закладок». У зв'язку з цим актуальнішею стає проблема забезпечення технологічної безпеки програмного забезпечення КС різного рівня і призначення.

Безпека програмного забезпечення в широкому значенні є властивістю даного ПЗ функціонувати без прояву різних негативних наслідків для конкретної комп'ютерної системи. Під рівнем безпеки ПЗ розуміється вірогідність того, що за заданих умов в процесі його експлуатації буде одержано функціонально придатний результат, і цим результатом повинен користуватися тільки легальний користувач. Причини, що призводять до функціонально непридатного результату, можуть бути різними: збої комп'ютерних систем, помилки програмістів і операторів, дефекти ПЗ. При цьому дефекти можуть бути як навмисні, так і ненавмисні. Перші, як правило, є результатом зловмисних дій, другі – помилкових дій людини.

Крім того, при експлуатації програмних комплексів можливий певний алгоритм внесення програмного дефекту: дизасемблювання виконуваного програмного коду, отримання початкового тексту, привнесення в нього деструктивної програми, повторна компіляція, корегування програми (у зв'язку з необхідністю отримання програми, «схожої» з оригіналом). Маніпуляції подібного роду можуть зробити програмісти, які мають досвід розробки і налагодження програм на асемблерному рівні.

Таким чином, необхідність внесення в програмне забезпечення захисних функцій на всьому протязі його життєвого циклу від етапу з'ясування задуму на розробку програм до етапів випробувань, експлуатації, модернізації і супроводу програм не викликає сумнівів.

У зв'язку з цим у даному навчальному посібнику розглядаються методологічні основи побудови захищеного програмного забезпечення, розглянуті сучасні методи забезпечення технологічної і експлуатаційної безпеки програм. Важливе місце відводиться методам створення алгоритмічно безпечного програмного забезпечення, використання якого дозволяє виявляти і усувати програмні дефекти деструктивного характеру як на етапі створення, так і на етапі застосування програм.

1 ЗАГАЛЬНІ ПРИНЦИПИ ЗАХИСТУ ПРОГРАМ ВІД НЕСАНКЦІОНОВАНОГО ДОСЛІДЖЕННЯ

1.1 Принципи побудови систем захисту та їх функції

Під *несанкціонованим доступом* (НСД) – *unauthorized access* – розумітимемо нелегальні дії щодо використання, зміни та знищення виконуваних модулів.

Системою захисту від несанкціонованого доступу (*system of protection against unauthorized access*) називають комплекс програмних засобів, що забезпечують ускладнення або заборону нелегального розповсюдження, використання і/або зміну програмних продуктів. *Надійність системи захисту* (*reliability of protection*) – це здатність протистояти спробам проникнення в алгоритм її роботи і обходу механізмів захисту.

Під *зламом програми* (*breaking program*) матимемо на увазі порушення функціональності об'єктів захисту програмного забезпечення (адже їх може бути декілька). Для аналізу об'єктів захисту використовуються два основні види засобів: налагоджувачі, які дозволяють контролювати виконання програм і таким чином прослідкувати весь алгоритм на практиці і дизасемблери, що дозволяють одержувати асемблерний лістинг програми для його подальшого статичного вивчення. І тому, якою би мовою ми не писали, які б способи захисту не використовували, все, зрештою, перетвориться на команди процесора. Отже, одержавши будь-яким чином лістинг програми, зламник бачить вже знайомі йому асемблерні команди, причому для аналізу алгоритму роботи програми йому в більшості випадків не важливо, за допомогою чого вона була створена. Достатньо лише добре розбиратися в роботі процесора і в архітектурі комп'ютерної системи.

Отже, ***основними принципами***, якими повинен користуватись розробник при реалізації захисту програм, є такі.

- Простота механізму захисту, оскільки ретельне тестування засобу захисту можливе тільки для простих і компактних схем.
- У механізмі захисту в нормальних умовах доступ повинен бути відсутнім.
- Механізм захисту можна не засекречувати, тобто немає сенсу засекречувати деталі реалізації систем захисту, призначеного для широкого користувача. Ефективність захисту не повинна залежати від досвідченості потенційного порушника.
- Доцільним є розділення повноважень, тобто застосування декількох ключів захисту. У комп’ютері право на доступ повинно визначатися виконанням ряду умов.
- При розробці систем захисту необхідно передбачувати максимальну відособленість механізму захисту. З метою заборони обміну інформацією між користувачами рекомендується при проектуванні схем захисту зводити до мінімуму число загальних для декількох користувачів параметрів і характеристик механізму захисту.

Отже, системи захисту від НСД повинні виконувати такі функції:

- ідентифікація ресурсів, що захищаються;
- автентифікація ресурсів, що захищаються, тобто установлення їх істинності на основі порівняння з еталоном;
- визначення прав доступу до ресурсів, що захищаються;
- реєстрація входу користувача в систему та реєстрація виходу з неї;
- реєстрація реакції на порушення прав доступу;
- обробка реєстраційних журналів;
- контроль цілісності і працездатності. Контроль і розмежування може здійснюватися на основі таблиць.

1.2 Основні методи та засоби дослідження програм

Перед тим, як розглянути способи захисту програм від НСД, необхідно з'ясувати, яким же чином здійснюється таке дослідження.

Всі засоби дослідження роботи програмних продуктів, у тому числі і захищених, можна розбити на чотири класи:

1. *Статичні засоби*, які оперують початковим кодом програми як даними і будують її алгоритм без виконання. Ці засоби є більш універсальними в тому значенні, що теоретично можуть одержати алгоритм усієї програми, у тому числі і тих блоків, які ніколи не отримують управління.
2. *Динамічні засоби*, які вивчають програму, інтерпретуючи її в реальному або віртуальному обчислювальному середовищі. Ці засоби можуть будувати алгоритм програми тільки на підставі конкретної її траси, одержаної при певних вхідних даних. Тому задача отримання повного алгоритму програми в цьому випадку еквівалентна побудові вичерпного набору текстів для підтвердження правильності програми, що практично неможливо, і взагалі при динамічному дослідженні можна говорити тільки про побудову деякої частини алгоритму.
3. *Синтаксичні методи.* До цієї групи відносяться методи, що ґрунтуються тільки на результатах лексичного, синтаксичного і семантичного аналізу програми.
4. *Статистичні методи.* Статистичні методи використовують інформацію, зібрану в результаті значної кількості запусків програми на великій кількості наборів вхідних даних

Розглянемо конкретні інструменти, що використовуються зламниками для несанкціонованого дослідження захищених програм.

Налагоджувачі і дизасемблери. Традиційно обидва ці типи інструментів використовуються в парі, оскільки дизасемблер видає лише "чистий код". Але сучасні дизасемблери здатні також розпізнати виклики стандартних функцій, виділити локальні змінні в процедурах і надати інший подібний сервіс. Користуючись дизасемблером, можна лише здогадуватися про те, які дані одержує та або інша функція як параметри і що вони

означають. Щоб з'ясувати це, найчастіше потрібне вивчення якщо не всієї програми, то досить значної її частини.

Налагоджувачі виконують принципово інші функції, вони дозволяють аналізувати код в процесі його роботи, відстежувати і змінювати стан регістрів і стека, правити код "на льоту" – загалом, спостерігати за роботою програми і навіть активно в неї втручатися. Зворотним боком цього є "неінтелектуальність" багатьох налагоджувачів – їх природжені здібності до аналізу коду рідко виходять за межі визначення напряму переходу. SoftIce, один з кращих налагоджувачів, наприклад, нічого не знає про типи даних і не здатний відрізнити звичний DWORD від покажчика на ASCIIZ-рядок, хоча і надає користувачу можливість перевірити це вручну. Втім, існують налагоджувачі, що поєднують високу якість дизасемблювання і аналізу коду з широкими можливостями з його налагоджування. Прикладом може служити OllyDebug, що виконує евристичний аналіз коду, виділяє локальні змінні, "знає" про типи даних, переданих функціям WinAPI і при цьому здатний у багатьох випадках автоматично відрізнити звичайне число від покажчика на рядок.

Декомпілятори і вузькоспеціалізовані налагоджувачі. Із зростанням потужності ЕОМ досить широке поширення набули компілятори, що створюють не чистий машинний код, а деякий набір умовних інструкцій, який виконується за допомогою інтерпретатора. Інтерпретатор може як поставлятися окремо (Java), так і бути прикріпленим до самої програми (хоча формально не інтерпретатор прикріплюється до програми, а програма до інтерпретатора. Прикладом може служити Visual Basic в режимі компіляції в p-code). Інтерпретаторами є практично всі інсталятори (у їх основі лежить інтерпретатор інсталяційної програми, хоча сам процес створення такого скрипта може бути прихований за допомогою візуальних засобів). Та і звичайні компілюючі мови можуть створювати код, прямий аналіз якого досит складний. Для аналізу таких програм використовуються спеціалізовані утиліти, що переводять код, зрозумілий лише інтерпретатору, у форму, зручнішу для розуміння людиною. Також деякі декомпілятори можуть витягувати інформацію про елементи інтерфейсу, створені візуальними засобами. У будь-якому випадку, не слід чекати від декомпіляторів повного відновлення початкового тексту програми.

Утиліти для розпакування та дампінга процесів. Дизасемблювати запаковану або зашифровану програму напряму неможливо. Але якщо дуже хочеться одержати хоч якийсь лістинг, можна спробувати витягнути з пам'яті комп'ютера знімок (дамп) програми у момент її роботи. Цей дамп вже можна більш менш успішно дизасемблювати. Більш того, на основі дампу можна відтворити виконуваний файл програми, причому цей файл успішно завантажуватиметься, запускатиметься і працюватиме. Саме на цьому принципі і основана робота більшості сучасних розпакувальників: піддослідна програма запускається під управлінням розпакувальника. Розпакувальник очікує на деяку подію, яка свідчить про те, що програма повністю розпакувалася, і тут же "заморожує" програму і скидає її дамп на

диск. Захисні системи нерідко намагаються протидіяти отриманню працездатного дампу за допомогою маніпуляцій з сегментами і таблицями ім.-порту-експорту. У цих випадках доводиться застосовувати PE-реконструктори, тобто утиліти, що знаходять в дампі некоректні посилання на функції і намагаються їх відновити. Більшість сучасних дамперів і розпакувальників мають вбудовані засоби відновлення секцій імпорту.

Утиліти аналізу файлів. Дуже часто виникає необхідність швидко дізнатись, яким пакувальником або захисним програмним забезпеченням оброблена та або інша програма, знайти всі текстові рядки в дампі пам'яті, проглянути вміст файлу у вигляді таблиці записів, вивести у файл список функцій, що імпортуються і експортуються програмою, і багато іншого. Для таких цілей існує величезна кількість спеціалізованих утиліт, що дозволяють швидко проаналізувати файл на наявність тих або інших ознак. Ці утиліти, як правило, не є життєво необхідними, але, при їх належній якості, здатні заощадити час і сили.

Шістнадцяткові редактори і редактори ресурсів. Це, поза сумнівом, найстародавніші з інструментів програмістів, які ведуть свою історію з тих часів, коли програмісти ще уміли читати і правити виконуваний код, не вдаючись до допомоги дизасемблерів. Мистецтво читання коду останніми роками практично втратило актуальність. І тепер лише крекери практикують цей прийом, виправляючи в неправильних програмах, наприклад, ідеологічно чужий опкод 75h на істинний і досконалий 0EBh.

Редактори ресурсів, у принципі, займаються тим самим, але по відношенню до прикріплених (прилінкованих) до виконуваного файлу ресурсів. Саме за допомогою редакторів ресурсів виконується значна частина робіт по незалежній русифікації програм і доробці інтерфейсів. Поруч із редакторами йдуть патчери, які дозволяють створити невеликий виконуваний файл, що автоматично вносить зміни в оригінальний файл програми або в код цієї програми безпосередньо в пам'яті.

API-шпигуни і інші утиліти моніторингу. Дуже часто буває необхідно знати, які саме дії виконує та або інша програма, звідки читає і куди записує дані, які стандартні функції і з якими параметрами вона викликає. Одержати ці відомості якраз і допомагають утиліти моніторингу. Такі утиліти діляться на дві великі групи:

- ті, які відстежують сам факт виникнення яких-небудь подій;
- ті, які дозволяють виявити один або декілька специфічних типів змін, що відбулися в системі за певний проміжок часу.

До першої групи відносяться всілякі API-шпигуни, що перехоплюють виклики системних функцій (деякі API-шпигуни вміють перехоплювати не тільки системні функції), добре відомі утиліти Reg-, File-, PortMon і інші, перехоплювачі системних повідомлень і багато інших. Ці утиліти, як правило, надають докладну інформацію про відстежувані події, але генерують досить об'ємні і незручні для аналізу протоколи, якщо відстежувані події відбуваються досить часто.

Друга група представлена програмами, що створюють знімки реєстру, жорсткого диска, системних файлів і т. п. Ці програми дозволяють порівняти стан комп'ютера до і після якоїсь події, побудувати список відмінностей між станами і на основі цього списку робити певні висновки.

Інші утиліти. Існує величезна кількість утиліт, що не вписуються в наведені вище категорії або потрапляють відразу в декілька категорій.

Отже, злам програм або крекінг – заняття вельми багатогранне, і такі ж багатогранні інструменти, які в ньому використовуються. Більш того, деякі з утиліт, які можуть бути корисні для крекінгу, створювалися для абсолютно інших цілей (хорошим прикладом може служити програма GameWizard32, яка взагалі-то була призначена, щоб махлювати в комп'ютерних іграх, але виявилася корисною при розкритті програми з обмеженням на максимальне число записів, що вводяться).

1.3 Способи вбудовування захисних механізмів в ПЗ

Вбудовування захисних механізмів у захищувані програми можна виконати такими основними способами:

- вставкою перевірочного механізму в початковий код на етапі розробки і налагодження програмного продукту – вбудований захист;
- вставкою фрагмента перевірочного коду у виконуваний файл – навісний захист;
- перетворенням виконуваного файлу до невиконуваного вигляду (шифрування, архівація з невідомим параметром і т. д.) і застосуванням для завантаження не засобів операційного середовища, а деякої програми, в тілі якої і здійснюються необхідні перевірки;
- комбінуванням вказаних методів.

Стосовно конкретної реалізації захисних механізмів для конкретної обчислювальної архітектури можна говорити про захисний фрагмент у виконуваному або початковому коді.

До процесу і результату вбудовування захисних механізмів можна пред'явити такі ***вимоги***:

- висока трудомісткість виявлення захисного фрагмента при статичному дослідженні (особливо актуальна при вбудовуванні в початковий код програмного продукту);
- висока трудомісткість виявлення захисного фрагмента при динамічному дослідженні (налагоджуванні і трасуванні за зовнішніми подіями);
- висока трудомісткість обходу або редагування захисного фрагменту.

Можливість вбудовування захисних фрагментів у виконуваний код обумовлена типовою архітектурою виконуваних модулів різних операційних середовищ, що містять, як правило, адресу точки входу у виконуваний модуль. У цьому випадку додавання захисного фрагмента відбувається таким чином. Захисний фрагмент додається до початку або кінця виконуваного файлу, точка входу корегується так, щоб при завантаженні управління передалося додатковому захисному фрагменту, а у складі захисного

фрагмента передбачається процедура повернення до оригінальної точки входу. Достатньо часто оригінальний виконуваний файл піддається перетворенню. У цьому випадку перед поверненням управління оригінальній точці входу здійснюється перетворення образу оперативної пам'яті завантаженого виконуваного файлу до початкового вигляду.

У разі доповнення динамічних бібліотек можлива корекція вказаним чином окремих функцій.

Істотним недоліком розглянутого методу є його легка виявляємість і у разі відсутності перетворення оригінального коду виконуваного файлу – легка можливість обходу захисного фрагмента шляхом відновлення оригінальної точки входу.

1.4 Структура програм, захищених від дослідження

З вищезазначеного витікає, що для захисту програмного забезпечення від дослідження необхідно застосовувати методи захисту від дослідження файлу з виконуваним кодом програми, що зберігається на зовнішньому носії, а також використати методи захисту виконуваного коду, завантажуваного в оперативну пам'ять для виконання цієї програми.

У першому випадку захист може бути заснований, наприклад, на шифруванні конфіденційної частини програми, а в другому – на блокуванні доступу до виконуваного коду програми в оперативній пам'яті з боку налагоджувачів. Крім того, перед завершенням роботи захищуваної програми повинен онулюватися весь її код в оперативній пам'яті. Це запобігатиме можливості несанкціонованого копіювання з оперативної пам'яті дешифрованого виконуваного коду після виконання програми, що захищається.

Таким чином, програма, що захищається від несанкціонованого дослідження, повинна складатися з основних трьох компонент.

1. *Ініціалізація***,** яка повинна забезпечувати виконання таких функцій:
   - збереження параметрів операційного середовища функціонування (векторів переривань, вмісту регістрів процесора і т. д.);
   - заборону всіх внутрішніх і зовнішніх переривань, обробка яких не може бути запротокольована в захищуваній програмі;
   - завантаження в оперативну пам'ять і дешифрування (якщо захист відбувається за допомогою шифрування) коду конфіденційної (контрольної, секретної) частини програми;
   - передачу управління конфіденційній частини програми.
2. *Конфіденційна (секретна) частина програми*, призначена для виконання основних цільових функцій і повинна бути захищена, наприклад, шифруванням для попередження внесення в неї програмної закладки.
3. *Деініціалізатор (деструктор)***,** який після виконання конфіденційної частини програми повинен виконати такі дії:
   - онулення конфіденційного коду програми в оперативній пам'яті;

– відновлення параметрів операційної системи (векторів переривань, вмісту регістрів процесора і т. д.), які були встановлені до заборони неконтрольованих переривань;
– виконання операцій, які неможливо було виконати при забороні неконтрольованих переривань;
– звільнення всіх незадіяних ресурсів комп'ютера і завершення роботи програми.

Для більшої надійності ініціалізація може бути частково зашифрованою і у міру виконання дешифрувати саму себе. Дешифруватися у міру виконання може і конфіденційна частина програми. Таке дешифрування називається *динамічним дешифруванням* виконуваного коду. У цьому випадку чергові ділянки програм перед безпосереднім виконанням розшифровуються, а після виконання відразу знищуються.

Для підвищення ефективності захисту програм від дослідження необхідним є внесення в програму *додаткових функцій безпеки*, направлених на захист від трасування. До таких функцій можна віднести:
– періодичний підрахунок контрольної суми області оперативної пам'яті, зайнятої початковим захищуваним кодом; порівняння поточної контрольної суми із заздалегідь сформованою еталонною і вживання необхідних заходів у разі неспівпадіння;
– перевірку кількості оперативної пам'яті, яку займає захищувана програма, порівняння з об'ємом, до якого програма адаптована, і вживання необхідних заходів у разі невідповідності;
– контроль часу виконання окремих ділянок програми;
– блокування клавіатури на час відпрацювання особливо критичних алгоритмів.

Для захисту ПЗ від дослідження за допомогою дизасемблерів можна використовувати і такий спосіб, як ускладнення структури самої програми з метою заплутування зловмисника, який дизасемблює цю програму. Наприклад, можна використовувати різні сегменти адреси для звернення до однієї і тієї ж області пам'яті. У цьому випадку зловмиснику буде важко здогадатися, що насправді програма працює з однією і тією ж областю пам'яті. Детальніше про методи захисту від дизасемблерів і налагоджувачів йтиметься далі.

Контрольні питання

1. Які способи впровадження захисних механізмів існують?
2. Які основні функції повинні виконувати системи захисту від НСК?
3. В чому полягає статичне і динамічне дослідження програм і які засоби для цього використовують?
4. Як зрозуміти поняття статистичного і синтаксичного дослідження програм і які інструменти для цього можуть бути використані?
5. Якою повинна бути структура програм, захищених від дослідження, і з яких компонентів вона складається?

2 ЗАХИСТ ВІД ДИЗАСЕМБЛЮВАННЯ

2.1 Необхідність і доцільність захисту від дизасемблювання

Цілком зрозуміле бажання більшості програмістів працювати з твердою копією досліджуваної програми. Відсутність початкових текстів зовсім не є непереборною перешкодою для вивчення і модифікації коду додатку. Методики зворотного проектування дозволяють автоматично розпізнавати бібліотечні функції, локальні змінні, стекові аргументи, типи даних, розгалуження, цикли і т. д. З часом дизасемблери, ймовірно, навчаться генерувати лістинги, близькі на вигляд до мов високого рівня.

Сьогодні трудомісткість аналізу двійкового коду не настільки велика, щоб надовго зупинити зловмисників. Величезне число скоєних зламів – краще тому підтвердження. В ідеальному випадку знання алгоритму захисту не повинно впливати на стійкість до зламу, але це досягається далеко не завжди. Наприклад, якщо виробник серверної програми вирішить встановити в демонстраційній версії обмеження на кількість одночасно оброблюваних з'єднань, зловмиснику достатньо знайти інструкцію процесора, що здійснює таку перевірку і видалити її. Модифікації програми можна перешкодити постійною перевіркою контрольної суми, але знову таки код, який обчислює цю контрольну суму і звіряє її з еталоном, можна знайти і видалити.

Скільки б рівнів захисту не передбачено, один або мільйон, програма може бути зламана – це тільки питання часу і витрачених зусиль. Але у відсутності дієвих правових регуляторів захисту інтелектуальної власності розробникам доводиться більше покладатися на стійкість свого захисту, ніж на закон. Існує думка, що якщо витрати на нейтралізацію захисного механізму будуть не нижчими за вартість легальної копії, її ніхто не зламуватиме. Це невірно. Матеріальний стимул – не єдине, що рухає хакером. Набагато сильнішою мотивацією є інтелектуальна боротьба з автором захисту, спортивний азарт, цікавість, підвищення свого професіоналізму, та й просто цікаве проведення часу. Деякі люди можуть тижнями корпіти над налагоджувачем, знімаючи захист з програми вартістю в декілька доларів, або навіть поширюваної безкоштовно (наприклад, диспетчер файлів FAR абсолютно безкоштовний, але це не рятує його від зламу).

Внаслідок цього першочерговою задачею зловмисника при зламі практично будь-якого захисту є дизасемблювання коду програми, що виконується, і одержання лістингу з мнемонічним зображенням асемблерних команд, тобто можливість статичного дослідження програми.

Вхідними даними для дизасемблера є програма або окрема ділянка коду. Результатом його роботи є лістинг (вихідний текст програми мовою асемблера), який в ідеалі повинен бути максимально близький до оригіналу. Дизасемблер відновлює код програми, послідовно декодуючи команду за командою з того місця, куди програмі передається керування

при її запуску. Він намагається додержуватися порядку виконання інструкцій, відрізняючи їх у такий спосіб від даних.

Будь-який, навіть самий витончений захист від копіювання, що забезпечує майже 100% гарантію легальності копії, є марним, якщо код програмного продукту доступний для вивчення й аналізу. У ньому завжди знайдуться місця, злегка змінивши які, можна якщо не цілком відключити захист, то, принаймні, прив'язати його ще до декількох комп'ютерів.

2.2 Основні методи протидії дизасемблюванню програм

Виділимо декілька найзагальніших підходів до захисту програмного забезпечення від дизасемблювання.

1. *Шифрування коду* – один з найпоширеніших та надійних методів захисту від статичного дослідження.
2. *Маніпулювання заголовками EXE-файлів* – метод, побудований на використанні властивостей структури виконуваних файлів і розробці навісного захисту.
3. *Обман дизасемблера*. Методи цієї групи полягають у тому, щоб заплутати дизасемблер: підсунути дані замість коду, дезорієнтувавши його логіку, повести його по помилковому сліду, підсунути зайві фрагменти коду і т. д. Всі ці способи обману можна поділити на такі групи:
   - різноманітні методи обфускації коду;
   - ускладнення логіки програм;
   - різноманітні додаткові методи боротьби з інтерактивними та автоматичними дизасемблерами.

   Це авангардні і досить перспективні методи захисту ПЗ не тільки від дизасемблювання, а й від налагоджування.
4. Методи *емуляції*, серед яких виділяють:
   - емуляцію процесора;
   - емуляцію мультизадачості.

   Розглянемо детальніше кожен з цих методів.

2.3 Шифрування коду

Гарантовано перешкодити аналізу коду дозволяє тільки стійке шифрування програми. Але сам процесор не може безпосередньо виконувати зашифрований код, тому перед передачею управління його необхідно розшифрувати. Якщо ключ міститься всередині програми, стійкість такого захисту близька до нуля. Все, чого може домогтися розробник, – утруднити пошук і отримання цього ключа, тим або іншим способом перешкоджаючи несанкціонованому налагодженню і дизасемблюванню програми. Інша справа, коли ключ міститься поза програмою. Тоді стійкість захисту визначається стійкістю використовуваного криптоалгоритму (звичайно, за умови, що ключ перехопити неможливо). Існує багато криптостійких шифрів, злам яких є недоступним для рядових зловмисників.

Шифрування коду – це найбільш загальний і досить надійний метод захисту від дизасемблювання. Сутність цього методу полягає у наступному. Ділянка коду, що захищається, шифрується якимось алгоритмом, який реалізується за допомогою якоїсь математичної моделі, одночасно з цим використовуваної для генерації ключа. Різні гілки програми зашифровують різними ключами, і для обчислення цього ключа необхідно знати стан моделі на момент передачі управління на відповідну гілку програми. До програми додається код розшифровувача, що у потрібний момент розшифровує його і передає йому керування. Код динамічно розшифровується в процесі виконання, а для розшифрування його повністю, потрібно послідовно перебрати всі можливі стани моделі. Якщо їх кількість буде дуже велика, відновити весь код стане практично неможливо. Отже, при дизасемблюванні він стане перед зламником в зашифрованому вигляді (у такому, у якому він знаходиться в тілі програми до розшифрування) і, природно, буде сприйнятий дизасемблером неправильно.

Слід зауважити, що цей метод досить уразливий через те, що алгоритм розшифрування доступний зламнику. Адже він міститься в тій самій програмі. А отже, необхідно лише знайти його і розшифрувати код, використовуючи отриману інформацію. Однак, дуже часто і цього не потрібно: існує маса готових інструментів, що дозволяють розшифрувати програму навіть в автоматичному режимі.

Тут важлива не криптостійкість методу шифрування, а сам факт, що код якимось чином змінюється. Виходячи з цього, можна для захисту програм використовувати нижче наведені алгоритми шифрування.

1) *Гамування*, різноманітні *підстановки і перестановки*. Справа ускладнюється необхідністю зберігати таблиці підстановок перестановок або гами, хоча можна їх обчислювати в процесі роботи. Це зручно використовувати для організації перевірок ключових параметрів захисту.
2) Цікавим є *використання асиметричних алгоритмів* (RSA або El-Gamal). Тоді, навіть розібравшись в розшифрованій програмі в оперативному запам’ятовувальному пристрої (ОЗП) і зрозумівши, що потрібно "виправити", ці зміни не можна буде зашифрувати і вставити в програму, оскільки в програмі буде присутній тільки ключ для розшифрування.
3) Також можна спробувати *переміщення частин програми в ОЗП і зчитування її з диска вроздріб*. Це іноді робиться в реальних системах захисту (наприклад, у CERBERUS).
4) Як варіант шифрування можна використовувати *ущільнення коду*, що не тільки перешкодить дизасемблюванню, але ще і зменшить розмір файлу, що завантажується.

Таким чином, для ускладнення дизасемблювання краще всього підлодить шифрування окремих ділянок програм або всієї програми цілком. Наприклад, частину програми-інсталятора можна оформити у вигляді окремої COM-програми. Після трансляції початкового тексту цієї програми її можна зашифрувати тим або іншим способом, і в зашифрованому

вигляді підвантажувати в пам'ять як програмний оверлей. Після завантаження програму слід розшифрувати в пам'яті і передати їй управління.

Ще краще виконувати динамічне розшифрування програми у міру її виконання, коли ділянки програми розшифровуються безпосередньо перед використанням і після використання відразу ж знищуються.

При розшифруванні можна копіювати ділянки програми в інше місце оперативній пам'яті. Нехай, наприклад, програма складається з декількох частин. Після завантаження її в оперативну пам'ять управління передається першій частині програми. Ця частина призначена для розшифрування другої частини, розташованої в пам'яті слідом за першою.

Задача другої частини – переміщення третьої частини програми на місце вже використаної першої частини і розшифрування її там.

Третя частина, одержавши управління, може перевірити своє розташування щодо префікса програмного сегменту і, у разі правильного розташування (відразу вслід за PSP), почати завантаження сегментних регістрів такими значеннями, які необхідні для виконання четвертої – інсталяційної – частини програми.

Якщо спробувати дизасемблювати програму, складену таким чином, то з цього нічого не вийде.

Як зазначено вище, шифрування виконуваного коду є досить дієвим способом захисту програм від дизасемблювання. Але криптографічним методам захисту присвячені окремі дисципліни. Отож, у даному посібнику ми приділимо увагу іншим методам захисту програмного забезпечення.

2.4 Маніпулювання EXE-заголовком

Модифікація заголовка EXE-файлу використовується в різних сучасних системах захисту. Але для того, щоб вміти змінити EXE-заголовок виконуваного файлу так, щоб не зіпсувати роботу самої програми, треба мати уявлення, що ж являє собою цей самий заголовок.

Дизасембльований модуль, який можна отримати в результаті роботи дизасемблера, складається, як мінімум, з двох частин.

Перша з них – це безпосередньо виконуваний код. Аналіз цієї частини програми дозволяє визначити, яким чином змінюються дані у програмі. На цьому етапі задачею дослідника є відновлення ходу думки програміста, що писав дану програму. Це задача досить творча, успіх її реалізації залежить, перш за все, від кваліфікації людини, яка здійснювала дослідження дизасембльованого модуля. Формалізувати цей процес, а тим більше автоматизувати, практично неможливо.

Друга частина містить дані, використовувані програмою і операційною системою. Саме ця частина може дати багато інформації про те, для якої операційної системи створено дану програму, який компілятор використано, який можливий склад основних компонентів програми і т.д.

2.4.1 Загальна структура виконуваного файлу

В операційних системах, що входять у сімейство Win32, існує декілька типів виконуваних файлів. Ці типи розрізняються за розширеннями файлів, а також за сигнатурою – визначеною послідовністю байтів, що знаходиться за певним зміщенням у тілі виконуваного файлу. Сигнатури виконуваних файлів перераховані у заголовочному файлі <*winnt.h*> (табл.1).

Таблиця 1 – Сигнатури виконуваних файлів

Сигнатура	Значення	Опис
IMAGE_DOS_SIGNATURE	0x5A4D	MZ
IMAGE_OS2_SIGNATURE	0x454E	NE
IMAGE_OS2_SIGNATURE_LE	0x454C	LE
IMAGE_VXD_SIGNATURE	0x454C	LE
IMAGE_NT_SIGNATURE	0x00004550	PE00

Ми розглянемо лише формат PE-файлів, оскільки саме з цими типами виконуваних файлів ми найчастіше маємо справу.

Структурно PE-файл складається з *заголовка* (header), *сторінкового іміджу* (image page) і необов'язкового *оверлею* (overlay). Представлення PE-файлу в пам'яті називається його *віртуальним образом* (virtual image), або просто *образом*, а на диску – *файлом,* або *дисковим образом.*

Образ характеризується двома фундаментальними поняттями – *адресою базового завантаження* (image base) і *розміром* (image size). При наявності переміщуваної інформації (relocation/fixup table) образ може бути завантажений за адресою, відмінною від image base, призначеною самою операційною системою.

Образ поділяється на *сторінки* (pages), а файл – на *сектори* (sectors). Віртуальний розмір сторінок/секторів явно задається у заголовку і не обов'язково повинен співпадати з фізичним.

Як правило, після сигнатури файлу розташовується заголовок виконуваного файлу, причому спочатку йде заголовок виконуваного файлу DOS. Чому саме DOS? Тому що виконуваний файл Windows є одночасно і виконуваним файлом DOS. Загальна схема виконуваного PE-файлу така:

old-exe-заголовок – заголовок EXE-файлу DOS
PE File Signature – сигнатура PE-файлу
PE-заголовок – заголовок виконуваного файлу Windows
Таблиця об'єктів (розділів, сегментів)
Розділ .... Розділ

Заголовок DOS

Exe-програми можуть складатися з декількох сегментів (кодів, даних, стека). Exe-файл має заголовок, що використовується при його завантаженні. Заголовок складається з форматованої частини, що містить сигнатуру і дані, необхідні для завантаження Exe-файлу, і таблиці для

настроювання адрес (*Relocation Table*). Таблиця складається зі значень у форматі `<сегмент : зсув>`. До зсувів у завантажувальному модулі, на яких указують значення в таблиці, після завантаження програми в пам'ять повинна бути додана сегментна адреса, з якої завантажена програма.

Заголовок розташовується на початку файлу EXE-програми. Опис його знаходиться у заголовочному файлі <*winnt.h*> і має таку структуру:

```
typedef struct _IMAGE_DOS_HEADER // DOS .EXE header
{
    WORD   e_magic;       // сигнатура EXE-файла (4D5Ah – 'MZ'). Ініціали
                          // Марка Збиковскі,  провідного розробника
                          // MS-DOS і архітектора EXE-формата
    WORD   e_cblp;        // довжина останньої сторінки
    WORD   e_cp;          // довжина файлу в сторінках (по 512 байтів)
    WORD   e_crlc;        // число елементів у таблиці настроювання адрес
    WORD   e_cparhdr;     // довжина заголовка в параграфах (по 200h байтів)
    WORD   e_minalloc;    // мінімальний і максимальний обсяг пам'яті,
    WORD   e_maxalloc;    // яку потрібно виділити після
                          // завантаженого модуля (у параграфах)
    WORD   e_ss;          // зсув сегмента стека (для установлення SS)
    WORD   e_sp;          // значення SP, установлюване при вході
    WORD   e_csum;        // контрольна сума
    WORD   e_ip;          // значення IP при вході (стартова адреса)
    WORD   e_cs;          // зсув сегмента коду (для установлення CS)
    WORD   e_lfarlc;      // зсув 1-го елемента таблиці настроювання адрес
    WORD   e_ovno;        // номер оверлею (0 – для кореневого модуля)
      . . .
    LONG   e_lfanew;      // зміщення PE-заголовка в байтах від початку файлу
  }  IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Саме з цього заголовка починаються всі виконувані файли Windows. При запуску Windows-програм з-під DOS в більшості випадків видається повідомлення про те, що програму слід запустити у середовищі Windows. Це повідомлення як раз і видається DOS-частиною програми.

Перший байт образу завжди починається з сигнатури “MZ”, в чому легко переконатись, завантаживши файл у будь-який налагоджувач, дизасемблер або шістнадцятковий редактор, і переглянувши його дамп.

Якщо значення у слові, яке знаходиться зі зміщенням 18h, більше або рівне 40h, то за зміщенням 3Ch знаходиться значення зміщення, за яким знаходиться сигнатура якогось іншого виконуваного файлу. І якщо ця сигнатура представлена значенням «PE00», то файл є виконуваним файлом Windows. Отже, виконуваний файл Windows сам по собі існувати не може.

Таким чином, ми визначили зміщення на початок виконуваного файлу Windows. А він, як і кожний інший виконуваний файл, також починається з заголовка.

Заголовок виконуваного файлу Windows

У заголовочному файлі <*winnt.h*> цей заголовок описаний так:

```
typedef struct _IMAGE_ROM_HEADERS
{   DWORD  Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_ROM_OPTIONAL_HEADER OptionalHeader;
} IMAGE_ROM_HEADERS, *PIMAGE_ROM_HEADERS;
```

З цього опису видно, що заголовок виконуваного файлу складається з трьох частин:

- сигнатура виконуваного файлу;
- обов'язкова частина заголовка PE-файлу;
- необов'язкова частина заголовка PE-файлу.

Обов'язкова частина заголовку має таку структуру:

```
typedef struct _IMAGE_FILE_HEADER
{  WORD  Machine;   // для якого процесора скомпільовано даний код:
                    // 0 – невідомий процесор; 0x14C – Intel 386,
                    // 0x14D – Intel 486, 0x14E – Intel Pentium і т.д.
   WORD  NumberOfSections; // кількість розділів (розділи ресурсів, даних,
                           // коду, експортованих, імпортованих функцій, ...
   DWORD TimeDateStamp;    // мітка часу створення файлу
   DWORD PointerToSymbolTable;   // зміщення таблиці символів, якщо це
                                 // передбачено у скомпільованому модулі
   DWORD NumberOfSymbols;        // кількість символів у таблиці символів
   WORD  SizeOfOptionalHeader;   // розмір необов'язкової частини заголовка
   WORD  Characteristics; // прапорці: 0x0100 – для 32-бітного комп'ютера,
                          // 0x1000 – системний файл, 0x2000 – Dll-файл
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Необов'язкова частина заголовка виконуваного файлу представлена структурою, яка наведена нижче. У коментарях пояснимо лише деякі поля цієї структури. Для більш детального ознайомлення з цією частиною заголовка слід звернутись до оригінального коду заголовка <*winnt.h*>.

```
typedef struct _IMAGE_OPTIONAL_HEADER
{          // Standard fields:
    WORD  Magic;                  // 0x10b
    BYTE  MajorLinkerVersion;     // старші і молодші  розряди номера версії
    BYTE  MinorLinkerVersion;     // лінкера, використаного при створенні програми
    DWORD SizeOfCode;             // округлений сумарний розмір всіх розділів
    DWORD SizeOfInitializedData; // сумарний розмір ініціалізованих даних
    DWORD SizeOfUninitializedData;
    DWORD AddressOfEntryPoINT ;  // відносна віртуальна адреса точки входу
                                 // у програму (база – у змінній нижче)
    DWORD BaseOfCode;            // відносна адреса програмної секції
    DWORD BaseOfData;            // відносна адреса секції даних
                                 // NT additional fields:
    DWORD ImageBase;             // комірка для базової адреси, за якою файл
                                 // буде відображено у пам'яті
    DWORD SectionAlignment;      // поля для вирівнювання
    DWORD FileAlignment;
    WORD  MajorOperatingSystemVersion;  // старші і молодші розряди номера
    WORD  MinorOperatingSystemVersion;  // старої версії, яка дозволить
    WORD  MajorImageVersion;            // виконувати цей файл0
    WORD  MinorImageVersion;
    WORD  MajorSubsystemVersion;
    WORD  MinorSubsystemVersion;
    DWORD Reserve1;        // резервні
    DWORD SizeOfImage;     // загальний розмір завантаж. модуля у пам'яті
    DWORD SizeOfHeaders;   // сумарний розмір заголовків і таблиці розділів
    DWORD CheckSum;        // контрольна сума виконуваного файлу
    WORD  Subsystem;       / тип підсистеми для здійснення інтерфейсу
    WORD  DllCharacteristics;    // 1 – виклик Dll при завантаженні, 2 – інакше
    DWORD SizeOfStackReserve;    // розмір потрібного стека
    DWORD SizeOfStackCommit;
    DWORD SizeOfHeapReserve;
    DWORD SizeOfHeapCommit;
    DWORD LoaderFlags;           // не використовується
    DWORD NumberOfRvaAndSizes;   // кількість елементів наступного масиву
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

Використання заголовків при завантаженні файлів у пам'ять

Як видно з опису полів у DOS-заголовку, певні поля визначають місце розташування точки входу в програму, місце розташування стека, розмір кореневого сегмента програми.

Після DOS-заголовка розташовується Window-заголовок, де зберігаються адреси тих слів у коді програми, що модифікуються операційною системою під час завантаження програми.

При запуску Exe-програми системним завантажувачем (викликом функції DOS 4Bh) виконуються такі дії.

1. Визначається сегментна адреса вільної ділянки пам'яті, розмір якого достатній для розміщення програми.
2. Створюється і заповнюється блок пам'яті для змінних середовища.
3. Створюється блок пам'яті для префікса програмного сегмента (PSP – Program Segment Prefics) і програми. При завантаженні програми сегменти розміщуються в пам'яті так, як показано на рисунку1. Образ програми в пам'яті починається з PSP, який створюється і заповнюється системою. PSP завжди має розмір 256 байтів і вміщує таблиці, поля даних, які використовуються системою в процесі виконання програми (таблиця сегментів, таблиця ресурсів, таблиця резидентних імен, таблиця посилань на модулі, таблиця переміщень і т.д.).

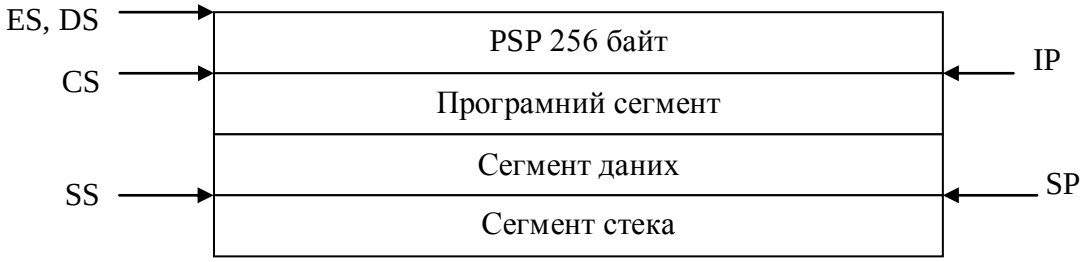


Рисунок 1 – Образ пам'яті EXE-програми

4. У робочу область завантажувача зчитується форматована частина DOS-заголовка Exe-файлу.
5. Обчислюється довжина завантажуваного модуля за формулою:
   `Size = ((e_cp◎512) - (e_cparhdr◎16)) - e_cblp.`
6. Визначається зсув завантажувального модуля, рівний `e_cparhdr*16`.
7. Обчислюється сегментна адреса (`START_SEG`) для завантаження.
8. Зчитується в пам'ять завантажувальний модуль (починаючи з адреси `START_SEG:0000`).
9. Розподіляється пам'ять для програми відповідно до `e_minalloc` і `e_maxalloc`.
10. Ініціалізуються сегментні регістри:
    a) ES, DS – вказують на початок PSP (що дає можливість зберігати їх зміст, а потім звертатися в програмі до PSP);
    b) AX – результат перевірки правильності ідентифікаторів драйверів, зазначених у командному рядку;
    c) `SS = START_SEG + e_ss` – початок сегмента стека;
    d) покажчик стека `SP = e_sp` – зміщення кінця сегмента стека;

e) `CS = START_SEG + e_cs` – початок програмного сегмента,

f) покажчик команд `IP = e_ip` – відносна адреса точки входу в програму (із операнда директиви END).

Отже, після завантаження програми в пам'ять адресними є всі сегменти, крім сегмента даних. Ініціалізація регістра DS у перших рядках програми дозволяє зробити адресним і цей сегмент.

Сутність найпростішого методу захисту полягає у тому, що змінивши, наприклад, поле SS:SP, можна оголосити (для дизасемблера) найважливішу частину програми, наприклад, стеком, що призведе до неправильної її обробки. Також можна змінювати й інші поля EXE-заголовка. Але в цілому цей метод має мало сенсу, якщо використовується в сполученні із шифруванням коду.

Таблиця об'єктів (object table)

Таблиця об'єктів (або таблиця розділів) – сукупність даних певного призначення: про експортовані та імпортовані функції, про ресурси, про переміщення (relocations) і т. д., які компактно розміщені у виконуваному файлі. Тобто, таблиця об'єктів – це просто інформація про зміст розділів.

Таблиця об'єктів розміщається відразу ж після заголовка PE-файлу. Часто для дослідження програми або з метою приховування механізмів захисту виникає необхідність обчислити зміщення початку таблиці об'єктів. Звичайно, це можна зробити так: взяти зміщення PE-заголовка і додати до нього розмір заголовка. Але у заголовочному файлі <*winnt.h*> для цієї мети описано макрос:

```
#define IMAGE_FIRST_SECTION().
```

Природно, що описи розділів мають один і той самий формат. Отже, вся таблиця об'єктів являє собою масив. Таким чином, нам необхідно знати, що являє собою кожний елемент цього масиву і де знайти кількість його елементів. Щодо кількості елементів цього масиву, то слід згадати значення поля NumberOfSections в обов'язковій частині заголовка. Що ж стосується формату елементів цього масиву, то знову ж таки у заголовочному файлі <winnt.h> існує опис структури для цієї мети:

```
typedef struct _IMAGE_SECTION_HEADER      // заголовок образа розділу
{  BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
   union {  DWORD   PhysicalAddress;      // адреса і розмір розділу
            DWORD   VirtualSize;
         } Misc;
   DWORD VirtualAddress;       // зміщення, за яким буде розміщено цей розділ
   DWORD SizeOfRawData;        // округлений розмір розділу
   DWORD Point erToRawData;  // зміщення початку розділу відносно початку файлу
   DWORD PointerToRelocations;      // в exe-файлах не використовують
   DWORD PointerToLinenumbers;      // в exe-файлах не використовують
   WORD  NumberOfRelocations;// в exe-файлах не використовують
   WORD  NumberOfLinenumbers;// практично не використовують
   DWORD  Characteristics;   // атрибути розділу (для читання, для запису,
                     // кешується чи ні, чи містить ініційовані дані і т. д.
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Розмір цього заголовка становить 40 байтів.

Масив Name розміром IMAGE_SIZEOF_SHORT_NAME = 8 задає найменування відповідного розділу програми.

Розділи у виконуваному файлі

Ми вже знаємо, що виконуваний файл складається з сигнатури, заголовків, таблиці розділів і безпосередньо розділів. Розглянемо розділи, які зустрічаються частіше і яку інформацію вони містять.

.text, CODE – секція програмного коду. Ця секція містить код, який генерує компілятор. В компіляторах фірми Microsoft ця секція називається .text, а в компіляторах фірми Borland (нині Inprise) вона зазвичай називається CODE. З цією секцією нерозривно пов'язані секції імпорту і експорту функцій.

Звичайними засобами операційних систем переглянути вміст цих секцій неможливо. Для їх перегляду використовуються різні дизасемблери, налагоджувачі, монітори і т. д.

.edata, DATA – секція ініціалізованих даних. Назва цієї секції також залежить від виробника: .edata – для компіляторів Microsoft, DATA – для компіляторів Borland. У даній секції містяться глобальні та статичні змінні, літерали.

.bss – секція неініціалізованих даних. Оскільки ніяких даних у цій секції зберігати не потрібно, то зазвичай її розмір у виконуваному файлі дорівнює 0. точніше, насправді такої секції у виконуваному файлі немає, а є лише згадування про неї у таблиці секцій. Компілятори фірми Borland цієї секції не створюють.

.idata, ImportDATA – секція імпортованих функцій. У цій секції знаходяться таблиці, що забезпечують імпорт функцій з інших бібліотек динамічної компанування.

.edata,ExportDATA – секція експортованих даних. Секція містить дані про ті функції, які експортуються даним модулем.

.rsrc, ReSouRCes – секція ресурсів. Вона зберігає всі ресурси, які присутні у виконуваному файлі.

.reloc – секція містить таблицю базових поправок. Вона використовується лише тоді, коли завантажувач може завантажити файл за адресою, починаючи з якої планував здійснити завантаження редактор зв'язків.

.tls (Thread Local Section). Якщо передбачається використовувати локальну пам'ять потоку, то ці дані не попадають ні в секцію ініціалізованих даних, ні в секцію неініціалізованих даних. Замість цього вони попадають саме в дану секцію.

У виконуваних файлах зустрічаються і інші секції.

Таким чином, розглянувши загальну структуру виконуваних файлів, можна зробити висновки про те, що у exe-файлах існує багато місць, в які можна втрутитись, не змінюючи при цьому функціональних можливостей самої програми, або просто грамотно змінити певні адреси у заголовках для того, щоб вставити власний фрагмент захисного коду і передати йому управління програмою.

2.4.2 Механізми впровадження захисного коду в PE-файли

Механізми впровадження захисних механізмів у виконувані файли можна класифікувати по-різному:

- за місцем впровадження (почало, кінець, середина);
- за «геополітикою» (затирання початкових даних, впровадження у вільний простір, переселення початкових даних на нове місце);
- за надійністю (гранично коректне, цілком коректне і вкрай некоректне впровадження);
- за реєрентерабельністю (реєрентерабельне чи нереєнтерабельне) і т. д.

Відштовхуючись від характеру дії на фізичний і віртуальний образ піддослідної програми, всі існуючі механізми впровадження розділяють на чотири категорії, позначені латинськими літерами A, B, C і Z.

До **категорії A** відносяться механізми, що *не викликають зміни адресації ні фізичного, ні віртуального образів.* Після впровадження у файл ні його довжина, ні обсяг виділеної при завантаженні пам'яті не змінюються і всі базові структури лишаються на своїх адресах. Цій умові відповідають:

- впровадження в порожнє місце файлу (PE-заголовок, хвости секцій, регулярні послідовності);
- впровадження шляхом ущільнення частини секції і створення нового NTFS-потоку усередині файлу.

До **категорії B** відносять механізми, що *викликають зміни адресації тільки фізичного образу*. Після впровадження у файл його довжина збільшується, проте кількість виділеної при завантаженні пам'яті не змінюється і всі базові структури проектуються за тими самими адресами, але їх фізичні зсуви змінюються, що вимагає повної або часткової перебудови структур, які прив'язуються до своїх фізичних адрес. І якщо хоча б одна з них залишиться не скоректованою (скоректована неправильно), файл-носій з великою імовірністю відмовить у роботі. Категорії B відповідають:

- розсунення заголовка;
- скидання частини оригінального файлу в оверлей;
- створення свого власного оверлею.

До **категорії C** відносять механізми, що *викликають зміни адресації як фізичного, так і віртуального образів.* Довжина файлу і пам'ять, що виділяється при завантаженні, збільшуються. Базові структури можуть або залишатися на своїх місцях (тобто змінюються лише зсуви, відлічувані від кінця образу/файлу), або переміщатися по сторінковому іміджу довільним чином, вимагаючи обов'язкової корекції. Цій категорії відповідають:

- розширення останньої секції файлу;
- створення своєї власної секції;
- розширення серединних секцій.

До **категорії Z** («засекреченої») відносяться механізми, що взагалі не торкаються файла-носія і впроваджуються у його адресний простір непрямим шляхом, наприклад, модифікацією ключа реєстру, відповідального за автоматичне завантаження динамічних бібліотек. Цією технологією цікавляться в першу чергу мережеві черв'яки і шпигуни. Віруси до неї байдужі.

2.4.3 Х-код та вимоги до його якості і функціонування

Як бачимо, механізми впровадження в PE-файли досить різноманітні. Називатимемо частину коду, яку ми збираємося впроваджувати, ***Х-кодом.*** Х-код слід проектувати з урахуванням усіх вимог, що пред'являються середовищем того коду, в яку він буде впроваджений:

- Х-код повинен *бути повністю переміщуваним*, тобто зберігати свою працездатність незалежно від базової адреси завантаження (це досягається використанням відносної адресації);
- грамотно сконструйований Х-код *не повинен модифікувати своє середовище*, оскільки не відомо, чи є у нього права. Стандартна секція коду позбавлена атрибуту IMAGE_SCN_MEM_WRITE, і привласнювати його вкрай небажано, оскільки це не тільки демаскує Х-код, але і знижує імунітет програми-носія. При впровадженні в секцію даних це обмеження втрачає свою актуальність, проте далеко не у всіх випадках запис в секцію даних дозволений;
- Х-код *повинен бути компактним*, оскільки об'єм простору, придатного для впровадження, часом дуже обмежений. Має сенс розбити Х-код на дві частини: крихітний завантажувач і протяжний хвіст. Завантажувач краще всього розмістити в PE-заголовку або у регулярній послідовності всередині файлу, а хвіст скинути в оверлей або NTFS-потік, комбінуючи тим самим різні методи впровадження;
- Х-код *не повинен затримувати управління* більш ніж на декілька сотих, від сили десятих часток секунди, інакше факт впровадження стане дуже помітним, чого допускати у жодному випадку не можна.

Продумавши і сформувавши Х-код згідно цих вимог, його треба коректно впровадити у тіло захищуваної програми і заставити працювати.

Х-код повинен **функціонувати** у такому порядку:

1. *Розмістити своє тіло* всередині піддослідного файлу шляхом:
   - затирання – розміщення Х-коду поверх оригінальної програми (оскільки цей спосіб приводить до необоротної втрати працездатності початкової програми і реально застосовується тільки у вірусах, ми його не розглядатимемо);
   - інтеграція – розміщення Х-коду у вільному місці програми;
   - дописування Х-коду в початок, середину або кінець файлу із збереженням оригінального вмісту;
   - розміщення Х-коду поза основним тілом файла-носія, завантажуваного «головою» Х-коду, впровадженого у файл попередніми способами.
2. *Перехопити управління* до початку виконання основної програми або в процесі виконання. Це здійснюється такими шляхами:
   - встановленням заново точки входу на тіло Х-коду;
   - впровадженням в околі оригінальної точки входу EP (Entry Point) команди переходу на Х-код (перед передачею управління Х-код повинен видалити команду, відновивши початковий вміст EP);
   - встановленням заново довільно взятої команди JMP/CALL на тіло Х-коду з подальшою передачею управління на оригінальну адресу (це не

гарантує, що X-коду взагалі вдасться отримати управління, зате забезпечує йому скритність і захищеність від програм хакерів);
- модифікацією одного або декількох елементів таблиці імпорту з метою підміни функцій, що викликаються, своїми власними (такою технологією в основному користуються стелс-віруси, що вміло приховують свою присутність у системі).

3. *Визначити адреси API-функцій*, життєво важливих для власного функціонування. Це здійснюється такими шляхами:
   - пошуком необхідних функцій у таблиці імпорту файлу-хазяїна (якщо їх не виявиться, то слід або відмовитися від впровадження);
   - пошуком LoadLibrary()/GetProcAddress() у таблиці імпорту файлу-господаря з подальшим імпортуванням всіх необхідних функцій вручну (цих функцій в таблиці імпорту також може не виявитися);
   - прямим викликом API-функцій за їх абсолютними адресами, прописаними всередині X-коду. Адреси функцій KERNEL32.DLL/ NTDLL.DLL непостійні і змінюються від однієї версії системи до іншої, а адреси USER32.DLL і всієї решти призначених для користувача бібліотек непостійні навіть в рамках однієї конкретної системи і варіюються залежно від ImageBase решти завантажуваних бібліотек, тому даний спосіб допустимий не завжди;
   - додаванням в таблицю імпорту необхідних X-коду функцій (ними, як правило, є LoadLibrary()/GetProcAddress(), за допомогою яких можна витягнути з надр системи всі інші функції – достатньо надійний, хоча і дуже помітний спосіб);
   - безпосереднім пошуком функцій LoadLibrary()/GetProcAddress() в пам'яті. Оскільки KERNEL32.DLL проектується на адресний простір усіх процесів, а її базова адреса завжди вирівняна на межу 64 Кбайтів, треба всього лише просканувати першу половину адресного простору процесу для пошуку сигнатури MZ. Якщо її знайдено, переконуємося у наявності сигнатури PE, розташованої зі зсувом e_lfanew від початку базової адреси завантаження. Якщо вона присутня, аналізуємо поле DATA_DIRECTORY і визначаємо адресу таблиці експорту, в якій треба знайти функції LoadLibrary() і GetProcAddress(). Якщо ж хоча б одна з цих умов не виконується, зменшуємо покажчик на 64 Кбайти і повторюємо всю процедуру заново. Але перш, ніж щось читати з пам'яті, слід викликати функцію IsBadReadPtr(), переконавшись, що ми маємо право це робити;
   - визначенням адреси функції KERNEL32!_except_handler3, на яку вказує обробник структурних виключень за замовчуванням. Ця функція не експортується ядром, проте присутня в налагоджувальній таблиці символів. Це робиться так:

```
mov esi,  fs: [0]
lodsd
lodsd
```

   Після виконання коду регістр EAX містить адресу, що лежить десь в глибині KERNEL32. Вирівнюємо її на межу 64 Кбайтів і шукаємо MZ/

PE-сигнатури, як показано в попередньому пункті (це найкоректніший і надійніший спосіб пошуку, рекомендований до використання);
– використанням native API-функції операційної системи, взаємодія з яким здійснюється або через INT 2Eh (Windows NT, Windows 2000), або через машинну команду syscall (Windows XP). Це найбільш трудомісткий і якнайменше надійний спосіб зі всіх. Мало того, що native API-функції не документовані і схильні до постійних змін, так вони ще і дуже примітивні (тобто реалізують прості низькорівневі функції, непридатні до безпосереднього використання).

2.4.4 Деякі алгоритми впровадження

Розглянемо деякі приклади впровадження X-коду. Візьмемо такий випадок**:** механізми, що не викликають зміни адресації ні фізичного, ні віртуального образів (категорія А). Серед усіх варіантів візьмемо розміщення X-коду у вільному місці програми (інтеграція): у PE-заголовок; у хвостові частини секцій; у регулярні послідовності.

Розміщення у PE-заголовку

Типовий PE-заголовок разом з DOS заголовком і заглушкою займає порядку 300h байтів, а мінімальна кратність вирівнювання секцій складає 200h. Таким чином, між кінцем заголовка та початком першої секції практично завжди є приблизно 100h вільних байт, які можна використовувати для "виробничих цілей", розміщуючи тут або всю впроваджувану програму цілком, або тільки завантажувач X-коду, що прочитує своє продовження з дискового файлу або реєстру (рис. 2):

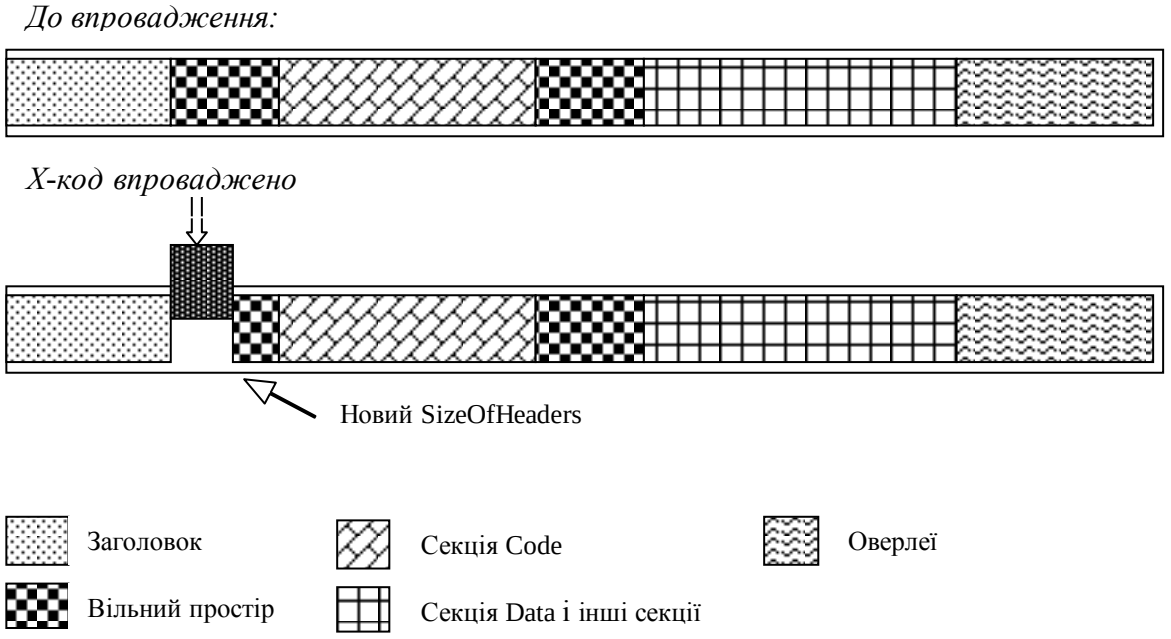


Рисунок 2 – Схема впровадження X-коду у заголовок файлу

Алгоритм впровадження у хвіст PE-заголовка з точку зору програмної реалізації може бути такий:

1. Прочитуємо PE-заголовок і приступаємо до його аналізу.
2. Якщо

   (SizeOfHeaders < FS.r_off) і (SizeOfHeaders + sizeof (X-code))< FS.r_off),

   то

   { а) збільшуємо SizeOfHeaders на sizeof (X-code) або ж просто підтягуємо його до raw offset першої секції;
   б) записуємо X-код на місце, що утворилося.
   }

   інакше

   { а) скануємо PE-заголовок на предмет пошуку безперервного ланцюжка нулів і, якщо він буде знайдений, впроваджуємо своє тіло, починаючи з байта 10h від її початку;
   б) впроваджуємо X-код в DOS-заглушку, не зберігаючи її старого вмісту.
   }
3. Якщо впровадження пройшло успішно, перехоплюємо управління на X-код.

Впровадження в хвіст секції

Операційна система Windows 9x вимагала, щоб фізичні адреси секцій вирівнювалися щонайменше на 200h байт, а Windows NT – на 002h. Тому між секціями практично завжди є деяка кількість вільного простору, що можна використати для реалізації захисту.

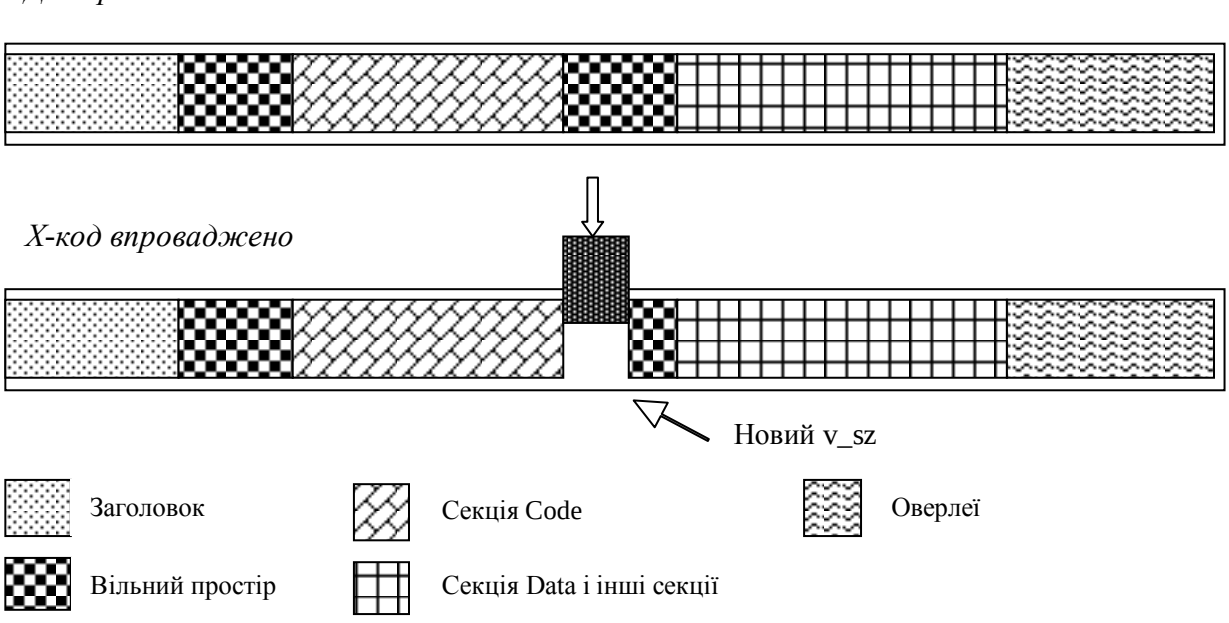


Рисунок 3 – Схема впровадження X-коду у хвіст секцій

Узагальнений алгоритм впровадження у хвіст секції може бути побудований таким чином.

1. Прочитуємо PE-заголовок.
2. Аналізуємо Section Table, порівнюючи фізичну довжину секцій з віртуальною.
3. Шукаємо секції, у яких r_sz>v_sz, і записуємо їх в кандидати на впровадження, попередньо переконавшись, що у хвості цих секцій містяться одні нулі.
4. Якщо r_sz-v_sz>=fa, то не чіпаємо таку секцію, оскільки, швидше за все, вона містить оверлей.
5. Якщо кворуму набрати не вдалося, шукаємо секції, у яких r_sz <= v_sz, і намагаємося знайти безперервний ланцюжок нулів у їх кінці.
6. З усіх кандидатів відбираємо секції з найбільшою кількістю вільного простору.
7. Знаходимо секцію, атрибути якої розташовують до впровадження, проаналізувавши певні прапори:

   (IMAGE_SCN_MEM_SHARED, IMAGE_SCN_MEM_DISCARDABLE скинуті, IMAGE_SCN_MEM_READ або IMAGE_SCN_MEM_EXECUTE встановлені, IMAGE_SCNLCNT_CODE або IMAGE_SCN_CNT_INITIALIZED_DATA встановлені),

   а якщо таких серед кандидатів, що залишилися, таких немає, то або коректуємо атрибути самостійно, або відмовляємося від впровадження.
8. Якщо v_sz!=0 і v_sz<r_sz, збільшуємо v_sz на sizeof(x-code) або підтягуємо до v_a наступної секції.

2.4.5 Запобіжні заходи при впровадженні Х-коду

При впровадженні захисних механізмів у захищувану програму слід дотримуватись певних запобіжних заходів. Наведемо деякі з них.

1. Нижче наведені випадки, коли від вставки Х-коду *слід відмовитися*:
   - якщо файл розташований на носії, захищеному від запису, або у нас не досить прав для його запису/читання (наприклад, файл заблокований іншим процесом), відмовляємося від впровадження;
   - якщо файл має атрибут, що забороняє модифікацію, або знімаємо цей атрибут, або відмовляємося від впровадження;
   - якщо Subsystem>2h або Subsystem<3h, відмовляємося впроваджувати;
   - якщо FA<200h або SA<1000 (FA – File Alignment, SA – Section Alignment), це, найімовірніше, драйвер, і в нього краще нічого не впроваджувати;
   - якщо файл імпортує одну або декілька функцій з hal.dll і/або ntoskrnl.exe, відмовляємося від впровадження;
   - якщо файл містить секцію INIT, він, можливо, є драйвером пристрою, і без особливої потреби краще сюди нічого не впроваджувати;
   - якщо DATA_DIRECTORY містить посилання на таблиці, що використовують фізичну адресацію, або відмовляємося від впровадження, або приймаємо на себе зобов'язання коректно «розпатрати» все ієрархію структур даних і скоректувати фізичні адреси;

– якщо ALIGN_UP(LS.r_off+LS.r_sz,A)>SizeOfFile (LS – Last Section, r_off – фізична адреса початку секції), файл, швидше за все, містить оверлей, і впроваджуватися в нього можна тільки методом 1;
– якщо фізичний розмір однієї або декількох секцій перевищує віртуальний на величину, більшу або рівну FA, і при цьому віртуальний розмір не рівний нулю, піддослідний файл містить оверлей, допускаючи тим самим використання впроваджень тільки типу 1.

2. Слід пам'ятати про необхідність *відновлення атрибутів файлу* і часу його створення, модифікації і останнього доступу (більшість розробників захисту обмежується лише часом модифікації, що демаскує факт впровадження).
3. Якщо поле контрольної суми не рівне нулю, слід або залишити такий файл у спокої, або розрахувати нову контрольну суму самостійно, наприклад шляхом виклику API-функції CheckSumMappedFile(). Онулювати контрольну суму неприпустимо, оскільки при активних сертифікатах безпеки операційна система просто відмовить файлу в завантаженні.
4. Останнім часом все частіше доводиться стикатися з виконуваними файлами величезного обсягу. Обробляти їх по частинах – нудно і складно. Завантажувати весь файл цілком – дуже повільно (чи дозволить Windows виділити таку кількість пам'яті!). Тому має сенс скористатися файлами, спроектованими в пам'ять (Memory Mapped File()), керованими функціями CreateFileMapping(), MapViewOfFile() і UnmapViewOfFile(). Це не тільки збільшує продуктивність, спрощує програмування, але і ліквідовує всі обмеження на гранично допустимий обсяг, який тепер може досягати 18 екзабайтів, що відповідає 11529215046068469 76 байтів). Як варіант, можна обмежити розмір оброблюваних файлів декількома мегабайтами, що є легко копійованими в оперативний буфер і зводять кількість «обв'язувального» коду до мінімуму.
5. Запобігання *повторного впровадження*. Х-код, що зберігає працездатність при багатократному впровадженні, *називають реєрентерабельним*. Реєнрентерабельність пред'являє жорсткі вимоги як до алгоритмів впровадження в цілому, так і до стратегії поведінки Х-коду. Класичним прикладом реєрентерабельності є Х-код, що дописує себе і кінець файлу і після здійснення всіх запланованих операцій повертає управління програмі-носію. При багатократному впровадженні Х-коду ніби «розмотуються», передаючи управління, як по естафеті, але якщо нитка управління заплутається, захист не буде здійснено. Припустимо, Х-код прив'язується до свого фізичного зсуву, відраховуючи його від кінця файлу. Тоді при багатократному впровадженні за цими адресами будуть розташовані зовсім інші комірки, що належать чужому Х-коду, і поведінка обох стане невизначеною. Отже, перед впровадженням у файл нереєнтерабельного Х-коду необхідно заздалегідь переконатися, що у файл не було впроваджене щось ще. На жаль, універсальних шляхів рішення не існує, тому доводиться вдаватися до різних евристичних прийомів, що розпізнають присутність чужерідного Х-коду за непрямими ознаками.

2.5 Захист програм шляхом обфускації

2.5.1 Поняття обфускації та його види

Обфускація ("obfuscation") – це один з методів захисту програмного засобу, який дозволяє ускладнити процес реверсивної інженерії початкового коду програми, що захищається. Суть процесу обфускації полягає в тому, щоб заплутати програмний код і усунути більшість логічних зв'язків у ньому, тобто трансформувати його так, щоб він був дуже важкий для вивчення і модифікації сторонніми особами (будь то зловмисники, або програмісти які збираються дізнатися унікальний алгоритм роботи програми, що захищається). Обфускація може застосовуватися не тільки для захисту програмних продуктів, вона має ширше застосування (наприклад, може бути використана творцями вірусів для захисту їх творінь і т. д.).

Помилковою є думка, що обфускація – це панацея від будь-якого роду зламу програм, чи то злам trial-версії комерційного продукту, чи то захист від переглядання використаного алгоритму. Обфускатори здебільшого розробляються з досить скромною метою – якомога сильніше утруднити зламнику шлях до розуміння коду і алгоритму, застосованого в програмі. Захист від зламу – це ніяк не задача обфускаторів, хоча деякі з них дозволяють в деякій мірі розв'язати і цю проблему. Можна виділити мінімум два аспекти, які дозволяють визначити, що робота обфускатора була виконана недаремно:

- по-перше, час, витрачений на розуміння коду зловмисником перевищує час, протягом якого актуальність алгоритму залишається значущою;
- по-друге, вартість деобфускації перевищує вартість самого продукту.

Розглядаючи процес обфускації з точки зору захисту ПЗ і трансформації початкового коду програми PR1 у вихідний код PR2 без можливості повернення до його початкового вигляду (трансформація в “один бік”) можна сказати, що процес трансформації буде вважатися процесом обфускації якщо будуть задовольнятися такі ***вимоги***:

1. Код програми PR2 в результаті трансформації буде суттєво відрізнятися від коду PR1, але при цьому буде дієздатним і виконуватиме ті самі функції.
2. Вивчення принципу роботи, тобто процес реверсивної інженерії програми PR2 буде більш складним, трудомістким і займатиме більше часу, ніж PR1.
3. При кожному процесі трансформації одного і того ж коду програми “PR1” код програми “PR2” буде різним.
4. Створення програми, яка детрансформує програму “PR2” в її найбільш схожий початковий вигляд, буде неефективним.

Процес обфускації може бути здійснений над будь-яким із перерахованих видів представлення програмного коду, тому прийнято виділяти такі ***рівні процесу обфускації***:

Низький рівень, коли процес обфускації здійснюється над асемблерним кодом програми, або навіть безпосередньо над двійковим файлом програми, який зберігає машинний код.

Високий рівень, коли процес обфускації здійснюється над вихідним кодом програми, написаної мовою високого рівня.

Здійснення обфускації на низькому рівні вважається менш комплексним процесом, але при цьому його важче реалізувати по ряду причин. Одна з таких причин полягає в тому, що мають бути враховані особливості роботи більшості процесорів, оскільки спосіб обфускації, прийнятний на одній архітектурі, може виявитись неприйнятним на іншій. Крім того, на сьогоднішній день процес низькорівневої обфускації досліджений мало і не встиг отримати широкої популярності. Більшість існуючих алгоритмів і методів обфускації можуть бути використані для здійснення процесу обфускації як на низькому, так і на високому рівнях.

Слід зауважити, що може бути неефективно піддавати обфускації весь код програми (наприклад, через те, що в результаті може знизитись час виконання програми), в таких випадках доцільно здійснювати обфускацію тільки на найбільш важливих ділянках коду.

Зауваження. Оскільки код, який отримуємо після здійснення обфускації над однією і тією ж програмою, різний, то *процес обфускації можна використовувати для швидкої локалізації порушників авторських прав* (тобто тих покупців, які будуть займатись нелегальним розповсюдженням куплених копій програм). Для цього визначають контрольну суму кожної копії програми, яка пройшла обфускацію і записують її разом з інформацією про покупця у відповідну базу даних. Після цього для визначення порушника достатньо буде, визначивши контрольну суму нелегальної копії програми, порівняти її з інформацією, що зберігається в базі даних.

Обфускація буває декількох видів:

- лексична обфускація (символьна обфускація);
- обфускація даних;
- обфускація графа потоку керування.

2.5.2 Лексична обфускація

Лексична обфускація полягає в форматуванні коду програми, зміні його структури таким чином, щоб він став нечитабельним, менш інформативним і важчим для дослідження дизасемблерами і декомпіляторами.

Обфускація такого виду включає в себе певні складові.

1. *Видалення необов'язкових конструкцій* мови програмування:
   - видалення усіх коментарів в коді або зміна їх на дезінформуючі;
   - видалення різноманітних пробілів, відступів, які зазвичай використовуються для кращого візуального сприйняття коду програми.
2. *Додавання різноманітних (сміттєвих) операцій*. Це досягається шляхом використання непрозорих змінних і предикатів, надлишкового коду, мертвого коду та деяких інших заплутуючих виразів і фрагментів коду.

3. *Зміна розміщення блоків* (функцій, процедур) програми так, щоб це ні в якому разі не вплинуло на її дієздатність. Цей вид лексичної обфускації може бути представлений за допомогою перемішування інструкцій.
4. *Символьна обфускація* – заміна імен ідентифікаторів (імен змінних, масивів, структур, функцій, процедур і т. д.) на самовільні довгі набори символів, які важко сприймати людині. Символьна обфускація найбільш легко реалізується.

Зміну глобальних імен ідентифікаторів слід проводити в кожній одиниці трансляції (один файл вихідного коду) так, щоб вони мали однакові імена (в іншому випадку програма, яка захищається, може стати не функціональною). Також слід враховувати специфічні ідентифікатори, прийняті в тій мові програмування, на якій написана захищувана програма. Імена таких ідентифікаторів краще не змінювати.

Обфускатори, які використовують символьну обфускацію, називають обфускаторами першого покоління. Вони застосовують такі методи.

1) *Перейменування методів, змінних і т.д. в набір безглуздих символів.* Наприклад, був метод класу GetPassword(), після обфускації даний метод матиме ім'я KJHS92DSLKaf(). Поза сумнівом, така нісенітниця відкине зламника на пару годин від явної мети. Проте існує одна проблема – багато декомпіляторів, зустрічаючи на своєму шляху подібного роду імена, замінюють їх на більш читабельні (method_1, method_2), тим самим зводячи всю роботу обфускатора нанівець.

2) *Перейменування найменувань змінних і методів у коротші.* Проходячи по всіх класах, методах, параметрах, вони замінюють імена на їх порядкові номери. Наприклад, був метод GetConnectionString(), а став _0(). До того ж, у зв'язку з тим, що символьної інформації тепер в збірці зберігатиметься менше, розмір самої збірки, відповідно, теж значно скоротиться. Подібні обфускатори можна також використовувати як оптимізуючі компілятори. Також подібне рішення добре тим, що існує вірогідність того, що одне і те ж ім'я буде використано для іменування класу, методів класу (що, наприклад, відрізняються тільки типом повернення). Це дозволить також заблокувати роботу деяких дизасемблерів.

3) *Використання для імен змінних нечитабельних символів.* Частина обфускаторів вставляють в імена нечитабельні символи, наприклад, символи японської мови. Хоча .Net і працює з кодуванням UTF8, не всі декомпілятори адекватно обробляють його символи. Деякі замінюють імена з такими символами на зрозуміліші, деякі проставляють замість незрозумілих символів їх код, деякі просто відмовляються працювати з даними символами.

4) *Використання ключових слів мов програмування.* Цей вид символьної обфускації дозволить захиститися від найпримітивніших декомпіляторів, які побачивши в якості імені зарезервоване слово вважають, що збірка не валідна і відмовляються з нею працювати.

5) *Використання імен, що міняють сенс*. Тут швидше діє психологічний чинник. Припустимо, клас SecurityInformation з методом GetInformation(), а став класом Car з методом Wash(). Звичайно, це може заплутати недосвідченого зламника, але процесу декомпіляції ніяк не пошкодить.

Наприклад, до лексичної обфускації програмний код мав такий вигляд (мова PERL):

```
my $filter;
if (@pod)
{  my ($buffd,$buffer)=File::Temp::tempfile(UNLINK=>1);
   print  $buffd "";
   print  $buffd @pod or die "";
   print  $buffd
   close $buffd  or die "";
   @found = $buffer;
   $filter = 1;
} exit;
sub is_taINT ed
{ my $arg = shift;
   my $nada = substr($arg, 0, 0);  # zero-length
   local $@;  # preserve caller's version
   eval { eval "#" };
   return length($@)!= 0;
}
sub am_taint _checking
{        my($k,$v)= each %ENV;
         return is_taINT ed($v);
}
```

Після лексичної обфускації програма набула вигляду:

```
sub z109276e1f2{(my $z4fe8df46b1 = shift(@_));
(my $zf6f94df7a7=substr($z4fe8df46b1,(0x1eb9+765-0x21b6),
(0x0849+1465-0x0e02)));local $@;eval {eval(""));};
return((length($@)!=(0x26d2+59-0x270d)));}   my($z9e5935eea4);
if(@z6a703c020a){(my($z5a5fa8125d,$zcc158ad3e0)=
File::Temp::tempfile("",(0x196a+130-0x19eb)));
print ($z5a5fa8125d"");(print($z5a5fa8125d @z6a703c020a) or
die((((".$zc158ad3e0)."\x3a\x20").$!)));
print ($z5a5fa8125d "");(close($z5a5fa8125d) or
die((("")));(@z8374cc586e=$zcc158ad3e0);
($z9e5935eea4=(0x1209+1039-0x1617));}exit;
sub z021c43d5f3{(my($z0f1649f7b5,$z9e1f91fa38)=
each(%ENV)); return(z109276e1f2($z9e1f91fa38));}
```

Дослідження дозволило зробити певні висновки про можливість і доцільність використання даного методу обфускації (заміна імен ідентифікаторів) для захисту програми. Кількісно це виражається таким чином: від 15 до 55% слів програми (що приблизно відповідає тій же частці коду програми) піддається зміні. Проте висновки цим твердженням не обмежуються: цю частку програми можна використовувати і для обфускації, і для внесення в код цифрових підписів, водяних знаків і т. д.

2.5.3 Обфускація даних

До складніших обфускаторів, які дають більше гарантій того, що наш код не зможуть зрозуміти зловмисники, відносяться обфускатори другого порядку. Вони застосовують методи обфускації даних і графа потоку управління.

Обфускація даних пов'язана із трансформацією структур даних. Вона вважається більш складною, і є найбільш сучасною й часто використовуваною. Її прийнято ділити на три основні групи, які описані нижче.

Обфускація зберігання

Обфускація зберігання полягає в трансформації сховищ даних, а також самих типів даних (наприклад, створення й використання незвичайних типів даних, зміна подання існуючих і т. д.). Нижче наведені основні методи, що дозволяють здійснити таку обфускацію.

1. *Зміна інтерпретації даних певного типу*. Як відомо, збереження даних у сховищах (змінних, масивах і т.д.) певного типу (ціле число, символ) у процесі роботи програми є дуже поширеним явищем. Наприклад, для переміщення по елементах масиву дуже часто використовують змінну типу "ціле число", що виступає в ролі індексу. Використання в цьому випадку змінних іншого типу можливо, але це буде не тривіально й може бути менш ефективно. Інтерпретація комбінацій розрядів даних, що містяться у сховищі, здійснюється залежно від його типу. Так, наприклад, можна сказати, що 16-розрядна змінна цілого типу, яка містить 0000000000001100, представляє ціле число 12, але ж дані в такій змінній можна інтерпретувати по-різному (не обов'язково як 12, а, наприклад, як 1100 і т. д.).
2. *Зміна терміну використання сховищ даних*, наприклад, перехід від локального їх використання до глобального і навпаки. Як приклад можна навести дві різні функції:

```
void func1 { ... int  a ...  }
void func2 { ... int  b ...  }
```

   Якщо ці дві функції не можуть виконуватися в процесі програми одночасно, виходить, для них може бути створена одна глобальна змінна, яка буде заміщати змінні *a* і *b*, наприклад:

```
int  AB = 0 ;
void func1 { ... int  AB ...  }
void func2 { ... int  AB ...  }
```

3. *Перетворення статичних (незмінних) даних у процедурні*. Більшість програм у процесі роботи виводять різну інформацію, що найчастіше в коді програми представляється у вигляді статичних даних, таких як рядки, які дозволяють візуально орієнтуватися в її коді й визначати виконувані операції. Такі рядки також бажано піддати обфускації, це можна зробити, просто записуючи кожен символ рядка, використовуючи його ASCII-код, наприклад символ "A" можна записати як шістнадцяткове число "0x41", але такий метод занадто простий. Ефектив-

нішим є метод, коли в код програми в процесі здійснення обфускації додається функція, що генерує необхідний рядок відповідно до переданих їй аргументів, після чого рядки в цьому коді видаляються, і на їх місце записується виклик цієї функції з відповідними аргументами. Наприклад, фрагмент коду (мова PERL):

```
print  "LOL\n" ;
$var = "101" ;
```

після обфускації, буде схожий на такий:

```
sub string {
my ($i) = @_ ;
my $k = 0 ;
$str = "" ;
while (1)
{  l1: if ($i == 1) {$str .= "L";$k = 0;goto l7;}
   l2: if ($i == 4) {$str .= "S";$k = 0;goto l7;}
   l3: if ($i == 3) {$str .= "1";$k = -1;goto l7;}
   l4: if ($i == 4) {$str .= "m";$k = 3;goto l7;}
   l5: return $str;
   l6: if   ($k == 0) {$str .= "1";$k += 2;goto l5;}
       else {$str .= "L";$k -= 1;goto l5;}
   l7: if ($k < 1) {$str .= "0";$k++;goto l6;}
}
}
...
print  string(1)."\n" ;
$var = string(3) ;
```

Крім того, так само можна вчинити і з числовими константами, які можуть бути також трансформовані, наприклад, число 1 можна представити як *(a+1-b)*, де *a = b*.

4. *Розділення змінних*. Змінні фіксованого діапазону можуть бути розділені на дві й більше змінних. Для цього змінну *V*, що має тип *x* розділяють на *k* змінних $v_1,...,v_k$ типу *y* тобто $V=v_1,...,v_k$. Потім створюється набір функцій що дозволяють витягати змінну типу *x* зі змінних типу *y* і записувати змінну типу *x* у змінні типу *y*. Як приклад розділення змінних можна розглянути спосіб подання однієї змінної *B* логічного типу (*boolean*) двома змінними b_1 і b_2 типу короткого цілого (*short*), значення яких буде інтерпретуватися таким чином:

B	b_1	b_2
false	0	0
true	0	1
true	1	0
false	1	1

Тоді фрагмент коду (мова C++):

```
bool B ;
B = false ;
if (B) {  ...  }
```

буде перетворено у такий:

```
short b1, b2 ;
b1 = b2 = 1 ;  // або b1 = b2 = 0
if (!(b1 & b2)) {  ...  }
```

5. *Зміна представлення* (або кодування). Наприклад, цілочисельну змінну *i* у нижче представленому фрагменті коду (мова С):

```
...
int  i=1;
...
while (i<1000)
{
    ... A[i] ...
    i++ ;
}
```

можна замінити, виразом $i=c_1 \cdot i+c_2$, де c_1, c_2 – константи. В результаті наведений код зміниться й буде складніший для сприйняття:

```
...
int  i=11;
int  c1=8, c2=3 ;
...
while (i<8003)
{ ... A[(i-3)/8] ...
  i+=8 ;
}
```

Обфускація з'єднання

Один з важливих етапів у процесі реверсивної інженерії програм базується на вивченні структур даних. Тому важливо намагатися у процесі обфускації ускладнити представлення використовуваних програмою структур даних. Наприклад, при використанні обфускації з'єднання це досягається об'єднанням незалежних даних чи розділенням залежних. Далі наведені основні методи, що дозволяють здійснити таку обфускацію.

1. *Об'єднання змінних*. Дві або більше змінних $v_1,...,v_k$ можуть бути об'єднані в одну змінну *V*, якщо їх спільний розмір не перевищує розміру змінної *V*. Наприклад, розглянемо простий приклад об'єднання двох цілочисельних змінних *X* та *Y* (розміром 16 біт) в одну цілочисельну змінну *Z* (розміром 32 біта). Для цього скористаємося формулою

$$Z(X,Y)=2^{16} \cdot Y+X,$$

яка дозволить, нехтуючи додаванням, визначати значення *Y*. Нехай

$$X=12,\ Y=4.$$

Тоді

$$Z=65536 \cdot 4+12=262156.$$

Тепер, знаючи *Z*, для знаходження *Y* можна знайти:

$$262156/65536=4.000183105 \approx 4.$$

При здійсненні арифметичних операцій над значеннями змінних X, Y, що зберігаються в Z, потрібно враховувати вище наведену формулу:

$$Z(X+n,Y)=2^{16} \cdot Y+(X+n)=Z(X,Y)+n\ ;$$

$$Z(X,Y-n)=2^{16} \cdot (Y-n)+X=2^{16} \cdot Y-2^{16} \cdot n+X=Z(X,Y)-2^{16} \cdot n;$$

В результаті код до обфускації (мова С):

```
short X=12, Y=4 ;
X+=5 ;
```

трансформується в такий:

```
int  Z=262156 ;
Z+=5 ;
```

2. *Реструктурування масивів* полягає в заплутуванні структури масивів, шляхом *розділення* одного масиву на декілька підмасивів, *об'єднання* декількох масивів в один, *згортання* масиву (збільшуючи його розмірність) і навпаки, *розгортання* (зменшуючи його розмірність). Наприклад, масив *A* можна розділити на декілька підмасивів *A1* і *A2*, при цьому масив *A1* буде містити парні, а *A2* – непарні позиції елементів масиву *A*. Тому такий фрагмент коду (мова С):

```
int  A[] = {a, b, c, d, e, f} ;
i = 3 ;
A[i] = ... ;
```

можна замінити на:

```
int A1 = {2, 4, 0} ;
int A2 = {1, 3, 5} ;
i = 3 ;
if (($i % 2) == 0) { $A1[$i / 2] = ... ; }
else { $A2[$i / 2] = ... ; }
```

Під *згортанням* масиву розуміється створення з одномірного масиву двовимірного. Наприклад, одномірний масив *A* розмірністю, можна замінити двовимірним масивом *B* розмірністю 2×3, після чого код

```
int  A[] = {1, 2, 3, 4, 5, 6} ;
for (int  i = 0 ; i < 6 ; i++)
{      A[i] = A[i] + 1 ;
       print f("%d\n", A[i]) ;
}
```

можна змінити на такий:

```
int  A[2][3] = { {1,2,3}, {4,5,6} };
for (int  i = 0 ; i < 2 ; i++)
{      for (int  ii = 0 ; ii < 3 ; ii++)
       {      A[i][ii] = A[i][ii] + 1 ;
              print f("%d\n", A[i][ii]) ;
       }
}
```

3. *Зміна ієрархій успадкування класів* здійснюється шляхом ускладнення ієрархії успадкування за допомогою створення додаткових класів або використання помилкового ділення класів.

Обфускація переупорядкування

Полягає в зміні послідовності оголошення змінних, внутрішнього розташування сховищ даних, а також переупорядкування методів, масивів (використання нетривіального подання багатомірних масивів), певних полів у структурах і т.д.

Модифікація області дії змінних

1. *Локалізація змінних у базовому блоці***.** Це перетворення локалізує використання змінних одним базовим блоком. Для кожного базового блоку функції, що заплутується, створюється свій набір змінних. Всі використання локальних і глобальному змінних у вихідному базовому блоці заміняються на використання відповідних нових змінних. Щоб забезпечити правильну роботу програми між базовими блоками вставляються так звані сполучні (connective) базові блоки, завдання яких

скопіювати вихідні змінні попереднього базового блоку у вхідні змінні наступного базового блоку. Застосування такого заплутуючого перетворення приводить до появи у функції великої кількості нових змінних, які, однак, використовуються тільки в одному-двох базових блоках, що заплутує людину, яка аналізує програму.

2. *Розширення області дії змінних*. Дане перетворення за змістом обернене попередньому. Це перетворення намагається збільшити час життя змінних настільки, наскільки можна. Наприклад, виносячи блокову змінну на рівень функції або виносячи локальну змінну на статичний рівень, розширюється область дії змінної й ускладнюється аналіз програми. Тут використовується те, що глобальні методи аналізу (тобто, методи, що працюють над однією функцією в цілому) добре обробляють локальні змінні, але для роботи зі статичними змінними потрібні більше складні методи міжпроцедурного аналізу.

2.5.4 Обфускація графа потоку управління

Найскладнішою в плані реалізації, але найстійкішою до спроб зламу є обфускація графа потоку управління. На даний час існує декілька подібних обфускаторів. Обфускатори цього типу застосовують такі методи.

Перетворення обчислень. Даний метод полягає у вставці в алгоритми помилкових умов. Наприклад, перед входом в цикл можна вставити помилкову умову на зразок такої: *if (3==2) {} else {}*; при цьому дугу, яка відповідає *true,* кинути у середину циклу, а дугу, яка відповідає *false,* кинути на початок циклу. Але, швидше за все, цей прийом не зіб'є з пантелику більшість декомпіляторів. Можна зробити цикл з двома і більш входами, наприклад, перед початком циклу вставити умову, яка завжди істинна, істинну дугу кинути на початок циклу, помилкову – на базовий блок, який поміщається в кінець фізичної послідовності, що представляє даний граф, а вихідну дугу з цього блоку на початок циклу. Таким чином можна одержати цикл з декількома входами. Проводячи маніпуляції з графом виключних ситуацій, також можна добитися непоганих результатів. Наприклад, якщо в графі потоку управління виключення відсутні, тим не менш можна наділити цей граф деякими виключеннями.

Видалення або додавання абстракцій коду. Видалення абстракцій коду дозволяє крім рішення задачі обфускації також оптимізувати роботу програми. Наприклад, обфускатор може замінити виклик якої-небудь функції безпосередньо тілом функції, або навпаки одну функцію замінити на декілька маленьких функцій.

Перемішування випадковим чином лінійних ділянок.

Дані методи більш детально будуть розглядатися у подальших підрозділах. Хоча вони взагалі-то і відносяться до методів обфускації, ми надалі виділимо їх в окремі групи.

2.5.4 Аналіз ефективності роботи обфускаторів

Обфускатор, як правило, аналізує метадані, але не всі члени збірки він може обфускувати. Наприклад, обфускатору не варто замінювати імена конструкторів типу – це може призвести до небажаних наслідків. Хоча в деяких випадках це буває можливо. Також іноді неможлива обфускація віртуальних або абстрактних методів.

Припустимо, список членів збірки для обфускації готовий. Обфускатор привласнює їм нові імена, базуючись на певному алгоритмі. Одні обфускатори привласнюють імена такої ж довжини що і були але позбавлені колишнього сенсу, інші базуються на нумерації всіх членів і обфускують їх у відповідності з номером члена збірки, треті – базуючись на токенах (token) члена збірки – унікальний ідентифікатор члена збірки, четверті прагнуть мінімізувати довжину імені і частіше використовувати одне і те ж ім'я серед членів збірки – це дає максимальний ефект обфускації (наприклад, якщо у розгляданого класу всі методи і поля названі як “1”). Після цього проводиться запис даних назад в збірку або генерація нової збірки. Деякі обфускатори здійснюють ще й оптимізацію – видалення непотрібної інформації зі збірки (дебаг-інформація, невживані методи, поля, класи). Фактично збірка після обфускації відрізняється від початкової однією деталлю – модифікованим розділом рядків (String Heap або #Strings) – одним з п'яти розділів метаданих, – це підсумок символьної обфускації, коли обфускатор не модифікує тіла методів.

Отже, у обфускаторів можна виділити і позитивні, і негативні риси.

До п*озитивних рис* роботи обфускаторів відносять такі:

- обфускатори роблять дизасембльований код важким для вивчення, перетворюючи *IsLicensed()* у *x();*
- деякі обфускатори використовують баги ILDASM для захисту від дизасемблювання в ньому (обфускатор Salamander);
- деякі обфускатори навіть конвертують код у native-код, роблячи марним дизасемблювання (обфускатор Salamander);
- деякі обфускатори шифрують і пакують exe-файл і разом з ним referenced збірки в один exe-файл, так що розмір програми може зменшитися 2-4 рази і не піддаватиметься дизасемблюванню (обфускатор Thinstall).

Негативні риси роботи обфускаторів такі:

- продукт все одно залишається таким, що дизасемблюється. Зібрати збірку після дизасемблювання не складе труднощів для досвідчених хакерів;
- обфускований код все одно лишається в якійсь мірі доступним для читання і розуміння в порівнянні з асемблер ним;
- захист обфускаторів, які використовують шифрування символьної частини метаданих, рядкових і бінарних ресурсів заважає користувачам продукту налагоджувати і тестувати свої продукти. Окрім цього – це ризик, оскільки деякі символьні дані можуть бути необхідними для подальшого використання.

Існує багато методів визначення ефективності застосування того або іншого процесу обфускації, до конкретного програмного коду. Ці методи прийнято розділяти на дві групи: аналітичні і емпіричні.

Аналітичні методи визначення ефективності обфускації ґрунтуються на трьох величинах, які характеризують наскільки ефективний той або інший процес обфускації:

- *стійкість* – вказує на ступінь складності здійснення реверсивної інженерії над кодом що пройшов процес обфускації;
- *еластичність* – вказує на те, наскільки добре даний процес обфускації захистить програмний код від застосування деобфускаторів;
- *вартість перетворення* – дозволяє оцінити, наскільки більше потрібно системних ресурсів для виконання коду, що пройшов процес обфускації, ніж для виконання оригінального коду програми.

Аналітичні методи найефективніше застосовувати при порівнянні різних алгоритмів обфускації, але при цьому вони не можуть дати абсолютної відповіді на питання, наскільки ефективно застосування того або іншого алгоритму, саме до даного програмного коду.

Емпіричні ж методи можуть дати прийнятну відповідь на таке питання, оскільки вони ґрунтуються на статистичних даних одержуваних в результаті досліджень. Для проведення одного з таких досліджень потрібні група людей (добре знайомих з реверсивною інженерією), фрагмент коду програми, що захищається, і набір різних алгоритмів обфускації.

Результати такого дослідження включатимуть мінімальну кількість часу, який було потрібно групі людей, для того, щоб вивчити кожен фрагмент коду, що пройшов один з алгоритмів обфускації.

На сучасному ринку програмних засобів вже існує ряд програм-обфускаторів, кожен з яких має свої особливості.

RemoteSoft Slamander Obfuscator. Цей обфускатор має навігатор, що дозволяє змінювати не тільки метадані, але і структуру PE-файлу, аналізувати ресурси, бінарні дані і може використовуватися як редактор початкових кодів. В ньому свій дизасемблер, але дизасемблює він відразу весь клас. Даний обфускатор реалізовує символьну обфускацію. Також даний засіб пропонує Protector, який перетворює методи на native code, що не дизасемблюються, але залежні від поточної версії .Net Framework. Сам обфускатор написаний у вигляді native code.

9Rays.Net ILObfuscator (ILO). Цей обфускатор має GUI, що полегшує завдання роботи зі збірками – навігатор по збірках, дизасемблер, система роботи з проектом обфускації, в якій можна переглядати і екстрагувати ресурси, що містяться в збірці. Поставляється разом з SDK, що дозволяє створити свою власну систему обфускації, має колекції замін і виключень, гнучкі можливості по заплутуванню, має набір варіантів найменувань обфускованих класів, включає функції оптимізації збірки після обфускації.

Wise Owl Demeanor Цей обфускатор також багато чого вміє і може: видаляє непотрібну інформацію з метаданих, шифрує символьні змінні, працює з мультимодульними збірками і інтегрується з VS.Net.

2.6 Способи реалізації ускладнення логіки

Як один з методів захисту від зворотної інженерії застосовується маскування програм. Кажучи неформально, *маскування програми* – це таке перетворення її тексту, яке повністю зберігає функціональність, але робить розуміння, зворотну інженерію і модифікацію тексту програми завданням неприйнятно високій вартості. Можна сказати, що маскування програм – це обфускація графа потоку керування програмою.

Об'єктами маскування є тексти реальних програм, що складаються з сотень функцій по декілька сотень рядків кожна. Замасковані програми повинні вкладатися в обмеження обчислювальної системи, що не може не відбитися на використовуваних методах маскування.

Загальна ідея цих методів може бути охарактеризована таким чином.

По-перше, значно збільшити складність графа потоку управління, але так, щоб всі дуги графа потоку управління, внесені при маскуванні, проходилися при виконанні програми.

По-друге, збільшити складність потоків даних маскованої функції, "наклавши" на неї програму, яка свідомо не впливає на оточення маскованої функції і, як наслідок, не змінює роботи програми. "Холоста" функція будується як з фрагментів маскованої функції, семантичні властивості яких свідомо відомі, так і з фрагментів, узятих з бібліотеки маскуючого транслятора. Для утруднення завдання виявлення "холостої" частини замаскованої функції використовуються мовні конструкції, які важко піддаються аналізу (наприклад, покажчики) і математичні тотожності.

Процес маскування можна розбити на декілька етапів:

1. *Збільшення розміру графа* потоку управління функції. На цьому етапі виконуються різні перетворення, які змінюють структуру циклів в тілі функції, а також може здійснюватись клонування базових блоків.
2. *Руйнування структури графа потоку* управління функції, коли в граф потоку управління вноситься певна кількість нових дуг. При цьому існуючі базові блоки можуть виявитися розбитими на декілька менших, а у графі потоку управління можуть з'явитися нові, поки що порожні, базові блоки. Мета цього етапу – підготувати місце, в яке надалі буде внесений неістотний код.
3. *Генерація неістотного коду.* На цьому етапі порожні базові блоки графа потоку управління заповнюються інструкціями, що не впливають на результат роботи програми. Неістотна, "холоста" частина поки ніяк не стикається з основною, функціональною частиною програми.
4. *Зчеплення "холостої" і основної програм*. Для цього використовуються як властивості програм, що важко аналізуються (наприклад, покажчики), так і різноманітні математичні тотожності і нерівності.

Розглянемо деякі маскуючи перетворення, які істотно утрудняють статичний аналіз програм.

2.6.1 Маніпулювання функціями

Відкрита вставка функцій (*function inlining*) полягає в тому, що тіло функції підставляється в місце її виклику. Це перетворення є стандартним для оптимізуючих компіляторів. Також воно є однобічним, тобто по перетвореній програмі автоматично відновити вставлені функції неможливо.

Винесення групи операторів (*function outlining*). Дане перетворення є оберненим до попереднього і добре доповнює його. Деяка група операторів вихідної програми виділяється в окрему функцію. При необхідності створюються формальні параметри. Перетворення може бути легко повернене компілятором, тобто він (як було сказано вище) може підставляти тіла функцій у місця їхнього виклику.

Усунення бібліотечних викликів (*eliminating library calls*). Більшість програм мовою Java широко використовують стандартні бібліотеки. Оскільки семантика бібліотечних функцій добре відома, такі виклики можуть дати корисну інформацію при зворотній інженерії програм. Проблема збільшується ще й тим, що посилання на класи бібліотеки Java завжди є іменами, і ці імена не можуть бути перекручені. У багатьох випадках можна обійти цю обставину, просто використовуючи в програмі власні версії стандартних бібліотек. Таке перетворення не змінить істотно час виконання програми, але значно збільшить її розмір.

Переплетення функції (*function interleaving*). Ідея цього перетворення в тому, що дві або більше функцій поєднуються в одну. Списки параметрів вихідних функцій поєднуються і до них додається ще один параметр, який дозволяє визначити, яка функція в дійсності виконується. Технічно це може бути зроблено перенумеруванням усіх базових блоків і введенням нової змінної, наприклад state, що містить номер поточного базового блоку. Заплутана функція замість операторів if, for і т.д. буде містити оператор switch, розташований всередині нескінченного циклу.

Клонування функцій (*function cloning*). При зворотній інженерії функцій у першу чергу вивчається її сигнатура, а також те, як ця функція використовується, у яких місцях програми, з якими параметрами і як викликається. Аналіз контексту використання функції можна утруднити, якщо кожен виклик деякої функції буде виглядати як виклик якоїсь іншої, щоразу нової функції. Може бути створено декілька клонів і до кожного з них буде застосований різний набір заплутуючих перетворень.

2.6.2 Використання непрозорих предикатів

Основною проблемою при проектуванні перетворень, що заплутують граф потоку керування, є те, як зробити їх не тільки дешевими, але й стійкими. Для забезпечення стійкості більшість перетворень ґрунтується на введенні *непрозорих змінних* і *непрозорих предикатів* (*opaque predicates*). Сила таких перетворень залежить від складності аналізу непрозорих предикатів і змінних. Змінна є *непрозорою*, якщо існує деяка властивість щодо цієї змінної, яка відома в момент заплутування програми, але важко від-

новлюється після того, як заплутування завершене. Аналогічно, предикат *P* називається *непрозорим*, якщо його значення відоме в момент заплутування програми, але важко відновлюване після його завершення.

Непрозорі предикати можуть бути трьох видів:

P^F – предикат, що завжди має значення "false",

P^T – предикат, що завжди має значення "true",

$P^?$ – предикат, що може приймати те або інше значення, і в момент заплутування поточне значення предиката відомо.

Наприклад, на рисунку 4 (1) один блок програми *A* розбитий (трансформований) на декілька незалежних блоків *A1* і *A2*, які з'єднані за допомогою непрозорого предиката P^T, що повертає завжди значення *TRUE*. У результаті такої трансформації виникає уявлення, що блок *A2* виконується не завжди, і тому для визначення умови виконання блоку *A2* зловмиснику треба буде знайти значення, яке повертає непрозорий предикат P^T.

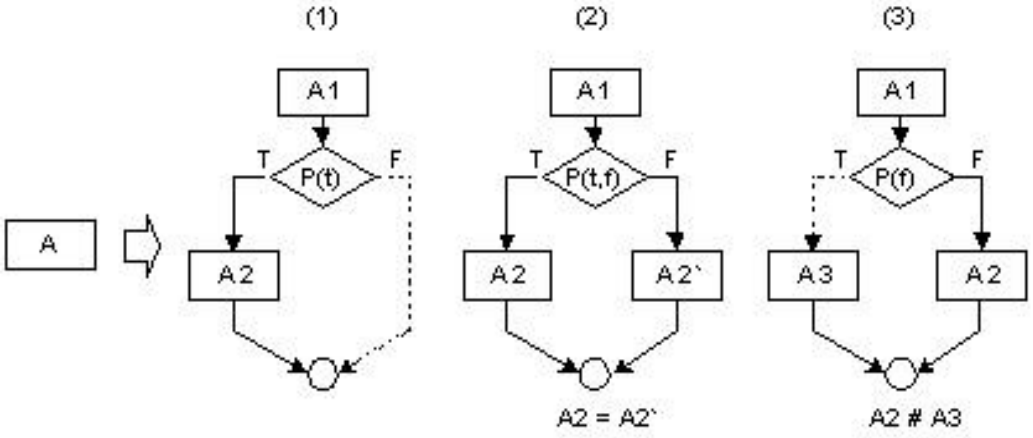


Рисунок 4 – Використання простих непрозорих предикатів

На рисунку 4 (2), крім блоку *A2* використовується ще один блок *A2,* над яким був здійснений процес обфускації (позначимо його як *A2′*), тобто ці два блоки *A2* і *A2′* мають різний код, але виконують однакові функції, отже, можна стверджувати, що *A2=A2′*, і тому не важливо, який із цих блоків буде виконаний у процесі роботи програми. З цього випливає, що для з'єднання блоку *A1* з *A2* і *A2′* ефективним буде використати непрозорий предикат, що може повернути кожне зі значень, а саме P^T або P^F.

На рисунку 4 (3) використовується новий блок *A3*, який містить довільний набір будь-яких операцій (недосяжний код). Це може спантеличити зловмисника, оскільки самі блоки *A1* і *A2* з'єднані за допомогою засобів непрозорого предиката P^F, що завжди повертає значення FALSE.

Ефективність обфускації керування в основному залежить від використаних непрозорих предикатів: складних для вивчення або простих та гнучких у використанні. Але рівною мірою також не менш важливу роль має термін їх виконання і кількість виконуваних операцій. Крім того, предикат не сильно повинен відрізнятися від тих функцій, які виконує сама програма, і не повинен містити надмірну кількість обчислень, у протилежному ж випадку зловмисник зможе відразу його виявити.

Використання різних способів доступу до елементів масиву. Нехай у програмі створено масив *a*, що ініціалізується заздалегідь відомими значеннями, а далі в програму додаються змінні *i* і *j*, в яких зберігатимуться індекси елементів цього масиву. Тепер непрозорі предикати можуть мати вигляд: *a[i]==a[j]*. Якщо до того ж змінні *i* й *j* у програмі змінюються, то існуючі зараз методи статичного аналізу дозволять тільки визначити, що і *i*, і *j* вказують на будь-який елемент масиву *a*.

Як простий спосіб можна використати розміщення всіх локальних змінних одного типу в масиві. У тексті функції замість імені змінної тепер використовуватиметься індекс масиву. Навіть у випадку, коли індекс завжди є літеральною константою, буде досить, щоб збити з пантелику прості алгоритми аналізу аліасів, які розглядають масив як єдине ціле.

У момент маскування програми відомо, які елементи масиву зайняті істотними, а які – неістотними змінними. Більш того, розподіл неістотних змінних по масиву може вибиратися довільним чином. Це може використовуватися для побудови залежностей між даними. Як приклад застосування запропонованого методу маскування візьмемо невелику програму, яка вирішує задачу про 8 ферзів. Далі для прикладу наведено текст цієї програми і схематичний вигляд графа потоку управління однієї з її процедур *queens()* до здійснення процесу обфускації (рис. 5) і після нього (рис. 6, 7).

```
static int  up[15], down[15],
           rows[8], x[8];
static void print (void)
{   int  k;
    for (k=0; k<8; k++)
      print f("%c\n", x[k]+'1');
}
static void queens (INT  c)
{ int  r;
  for (r=0; r<8; r++)
    if (rows[r]&&up[r-c+7]&&down[r+c])
    {  rows[r]=up[r-c+7]=down[r+c]=0;
       x[c] = r;
       if (c==7) print ();
       else  queens(c+1);
       rows[r]=up[r-c+7]=down[r+c]=1;
    }
}
int  main(void)
{   int  i;
    for (i=0; i<15; i++)
        up[i]=down[i]=1;
    for (i=0; i<8; i++)
        rows[i]=1;
    queens(0);
    return 0;
}
```

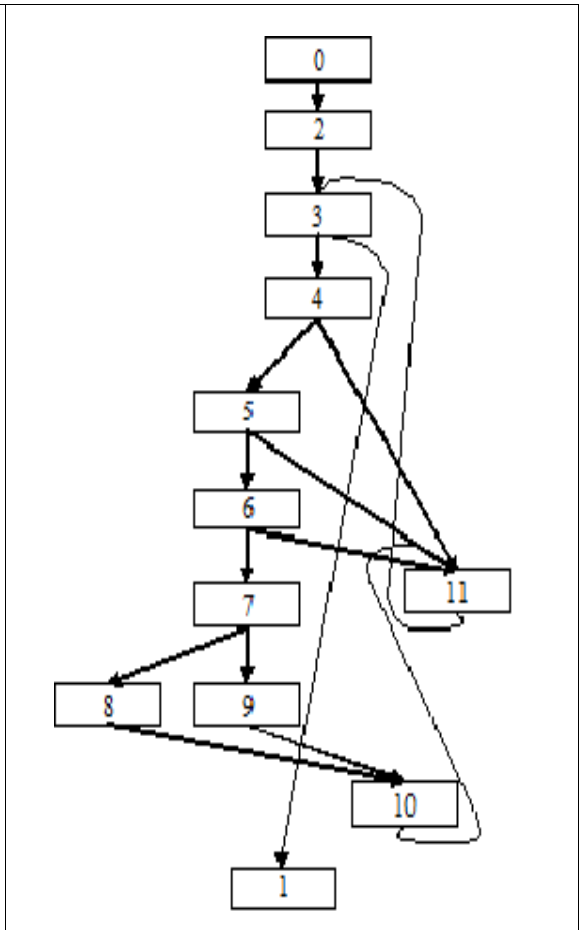


Рисунок 5 – Текст програми і граф потоку управління функції *queens()*

```
static void queens(INT  c)
{
  int  q[13u];
  q[10] = 0;
  q[8] = 0;
  q[1] = (c + 7);
  q[4] = 0;
  q[7] = 0;
  q[2] = -1;
  q[11] = 0;
L127:  if (!q[2]) goto L128;
  q[11]++;
  q[2] = (unsigned) q[2] >> 1;
  goto L127;
L128:  q[12] = q[11]+q[2]-19;
  q[11] = q[11] % q[12];
L111:
  if (q[q[11]+4] >= 8) goto L45;
  q[3] = q[10];
  q[q[11]+2] = ((q[8] + q[7]) + 1);
  q[9] = (q[10] + 1);
  q[1] = ((q[1] + c) + 2);
  q[q[11]+4] = (q[10] + 2);
  q[7] = ((7 + q[q[11]+2]) - q[1]);
  if ((v3)[q[3]] == 0) goto L94;
  q[4]=((v8)[q[3]]+(v2)[((q[3]-c)+7)]);
  if((v7)[((q[3]-c)+7)]==0) goto L94;
  if (q[7] < 0) goto L96;
  if (q[7] <= 7) goto L97;
L96:
  q[7]=((q[3] - c) + 7);
L97:  if ((v4)[(q[3] + c)] == 0) goto
L94;
  q[1] = (q[8] + (v2)[q[7]]);
  q[2] = (((c * (c + 1)) * q[3]) % 4);
  goto L99;
L122:  if (q[5] <= 0) goto L101;
  (v4)[(q[3] + c)] = 0;
  (v2)[((q[3] - c) + 7)] = 0;
  (v7)[((q[3] - c) + 7)] = 0;
  (v3)[q[3]] = 0;
  (v5)[c] = q[3];
  (v8)[c] = q[q[11]+2];
  q[5] = ((q[5] + 1) != 2);
  goto L102;
L101:  q[7] = (q[7] + 2);
  if (q[7] <= 14) goto L103;
  q[7] = (q[3] + c);
L103:  q[4] = (v2)[q[7]];
  (v4)[(q[3] + c)] = 0;
  q[1] = 0;
  (v7)[((q[3] - c) + 7)] = 0;
  (v8)[q[3]] = q[1];
  (v3)[q[3]] = 0;
  (v5)[c] = q[3];
  q[5] = ((q[5] + 4) != 5);
  q[1] = (v8)[c];
L102:  if (c != 7) goto L105;
L104:  print ();
  q[4] = ((c * 5) + 4);
  q[q[11]+2] = (v8)[c];
  (v8)[c] = q[4];
  goto L106;
L105:  q[q[11]] = ((c * 5) + 3);
  (v2)[((q[3]-c)+7)]=((q[8]+q[1])-7);
L115:
  if ((q[6] % 5) < 2) goto L108;
  q[4] = ((q[q[11]] % 5) + c);
  if ((q[4] % 7) != 0) goto L109;
  q[4] = 1;
L109:  q[1] = ((q[4] * q[4]) % 7);
  q[1] = ((q[1] * q[1]) * q[1]);
  c = ((c + (q[1] % 7)) - 1);
  queens(((c + 1)));
  q[11] = (q[q[11]+5] + q[12]) % 13;
L106:  (v4)[(q[3] + c)] = 1;
  (v2)[(q[3] + c)] = q[q[11]+2];
  (v7)[((q[3] - c) + 7)] = 1;
  (v8)[q[3]] = 1;
  (v3)[q[3]] = 1;
  q[4]=(((v2)[(q[3]+c)]+c)+7);
L94:  q[1] = (q[4] > 5);
  if ((v3)[q[9]] == 0) goto L111;
  if ((v7)[((q[9]-c)+7)]!=0) goto L112;
  if (q[1] != 0) goto L111;
  if (q[4] <= 5) goto L111;
L112: if((v4)[(q[9]+c)]==0) goto L111;
  q[6] = (((q[9] + c) * 5) + 1);
  q[1] = ((q[q[11]] + q[8]) + 1);
  goto L115;
L108:  (v2)[((q[9]-c)+7)]=q[5];
  (v4)[(q[9] + c)] = 0;
  (v8)[q[9]] = (v3)[q[9]];
  (v7)[((q[9] - c) + 7)] = 0;
  q[7] = (q[9] + c);
  (v3)[q[9]] = 0;
  q[8] = ((q[5] + q[2]) + 7);
  (v5)[c] = q[9];
  (v8)[c] = q[q[11]+2];
  if (c != 7) goto L117;
L116: print ();
  q[1] = (v8)[c];
  (v8)[c] = q[4];
  goto L118;
L117:  (v8)[(c + 1)] = q[1];
  queens(((c + 1)));
L118:  q[8]=((v2)[(q[9]+c)]+q[1]);
  q[1] = ((q[8] + q[7])+q[5]);
  if (((v1)[((((q[9]-
   c)+7)^q[5])%4)]%4)!=1) goto L120;
  (v2)[((q[9]-c)+7)]=((q[7]+q[9])-c);
  (v4)[(q[9] + c)] = 1;
  q[4] = (q[q[11]+2] + 1);
  (v8)[q[9]] = 1;
  (v7)[((q[9] - c) + 7)] = 1;
  (v3)[q[9]] = 1;
  q[11] = (q[11] + q[q[11]+6]) % 13;
  goto L111;
L120: q[2]=((((v7)[(q[9]-c)]+7)|1211)%6);
  q[4] = (q[8]+((v7)[(q[9]-c)]|1211));
  q[7] = (q[7] + 1);
L99:if((v6)[q[2]]>(v6)(q[2]+1) goto L122;
  (v2)[(q[9]+c)]=((v6)[q[2]]+c);
  q[8] = (((q[1]+q[4])+q[9])-7);
  (v4)[(q[9]+c)]=1;
  q[4] = (v8)[q[9]];
  (v8)[q[9]] = 1;
  (v7)[((q[9] - c) + 7)] = 1;
  q[7] = (q[7] - 1);
  (v3)[q[9]] = 1;
  q[q[11]+5] = (q[11] + q[12]) % 13;
  goto L111;
L45: ;
}
```

Рисунок 6 – Текст функції *queens()* після заплутування

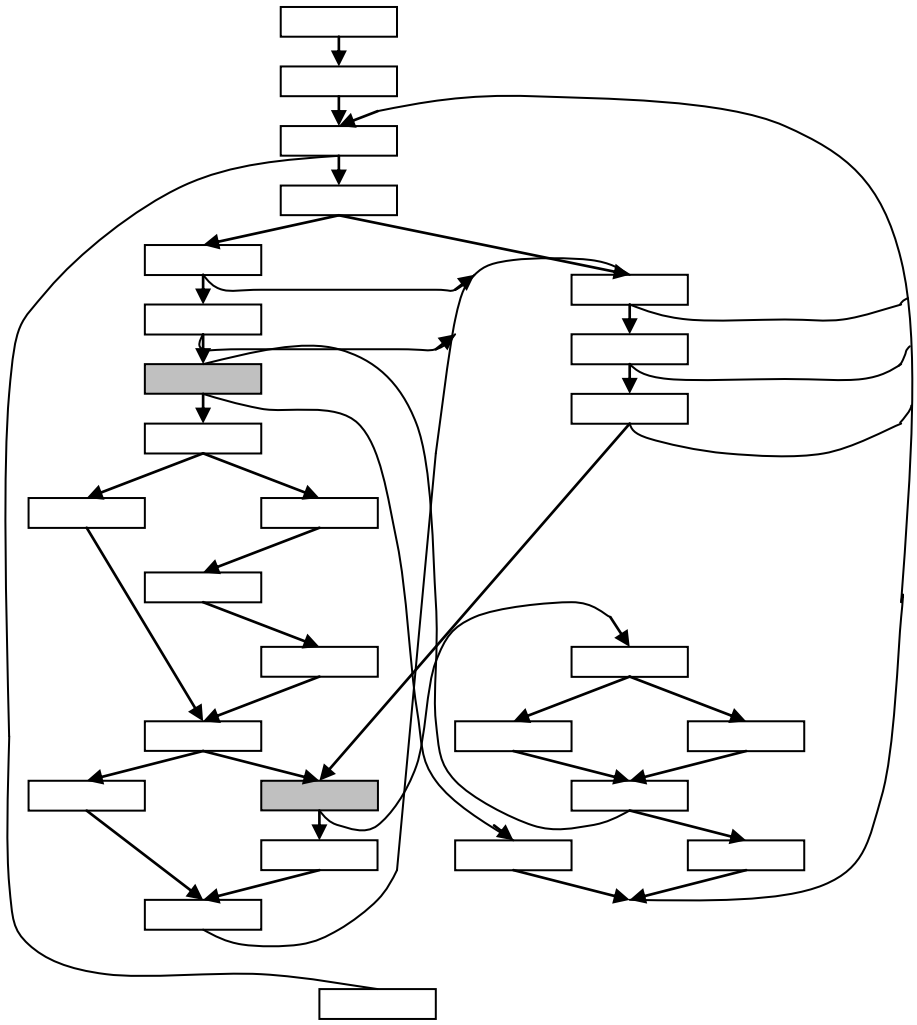


Рисунок 7 – Граф потоку управління замаскованої функції *queens()*

Як бачимо, зміни досить суттєві, а для хакера кількість роботи збільшилася на декілька порядків.

Використання покажчиків на спеціально створювані динамічні структури. У цьому підході в програму додаються операції по створенню структур даних (списків, дерев), і додаються операції над покажчиками на ці структури, підібрані таким чином, щоб зберігалися деякі інваріанти, які й використовуються як непрозорі предикати.

Один з простих способів використання динамічних структур даних для внесення залежностей за даними полягає в тому, що всі значення змінних усіх типів зберігаються в списку, який розміщується в динамічній пам'яті. Для доступу до змінних замість їх імен використовується розіменування спеціальних покажчиків. Крім того, покажчики на неістотні змінні час від часу змінюють своє положення у списку. В результаті всі звернення до колишніх локальних змінних функції перетворяться на звернення до об'єктів в області динамічної пам'яті. Для розділення істотних і неістотних змінних потрібно буде здійснити аналіз аліасів в динамічній пам'яті, працюючи при цьому з динамічними структурами даних довільної глибини. В даний час не існує такого методу статичного аналізу аліасів.

Цей підхід проілюстрований на рисунку 8. Функція містить чотири змінні: *a, b, c* і *d*. Нехай, для визначеності, змінні *a* і *b* – істотні, а *c* і *d* – неістотні. У замаскованій функції замість цих локальних змінних вводяться змінні *p1* і *p2*, що вказують на ланки списку, розміщеного в динамічній пам'яті. Змінна *a* доступна як *p1→prev* (або як *p2→next)*, а змінна *c* доступна як *p1→next* (або як *p2→prev)*.

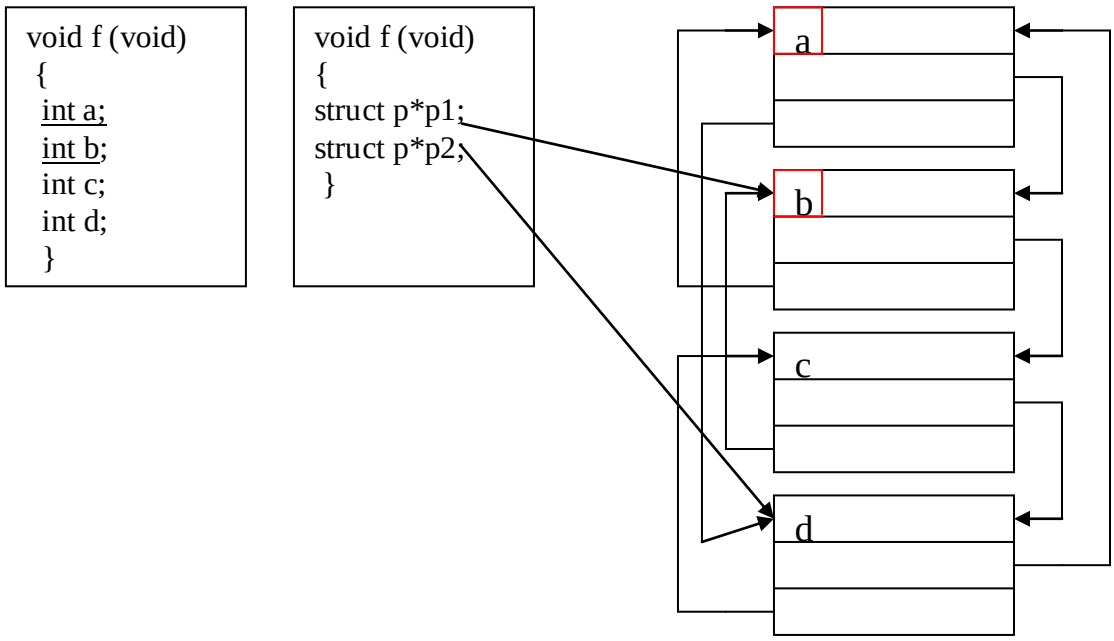


Рисунок 8 – Використання списків для внесення залежностей по даних

Побудова складних булевих виразів за допомогою еквівалентних перетворень із формули *true*. У найпростішому випадку ми можемо взяти *k* довільних булевих змінних $x_1 ... x_k$ і за допомогою еквівалентних алгебраїчних перетворень, коли частина дужок або всі дужки розкриваються, побудувати з них тотожності, наприклад:

$$1 = (x_1 \cup x_1) \cap ... \cap (x_k \cup x_k) \quad ; \; 0 = (x_1 \cup \bar{x}_1) \cap ... \cap (x_k \cup \bar{x}_k).$$

Це можна використати у багатьох випадках. Так, фрагмент коду:

```
i = 1 ;
while (i<101)
{      ...
      i++ ;
}
```

після розширення умов циклу матиме вигляд:

```
i = 1 ;
j = 100 ;
while ((i<101) && (j*j*(j+1)*(j+1)%4 == 0)(t))
 {       ...
         i++ ;
         j = j*i+3 ;
 }
```

Використання комбінаторних тотожностей, наприклад,

$$2^n = \sum_{k=0}^{n} C_n^k .$$

$$0 = \sum_{k=0}^{n} (-1)^k C_n^k$$

$$n \cdot 2^n = \sum\nolimits_{k=0}^{n} k \cdot C_n^k$$

$$n(n-1)^{n-2} = \sum_{k=0}^{n} k(k-1) C_n^k$$

$$n = \sum_{k=0}^{n} (-1)^k 4^{n-k} C_{2n-k+1}^k - 1$$

Програмування комбінаторних тотожностей може бути використано для ускладнення логіки програми всюди: і при вставлянні функцій, і при клонуванні функцій, і при розгортанні циклів, і при вставлянні мертвого та недосяжного кодів тощо.

2.6.3 Внесення недосяжного, мертвого або надлишкового коду

Якщо в програму внесені непрозорі предикати видів P^F або P^T, гілки умови, які відповідають умові "true" у першому випадку й умові "false" у другому випадку, ніколи не будуть виконуватися. Фрагмент програми, що ніколи не виконується, називається ***недосяжним кодом*** (*unreachable code*). Ці гілки можуть бути заповнені довільними обчисленнями, які можуть бути схожі на дійсно виконуваний код, наприклад, зібрані із фрагментів тієї ж самої функції. Оскільки недосяжний код ніколи не виконується, дане перетворення впливає тільки на розмір заплутаної програми, але не на швидкість її виконання. Загальне завдання виявлення недосяжного коду, як відомо, алгоритмічно нерозв'язне. Це означає, що для виявлення недосяжного коду повинні застосовуватися різні евристичні методи, наприклад, засновані на статистичному аналізі програми.

На відміну від недосяжного коду, ***мертвий код*** (*dead cod*e) у програмі виконується, але його виконання ніяк не впливає на результат роботи програми. При внесенні мертвого коду розробник повинен бути впевнений, що вставляє фрагмент, який не може впливати на код, що обчислює значення функції. Це практично означає, що мертвий код не може мати побічного ефекту, навіть у вигляді модифікації глобальних змінних, не може змінювати оточення працюючої програми, не може виконувати ніяких операцій, які можуть викликати виключення в роботі програми.

Надлишковий код (*redundant code*), на відміну від мертвого коду, виконується, і результат його виконання використовується надалі в програмі, але такий код можна спростити або зовсім видалити, оскільки обчислюється або константне значення, або значення, уже обчислене раніше. Для внесення надлишкового коду можна використати алгебраїчні перетворення виразів вихідної програми або введення в програму математичних тотожностей. Наприклад,

$$sin^2x+cos^2x = 1;$$

або

$$x*p/q,$$

де p й q гарантовано мають однакові значення.

Також при внесенні надлишкових кодів можна використовувати непрозорі предикати у вигляді комбінаторних тотожностей, наприклад:

$$\sum_{i=0}^{8} C_8^i = 2^8 = 256.$$

Але при введенні алгебраїчних виразів слід бути досить уважними, оскільки при їх обчисленні можливі переповнення.

Етап генерації неістотного (мертвого, надлишкового або недосяжного) коду складається з наступних підетапів:

- генерація пулу типів;
- генерація пулу змінних;
- генерація неістотної коди.

Генерація пулу типів. Готуються визначення типів, які потім використовуватимуться у "холостому" коді. Типи даних для "холостого" коду будуються одним з наступних способів:

- безпосередньо використовуються типи, визначені в маскованій програмі;
- використовуються вбудовані типи;
- з вбудованих типів і типів, визначених в маскованій програмі, шляхом вбудовування в шаблонні типи маскуючого компілятора. Наприклад, з типу *t*, визначеного в маскованій програмі, можуть бути побудовані типи масиву елементів типу *t*, списки, дерева з елементами типу *t*;
- модифікацією структурних типів, визначених в маскованій програмі.

Для кожного типу, що додається в масковану програму, в маскуючому компіляторі будується *клас-реалізатор*, на який покладаються всі функції з генерації інструкцій для маніпуляцій з цим типом, а кожна неістотна змінна, додана в масковану функцію, в маскуючому компіляторі представлена класом.

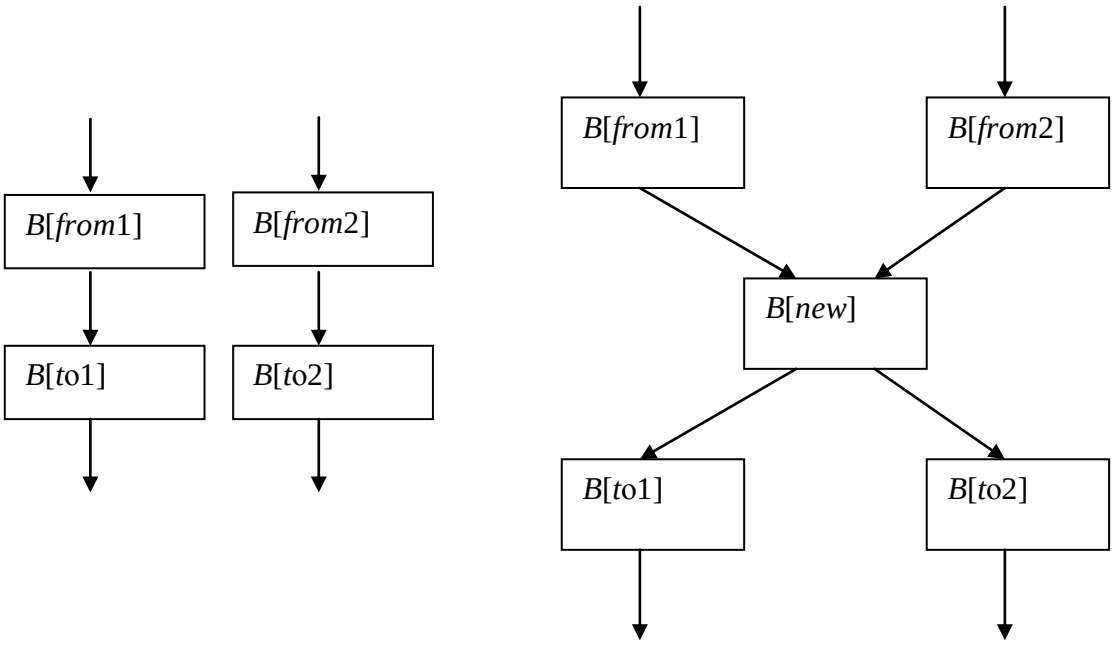
*Генерація пулу змінних***.** Для генерації пулу холостих змінних використовуються типи, побудовані на попередній стадії. Змінні розміщуються як на рівні локальних змінних функції, так і на рівні глобальних змінних.

*Генерація неістотного коду***.** Інструкції неістотного коду розміщуються в базових блоках упереміш з інструкціями початкової функції і керуючими інструкціями.

2.6.4 Зчеплення дуг

Щодо зчеплення дуг, то схема такого перетворення показана на рисунку 9. Для перетворення вибираються дві випадкові дуги графа потоку управління функції. При цьому перевага віддається "далеким" один від

одного дугам, де відстань вимірюється як мінімальне з довжин двох найкоротших шляхів уздовж графа від кінця однієї дуги до початку іншої. Дві вибрані дуги не повинні мати загального початку або загального кінця. Ключовим для забезпечення надійності зачеплення дуг є предикат *P*, який в кінці виконання нового базового блоку *B[new]* гарантує повернення на "правильний" шлях виконання – так званий повертаючий предикат.



а (початковий граф) б (перетворений граф)

Рисунок 9 – Схема перетворення зачеплення дуг

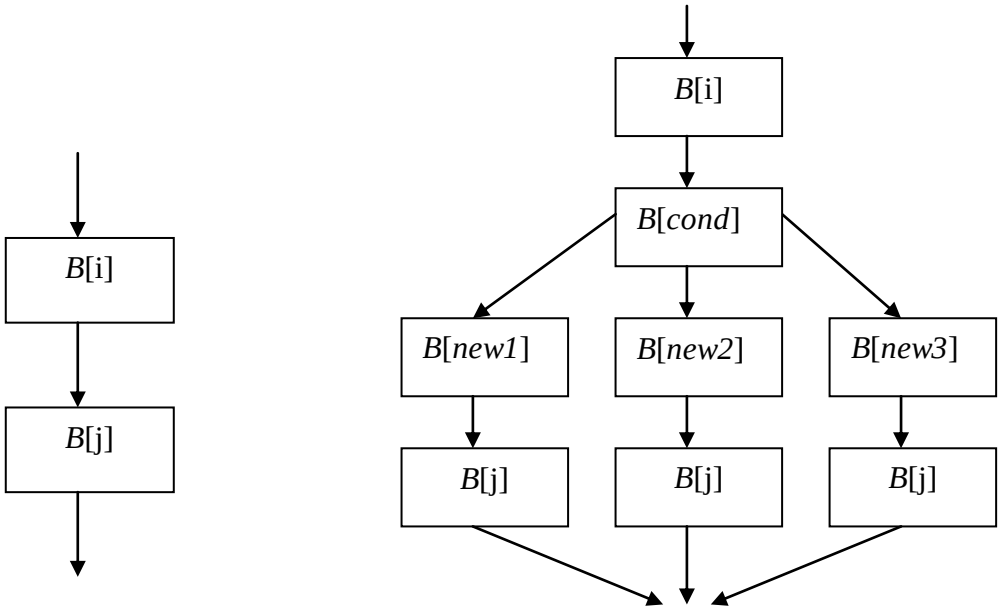
До перетворення після виконання базового блоку *B[from1]* завжди виконувався базовий блок *B[to1]*, а після базового блоку *B[from2]* – блок *B[to2]*. В результаті виконання даного перетворення створюється новий базовий блок *B[new]*, який виконується і після *B[from1]*, і після *B[from2]*. Новий базовий блок завершується обчисленням предиката *P*, в залежності від якого управління передається або на базовий блок *B[to1]*, або на базовий блок *B[to2]*. Предикат *P* повинен гарантувати, що управління повернеться на ту гілку, з якого воно прийшло в блок *B[new]*.

Найпростіший повертаючий предикат – це звичайна булева змінна, яка може бути оголошена на рівні локальних або глобальних змінних.

Повертаючі предикати можуть бути побудовані на основі хеш-функцій. Нехай хеш-функція *f* відображає цілочисельний тип в булевий. Введемо змінну *v*, яку використовуватимемо як аргумент *f*. Таким чином, предикат *P* дорівнює *f(v)*. Установлення значення *P* в *true* еквівалентне привласненню змінній *v* будь-якого значення *x*, на якому *f(x)=true*. Такі значення *x* можуть братися з деякого масиву P_{true}, який індексується довільним виразом *e*. Тоді установка значення предиката *P* в *true* виконується привласненням *v* значення $P_{true}[e]$. Установлення значення предиката *P* в *false* виконується аналогічно. Для побудови повертаючих предикатів можуть бути використані динамічні структури даних, аналогічно тому, як вони використовувалися для побудови лічильників.

2.6.5 Клонування базових блоків

Перетворення клонування базових блоків полягає в заміні послідовного виконання двох базових блоків (наприклад, *B[i]* і *B[j])* на оператор розгалуження з виконанням копії базового блоку *B[j]* на кожній з гілок. Для цього базовий блок *B[j]* повинен бути скопійований необхідне число разів. На рисунку 10 наведена відповідна схема перетворення.



а (початковий граф) б (перетворений граф)

Рисунок 10 – Схема перетворення клонування базових блоків

Тут базовий блок *B[j]* був розмножений двічі. Базовий блок *B[cond]* міститиме інструкції, необхідні для того, щоб кожна з трьох копій базового блоку *B[j*] виконувалася приблизно з однаковою частотою. Блоки *B[new1], B[new2], B[new3]* також можуть містити інструкції для підтримки рівномірного розподілення потоку виконання за трьома альтернативами.

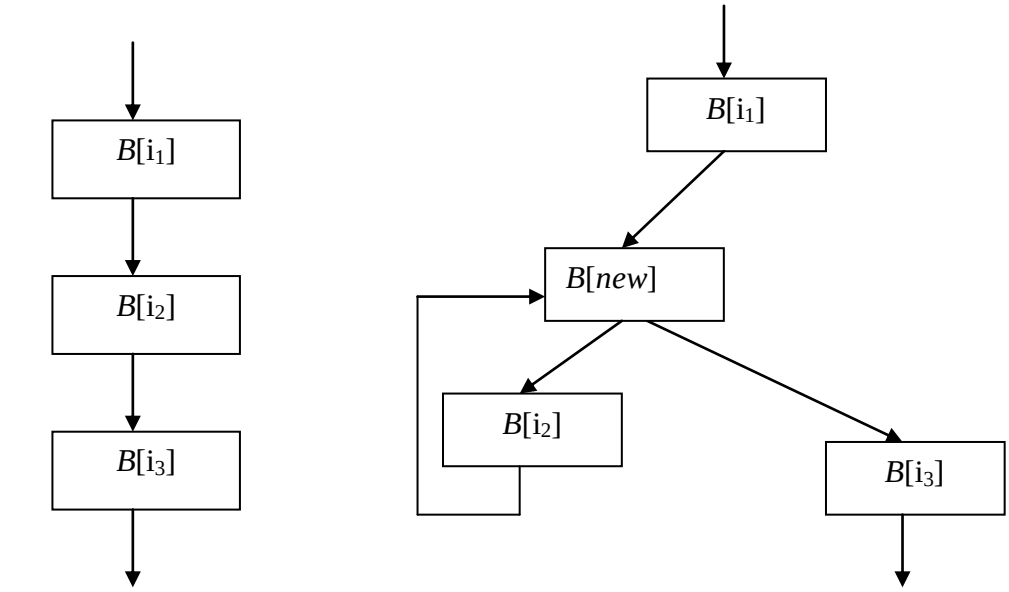
Вже на етапі клонування базових блоків починається побудова паралельної "холостої" функції, яка надалі буде об'єднана з основною функцією для отримання результуючої замаскованої функції. Спочатку "холоста" програма будується таким чином, що містить лише типи даних, змінні і інструкції, необхідні для коректної роботи програми з клонованими базовими блоками. При створенні паралельної функції одночасно будуються всі структури, що містять результати аналізу потоку управління і потоків даних паралельної функції. Використовуються такі методи побудови паралельної функції, при яких її семантичні властивості свідомо відомі.

2.6.6 Використання різноманітних перетворень циклів

Створення псевдоциклів. Перетворення полягає у внесенні в граф потоку управління функції зворотної дуги. При цьому контролюється, щоб тіло отриманого циклу виконувалося тільки один раз. Схема перетворення

показана на рисунку11. Тут зчіплюються дуги $B[i_1]$-$B[i_2]$ *і* $B[i_2]$-$B[i_3]$. Предикат *P*, що знаходиться в кінці базового блоку *B[new]*, повинен забезпечити одноразове виконання базового блоку $B[i_2]$.

Стійкість такого перетворення до аналізу визначається стійкістю повертаючого предиката. На відміну від зчеплення дуг, де інструкції встановлення значення повертаючого предиката *P* можуть бути розміщені далеко від точки зчеплення дуг, перетворення створення псевдоциклу більш обмежене. Встановлення значення предиката *P*, щоб управління покинуло псевдоцикл, не може бути винесене за межі блоку $B[i_2]$.



а (початковий граф) б (перетворений граф)

Рисунок 11 – Схема побудови "псевдоциклу"

Розгортання циклів. Цей метод застосовується в оптимізуючих компіляторах для прискорення роботи циклів або їх розпаралелювання. Розгортання циклів полягає в тому, що тіло циклу розмножується два або більше разів, умова виходу з циклу і оператор збільшення лічильника відповідним чином модифікуються. Якщо кількість повторень циклу відома в момент компіляції, цикл може бути розгорнутий повністю. Наприклад, цикл

```
for (i=1; i=<n; i++)
{
    // тіло циклу
}
```

після розгортання може мати вигляд:

```
for (i=1; i<n-1; i++)
{
    // тіло циклу
}
// тіло циклу
```

Пониження розмірності індексного простору. На рисунку 12 наведено приклад застосування перетворення пониження розмірності індексного простору для функції перемножування двох матриць. Перетворення дозволило перейти від триразового вкладеного циклу до єдиного циклу.

```
enum {N=128};
typedef double matr_t[N][N];
void mm(matr_t m1,matr_t m2,
        matr_t r)
    {
     int  i,j,k;
     for(i=0;i<N;i++)
        for(j=0;j<N;j++)
           r[i][j]=0;
     for(i=0;i<N;i++)
        for(j=0;j<N;j++)
           for(k=0;k<N;k++)
              r[i][j]+=
              m1[i][k]*m2[k][j];
     . . . .
    }
```

а (функція до перетворення)

```
enum {N=128};
typedef double matr_t[N][N];
void mm (matr_t m1,matr_t m2,
         matr_t r)
    {
     int  i,j,k;
     for(i=0;i<N*N;i++)
        r[i/N][i%N]=0;
     for(i=0;i<N*N*N;i++)
        r[i/(N*N)][i%(N*N)/N]+=
        m1[i/(N*N)][i%N]*
        m2[i%N][i%(N*N)/N];
     . . . .
    }
```

б (функція після перетворення)

Рисунок 12 – Приклад перетворення пониженням розмірності

Розкладання циклів. Розкладання циклів полягає в тому, що цикл зі складним тілом розбивається на декілька окремих циклів із простими тілами й з тим самим простором ітерування. Наприклад, цикл

```
for (i=1; i<n; i++)
{
    a[i]+= c ;
    x[i+i]=d+x[i+1]*a[i] ;
}
```

після розкладання може мати вигляд:

```
for (i=1; i<n; i++)
{
    a[i] += c ;
}
for (i=1; i<n; i++)
{
    x[i+i]=d+x[i+1]*a[i] ;
}
```

Як правило, наведені методи ускладнення логіки програмно реалізують на мовах програмування високого рівня (C, C++, Perl, Java та ін.).

2.7 Додаткові методи бборотьби з автоматичними і інтерактивними дизасемблерами

2.7.1 Захист від автоматичних дизасемблерів

Автоматичні дизасемблери аналізують код виконуваного файлу й формують відповідний йому вихідний текст або лістинг. Синтаксичний аналіз коду може звести нанівець всі зусилля зі створення протитрасувальної підсистеми. Переглянувши дизасембльований текст програми,

можна знайти й обійти всі механізми захисту від налагодження. Тому необхідно посилити реалізацію підсистеми.

Захиститися від статичного дослідження програми можна, крім вище наведених методів, ще й різними асемблерними трюками, спрямованими на перекручування вихідного коду дизасемблера. Розглянемо деякі з них.

Перемішування команд. Цей метод можна, мабуть, назвати най простішим із запропонованих, оскільки він практично не вимагає зміни основного алгоритму роботи програми. Метод полягає в наступному.

Якщо команди випадковим чином поміняти місцями (залишивши першу на місці) і перед кожною наступною вставити команду переходу на неї, то послідовність виконання команд не зміниться і буде мати той самий фактичний зміст. Нехай оригінальний код такий:

```
start:
    instruction 1
    instruction 2
    instruction 3
    instruction 4
 end:
```

Тоді перетворений код може бути таким:

```
start:    jmp   @@1
 @@4:    instruction  4
          jmp         end
 @@2:    instruction  2
          jmp         @@3
 @@1:    instruction  1
          jmp         @@2
 @@3:   instruction   3
          jmp         @@4
 end:
```

Розглянемо приклад. Нехай у нашому програмному коді є типова ділянка якогось класичного захисту:

```
0000000: 16            push      ss
0000001: 17            pop       ss
0000002: 9C            pushf
0000003: 58            pop       ax
0000004: A90001        test      ax
0000007: 740C          je   RealModeDebuggerDetected
```

Тут вся послідовність команд у нас перед очима і ніщо не складає труднощів проаналізувати її. Тепер "перемішуємо" цей код. Наведений фрагмент може в результаті виглядати приблизно так:

```
00000000: 16           push ss
00000001: EB07         jmps 00000000A
00000003: 58           pop  ax
00000004: EB08         jmps 00000000E
00000006: 740B         je   RealModeDebuggerDetected
00000008: EB09         jmps 000000013
0000000A: 17           pop  ss
0000000B: 9C           pushf
0000000C: EBF5         jmps 000000003
0000000E: A96400       test ax,00064
00000011: EBF3         jmps 000000006
```

Тепер провести аналіз коду буде вже набагато складніше: під дизасемблером і у налагоджувачі ми будемо спостерігати постійні стрибки з місця на місце, причому відстежити, звідки була виконана чергова команда переходу, дуже проблематично (не всі налагоджувачі мають подібні функції). Якщо дизасемблювати такий виконуваний файл, то ми одержимо лістинг, що прийдеться постійно перегортати взад-вперед, відслідковуючи порядок виконання програми.

Приховування команд передачі управління приводить до того, що дизасемблер не може побудувати граф передачі управління. Це можна здійснити з допомогою:

- непрямої передачі управління;
- модифікації адреси переходу в коді програми.

Переходи і виклики підпрограм по динамічно змінюваних адресах мають на увазі модифікацію байтів адрес переходу або виклику підпрограми, що знаходяться за першим байтом команди (байтом коду операції). Наведемо декілька прикладів реалізації цього методу(табл.3).

Таблиця 3 – Приклади реалізації приховування команд

Фрагмент коду	Пояснення

`mov word ptr cs:m[1],1234h`
`. . .`
`m:`
`jmp place`
`. . .`
	модифікація адреси переходу – підстановка нової адреси (адреса переходу або виклику підпрограми, що знаходиться за першим байтом команди)

`mov word ptr cs:m[1],es`
`mov word ptr cs:m[3],5678h`
`. . .`
`m:`
`call far 0000h`
	модифікація адреси підпрограми, що викликається (адреса переходу або виклику підпрограми, що знаходиться за першим байтом команди)

`mov bx,1234h` `jmp dword ptr cs:[bx]`	модифікація адреси – непрямі переходи
`call word ptr es:[si]`	модифікація адреси – непрямі виклики

Використання коду, що перекривається. Код можна зробити скільки завгодно багаторівневим. Наведемо лише простий приклад.

```
eat_sr:
       mov ax,02EBh
       jmp $-2
       ...         ; решта коду
```

Перша інструкція заносить "ліве" значення в AX. Друга робить перехід на значення операнду команди `MOV AX, 02EBh`. Тут '`02EB`' означає команду '`jmp$+2`'. Цей перехід перестрибує перший `JMP` і продовжує виконувати команду далі за кодом.

А ось інший приклад коду, що перекривається.

```
mov   ax, 0FE05h
jmp   $-2          ; Перехід на 05h FEh, тобто на
                   ; команду add ax, 0EBFEh, за якою
                   ; слідують cld і add ah, 3Bh
add ah, 03Bh       ; AX = 2503h
```

Реальне значення AX буде невідомо доти, поки не буде поміщено в регістр. І цей факт можна надалі використати.

Використання нестандартних способів передачі управління (заміна `jmp` через `ret`, `ret` і `call` через `jmp`) (табл. 4).

Таблиця 4 – Приклади нестандартних способів передачі управління

Первинний код	Альтернативний код

`jmp m`
`. . .`
`m:`
	`mov ax,offset m ; занести в стек адресу мітки`
	`push ax`
	`ret ; перейти на мітку`
	`. . .`
	`m:`

`. . .`
`call subr`
`m:`
`. . .`
`subr:`
`. . .`
`ret`
`. . .`
	`mov ax,offset m ; занести в стек адресу повернення`
	`push ax jmp subr ; перейти на підпрограму`
	`m:`
	`. . .`
	`subr:`
	`. . .`
	`ret`
	`. . .`

`. . .`
`INT 13h`
	`. . .`
	`pushf ; занести у стек прапори`
	`push cs ; занести у стек CS`
	`mov ax,offset m ; занести у стек адресу повернення`
	`push ax`
	`xor ax, ax`
	`mov es, ax`
	`jmp dword ptr es:[13h*4]`
	`m:`

`ret`	`pop bx ; взяти зі стеку адресу повернення` `jmp bx ; перейти на нього`

`iret`
`. . .`
	`mov bp,sp ; перехід на точку повернення`
	`jmp dword ptr [bp] ; з переривання`
	`. . .`
	`add sp,4 ; точка повернення`
	`popf`

Заплутування за допомогою безумовних переходів. Це є найпростішим способом заплутування коду. Після команди переходу на приховувану команду ставиться декілька команд, що виглядають осмислено і код операції і/або префікс команди, що має великий розмір у байтах. Довжина в кожному конкретному випадку підбирається з таким розрахунком, щоб кінець цієї фіктивної команди потрапив у середину однієї з дійсних. Це приводить до того, що дизасемблер, починаючи з цієї команди, видає неправильну послідовність команд, а нерідко й взагалі не може нічого декодувати і тільки пише послідовність директив оголошення даних (DB, DW, ...). Крім того, хороші налагоджувачі (InSight, DeGlucker, Meffistofel, TR) показують у вікні коду таку ж невірну послідовність команд доти, поки ко-

манда переходу не буде виконана, а неякісні (Microsoft Debug, Microsoft CodeView, Turbo Debugger) не відображають правильної команди навіть після виконання команди JMP (коли CS:IP зберігають адреси команди).

Наведемо приклад обману дизасемблера за рахунок приховування асемблерної команди в більше довгій (а насправді її код операції обходиться). При цьому для обходу байтів, що засмічують програму, використовується команда безумовного переходу.

```
            jmp   Hidden
            mov   ax, 3000
            INT   21h
            DB    0EAh       ; КОП подальшого переходу
      Hidden:
            mov   bx, OFFSET BeingDebugged
```

Починаючи зі зсуву, позначеного як Hidden, дизасемблер видасть або неправильну послідовність команд, або серію директив DB. Спосіб дуже легко реалізується, однак і зрозуміти, у чому тут справа, відносно просто.

Більш ефективна і більш компактна варіація цього методу, при якому для прихованої захищеної команди застосовується *умовний* перехід за раніше відомою істинною умовою. Однак істинність цієї умови не повинна бути "зрозумілою" для компілятора, а тим більше для дизасемблера. Перевагою цього методу є й те, що він не вимагає внесення додаткових, ніколи не виконуваних команд у текст програми. Цей спосіб може бути й непоганим протиналагоджувальним прийомом – одночасно спантеличувати і дизасемблер, і налагоджувач.

```
            mov   ax, BeingDebugged
            cmp   ax, 0
            je    NormalRun
            DB    0EAh
      NormalRun:
            call  SecretRoutine
```

Дизасемблер не розуміє, що умова свідомо істина, а "дальній перехід", сфабрикований нами, ніколи керування не одержить. Тому він сумлінно дизасемблює неіснуючий ланцюжок команд.

Також дизасемблер збиває ***нестандартний формат завантажуваного модуля*** (наприклад, перекрити весь сегмент коду стеком).

2.7.2 Протистояння інтерактивним дизасемблерам

Інтерактивні дизасемблери формують вихідний текст/лістинг по виконуваному коду програми так само, як це роблять автоматичні дизасемблери. Однак інтерактивні дизасемблери відрізняються від автоматичних наявністю потужного користувацького інтерфейсу, що значно полегшує аналіз дизасемблерної програми. Вони дозволяють:

- змінювати імена змінних, міток, підпрограм і т. д., вводити імена для нових адрес, видаляти наявні мітки/імена;
- шукати послідовності символів у результуючому тексті і послідовності байтів у виконуваному коді;

– повторно дизасемблювати ділянки коду в послідовність асемблерних команд або директив DB;
– задавати коментарі до підпрограм, переривань і т. д., які автоматично розставляються біля всіх відповідних викликів;
– переглядати перелік сегментів програми;
– редагувати дизасемблерний текст із автоматичною модифікацією здійсненного коду або без неї.

Різні інтерактивні дизасемблери надають також й інші можливості. Самі досконалі з них (наприклад, IDA) дозволяють не тільки змінювати вже дизасембльований код, але й втручатися в сам процес виконання.

Наведені методи добре діють проти автоматичних дизасемблерів. Однак заплутати такий дизасемблер, як IDA (або будь-який інший) тільки з їх допомогою не вдасться. Точніше, вдасться, але лише до того моменту, поки зламник не здогадається примусово позначити байти, що засмічують, як 'Undefined', а всі після них – як код. Відразу ж після цього він матиме можливість аналізу захищуваної програми у середовищі дизасемблера.

Проти цього існує складний, але ефективний прийом, що одержав назву ***"динамічний фуфель".*** Суть прийому полягає в тому, що байти для засмічування не обмежуються командами передачі керування. Вони замі-щаються простими командами (NOP, STI й ін.) уже в ході виконання програми, але до першого запуску програм, що містять "фуфелі". Інакше кажучи, захищуваний від дизасемблювання фрагмент програми дійсно не може бути запущений у тому вигляді, у якому програма перебуває на диску – швидше за все, це приведе до зависання комп'ютера. Однак, запус-тившись, програма зчитує звідкись дані, необхідні для усунення байтів і заміщає їх на команди, що ніяк не впливають на хід виконання програми.

Злам програми, що захищені таким методом, – тривалий і трудомісткий процес, навіть якщо дані для "дефуфелізації" містяться в коді самої програми.

Можна й ще ускладнити злам, зберігаючи ці дані на нульовій доріжці або в навмисно залишеному проміжку між розділами дискової підсистеми. У цьому випадку саме по собі копіювання секретної програми нічого супротивникові не дасть – він втратить дані, необхідні для її перетворення до нормально працюючого стану.

2.7.3 Висновки щодо ефективності використання методів ускладнення логіки

З усього вищезазначеного можна зробити відповідні висновки.

До *позитивних* рис запропонованих методів можна віднести те, що вони при вдалій реалізації практично виключають можливість абстра-гування від утрудненого коду. Отже, чим глибше і гармонійніше переміш-шування коду інтегрується в оригінальний код програми, тим ефективніше його протидія людському фактору.

Залишається тільки відкритим питання про автоматичну чи хоча б про напівавтоматичну реалізацію цих методів. На даний час вже розробляють-

ся засоби (обфускатори, скремблери), призначені для здійснення автоматичного ускладнення аналізу початкового коду програм.

Слід зазначити головний істотний *недолік* застосування даних методів: невиправдане зростання обсягу коду програми.

Не слід також забувати про *можливість атак зловмисників* при використанні даних методів. Здавалося б, що відновити початковий вид коду після такого перетворення є не дуже складною задачею: просто йти по "ланцюжку" команд переходу або інших інструкцій, послідовно зберігаючи інструкції, що не відносяться до команд передачі керування. На практиці ж виникає ряд труднощів:

- необхідно написати подобу дизасемблера (треба, як мінімум, знати довжину всіх інструкцій), який буде виконувати таку задачу;
- задача ускладнюється на кілька порядків, якщо для передачі керування на наступну інструкцію використовувати команди умовного переходу. Тоді справа вже не обмежиться дизасемблером: прийдеться створювати цілий емулятор процесора, щоб правильно визначати стан прапорців реального процесора перед виконанням переходу і, відповідно, його кінцеву мету;
- при зворотній перестановці коду зміниться положення інструкцій з відносною адресацією (JMP, CALL і ін.) і зміст коду також істотно спотвориться.

Крім того, зовсім неможливо визначити, чи була якась команда переходу додана автоматично, щоб заплутати зламника, чи ця команда передбачена оригінальним алгоритмом.

Таким чином, наведені методи цілком відповідають концепції ускладнення нелінійного аналізу: зламник у будь-якому випадку зіштовхується з величезною кількістю переходів і інших «неприємностей», обсяг яких може багаторазово перевищувати розмір самого коду.

При правильній реалізації, а також використовуючи нескладні хитрування, що роблять неможливим автоматичне відновлення оригінального коду, у зламника не залишиться ніякої можливості проігнорувати ці підходи і він повинен буде «прокрокувати» всі зустрічні ним помилкові і реальні інструкції.

2.8 Емуляція процесора та мультизадачності як способи протистояння статичному вивченню програм

2.8.1 Суть захисту за допомогою емуляції процесора

Як уже було сказано, зламник, одержавши якимось чином лістинг програми, бачить уже знайомі йому інструкції процесора і питання швидкості зламу багато в чому залежить лише від того, наскільки хороші його навички при роботі з процесором. Саме проти цього факту і спрямована запропонована методологія.

Філософія підходу така: навіть якщо зламник є як завгодно досвідченим в архітектурі одного процесора, то, зіштовхнувшись з якимось

іншим і не володіючи технічною інформацію з цієї архітектури, йому доведеться витратити досить багато часу на засвоєння нової архітектури.

Суть методології в тому, що критичні ділянки коду програми (ідентифікація, автентифікація, критичні функції і процедури чи будь-який інший об'єкт захисту, нелінійний аналіз коду яких необхідно утруднити) або навіть вся програма цілком повинні бути написана під "саморобний" процесор, інструкції якого виконуються емулятором на реальному процесорі x86. Очевидно, що ми можемо розробити як завгодно складну, більш заплутану і нелогічну, ніж x86, архітектуру і систему команд під наш саморобний процесор. Обмеженням тут може бути лише фантазія розробника.

Вважатимемо, що поняття архітектури цілком може бути застосовано до псевдопроцесорів. З тією лише різницею, що для реальних процесорів архітектуру складають принципи роботи з "залізом", формати організації і представлення даних всередині процесора, а для емулятора – внутрішні формати даних, алгоритми роботи (виконання псевдокоманд) з позиції емульованого коду.

Починати проектування "процесора" слід із визначення кола задач, які йому потрібно буде вирішувати. Наприклад, якщо програма зашифрована, то написати її розшифровувач у такий спосіб є непоганою ідеєю. І хоча такий "процесор" може бути набагато менш ефективним у плані продуктивності, ніж той, на якому "крутиться" сам емулятор, деякі, не критичні за часом процедури, які вимагають захисту, можна переписати під нього. Такими "вузькими місцями" можуть бути перевірка серійного номера чи якісь "цінні функції" (недоступні, наприклад, у ShareWare-версії продукту). Слід зазначити, що більшість функцій середньостатистичної комерційної програми взагалі не є критичними до продуктивності, тому їх "перенесення" під емульований процесор може цілком бути виправданим.

І хоча на перший погляд задача може здаватися складною, такому "емулятору" не потрібно дуже багато "уміти". Основні арифметичні і логічні операції, команди передачі керування (JMP, CALL та інші), команди доступу до зовнішньої (відносно нашого емулятора) пам'яті – до пам'яті програми. Побудувати емулятор з деякою кількістю регістрів і не “навороченою” лінійною адресацією пам'яті не так складно. А цього вистачить для більшості задач. Маючи базові знання про кодогенерацію асемблерів IBM PC, неважко придумати що-небудь своє. Нехай, наприклад, усі команди нашого "процесора" будуть однобайтовими, з додатковими байтами-параметрами команди. Усе, що залишається зробити – процедуру, яка буде приймати як параметр адреси на процедуру в наших "опкодах" і послідовно їх вибирати, ідентифікувати та виконувати.

Потім, якщо це не команда переходу, збільшувати покажчик поточної команди на розмір команди (а якщо це команда переходу чи виклик процедури, то відповідним чином змінювати стан стека і покажчик наступної команду). І припиняти виконання, наприклад, по спеціальному опкоду.

Фактично, ми "видумуємо" і імплементуємо ("натягаємо" на реальний, процесор x86) свій власний асемблер, побічна перевага якого в тому, що ми можемо самі придумувати будь-яку систему команд, яку побажаємо.

Наприклад, для полегшення реалізації конкретного алгоритму можна подбати про створення інструкцій, що, за незрозумілими для нас причинами, відсутні у x86, у якої обидва операнди – комірки пам'яті.

Для ускладнення розбирання емулятора зламниками можна придумати безліч способів. Наприклад, можна, замість багатьох однозначних арифметичних інструкцій ввести одну універсальну, яка буде відразу складати два спеціальних регістри, відведених для додавання, множити два регістри, відведених для множення. Потім (вже в коді, що емулюється) просто завантажити потрібні дані в потрібні регістри і виконати цю інструкцію. Оскільки результуючі дані можуть бути використані не відразу, то визначити, у якому сенсі була викликана ця інструкція в тому чи іншому випадку, буде проблематично.

Код процедур для емуляції можна зашифрувати або ущільнити, а потім відновлювати "на льоту", по одній команді самим емулятором. Без покрокового виконання програми побачити реальний вміст коду, навіть здогадуючись про зміст наших "опкодов" чи вірогідно знаючи його, буде також проблематично.

Таким чином, щоб одержати хоч якесь представлення про роботу таких ділянок "коду", зламнику необхідно буде вникнути в архітектуру і систему команд нашого емулятора. Зробити це без технічної документації, маючи в наявності лише сам емулятор, буде непросто.

Крім всього іншого, для зламника виключається можливість налагодити нашу критичну ділянку, оскільки такі методи, як вставка точок зупину (break points), покрокове виконання і т.д., можуть бути застосовні лише до коду емулятора, що вибирає команди, але не до емульованого коду. Щоб здійснити це, прийдеться як мінімум писати й “знеглючувати” свій налагоджувач, у який буде вбудований такий самий емулятор.

Але при всьому цьому *атаки на емулятори процесорів* все ж можливі. По-перше, зламник може розібрати команди. Якщо емулятор сам розбирає виконуваний їм псевдокод – команду за командою – то потенційний зламник теж у змозі це зробити. Для цього йому фактично прийдеться написати дизасемблер для процесора з новою, нікому досі невідомою архітектурою. Причому, всі нюанси цієї архітектури йому доведеться розбирати самому, без будь-якої літератури, яка звичайно супроводжує промислово стандартизовані архітектури.

По-друге, можна використати уразливість емуляторів з роздільними даними і кодом. Слід звернути увагу на те, як особливості архітектури можуть вплинути на уразливість захисту, побудованого на ній. Мова йде про поділ даних і коду в адресному просторі псевдопроцесора. Реалізація емулятора, при якій код і дані емульованого процесу розділені архітектурно (тобто емульована процедура адресує дані і код різними командами: це можна спостерігати й у реальних мікроконтролерах), більш уразлива для дизасемблерів. Це відбувається через те, що дизасемблеру вже не треба визначати (а визначаючи автоматично – помилятися), які з дизасембльованих даних представити у вигляді команд, а які – у вигляді даних. І,

навпаки, процес аналізу коду архітектури, подібної x86, ускладнює процес аналізу коду.

Наступний приклад ілюструє можливість саме такої помилки дизасемблера. Вихідний код:

```
...
INT  20h
db   "THIS_IS_PROGRAM_DATA"
...
```

От як він може бути сприйнятий дизасемблером:

```
00000000: CD 20    INT       20
00000002: 54       push      sp
00000003: 48       dec       ax
00000004: 49       dec       cx
00000005: 53       push      bx
00000006: 5F       pop       di
00000007: 49       dec       cx
00000008: 53       push      bx
00000009: 5F       pop       di
0000000A: 50       push      ax
0000000B: 52       push      dx
0000000C: 4F       dec       di
0000000D: 47       inc       di
0000000E: 52       push      dx
0000000F: 41       inc       cx
00000010: 4D       dec       bp
 . . .
```

Природно, ці зовсім безглузді інструкції аналізувати даремно.

Логічно припустити, що чим витонченішою і нелогічнішою буде архітектура псевдопроцесора, тим більше сил і часу прийдеться витратити зламнику, щоб розібратися в ній.

Швидше за все, реалізація даного методу повинна являти собою деякий компілятор (з асемблера чи з будь-якої мови високого рівня), що збирає емулятор з унікальною системою команд і архітектурою, і що шифрує чи пакує псевдокод програми. Цей компілятор можна використовувати багаторазово: цілком реально формувати матрицю опкодів зовсім випадково.

2.8.2 Емуляція мультизадачості

Метод оснований на ускладненні налагодження шляхом виконання декількох гілок алгоритму "паралельно". Причому, в залежності від інструментарію зламника і від конкретної реалізації багатозадачності, можна або цілком виключити налагодження (якщо скористатися апаратними можливостями процесорів сімейства x86, але це виключить сумісність із сучасними, мультизадачними і не дуже, операційними системами), або логіка програми буде змінюватися при виконанні під налагоджувачем: деякі з гілок алгоритму взагалі можуть не виконуватися.

Підійти до реалізації багатозадачності можна з двох позицій.

1. Реалізувати "*емулятор багатозадачного процесора*". Для цього варіанта слід буде лише ввести кілька буферів для команд і даних (тут можливі варіанти – можна створити як окремі області пам'яті для всіх задач, так і емулювати все в окремому адресному просторі псевдопроцесора). Після цього просто виконувати команди по черзі з усіх задач, стежачи за адресацією, якщо пам'ять задач розділена.
2. Реалізувати "*паралельне виконання*" декількох ділянок коду дійсного процесора. Для реалізації цього методу треба продумати механізм збереження контексту виконуваної "задачі" і переключення в іншу "задачу". Так, можна встановити оброблювач переривання таймера, що буде зберігати регістри і запам'ятовувати те місце, куди повернутися при виконанні перерваної задачі, потім відновлювати регістри і прапорці задачі, якій варто передати керування. Потім просто змінити адресу повернення в стеці на адресу, з якої почнеться виконання процедури.

Перший метод успадковує всі недоліки емуляторів (менша швидкість роботи і т. п.), зате дозволяє організувати кращий розподіл ресурсів і "процесорного" часу, наприклад, виконувати по одній команді з кожної виконуваної гілки коду, чого на реальному процесорі домогтися важко.

Другий метод обмежений у використанні системних функцій у паралельно виконуваних процедурах (оскільки деякі системні сервіси можуть бути нерентабельні, а передача керування на іншу ділянку коду може здійснитися і під час відпрацьовування системної функції, всередині тіла функції). Іншим недоліком можна вважати труднощі розподілу системних ресурсів і рівномірного розподілу процесорного часу між виконуваними паралельно процедурами.

Якщо система реалізована на основі емулятора, то тут можна застосувати налагоджувач, переключення задач у загальному випадку повинно продовжувати функціонувати так само, як і без налагоджувача. А от якщо переключення задач реалізоване, наприклад, на основі переривань, то використання налагоджувача може порушити (змінити) хід виконання програми. Тоді залишається аналізувати процедури лінійно...

Запропонована методологія є авангардним підходом до захисту ПЗ і має солідні перспективи.

Контрольні питання

1. В чому доцільність і необхідність захисту від статичного дослідження?
2. Класифікуйте методи захисту від статичного дослідження.
2. Дайте загальну характеристику шифруванню як одному з методів протидії дизасемблюванню.
3. В чому сутність методу емуляції процесора з точки зору захисту програмного забезпечення?
4. Як може допомогти емуляція мультизадачності у боротьбі зі статичним дослідженням?
5. Охарактеризуйте структуру виконуваних файлів

6. Що таке заголовки виконуваних файлів, які їх види є? Для чого їх використовує операційна система?
7. Що таке таблиця об’єктів (розділів виконуваного файлу), які об’єкти існують у програмах від різних виробників?
8. Які способи впровадження захисних механізмів у виконувані файли ви знаєте?
9. Дайте поняття Х-коду та наведіть вимоги до нього.
10. Наведіть приклади впровадження Х-коду у заголовки виконуваних файлів. За якими алгоритмами вони здійснюються?
11. Що розуміють під терміном "обфускація"?
12. Які види обфускації ви знаєте?
13. Наведіть основні кроки для здійснення лексичної обфускації.
14. Які існують види обфускації даних?
15. Наведіть приклади обфускації даних і поясніть їх.
16. Що означає обфускація графа потоку керування?
17. Що означає поняття маскування програми? Які його етапи?
18. Які маніпулювання функціями допомагають захиститись від несанкціонованого дослідження?
19. Що таке непрозорі предикати?
20. Наведіть приклади використання непрозорих предикатів.
21. Як допомагає внесення недосяжного, мертвого або надлишкового коду захиститись від дослідження?
22. В чому сутність зчеплення дуг?
23. Що таке клонування базових блоків?
24. Які перетворення циклів допомагають запобіганню несанкціонованого дослідження програм?
25. У чому різниця між автоматичними і інтерактивними дизасемблерами?
26. Наведіть приклади автоматичних та інтерактивні дизасемблерів.
27. Які методи боротьби з автоматичними дизасемблерами ви знаєте?
28. Охарактеризуйте особливі методи боротьби з інтерактивними дизасемблерами.
29. Поясність такий “динамічний фуфель” як прийом для захисту від дослідження.
30. В чому сутність методології емуляції процесора?
31. Які шляхи реалізації емуляції багатозадачності для захисту від дослідження?
32. В чому полягає паралельне виконання гілок задачі з точки зору емуляції мультизадачності?
33. Які позитивні і негативні риси методів протистояння дизасемблерам ви можете назвати?

3 ЗАХИСТ ВІД НЕСАНКЦІОНОВАНОГО НАЛАГОДЖУВАННЯ

3.1 Огляд і класифікація налагоджувачів

Якщо дизасемблювання є першим кроком по статичному вивченню програми, і методи захисту від дизасемблювання виключають тільки найпримітивніші атаки, то наступним кроком “дослідника” програми буде вивчення її роботи в динаміці, під контролем налагоджувачів.

В даний час сучасні хакери використовують великий інструментарій, важливу частину якого складають налагоджувачі реального і захищеного режимів.

Налагоджувачі реального режиму

У дану групу входять "класичні" налагоджувачі реального режиму DOS, такі як DOS Debug, AFD, Turbo Debugger. В даний час налагоджувачі з цієї групи практично не використовуються для реінжинірингу, оскільки вони традиційно використовують для своєї роботи налагоджувальні і трасувальні переривання (INT 1 та INT 3), стек налагоджуваної програми, залишають сліди в тілі програми та у стеці. Навіть примітивні антитрасувальні заходи, наприклад, використання в захисних процедурах векторів переривань INT 1 і INT 3, ставлять зламника, що користується налагоджувачем з даної групи, в скрутне положення.

Найпоширеніші налагоджувальники описані далі.

Turbo Debugger by Borland International. Цей налагоджувач створений у 1988 р. братами Williams. Він має розвинений віконний інтерфейс, надає можливості з перегляду коду і вихідного тексту програми, шістнадцяткового дампу, змінних (при наявності налагоджувальної інформації), створенню макросів. Разом з тим налагоджувач має багато недоліків і помилок, активно використовуваних захистами: використання стека налагоджуваної програми; використання INT 1, INT 3 для трасування; перехоплення переривань INT 0, INT 1, INT 3; некоректна робота з відеобуфером; некоректне виставляння початкових значень регістрів; неправильне дизасемблювання інструкцій виду JMP $+1; API відсутній.

CodeView by Microsoft. За своїми помилками цей налагоджувач мало відрізняється від TurboDebugger. Підтримує власний формат налагоджувальної інформації.

AFD. Створений у 1988 році і надає такі можливості: покроковий режим виконання інструкцій, покрокове виконання підпрограм, збереження точок зупину у файлі користувача, пошук даних у пам'яті, створення макросів і запис їх у файл.

GameTools. Основне призначення – злам захистів у програмах, що вимагають RealMode для роботи, злам ігор “на життя”. Можливості цього налагоджувача такі: умовні точки зупину на переривання, робота разом із зовнішнім дебагером, невеликий об’єм необхідної пам'яті, зміна швидкості

таймера (включаючи зупинку), вивантаження пам'яті на диск із можливістю її подальшого порівняння з поточною, установка констант в пам'яті.

Quaid Analizer. Це інструмент для аналізу незнайомих програм; містить дуже зручні можливості трасування по перериваннях і використання будь-якого переривання для проставляння контрольних точок.

Налагоджувачі захищеного режиму

В дану групу входять налагоджувачі, що одержали найбільше поширення останнім часом. Вони надають набагато більше можливостей, ніж налагоджувачі з попередньої групи. З одного боку, налагоджувачі з даної групи залишають набагато менше слідів, по яких їх можна ідентифікувати. Наприклад, для установлення контрольної точки в тілі виконуваної програми використовується не стандартне переривання контрольної точки INT 3, яке можна легко знайти, а апаратне переривання по виконанню інструкції. З іншого боку, додаткові можливості у виді переривань для доступу до пам'яті чи до портів введення/виведення роблять такі налагоджувачі дуже могутнім інструментом для реінжинірингу.

SoftIce – це один з наймогутніших налагоджувачів. Характерні особливості його такі: він не є цілком stealth-налагоджувачем, оскільки залишає частину свого коду у пам’яті; існує API між програмою й налагоджувачем; підтримує налагоджувальну інформацію Microsoft ('NB' на початку налагоджувальної інформації), Borland (db0FB52h).

3.2 Захист від налагоджувачів реального режиму

Захиститися від дослідження під налагоджувачем можна двома шляхами:

- тим або іншим методом *виявити налагоджувач* і передати керування на деяку гілку реакції на налагоджувач;
- *“забруднити” програму* фрагментами коду, які нормально виконуються без налагоджувача, але під налагоджувачем приводять до аварійного завершення, зависання комп'ютера або перекручуванню ходу виконання програми.

3.2.1 Виявлення налагоджувача реального режиму

Налагоджувачі реального режиму досить просто виявити. Можна виділити дві основні групи методів їх виявлення:

- використання апаратних особливостей процесора, зокрема наявність черги команд, а також втрата трасувального переривання після виконання деяких інструкцій, наприклад зміни вмісту сегментних регістрів по командах MOV або POP;
- виявлення змін операційного середовища шляхом перевірки векторів переривань, перевірки часу виконання окремих ділянок програми, перевірки початкових станів регістрів при запуску програми і т. п.

Налагоджувачі у своїй роботі використають такі ресурси комп'ютера:

1) налагоджувальне переривання INT 1 – трасувальне переривання або переривання покрокової роботи;
2) налагоджувальне переривання INT 3 – переривання контрольної точки;
3) прапор трасування TF.

Все це може застосовується для виявлення несанкціонованого дослідження захищених програм під налагоджувачем.

1. ***Виявити установлення прапора трасування TF***. Справа в тому, що процесори Intel 8086 "втрачають" трасування однієї команди, якщо попередня команда змінювала значення сегментного регістра. Тому можна виявити установлення прапора трасування TF у процесі налагодження, наприклад, так.

```
mov  ax, ss
push ax
pop  ss
pushf
pop  ax
pushf
pop  bx
sub  ax, bx
mov  bx, OFFSET BeingDebugged
mov  [bx], ax
```

2. Варіацією цього методу є ***"ловля" налагоджувача на командах (префіксах) переустановлення сегмента***.

```
cs:
   pushf
   pop  ax
   pushf
   pop  bx
   sub  ax, bx
   mov  bx, OFFSET BeingDebugged
   mov  [bx], ax
```

При покроковому трасуванні даних фрагментів, наприклад, у середовищі TD змінній `BeingDebugged` буде присвоєно значення 100h. Однак цей метод не гарантує 100%-ї ймовірності виявлення налагоджувача, оскільки фрагмент може бути пройдений не по кроках, а відразу (команда Go to cursor / F4 або Step over / F8).

Крім того, хороші налагоджувачі реального режиму переривання INT 1 не використовують – тобто метод не працює, а з деякими слабшими налагоджувачами це також не проходить, оскільки вони використовують INT 1, але прапорець TF взагалі не спрацьовує. Тому треба застосовувати ще й інші методи виявлення налагоджувачів.

3. Самий очевидний з них – ***перевірка байта, що знаходиться за вектором переривання INT 3*** (можна і INT 1, але це менш надійно, оскільки гарні налагоджувачі реального режиму INT 1 взагалі не перехоплюють). Вектор краще одержувати безпосередньо з таблиці векторів переривань, а не викликом функції 35h переривання 21h.

```
xor  ax, ax
```

```
mov   es, ax
mov   bx, 0Ch
xor   dh, dh
mov   dl, BYTE PTR es:[bx]
mov   bx, OFFSET BeingDebugged
mov   ax, [bx]
sub   dl,0CFh     ; код команди IRET
add   ax, dx
mov   [bx], ax
```

Після виконання цього фрагмента під налагоджувачем реального режиму (навіть таким хорошим, як InSight!) змінна `BeingDebugged` буде мати ненульове значення. Тепер у будь-якому місці програми можна порівняти значення з нулем і, якщо воно не дорівнює нулю, реагувати на налагоджувач.

4. ***Виявлення переривання INT 3***. Цей метод виявлення налагоджувачів реального режиму оснований на реалізації в останніх механізмах точок зупину по INT 3. Коли в такому налагоджувачі ставлять точку зупину, байт за цією адресою заміщається на однобайтову команду INT 3 (код операції 0CCh). Відповідно, програма може виявити ці команди, виставити прапор виявлення налагоджувача (у наших прикладах `BeingDebugged`) і зреагувати на налагоджувач при перевірці цього прапора. Способів виявлення цих команд (і взагалі перекручування коду, оскільки є налагоджувачі, що ставлять замість INT 3 виклик іншого переривання) можна придумати багато – починаючи з пошуків байта 0CCh командою SCASB і закінчуючи знаходженням різних контрольних кодів цілісності із захищуваним фрагментом коду в якості аргументу.

5. ***Використання буфера передвибірки команд.*** На процесорах сімейства 486 можна виявити налагоджувач, використовуючи буфер передвибірки команд. Зміна коду команди, що вже обрана й перебуває в цій черзі, ніяк не вплине на хід виконання програми. Під налагоджувачем же черга передвибірки постійно скидається, і виконується змінена команда.

```
      mov   BYTE PTR Critical, 0F9h
            ; Код операції stc
Critical:
       clc
      jnc   Normal
      mov   bx, OFFSET BeingDebugged
      mov   [bx], 1
Normal:     ; Нормальне виконання програми
```

Тут перша команда ніяк не вплине на нормальне виконання програми, оскільки пересилання виконується в оперативну пам'ять, а команди вже перебувають у буфері передвибірки. Під налагоджувачем же виконається змінений код, і прапор налагодження буде виставлений. Однак потрібно пам'ятати, що цей спосіб придатний лише на допоміжну роль, оскільки навіть якщо програма буде виконуватись на вбудованій машині з аналогом 486-го процесора, досліджувати її, можливо, будуть на Pentium. А там конвеєризація реалізована інакше, і буфера передвибірки немає.

6. ***Використання особливостей ініціалізації регістрів налагоджувачами***. Цей метод виявлення налагоджувача можна застосовувати тільки

проти слабких налагоджувачів, таких, як CodeView або Turbo Debugger. Він оснований на тому, що при завантаженні програми певним чином ініціалізуються регістри. При цьому в регістр CX заноситься ненульове значення: у com-програмах – це довжина com-файлу, а в exe-програмах – розмір коду в оперативній пам'яті. Регістр DI встановлюється рівним SP, причому значення SP не дорівнює нулю. Дослідження ж програми під налагоджувачем вимагає кількаразових прогонів. CodeView і TD при першому прогоні програми занулюють регістри AX, BX, CX, DX, SI, DI, BP. При повторному прогоні програми CodeView знову занулює ці регістри, а Turbo Debugger взагалі не торкається сміття, що залишилося після попереднього прогону. Порівнюючи на початку програми значення регістрів з необхідними, можна виявити налагоджувач (наприклад, за умовою CX=0 або DI<>SP) і виставити прапор виявлення або відразу ж перейти на гілку реакції на налагоджувач. Проти налагоджувачів високої якості даний прийом марний.

7. Нарешті, розглянемо метод виявлення налагоджувача, заснований на ***пошуку точки зупину.***

```
        call MyProc
        …
MyProc:
        pop  bx
        push bx
        cmp  [bx], 0CCh ; Перевірка після виклику MyProc
        jz   Debug
        ret
```

3.2.2 Перекручування роботи програми під налагоджувачем реального режиму

Всі способи, описані вище, цілком застосовні для реальних захистів, але в них є істотний недолік: можна взагалі не розбиратися, як програма визначає факт роботи під налагоджувачем. Досить знайти команду переходу на гілку реакції на налагоджувач – і всі наші хитрування стають марними. Тому в реальних захистах їх потрібно доповнювати трюками, що спотворюють роботу програми без явних перевірок. Можна виділити такі методи перекручування ходу виконання програми під налагоджувачем:

- протидія установленню контрольних точок і зміні коду програми, наприклад, періодичною перевіркою контрольних сум різних ділянок програми, чергуванням команд заборони й дозволу переривань і т. п.;
- порушення інтерфейсу з користувачем, наприклад шляхом блокування клавіатури, перекручування виведення на екран і т. п.;
- використання налагоджувальних переривань (а іноді й не тільки налагоджувальних) для реалізації таких дій, як генерація ділянок коду, шифрування, виклик інших підпрограм системи захисту;
- визначення стека в області коду і кількаразова його зміна.

1. Почнемо з ***використання прапора виявлення налагоджувача*** (точніше кажучи, змінної статусу цього виявлення – вона зовсім не зобов'язана містити лише 2 значення – 0 і 1) для перекручування виконання програми.

```
mov   ax, BeingDebugged
shl   ax, 3
push  ax
retn
```

Якщо налагоджувач не виявлений (`BeingDebugged=0`), виконується повернення на наступну за RETN команду. Якщо ж `BeingDebugged<>0`, то виконується перехід на команду по зсуву `CS:AX`, де AX – це перетворене значення `BeingDebugged`. При цьому ми, як правило, потрапляємо в середину деякої послідовності команд або у середину багатобайтової команди, що приводить до зависання DOS і неможливості подальшої роботи.

2. Наступний метод спотворити виконання програми під налагоджувачем – ***додавати значення змінної*** **`BeingDebugged`** ***до зсуву*** в регістрі при непрямих викликах програм.

```
mov   bx, OFFSET SecretRoutine
mov   ax, BeingDebugged
ror   ax, 4
add   bx, ax
call  bx
```

Як бачимо, якщо налагоджувач не виявлений, засекречена підпрограма нормально викликається. Якщо ж його засікли, то зсув буде перекручено, і це, цілком ймовірно, приведе до зависання DOS.

3. Можна ***підмінювати вектори налагоджувальних переривань*** (INT 1, INT 3). Тут відкривається широкий простір для фантазії розроблювача – можна поміняти їх місцями, замінити на вектори INT 19h, INT 20h або будь-який інший вектор, зрушувати сегмент або зсув у цих векторах, модифікувати код оброблювача і т. д. Спроба трасування програми при таких перестановках знову ж приводить до зависання налагоджувача або DOS.

4. Наступний метод – ***блокувати відеопідсистему***, тобто використовувати послідовність асемблерних команд, що призводить до зависання будь-якого налагоджувача.

```
mov   ax, 1201h
mov   bl, 32h
int   10h
```

5. Можна ***використати заборону роботи із клавіатурою:***

```
mov   al, 0ADh
out   64h, al
```

6. Можна ***використати виклики INT 1 і INT 3 у перериванні від таймера,*** а не там, де їх звичайно шукають, і з активацією гілки не відразу, а після деякої тимчасової затримки. Очевидно, цей спосіб конфліктує з перестановкою векторів або заміщенням оброблювачів, і застосовувати їх можна тільки по черзі.

Комбінуючи описані методи, можна побудувати досить надійну систему захисту.

3.3 Боротьба з налагоджувачами захищеного режиму

Деякі налагоджувачі захищеного режиму ловляться на протиналагоджувальні трюки, призначені для боротьби з налагоджувачами реального режиму. Так, SoftIce попадається на прийоми, пов'язані із втратою одного трасувального переривання після команди CS (код операції 2Eh).

```
cs:
pushf
pop  ax
pushf
pop  bx
sub  ax, bx
add  BeindDebugged, ax
```

Інші прийоми проти налагоджувачів захищеного режиму засновані на представлених ними API. Так, DeGlucker, надаючи API по INT 15h (функції 0FFxx), намертво зависає на конструкції:

```
mov  ax, 0FF01h
INT  15h
```

Третя група прийомів боротьби з налагоджувачами захищеного режиму полягає в перекручуванні стану апаратних налагоджувальних засобів. Так, якщо відомо, що для трасування програми застосовується регістр DR1, то можна спотворити або його значення, або значення керуючих бітів у регістрі DR7. Однак, уже є налагоджувачі (DeGlucker 0.05), які самі використовують налагоджувальні регістри, а налагоджуваній програмі використати їх не дають, емулюючи звернення до них.

І нарешті, четверта група прийомів проти налагоджувачів захищеного режиму заснована на використанні помилок конкретних налагоджувачів. Так, SoftIce (принаймні, деякі версії) некоректно обробляє команди звернення до регістра DR7, що дозволяє ловити його на такому, наприклад, фрагменті коду:

```
mov  eax, dr7
or   eax, 2000h
mov  dr7, eax
mov  eax, dr7
test eax, 2000h
jnz  DebudderNotFound    ; гілка нормальної роботи
```

Далі йдуть команди реакції на налагоджувач або, допустимо, взведення прапора такої реакції.

3.4 Додаткові прийоми антиналагоджувального програмування

Прийоми захисту програм від налагоджувачів і дизасемблерів для програмування завдань відповідального цільового призначення, звичайно, гарні. Але, як ми вже відзначали, абсолютно надійних захистів не буває, і оскільки завдання розроблювача захисту – змусити супротивника витратити на злам час, достатній для прийняття контрзаходів, тільки протиналагоджувальними трюками обмежуватися не треба. Не зашкодить також

ускладнити дослідження програми й після того, як супротивник обійде або усуне всі пастки для хакерського інструментарію.

Для цього необхідно зробити програму високочитабельною для нас, але малозрозумілою в дизасеблерному вигляді, а під налагоджувачем – такою, що створює враження хаотичного набору умовних і безумовних переходів. Це завдання вирішується застосуванням так називаного "витонченого" програмування.

Можна виділити декілька основних направлень у цьому підході:

- екзотична, що має незвичайний вид реалізації алгоритмів і використання рідких команд процесора або їх нестандартних сполучень;
- реалізація декількох повністю еквівалентних варіантів того самого алгоритму, при кожному звертанні до якого випадковим чином вибирається один з варіантів його реалізації;
- засмічення коду "сміттям" – командами, що не впливають на обробку наших даних (крім деякого збільшення часу обробки на "засмічених" ділянках).

Розглянемо ці прийоми докладніше.

Екзотична реалізація алгоритмів. Припустимо, у нас є деякий прапор (або змінна), для якого критично важлива перевірка на 0. Однак ми не бажаємо явно писати команду "`CMP AX, 0`" і взагалі, по можливості, хочемо обійтися без команд передачі керування.

Перше, що спадає на думку – використати команди, придатні для неявної перевірки на 0. Наприклад, використати команди двійково-десяткової арифметики.

```
mov   ax, OurFlag
daa
push
pop   ax          ; Неявна перевірка прапора нуля
and   ax, 40h
jz    FlagIsZero
```

Зрозуміло, у реальній програмі одержання прапорів і перевірка наявності прапора нуля повинні бути рознесені для ускладнення дослідження. Однак цей варіант, хоча й застосовний, відносно простий для зламу.

Злам значно ускладнюється, якщо ми встановимо оброблювач нульового значення прапора на int 0 (ділення на 0), а перевірку реалізуємо як ділення будь-чого на значення прапора, завантажене в будь-який припустимий регістр. У цьому випадку ніякого переходу на оброблювач нульового значення прапора немає взагалі, а крім того, деякі налагоджувачі аварійно завершують програму при виконанні ділення на 0.

Зауважимо, що так можна реалізувати й перевірку ненульових значень, віднімаючи їх від значення нашої змінної і ділячи що-небудь на результат. При цьому, мабуть, значення потрібно примусово пересилати в будь-який допустимий регістр, а оброблювач int 0 повинен являти собою реалізацію деякого хеш-перетворення, що повертає індекс масиву адрес точок входу в оброблювачі конкретних значень (або сама адреса).

Реалізація еквівалентних гілок. Для ускладнення дослідження програми під налагоджувачем користь від цього прийому очевидна. Справді, якщо ми, перебуваючи в налагоджувачі, попадаємо то на одну команду, то на іншу, це явно не спростить розуміння алгоритму програми.

Наведемо простий приклад. Операція NEG (перетворення числа в додатковий код) еквівалента виключаючому АБО з "усіма одиницями" і інкремента результату. Напишемо макрос, що реалізує алгоритм із двома можливими гілками його виконання.

```
;===== NEG над 16-розрядним регістром. ==============
XNEG16     MACRO Reg
           local XNEG2, XNEGQ
           push  ax          ; Використовується процедура Random
           call  Random      ; генерація випадкових або
                             ; псевдовипадкових чисел
           cmp   al, 30
           ja    XNEG2
           pop   ax
           neg   Reg
           jmps  XNEGQ
XNEG2:
           xor   Reg, 0FFFFh
           inc   Reg
XNEGQ:
           ENDM
```

Це лише найпростіший приклад, що відносно легко піддається аналізу. Для ускладнення аналізу можна:

- збільшити число гілок (реалізується не для всіх алгоритмів);
- реалізувати гілки у вигляді оброблювачів переривань (INT 1, INT 3, INT 4, INT 6 і т.д.) і звертатися до них не прямо, а шляхом створення відповідних ситуацій (встановленням прапора TF, засиланням команди int 3 на місце заздалегідь поставленого NOP, діленням на 0 і ін.), що до речі, ще й розширить протиналагоджувальну підсистему;
- збільшити число перевірок випадкового числа.

```
;===== NEG над 16-розрядним регістром. ==============
XNEG16     MACRO Reg
           LOCAL XNEG1, XNEG2, XNEGQ
           push  ax    ; Використовується процедура Random
           call  Random      ; Деяка підпрограма
                             ; генерація випадкових або
                             ; псевдовипадкових чисел
           cmp   al, 30
           ja    XNEG2
XNEG1:     pop   ax
           neg   Reg
           jmps  XNEGQ
XNEG2:     pop   ax
           push  ax
           call  Random2     ; Інший генератор
           cmp   ah, 73
           jbe   XNEG1
           pop   ax
           xor   Reg, 0FFFFh
           inc   Reg
XNEGQ:     ENDM
```

Загалом кажучи, в антиналагоджувальному програмуванні немає готових рецептів. Тут все залежить від розроблювача конкретного захисту, від його фантазії й знання використовуваного процесора.

Засмічування коду. Під засмічуванням коду будемо розуміти штучне внесення в нього команд, що не мають відношення до реалізованого алгоритму, що або ускладнює його аналіз, або робить цей аналіз більше стомлюючим відповідно, і при цьому вимагає більше сил і часу.

Мова йде про таке:

- маніпуляції незадіяними регістрами;
- встановлення або скидання деяких прапорів та виконанням декількох команд, що на ці прапори не впливають, з наступним умовним переходом, який насправді свідомо виконується або свідомо не виконується;
- обробка декількох однакових структур, з яких тільки одна містить дані нашого алгоритму і т. д.

Все це не тільки збільшує обсяг дизасембльованого тексту, але й відволікає увагу від захищаючого алгоритму.

Корисно засмічувати код ще й макросами або викликами, що емулюють введення з клавіатури команд SoftIce, TR або комбінацій клавіш розповсюджених налагоджувачів, або навіть просте натискання деяких випадкових клавіш. Очевидно, при цьому потрібно почистити буфер перед викликом будь-якої підпрограми звернення до клавіатури.

Контрольні питання

1. В чому різниця між налагоджувачами реального і захищеного режимів?
2. Наведіть існуючі програми-налагоджувачі реального та захищеного режимів?
3. Яким чином можна виявити присутність в операційній системі налагоджувачів?
4. Які основні переривання використовують налагоджувані?
5. Наведіть додаткові методи захисту від налагоджувачів реального режиму.
6. Які методи захисту від дослідження програм під налагоджувачами захищеного режиму?
7. Яким чином реалізація еквівалентних гілок допоможе протистояти налагоджувачам?
8. В чому сутність захисту за допомогою засмічування коду?
9. Охарактеризуйте налагоджував SoftIce.
10. В чому, на вашу думку, проявляються позитивні і негативні риси налагоджувачів.

4 ВИКОРИСТАННЯ ХУКІВ У WINOWS

4.1 Поняття хуків та фільтруючих функцій

В операційній системі Microsoft Windows ***хуком*** називається механізм перехоплення особливою функцією подій (таких, наприклад, як повідомлення від маніпулятора миші або клавіатури) до того, як вони дійдуть до програми. Ця функція може потім реагувати на події, а в деяких випадках змінювати або відміняти їх. З цього випливає, що підключення хуків призводить до збільшення обсягу роботи системи, тобто робота системи сповільнюється. Отже, хуки повинні встановлюватися в самих крайніх випадках і видалятися відразу ж після того, як потреба в них відпала. Зазвичай обробка повідомлень в операційній системі відбувається так:

Повідомлення → Система

А у випадку застосування хуків це виглядає так:

Повідомлення → Хук_1→ Хук_2→ … → Хук_N → Система

Одне повідомлення може оброблятися декількома хуками, або, інакше кажучи, можуть бути створені ***ланцюжки хуків.*** Якщо в системі виникає повідомлення, асоційоване з якимось хуком, то Windows передає його на обробку першому хуку, потім другому й т.д. І лише після того, як всі хуки відпрацюють, повідомлення надійде за своїм прямим призначенням.

Функції, які одержують повідомлення про події, називаються ***фільтруючими функціями*** і відрізняються за типами перехоплених ними подій. Наприклад, це може бути фільтруюча функція для перехоплення всіх подій миші або клавіатури. Щоб Windows змогла викликати функцію-фільтр, ця функція повинна бути встановлена, тобто, прикріплена до хука (наприклад, до клавіатурного хука). Прикріплення однієї або декількох фільтруючих функцій до будь-якого хука називається ***встановленням хука***. Якщо до одного хука прикріплено декілька фільтруючих функцій, Windows реалізує чергу функцій, причому функція, прикріплена останньою, виявляється на початку черги, а найперша функція – у її кінці.

Якщо до хука прикріплена одна або більше функцій-фільтрів і відбувається подія, що приводить до спрацьовування хука, Windows викликає першу функцію з черги функцій-фільтрів. Ця дія називається ***викликом хука.*** Наприклад, якщо до хука CBT (цей хук відповідає за створення і відкриття вікон) прикріплена функція й відбувається подія, після якого спрацьовує хук (припустимо, відбувається створення вікна), Windows викликає CBT-хук, тобто першу функцію з його черги.

Для того, щоб користуватися хуками, необхідно знати таке:

- які існують типи хуків, що вони можуть робити, і яку інформацію (параметри) вони передають вашій функції;
- як використати функції Windows для додавання й видалення фільтруючих функцій із черги функцій хука;
- які дії повинна виконувати фільтруюча функція, яку встановлюємо.

4.2 Типи хуків

Деякі хуки можуть бути встановлені лише з системною областю дії, а деякі можна встановлювати як для усієї системи, так і для одного потоку, як показано в таблиці 5.

Таблиця 5 – Типи хуків та їх основні характеристики

Тип хука	Код	Область дії	Опис
WH_CALLWNDPROC	4	Потік або вся система	Windows викликає цей хук при будь-якому звертанні до ф-ії SendMessage (до передачі повідомлень віконної функції).
WH_CALLWNDPROCRET	12	Потік або система	Викликається після того, як віконна процедура завершила свою роботу.
WH_GETMESSAGE	3	Потік або система	Windows викликає цей хук перед виходом з функцій GetMessage
WH_CBT	5	Потік або вся система	Використовується для перехоплення повідомлень про створення вікон, переході в активний (неактивний) стан і т.д.
WH_DEBUG	9	Потік або система	Windows викликає цей хук перед викликом будь-якої фільтруючої функції.
WH_GETMESSAGE	3	Потік або система	Windows викликає цей хук перед виходом з функцій GetMessage
WH_JOURNALRECORD	0	Тільки система	Windows викликає цей хук при видаленні події із системної черги. Т.ч., фільтри цього хука викликаються для всіх клавіатурних подій і мишки
WH_MOUSE	7	Потік або вся система	Windows викликає цей хук після виклику функцій GetMessage за умови отримання повідомлення від миші.
WH_KEYBOARD	2	Потік або вся система	Windows викликає цей хук коли функції GetMessage збираються повернути повідомлення WM_KEYUP, WM_KEYDOWN, WM_SYSKEYUP, WM_SYSKEYDOWN або WM_CHAR.
WH_MSGFILTER	-1	Потік або вся система	Windows викликає цей хук, коли діалогове вікно, інформаційне вікно, смуга прокручування або меню отримують повідомлення, або коли користувач натискає комбінацію ALT+TAB (або ALT+ESC) при активному додатку, що встановив цей хук. Даний хук встановлюється для конкретного потоку
WH_SYSMSGFILTER	6	Тільки система	Ідентичний хуку WH_MSGFILTER, але що він має системну область видимості.
WH_SHELL	10	Потік або вся система	Windows викликає цей хук при певних діях з вікнами верхнього рівня – top-level windows (тобто, з вікнами, що не мають власника).

Для будь-якого даного типу хука, першими викликаються хуки потоків, і тільки потім системні хуки.

Є декілька причин, по яких краще використати потокові хуки замість системних. Хуки потоків мають такі переваги:

- не створюють проблем роботі додатків, не зацікавлених у виклику хука;
- не поміщають всі події, що ставляться до хука, у чергу (так, щоб вони надходили не одночасно, а одне за іншим). Наприклад, якщо додаток встановить клавіатурний хук для всієї системи, то всі клавіатурні повідомлення будуть пропущені через фільтруючу функцію цього хука, залишаючи невикористаними системні можливості багато-потокової обробки вводу. Якщо ця функція припинить обробляти клавіатурні події, система буде виглядати завислою, хоча насправді й не зависне. Користувач завжди зможе використати комбінацію CTRL+ALT+DEL для того, щоб вийти із системи (log-out) і вирішити проблему, але йому це навряд чи сподобається. До того ж, користувач може не знати, що подібну ситуацію можна вирішити, увійшовши в систему під іншим ім'ям (log-out/log-in);
- не вимагають знаходження функції-фільтра в окремій DLL. Всі системні хуки та хуки потоків в іншому додатку повинні бути в DLL;
- їм не потрібно розділяти дані між DLL, завантаженими в різні процеси. Фільтруючі функції із системною областю видимості, які зобов'язані перебувати в DLL, повинні до того ж розділяти необхідні дані з іншими процесами. Тому що таке поводження не є типовим для DLL, ви повинні вживати спеціальних заходів обережності при реалізації системних фільтруючих функцій. Якщо функція-фільтр не вміє розподіляти дані й неправильно використає дані в іншому процесі, цей процес може звалитися.

4.3 Встановлення та зняття хуків

Для встановлення й доступу до фільтруючих функцій Windows-програм використовують функції, описані в заголовному файлі "winuser.h": SetWindowsHookEx(), CallNextHookEx() і UnhookWindowsHookEx().

SetWindowsHookEx()

Функція SetWindowsHookEx() додає функцію-фільтр до хуку. Ця функція приймає чотири аргументи.

```
WINUSERAPI HHOOK SetWindowsHookEx (
                    int idHook, HOOKPROC lpfn,
                    HINSTANCE hmod, DWORD dwThreadId);
```

Параметри цієї функції такі:

idHook – цілочисельний код, що описує хук, до якого буде прикріплена фільтруюча функція;

lpfn – адреса функції-фільтра, яка вказує на процедуру хука (ця процедура повинна мати визначену кількість та типи аргументів, а призначення аргументів цієї функції залежить від типу хука);

hmod – хендл модуля, що містить фільтруючу функцію. В Win32 (на відміну від Win16), цей параметр повинен бути NULL, якщо процедура хука находиться всередині коду поточного процесу. При встановленні хука для всієї системи або для потоку в іншому процесі, потрібно використати хендл DLL, що містить функцію-фільтр.

dwThreadId – ідентифікатор (номер) потоку, для якого встановлюється хук. Якщо цей ідентифікатор ненульовий, установлена фільтруюча функція буде викликатися тільки в контексті зазначеного потоку Якщо ідентифікатор дорівнює нулю, установлена функція має системну область видимості й може бути викликана в контексті будь-якого потоку в системі. Додаток або бібліотека можуть використати функцію GetCurrentThreadId() для одержання ідентифікатора поточного потоку.

Функція SetWindowsHookEx() повертає хендл встановленого хука (типу HHOOK) або встановлює код останньої помилки в одне з наступних значень для індикації невдалого завершення функції:

ERROR_INVALID_HOOK_FILTER – невірний код хука;

ERROR_INVALID_FILTER_PROC – невірна фільтруюча функція;

ERROR_HOOK_NEEDS_HMOD – глобальний хук встановлюється з параметром *hInstance*, рівним NULL або локальний хук встановлюється для потоку, який не належить даному додатку;

ERROR_GLOBAL_ONLY_HOOK – хук, який може бути лише системним, встановлюється як потоковий;

ERROR_INVALID_PARAMETER – невірний ідентифікатор потоку;

ERROR_JOURNAL_HOOK_SET – для реєстраційного хука (journal hook) вже встановлена фільтруюча функція. У будь-який момент часу може бути встановлений тільки один записуючий або відтворюючий хук. Цей код помилки може також означати, що додаток намагається встановити реєстраційний хук у той час, як запущений зберігач екрану. Цей код помилки може також означати, що додаток намагається встановити реєстраційний хук у той час, як запущений зберігач екрану;

ERROR_MOD_NOT_FOUND – параметр *hInstance* у випадку, коли хук є глобальним, не посилається на бібліотеку. (Насправді, це значення означає лише те, що модуль User не зміг виявити даний хендл у списку модулів);

будь-яке інше значення – система безпеки не дозволяє встановити даний хук, або в системі закінчилась пам'ять.

UnhookWindowsHookEx()

Після того, як потреба у використання хуку відпала, його треба деінсталювати функцією UnhookWindowsHookEx(). Ця функція приймає хендл хука, отриманий від SetWindowsHookEx() і повертає логічне значення, що показує успіх операції (на даний момент завжди повертає TRUE).

```
WINUSERAPI BOOL WINAPI UnhookWindowsHookEx (HHOOK hook).
```

Коли хук уже встановлений, Windows викликає першу функцію в черзі, і на цьому її відповідальність закінчується. Після цього функція відповідальна за те, щоб викликати наступну функцію в ланцюжку.

CallNextHookEx()

В Windows є функція ***CallNextHookEx()*** для виклику наступного фільтру в черзі фільтрів. Ця функція приймає чотири параметри.

```
CallNextHookEx( hHook, nCode, wParam, lParam);
```

Перший параметр *hHook* – це значення, повернене при встановленні хуку функцією SetWindowsHookEx(). У цей час Windows ігнорує це значення, але в майбутньому це може змінитися. Наступні три параметри – *nCode*, *wParam*, та *lParam* – Windows передає далі по ланцюжку функції.

Windows зберігає у своїх внутрішніх структурах ланцюжок фільтруючих функцій і стежить за тим, яка функція викликається в даний момент. При виклику CallNextHookEx() Windows визначає наступну функцію в черзі й викликає її.

4.4 Опис функції-фільтру

Функція-фільтр повинна описувати дії, які необхідно виконувати при перехопленні будь-якого повідомлення.

```
… HookFunc (int nCode, WORD wParam, DWORD lParam).
```

Призначення аргументів цієї функції залежить від типу хука.

Фільтри можуть обробити повідомлення, але не можуть змінювати його (хоча це було можливо в Win16).

У системі існують такі фільтруючі функції:

```
CallMsgFilter()
CallWndProc()
CallWndRetProc()
CBTProc()
ForegroundIdleProc()
GetMsgProc()
JournalPlaybackProc()
JournalRecordProc()
DebugProc()
KeyboardProc()
LowLevelKeyboardProc()
LowLevelMouseProc()
MessageProc()
MouseProc()
ShellProc()
SysMsgProc()
```

4.5 Можливості хуків

Хуки надають потужні можливості для додатків Windows. Використовуючи хуки, додатки можуть:

- обробляти або змінювати всі повідомлення, призначені для всіх діалогових вікон (dialog box), інформаційних вікон (message box), смуг прокручування (scroll bar), або меню одного додатка (WH_MSGFILTER);
- обробляти або змінювати всі повідомлення, призначені для всіх діалогових вікон, інформаційних вікон, смуг прокручування, або меню всієї системи (WH_SYSMSGFILTER);
- обробляти або змінювати всі повідомлення в системі, одержувані функціями GetMessage() або PeekMessage() (WH_GETMESSAGE);
- обробляти або змінювати всі повідомлення (будь-якого типу), що посилають викликом функції SendMessage() (WH_CALLWNDPROC);
- обробляти, змінювати і видаляти клавіатурні події (WH_KEYBOARD);
- обробляти, змінювати або скасовувати події миші (WH_MOUSE);
- реагувати на певні дії системи, уможливлюючи розробку додатків комп'ютерного навчання – computer-based training (WH_CBT);
- запобігати виклику іншої функції-фільтра (WH_DEBUG);
- забезпечувати підтримку кнопки F1 для меню, діалогових й інформаційних вікон (WH_MSGFILTER);
- забезпечувати запис і відтворення подій миші й клавіатури. Так, програма Windows Recorder використовує хуки WH_JOURNALRECORD, WH_JOURNALPLAYBACK для запису і відтворення;
- стежити за повідомленнями, щоб визначити, які з них призначені певному вікну або які дії вони генерують. Так, утиліта Spy з Win32T SDK for Windows NTT використовує для цих цілей хуки WH_GETMESSAGE, WH_CALLWNDPROC (вихідні тексти Spy можна знайти в SDK);
- симулювати введення з миші та клавіатури (WH_JOURNALPLAYBACK). Хуки – єдиний надійний спосіб симуляції цих дій. Якщо спробувати імітувати їх через посилання повідомлень, не буде відбуватися відновлення стану клавіатури або миші у внутрішніх структурах Windows, що може привести до непередбаченого поводження. Якщо використовуються хуки, ці події обробляються в точності так, як і дійсне введення із клавіатури або миші. Так, Microsoft Excell використовує хуки для реалізації макрофункції SEND.KEYS;
- зробити можливим використання CBT додатками Windows (WH_CBT). Хук WH_CBT значно полегшує розробку CBT-додатків.

Найпоширеніше застосування подібна технологія знаходить у троянських програмах. Однак, хуки можна застосувати у побудові систем захисту програм від небажаного впливу. Наприклад, можна створити невидимий процес, приховати які-небудь файли на диску, сховати запис в реєстрі та приховати мережні з'єднання.

Зламники також використовують хуки для несанкціонованого доступу, налагодження та модифікації захищених програм.

Виклики WIN32, які найчастіше використовуються при зламах, знаходяться у бібліотеках gdi32.dll, kernel32.dll, user32.dll :

```
ReadFile, WriteFile, SetFilePointer, GetSystemDirectory,
GetSystemDirectoryA, GetPrivateProfileStringA,
GetPrivateProfileIntA, WritePrivateProfileStringA,
WritePrivateProfileIntA.
```

Також часто використовуються хуки і при перехопленні викликів, що призначені для створення і видалення ключів, читання, відкриття і закриття ключів реєстру:

```
RegCreateKey, RegCreateKeyA, RegDeleteKey, RegDeleteKeyA,
RegQueryValue, RegQueryValueA, RegCloseKey, RegCloseKeyA,
RegOpenKey, RegOpenKeyA.
```

При роботі з елементами керування діалогових вікон, вікон повідомлень та інших можливостей відображення тексту хуки використовують для перехоплення таких повідомлень:

```
GetWindowText, GetWindowTextA, GetDlgItemText,
GetDlgItemTextA, GetDlgItemInt, MessageBox, MessageBoxA,
MessageBoxExA, MessageBeep, SendMessage, WsPrintf,
```

Хуки використовують і при роботі з функціями отримання системного часу і дати:

```
GetSystemTime, GetLocalTime, SystemTimeToFileTime,
```

при роботі з вікнами:

```
CreateWindow, CreateWindowExA, ShowWindow, BitBlt,
```

та при роботі з зовнішніми пристроями:

```
GetDriveType, GetDriveTypeA, GetLogicalDrives,
GetLogicalDrivesA, GetLogicalDriveStrings,
GetLogicalDriveStringsA.
```

4.6 Приклади застосування хуків

Використання хука WH_CALLWNDPROC

Нехай маємо найпростішу програму, у якій є головне вікно й ми хочемо перехоплювати всі повідомлення, які надходять цьому вікну, і записувати їх у файл. Наведемо фрагменти програми.

```
     //  опис змінних і головна функція додатка
…
HHOOK hHook;
…
     //  віконна функція
{
   switch(msg)
   {     case WM_CREATE:

     hFile=CreateFile("asd",GENERIC_WRITE,0,NULL,CREATE_ALWAYS,
```

```
                              FILE_ATTRIBUTE_NORMAL,0);
          hHook=SetWindowHookEx(WH_CALLWNDPROC,
                              (HOOKPROC) MyHookProc,
                              NULL,GetCurrentThreadId());
          return 0;
     …
     case WM_DESTROY:
          UnhookWindowHookEx(hHook);
          CloseHandle(hFile);
          PostQuitMessage(0);
          return 0;
     …
}
…
     // функція-фільтр обробки хука
LRESULT CALLBACK MyHookProc(int nCode,
                                   WPARAM wParam, LPARAM lParam)
{
     char buff[256];

     …
     if (nCode<0)
          return CallNextHookEx(hHook,nCode,wParam,lParam);
     else
     {    if (nCode==HC_ACTION)
          {
               Sprintf(buff,,"nCode – %08x", nCode);
               WriteFile(hFile,buff,…);
               ……
               CallNextHookEx(hHook,nCode,wParam,lParam);
          }
     }
}
…
```

Проаналізуємо цей фрагмент коду.

Якщо перший аргумент функції MyHookProc() приймає значення HC_ACTION, процедура хука повинна обробляти повідомлення, інакше – передати повідомлення на обробку наступних хуків ланцюжку хуків за допомогою CallNextHookEx().

Якщо другий аргумент не дорівнює 0, то повідомлення відіслане поточним процесом, інакше він дорівнює 0.

Третій аргумент – покажчик на структуру:

```
struct CWPSTRUCT
{
   LPARAM  lParam;  // беремо у повідомлення, що перехоплює
   WPARAM  wParam;  // беремо у повідомлення, що перехоплює
   DWORD   message; // номер повідомлення
   HWND    hwnd;    // хендл вікна, якому відправлено
                    // повідомлення
};
```

Використання хука WH_CBT

Хуки цього типу використовують для того, щоб перехопити повідомлення про створення вікон, переході в активний стан, видалення, зміну розмірів, мінімізації й т.д. Ці хуки одержують керування перед виконанням системних команд, перед видаленням подій від мишки або клавіатури із системних черг. Звичайно ці хуки використовуються в системах навчання із застосуванням комп'ютерів.

Функція-фільтр для обробки цього хука теж містить три параметри. Як перший параметр можуть використовуватись наступні значення:

HCBT_ACTIVATE (5) – вікно готується до активації;
HCBT_CREATEWND (3) – вікно готується до створення;
HCBT_DESTROYWND (4) – вікно готується до видалення;
HCBT_MINMAX (1) – вікно готується до мінімізації або максимізації;
HCBT_MOVESIZE (0) – вікно готується до переміщення або зміни розмірів;
HCBT_SYSCOMMAND (8) – до видачі готується системна команда;
HCBT_CLICKSKIPPED (6) – із системної черги буде видалено повідомлення від мишки;
HCBT_KEYSKIPPED (7) – із системної черги буде видалено повідомлення від клавіатури;
HCBT_SETFOCUS (9) – вікно готується одержати клавіатурний фокус;
HCBT_QS (2).

Інтерпретація другого й третього аргументів залежить від того, що передано як перший аргумент. Наприклад, якщо перший аргумент дорівнює 5, другий – хендл вікна, третій – об'єкт, що викликав активацію. Якщо перший аргумент дорівнює 6, то другий – номер повідомлення мишки, третій – структура, у якій зазначені координати мишки, хендл вікна, місцезнаходження мишки (над меню, на лівій границі вікна й т. д.).

Контрольні питання

1. Дайте поняття хуків. Наведіть основні групи хуків та їх основні характеристики.
2. За якою схемою працюють хуки?
3. Наведіть коло задач, де доцільність їх застосування.
4. Які існують типи хуків, що вони можуть робити?
5. Як використати функції Windows для додавання й видалення фільтруючих функцій із черги функцій хука?
6. Що таке фільтруюча функція?
7. Які дії повинна виконувати фільтруюча функція?
8. Наведіть приклади використання у програмах перехоплень повідомлень за допомогою встановлення у них хуків.

5 СУЧАСНІ ТЕХНОЛОГІЇ ДАМПІНГА І ЗАХИСТУ ВІД НЬОГО

Як відомо, одним із найпоширеніших механізмів захисту ПЗ є використання навісного захисту. Основна ідея такого захисту – ускладнити аналіз роботи програми за допомогою шифрування коду програми і розшифровування його безпосередньо перед виконанням. Такий захист забезпечує відносно стійкий та дешевий захист. При його використанні оригінальний код програми шифрується, модифікується PE-заголовок, до програми додається розшифровувач. Перед виконанням програма розшифровується, частково відновлюється оригінальний PE-заголовок, і керування передається програмі. Але такий захист, як і будь-який інший, не є гарантією того, що програма буде повністю і назавжди захищеною. І одним з методів зняття захистів такого виду є використання дамперів, тобто програм, які можуть зберегти дамп пам’яті під час роботи програми в момент закінчення спрацьовування захисту.

Зняття такого захисту проходить в три етапи:

– знаходження моменту передачі керування оригінальній програмі;
– знімання дампу пам’яті програми;
– відновлення PE-заголовку.

З цього можна зробити такий висновок: посилити захист можна, максимально ускладнивши для хакера виконання цих операцій. Оскільки предметом даного розділу є дослідження можливості зняття образу програми з пам’яті, доцільно коротко розглянути деякі теоретичні аспекти, необхідні для глибинного розуміння цього питання.

Дамп пам'яті – це копія вмісту оперативної пам'яті, що знаходиться на жорсткому диску або іншому енергонезалежному пристрої пам'яті. Природно, дампом може бути не вся оперативна пам'ять, а тільки якась певна її частина, яка, так би мовити, цікавить в даний момент ту програму, яка робить цей дамп.

Дамп пам'яті, як правило, створюється під час різних збоїв в програмному забезпеченні (наприклад, в операційній системі), які призводять до його краху, але дозволяють запустити ту частину програми, яка призначена для збору інформації про причини збою. Власне, найбільш істотна область застосування дампа пам'яті як раз і полягає у зборі інформації про причини фатальних для програмного забезпечення збоїв у його власній роботі.

Дампи пам'яті можуть створюватися самими різними програмами – починаючи від невеликих утиліт, які копіюють на жорсткий диск тільки ту частину пам'яті, де знаходиться їх власний робочий код, до операційної системи, яка може скидати на диск весь вміст оперативної пам'яті. Також цим можуть займатися і антивіруси – дампи пам'яті потім, по ідеї, повинні відсилатися антивірусним компаніям, які будуть шукати в них тіло вірусу. Але на практиці це реалізується порівняно рідко, оскільки дамп пам'яті, що має великий обсяг, непросто пересилати через Інтернет.

5.1 Порядок завантаження програми і виділення пам’яті процесу

Як відомо, Windows – 32 або 64-розрядна операційна система. Перш за все, це означає, що запущена на виконання програма у 32-розрядній Windows може адресувати лінійний адресний простір розміром 2^{32} байтів (4 ГБ), при цьому адресація проводиться за допомогою 32-розрядних регістрів-покажчиків. Кожен запущений в системі процес володіє своїм власним адресним простором. Адресний простір одного процесу не перетинається з адресними просторами інших процесів. Програма може адресувати будь-який елемент пам'яті в діапазоні адрес свого адресного простору. Адресний простір в Windows – віртуальний, він не пов'язаний безпосередньо з фізичним простором оперативної пам'яті комп'ютера.

Механізм виділення пам'яті в Windows складається з двох фаз.

Перша фаза виділення пам'яті полягає в резервуванні ділянки необхідного розміру в адресному просторі процесу. При цьому не виділяється реальна пам'ять. Базова адреса реально зарезервованої ділянки пам'яті кратна 64 Кбайтам. При резервуванні ділянки в адресному просторі можна вказати бажаний атрибут (за нього відповідає певне поле у заголовку виконуваного файлу), який регулює доступ до цієї пам'яті: запис даних, читання даних, виконання коду або комбінацію цих ознак. Порушення правил доступу до пам'яті приводить до генерації системою виключення.

Друга фаза виділення пам'яті в Windows – це виділення реальної, фізичної пам'яті. Мінімальний блок реальної пам'яті, яким оперує система і який можна виділити, – це сторінка пам'яті. Розмір сторінки залежить від типу операційної системи і складає для Windows NT 8 Кбайтів. Виділення реальної пам'яті відбувається посторінково. Кожній сторінці може бути призначений свій власний атрибут доступу.

Віртуальний адресний простір кожного процесу розбивається на розділи (табл.6). Їх розмір і призначення якоюсь мірою залежать від конкретного ядра Windows.

Таблиця 6 – Розділи адресного простору процесу

Розділ	32-розрядна Windows 2000	64-разрядна Windows 2000
Для виявлення нульових покажчиків	0x00000000	0x00000000 00000000
	0x0000FFFF	0x00000000 0000FFFF
Для коду та даних користувача	0x00010000	0x00000000 00010000
	0x7FFEFFFF	0x000003FF FFFEFFFF
Закритий, розміром 64 Кбайти	0x7FFF0000	0x000003FF FFFF0000
	0x7FFFFFFF	0x000003FF FFFFFFFF
Для коду та даних ядра	0x300000000	0x00000400 00000000

Розділ для виявлення нульових покажчиків. Цей розділ адресного простору резервується для виявлення нульових покажчиків. Будь-яка спроба читання або запису в пам'ять по цих адресах викликає порушення доступу.

Розділ для коду і даних користувача. У цьому розділі розташовується закрита частина адресного простору процесу. Жоден процес не може отримати доступ до даних іншого процесу, що розміщені в цьому розділі. Основний обсяг даних, що належать процесу, зберігається саме тут.

Закритий розділ розміром 64 Байти. Цей розділ заблокований, і будь-яка спроба звернення до нього приводить до порушення доступу. Microsoft резервує цей розділ спеціально для того, щоб спростити внутрішню реалізацію дій операційної системи.

Розділ для коду і даних ядра. У цей розділ поміщається код операційної системи, зокрема драйвери пристроїв і код низькорівневого управління потоками, пам'яттю, файловою системою, мережевою підтримкою.

При запуску будь-якого додатку ОС виконує такі дії:

- функція *CreateProcess()* відшукує вказаний їй EXE-файл;
- створює новий об'єкт ядра "процес";
- створює адресний простір нового процесу; резервує регіон адресного простору – такий, щоб в нього помістився даний EXE-файл. Бажане розташування цього регіону вказується усередині самого EXE-файлу – поле *ImageBase* PE-заголовка (за замовчуванням базова адреса EXE-файлу – 0x00400000);
- спроектувавши EXE-файл на адресний простір процесу, система звертається до розділу EXE-файлу зі списком DLL, що містять необхідні функції. Після цього система, викликаючи *LoadLibrary()*, по черзі завантажує вказані (а при необхідності і додаткові) DLL-модулі;
- резервує регіон адресного простору – такий, щоб в нього міг поміститися заданий DLL-файл. Бажане розташування цього регіону указується усередині самого DLL-файлу;
- після завантаження EXE- і DLL-файлів з адресним простором процесу починає виконуватися стартовий код EXE-файлу.

Як було сказано у попередніх розділах, будь-який образ EXE- або DLL-файлу складається з групи розділів (їх перелік знаходиться у таблиці розділів виконуваного файлу). Наприклад, при компіляції програми код поміщається в розділ *.text*, неініціалізовані дані – в розділ *.bss*, а ініціалізовані – в розділ *.data*.

5.2 Доступ до пам’яті та списку процесів

В операційній системі Windows визначені дві основні функції для доступу до пам’яті процесу:

```
ReadProcessMemory() та WriteProcesssMemory().
```

Вони базуються на більш низькорівневих функціях:

```
NativeAPI(),
ZwReadVirtualMemory(),
ZwWriteVirtualMemory(),
```

що визначені в модулі <*ntdll.dll*>.

Прототипи цих функцій мають такий вигляд:

```
BOOL ReadProcessMemory (
     HANDLE  hProcess,          // дескриптор процесу
     LPCVOID lpBaseAddress,     // адреса початку зчитування
     LPVOID  lpBuffer,          // покажчик на буфер
     SIZE_T  nSize,             // розмір зчитуваної області
     SIZE_T* lpNumberOfBytesRead);// к-сть зчитаних байтів

ZwReadVirtualMemory (
     IN_HANDLE ProcessHandle,   // дескриптор процесу
     IN_PVOID  BaseAddress,     // адреса початку зчитування
     OUT_PVOID Buffer,          // покажчик на буфер
     IN_ULONG  BufferLength,    // розмір зчитуваної області
     OUT_PULONG ReturnLength OPTIONAL
     );                         // кількість зчитаних байтів
```

Як бачимо, функції ідентичні за своїми параметрами, і потребують покажчика процесу. Його можна отримати за допомогою функцій *OpenProcess()* або *ZwOpenProcess()* з ідентичними параметрами:

```
HANDLE OpenProcess(
     DWORD dwDesiredAccess,     // ідентифікує вид доступу
               // (напр., PROCESS_VM_READ для зчитування)
     BOOL bInheritHandle,
     DWORD dwProcessId    //ід-тор процесу. Його можна
               //отримати за допомогою диспетчера задач
     );
```

Після отримання дескриптора можна читати пам’ять процесу, за умови, що сторінки пам’яті на мають атрибутів доступу PAGE_GUARD або PAGE_NOACCESS.

Отже для зняття дампу процесу, зламнику потрібно знати його ідентифікатор (PID), а потім за ідентифікатором визначити дескриптор.

Для визначення ідентифікатора користуються функціями *NativeAPI*() або *ZwQuerySystemInformation()*, за допомогою яких отримаємо список процесів у системі. Наприклад, для цього можна використати такий код:

```
pBuffer = AllocMem(ReturnLength); // резервуємо буфер
     // далі заносимо в буфер інформацію про процеси
ret = ZwQuerySystemInformation
      (SystemProcessesAndThreadsInformation,
       pBuffer, ReturnLength, ReturnLength);
    //далі просто скануємо буфер і беремо необхідні данні.
```

Функція *ZwQuerySystemInformation()* дає досить розгорнуту інформацію про систему, в тому числі і інформацію про процеси, тому для використання цієї функції треба описати шість структур.

5.3 Отримання дампу пам’яті обраного процесу

Частина програми, що безпосередньо зчитує пам’ять, може бути реалізована у вигляді зовнішнього модуля – динамічної бібліотеки. Це значно спрощує код програми та дозволяє у майбутніх версіях змінити модуль зчитування на більш складний без значних змін у програмі. Тож розглянемо його більш детально.

```
                    // отримуємо дескриптор процесу
hProc=OpenProcess(PROCESS_VM_READ,FALSE,dwProcessId );
```

```
if (!hProc)
{
   lstrcpy(szErrorStr,"Error - querying process handle!");
   return FALSE;      // виводимо помилку, якщо отримати
                      // дескриптор не вдалося
}
```

Отримавши дескриптор, спробуємо прочитати всю необхідну пам'ять одразу:

```
bRet=ReadProcessMemory(hProc,pStartAddr,pDumpedBytes,
                       dwcBytes,&cb );
CloseHandle(hProc);
if (bRet) goto TidyUp; // якщо прочитали, виходимо
```

Якщо ж зчитати не вдалося, це означає що деякі сторінки пам'яті в заданому діапазоні мають атрибути захисту, що не дозволяють зчитувати їх. Тож будемо зчитувати пам'ять по сторінках перевіряючи атрибути доступу:

```
#define f  minfo.Protect
           // Отримуємо атрибути доступу в структурі minfo;
bRet=VirtualQueryEx(hProc,minfo.BaseAddress,
                    &minfo,sizeof(minfo));
           // Далі перевіряємо поле Protect в minfo.
if (((f&PAGE_GUARD)!=0)||((f&PAGE_NOACCESS)!=0))
     {     memset(
           MakePtr(PVOID,pDumpedBytes,dwcBytes-cb2Do),
           0, dwBlockSize);
           cbFailure += dwBlockSize;
           // Якщо не маємо доступу до сторінки, заповнюємо
           // нулями відповідну частину вихідного буферу
     }
     else
     {  // Якщо маємо доступ до сторінки, то зчитуємо її.
        // Якщо зчитати не вдалось - заповнюємо нулями
        // відповідну  частину вихідного буферу.
     }
```

Реалізований метод отримання дампу програми підійде лише для випадку, коли програма не захищена за допомогою драйверів, що забороняють доступ до пам'яті залежно від того, звідки іде звернення на читання/запис.

5.4 Програми для знання дампу і захист від них

На сучасному ринку програмного забезпечення існує ряд програм для зняття дампу пам'яті. Розглянемо одну з найвідоміших програм для здійснення дампінгу і не тільки для цього. PE Tools – повнофункціональна утиліта для роботи з PE-файлами. Вона включає в себе такі складові: редактор PE-файлів, Task Viewer, оптимізатор Win32 PE-файлів, детектор компілятора/пакувальника та багато чого іншого. Головною її особливістю є те , що вона є автономним дампером, тоді як SoftIce та OllyDbg – ні.

Встановлюється програма дуже просто і має зручний для користувача інтерфейс (рис. 13).

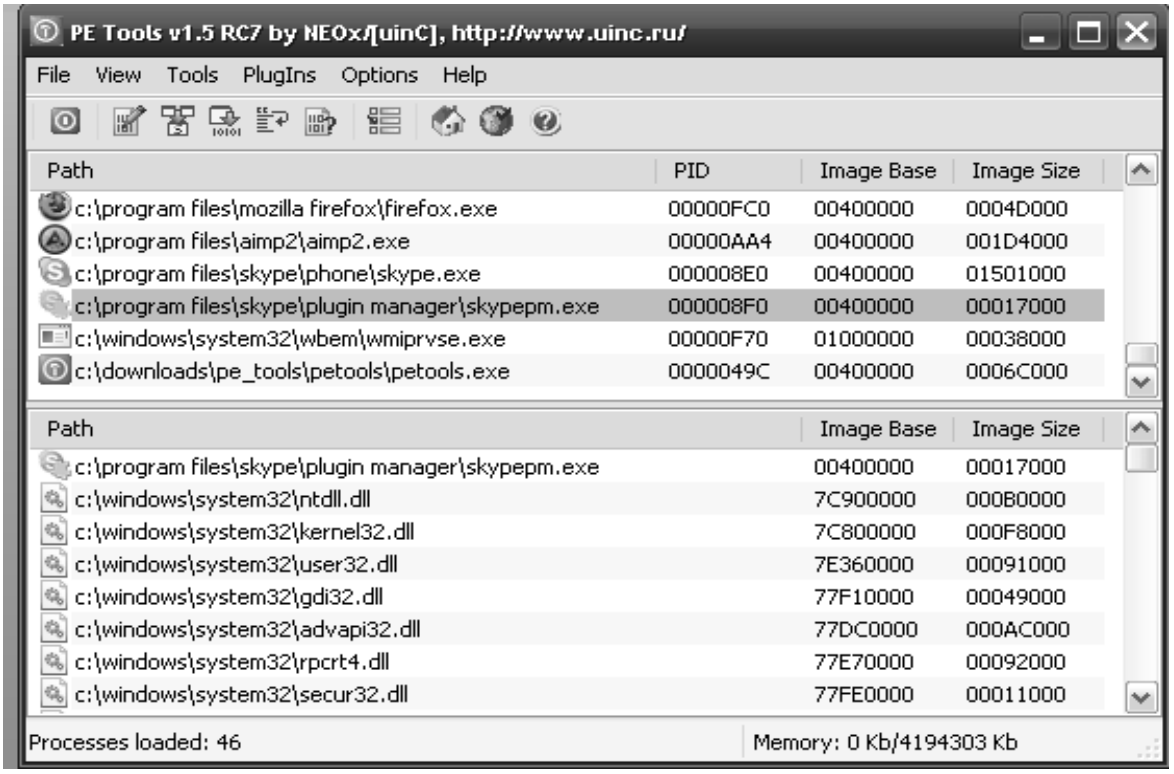


Рисунок 13 – Вигляд головного вікна програми PE Tools

Дана утиліта містить в собі багато корисних функцій, необхідних кожному хакеру. На даний час існує декілька стандартних плагінів програми, кожен з яких виконує ряд додаткових функцій, і є можливість підключення додаткових плагінів.

Серед основних опцій утиліти такі:

- Dump Full;
- Dump Partial;
- Dump Region

Якщо ми працюємо з налагоджувачем прикладного рівня, то зняти дамп з програми дуже просто. Достатньо перемкнутися на PE Tools, вибрати потрібний процес в списку натиснути праву кнопку миші і вибрати "Dump full" (рис. 14).

Зберігати дамп можна в папку з самим exe-файлом або вибрати власну. Назва здампованого (знятого зліпка пам’яті) файлу за замовчуванням буде «Dumped.exe». Тепер цю програму можна запускати.

Але з налагоджувачами рівня ядра (Microsoft Kernel Debugger) цей спосіб не пройде. Щоб отримати дамп, запам'ятовуємо перші два байти від початку OEP і записуємо сюди `EBFEh`, що відповідає машинній інструкції `JMP short $-2`, яка зациклює процес. Тепер можна сміливо виходити з налагоджувача, переходити в PE Tools, знімати дамп і відновлювати оригінальні байти в будь-якому hex-редакторі. Звичайні пакувальники (типу UPX) не чинять опору зняттю дампу, оскільки боротьба з хакерами в їх завдання не входить.

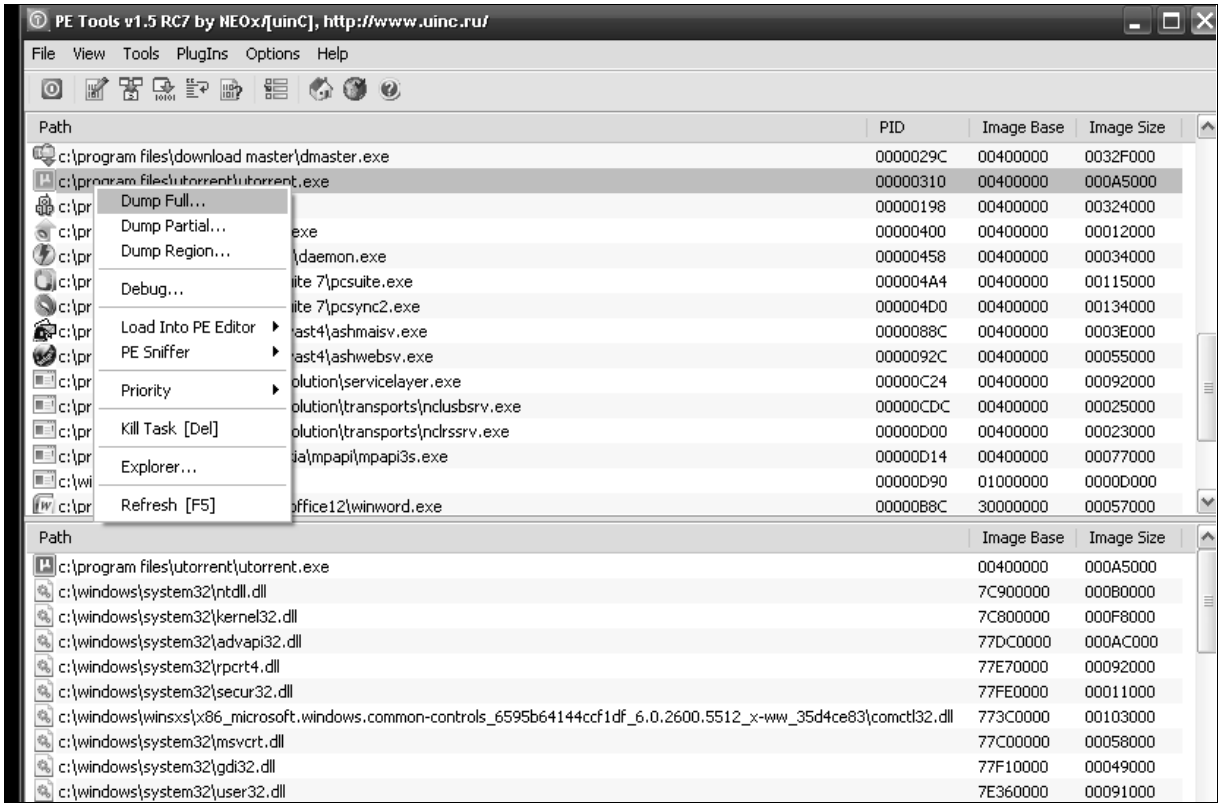


Рисунок 14 – Зняття дампу за допомогою функції "Dump full"

За допомогою функції «Dump partial» можна знімати не весь дамп, а тільки його частину. Для цього спочатку треба вибрати потрібний нам процес, натиснути праву кнопку миші і вибрати опцію «Dump partial». Далі необхідно обрати область пам’яті, звідки почнеться зняття дампу, і необхідний обсяг (рис.15) .

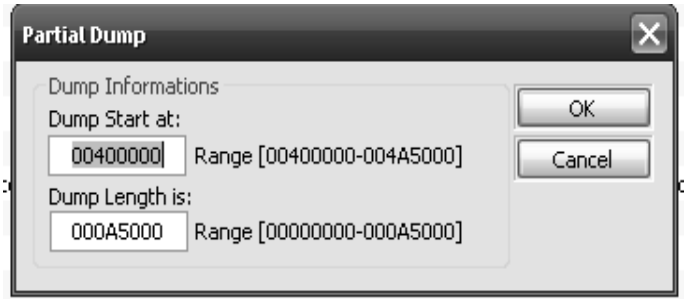


Рисунок 15 – Вікно вибору початку дампу і його довжини

Після цього отриманий дамп зберігається аналогічним способом.

У будь-якому випадку зняття дампу починається з визначення регіону пам'яті, що входить в склад виконуваного файлу (або динамічної бібліотеки). Прийоми, описані вище, повністю використовують PE-заголовок і таблицю секцій, потрібні операційній системі практично тільки на стадії завантаження файлу. Зокрема, нас цікавлять поля *Imagebase* (адреса базового завантаження), *SizeOfImage* (розмір образу) і вміст таблиці секцій. Протектори стирають ці поля після завершення розпаковування або підсовують свідомо некоректні значення. PE Tools в більшості випадків самостійно відновлює інформацію, якої бракує, але іноді це не спрацьовує.

Найпростіше досліджувати карту пам'яті потрібного процесу, яку повертають API-функції *VirtualQuery()* (*VirtualQueryEx()*). Регіони, помічені як *Mem_Image*, належать виконуваному файлу або одній з використовуваних ним DLL. У PE Tools за побудову карти пам’яті відповідає команда "Dump region" (рис. 16) .

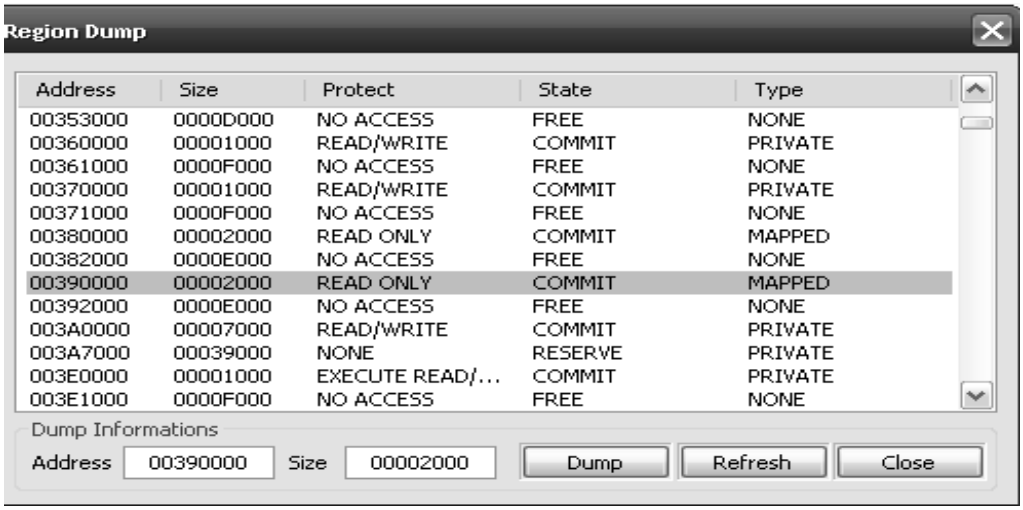


Рисунок 16 – Вікно перегляду карти пам'яті в PE Tools

Якщо жодним із способів визначити межі образу не вдається, доводиться знімати дамп фрагментів адресного простору, завантажуючи їх в IDA Pro, як двійкові файли із збереженням початкової адреси фрагмента.

5.5 Деякі методи захисту від дампінгу

5.5.1 Антидампінг у нульовому кільці

Як було зазначено вище, всі дампери процесів побудовані на функціях OpenProcess(), ReadProcessMemory(), VirtualQueryEx() та інших. Для отримання списку модулів, завантажених в процес, зазвичай використовуються функції Toolhelp API, які, у свою чергу, читають пам'ять процесу через ReadProcessMemory(). На рівні NATIVEAPI при цьому відбувається виклик функцій ZwOpenProcess() і ZwReadVirtualMemory(). Очевидний спосіб протидії дампу – це встановити драйвер, який перехопить у ядрі ці функції і заборонить доступ до процесу, що захищається.

Перехоплення функції ZwOpenProcess()

Одним з простих рішень є перехоплення тільки ZwOpenProcess(), оскільки для читання пам'яті процесу його потрібно спочатку відкрити. Код обробника перехоплення виглядатиме приблизно таким чином:

```
NTSTATUS NewNtOpenProcess (
        OUT_PHANDLE ProcessHandle,
        IN_ACCESS_MASK DesiredAccess,
        IN_POBJECT_ATTRIBUTES ObjectAttributes,
        IN_PCLIENT_ID ClientId OPTIONAL)
{
```

```
 HANDLE ProcessId;
 if ((ULONG)ClientId > *MmUserProbeAddress)
     return STATUS_INVALID_PARAMETER;
 _try
     {  ProcessId = ClientId->UniqueProcess;      }
 _except (EXCEPTION_EXECUTE_HANDLER)
     {
        DPRINT("Exception");
        return STATUS_INVALID_PARAMETER;
     }
 if (IsAdded(wLastItem, ProcessId))
     {
        DPRINT("Access Denied!");
        return STATUS_ACCESS_DENIED;
     }
  else
      return TrueNtOpenProcess(ProcessHandle,
            DesiredAccess, ObjectAttributes, ClientId);
}
```

У цьому коді спочатку безпечним чином витягується PROCESSID – ідентифікатор процесу, що відкривається, а потім приймається рішення про передачу управління оригінальній функції або повернення з кодом Status_access_denied. Оскільки захищених процесів в системі може бути декілька, то необхідно вести список цих процесів і коректно додавати в нього нові процеси і видаляти завершені. Для ведення цього списку об’явимо структуру, що описує захищуваний процес:

```
typedef struct _ProcessList
{     PVOID NextItem;
      HANDLE Pid;
} TProcessList, *PProcessList;
```

Наведемо код, що керує зв’язаним списком цих структур:

```
BOOLEAN IsAdded(PProcessList List, HANDLE Pid)
{     PProcessList Item = List;
      while (Item)
      {  if (Pid == Item->Pid) return TRUE;
         Item = Item->NextItem;
      }  return FALSE;
}
void DelItem(PProcessList *List, HANDLE Pid)
{     PProcessList Item = *List;
      PProcessList Prev = NULL;
      while (Item)
      {  if (Pid == Item->Pid)
         { if (Prev) Prev->NextItem=Item->NextItem;
           else *List=Item->NextItem;
           ExFreePool(Item);
           return;
         }
         Prev = Item;
         Item = Item->NextItem;
      }  return;
}
void FreePointers(PProcessList List)
{     PProcessList Item = List;
      PVOID Mem;
      while (Item)
```

```
        {  Mem = Item;
           Item = Item->NextItem;
           ExFreePool(Mem);
        }  return;
    }
void AddItem(PProcessList *List, HANDLE Pid)
{     PProcessList wNewItem;
      wNewItem=ExAllocatePool(NonPagedPool,
                  sizeof(TProcessList));
      wNewItem->NextItem = *List;
      *List = wNewItem;
      wNewItem->Pid = Pid;
      return;
}
```

Додавати процеси в список ми будемо при отриманні IOCTL запиту до драйвера, а видаляти – при завершенні.

Перехоплення функцій ZwReadVirtualMemory(), ZwWriteVirtualMemory() і ZwCreateThread()

Для надійнішого захисту можна ще перехоплювати функції ZwReadVirtualMemory(), ZwWriteVirtualMemory() і ZwCreateThread(), але тоді виникне необхідність отримувати за хендлом процесу його ідентифікатор PROCESSID. Це можна здійснити за допомогою функції ZwQueryInformationProcess(), але це не працюватиме для тих хендлів, які не мають прапора доступу Process_Query_Information, тому у цьому випадку краще за допомогою функції ObReferenceObjectByHandle() отримати покажчик на структуру EPROCESS, пов'язану з потрібним процесом, і витягнути PROCESSID безпосередньо з неї. Це можна зробити так:

```
ULONG GetPid(HANDLE PHanlde)
{   NTSTATUS st = 15;
    PEPROCESS process = 0;
    ULONG pId;
    st = ObReferenceObjectByHandle(PHanlde, 0, NULL,
                UserMode, &process, NULL);
    if (st == STATUS_SUCCESS)
    {    pId = *(PULONG)(((ULONG)process) + pIdOffset);
         ObDereferenceObject(process);
         return pId;
    }  return 0;
}
```

Тут `pidoffset` – зсув поля Processid в структурі EPROCESS. Ця константа може бути різною в різних версіях ядра системи, тому слід при старті драйвера перевіряти версію ядра і відповідно заповнювати значення змінної. При написанні обробників слід враховувати, що ZwOpenProcess() повинна працювати, якщо процес намагається відкрити сам себе (для порівняння слід розглянути функцію PsGetCurrentProcessId()), а при обробці функцій, що приймають хендли, потрібно враховувати, що існує псевдохендл (-1), який означає поточний процес.

Слід зауважити, що при практичній реалізації цього методу зустрічаються деякі підводні камені. Наприклад, в Windows XP існує служба стилів. Для правильної її роботи потрібно дозволяти доступ процесу сервера

підсистеми (*csrss.exe*) до пам'яті процесу, що захищається. У зв'язку з цим виникає необхідність у визначенні ID цього процесу. Питання це не зовсім просте, оскільки за іменем процесу його ID визначати не можна, оскільки процесів csrss.exe у системі може бути декілька (наприклад, навіть зламник для своєї роботи може перейменувати так свій дампер), тому потрібний надійніший спосіб однозначного визначення цього процесу.

Для реалізації цього можна використовувати те, що сервер підсистеми володіє деякими іменованими об'єктами, за якими його і можна визначити. Для прикладу візьмемо LPC-порт \Windows\ApiPort, який створюється *csrss.exe* у всіх версіях Windows NT. Для його визначення слід отримати всі відкриті хендли за допомогою функції ZwQuerySystemInformation(), скопіювати кожен з них в таблицю хендлів ядра, викликати функцію ZwQueryObject() і порівняти отримане ім'я з шуканим. У випадку збігу імен, ID процесу власника хендла і буде *csrss.exe*. Все це може здійснити код, наведений у додатку А. Така методика антидампінгу вико ристовується в протекторі Themida (Extreme Protector). В ранніх версіях протектора для боротьби з проблемами роботи GUI застосовувався метод, подібний до описаного вище, але в останніх версіях розробникам довелося від нього відмовитися (що погіршило захист). А от останні версії Themida забороняють доступ не до всього адресного простору процесу, що захищається, а тільки за діапазоном адрес, в якому знаходиться EXE-файл, а всі завантажені в його пам'ять DLL лишаються ніяк не захищеними.

Протидії дампінгу за допомогою драйвера

Зазвичай наведений вище метод показує себе досить добре, оскільки більшість зламників не досить чітко уявляють, що твориться в ядрі системи, але за наявності мінімальних знань такий антидампінг проблемою не буде. Один з методів зняття цього захисту – знайти адресу оригінального обробника, і прибрати хук, відкоригувавши SDT. Проблема в тому, що цей спосіб не універсальний, оскільки хук може бути встановлений не тільки через SDT, але й перехопленням int 2EH (для Windows 2000) або зміною обробника *sysenter* (для Windows XP). Прибирати хуки проектора – це не найкращий варіант, оскільки він може перевіряти їх наявність, і при знятті здійснювати деякі ворожі дії (в цьому один з недоліків StarForce). Тому тут можна запропонувати інший спосіб – знімати дамп за допомогою драйвера, не використовуючи при цьому ZwReadVirtualMermory().

У ядрі Windows NT є недокументовані функції *KeAttachProcess()* і *KeDetachProcess(),* які дозволяють драйверу змінювати поточний адресний простір. Ось прототипи цих функцій:

```
extern void KeAttachProcess(PEPROCESS Process);
extern void KeDetachProcess(void);
```

З їх допомогою можна підключитися до захищеного процесу і прочитати його пам'ять, але (на жаль зламникам або на радість розробникам) функції, що забезпечують безпеку операційної системи, забороняють дамп свого процесу у такий спосіб. Але захистом це назвати важко, оскільки підключитися до іншого процесу можна, наприклад, за допо-

могою функцій *KeStackAttachProcess()* і *KeUnstackDetachProcess()*, які протектор перехоплювати забуває (і захиститися від цього буде проблемою, оскільки ці функції використовуються системними драйверами). Тому для читання пам'яті можна використовувати саме їх. Ось прототипи цих функцій і структура, необхідна для їх роботи:

```
extern NTKERNELAPI void KeStackAttachProcess
                (PVOID Process, PKAPC_STATE ApcState);
extern NTKERNELAPI void KeUnstackDetachProcess
                (PKAPC_STATE ApcState);
typedef struct _KAPC_STATE
{     LIST_ENTRY ApcListHead[2];
      PVOID Process;
      BOOLEAN KernelApcInProgress;
      BOOLEAN KernelApcPending;
      BOOLEAN UserApcPending;
} KAPC_STATE, *PKAPC_STATE;
```

Якщо буде реалізовано перехоплення і цих функцій, то нескладною справою для зламників буде підключитися до потрібного процесу вручну, змінивши регістр CR3. Для читання пам'яті процесу можна передавати драйверу через IOCTL хендл процесу, адресу зчитаної пам'яті, адресу буфера в поточному процесі і число зчитаних байтів, після чого драйвер отримає покажчик на EPROCESS піддослідного процесу, підключиться до нього і прочитає потрібні дані. Тобто, фактично, це є ручна реалізація функції *ZwReadVirtualMemory()*. Код, який робить все це, виглядатиме так:

```
void CopyProcessMem
    (HANDLE hProcess, PVOID SrcAddr, PVOID DstAddr, ULONG *Size)
{   PEPROCESS process = NULL;
    NTSTATUS  st;
    PUCHAR pMem = NULL;
    ULONG Addr, Bytes;
    PUCHAR cPtr, dPtr;
    KAPC_STATE ApcState;
    st=ObReferenceObjectByHandle
                (hProcess,0,NULL,UserMode,&process,NULL);
    if (NT_SUCCESS(st))
    {      Bytes = *Size;
           pMem = ExAllocatePool(NonPagedPool, Bytes);
           dPtr = pMem;
           cPtr = (PUCHAR)SrcAddr;
           KeStackAttachProcess(process, &ApcState);
           _try { while (Bytes)
                  {      *dPtr = *cPtr;
                         cPtr++;  dPtr++; Bytes --;
                  }
                }
           _except(EXCEPTION_EXECUTE_HANDLER) {      }
           KeUnstackDetachProcess(&ApcState);
           Bytes = *Size - Bytes;
           _try { memcpy(DstAddr, pMem, Bytes);
                  *Size = Bytes;
                }
           _except(EXCEPTION_EXECUTE_HANDLER)    {   }
           ExFreePool(pMem);
           ObDereferenceObject(process);
```

```
    }return;
}
```

Створити захист від драйверного дампінгу набагато важче, оскільки практично неможливо визначити, звідки йде звернення до пам'яті: від дампера чи від системи. Але зробити це можна. Хоча не слід думати, що це спрацьовує з усіма протекторами, які використовують драйверний антидамп (але на даний момент допомагає від усіх).

На жаль, замінити *ZwWriteVirtualMemory()* своїм аналогом у багатьох випадках буде недостатньо, оскільки деякі протектори перехоплюють ще і *ZwOpenProcess()*. Щоб обійти це, можна не відкривати процес, а отримати його хендл неявним пошуком в процесі сервера підсистеми (використовуючи перехоплення API-функцій в Windows NT). У бібліотеці *advapihook* відповідна функція називається *OpenProcessEx()*. Також такий метод допомагає зламникам проти захистів, які змінюють маркер безпеки процесу, забороняючи відкриття його пам'яті на читання. Якщо цей метод отримання хендлів отримає популярність, то автори захистів швидше за все почнуть закривати хендли в сервері підсистеми. Хоча слід зауважити, що захисту від цього вони не отримають, оскільки це не може бути виконано коректно. Для отримання хендла процесу в драйвері, досить отримати покажчик на його EPROCESS з допомогою функції *PsLookupProcessByProcessId()*, після чого його можна додати в таблицю хендлів нашого процесу за допомогою *ObOpenObjectByPointer()*. Ці дії здійснює такий код:

```
HANDLE MyOpenProcess(HANDLE ProcessId)
{   PEPROCESS Process;
    NTSTATUS St;
    HANDLE hProcess = NULL;
    PsLookupProcessByProcessId(ProcessId, &Process);
    ObOpenObjectByPointer(Process,0,NULL,0,NULL,
                          UserMode,&hProcess);
    ObDereferenceObject(Process);
    return hProcess;
}
```

Можна перехоплювати і функцію *PsLookupProcessByProcessId()*, але це тільки відстрочить отримання дампу на п'ять хвилин, оскільки отримати хендл можна вручну, попорпавшись в структурах ядра, або взагалі зняти дамп без отримання хендла і без використання API.

Захист в нульовому кільці відкриває дійсно багаті можливості (обмежені тільки уявою). Наприклад, досить непоганий спосіб антидампу – руйнування таблиці сторінок процесу, що захищається. Для цього треба втрутитися в роботу планувальника і перехопити функцію *SwapContext()*, що не експортується, а викликається при зміні робочого потоку, в обробнику перехоплення, при перемиканні на процес, що захищається, потрібно відновлювати таблицю сторінок, а при відключенні від нього – руйнувати.

Це найпростіше, що можна протиставити драйверному дампу. Обійти такий захист теж нескладно, потрібно просто змусити працювати дампер в контексті захищеного процесу, що можна зробити як в ядрі (перехоплення обробника системних викликів), так і в режимі роботи.

5.5.2 Динамічне розпаковування

Інший розповсюджений метод захисту від зняття дампу – динамічне розпаковування. Суть його в тому, що протектор розпаковує захищувану програму не повністю, а частинами. Спочатку розпаковується перша сторінка, а коли відбувається звернення за її межі, протектор перехоплює виключення і розпаковує запитану сторінку, при цьому він може прибрати з пам'яті попередню сторінку. Таким чином, образ захищеного процесу в пам'яті ніколи не існує повністю, отже звичайним дампером його не зняти не можна. Така методика застосовується в досить розповсюдженому протекторі Armadillo і називається Copymem. Для перехоплення виключень і розшифровки коду створюється окремий процес, що підлагоджує захищений за допомогою DEBUGAPI. Для зняття цього захисту більшість хакерів обирають не зовсім оптимальний шлях – реверсинг і зміну коду протектора з метою змусити його розшифрувати код повністю. Це дуже трудомістка операція, тим більш, що в кожній новій версії автори міняють реалізацію цього захисного механізму, і тоді старі програми для зламу стають неактуальними. Але у цьому випадку можна дампувати процес зсередини. Для цього треба впровадити код дампера в адресний простір захищеного процесу і прочитати його пам'ять. При цьому виникають виключення, обробивши які протектор віддасть нам весь код у розшифрованому вигляді. Це найпростіший спосіб зняття захисту Copymem.

Контрольні питання

1. Що таке дамп пам’яті?
2. Який порядок завантаження програми і виділення пам’яті процесу?
3. Як отримати дамп отриманого процесу?
4. Які методи захисту від дампінгу ви можете назвати? Охарактеризуйте їх.
5. Які програми для зняття програм з пам’яті ви знаєте і як від них захиститись?

ВИСНОВКИ

Доцільність захисту програмного забезпечення обмежується конкурентною боротьбою – при інших рівних умовах на сьогоднішній день клієнт обирає той програмний продукт, який якнайкращим чином захистить його інтереси.

При розробці систем захисту існує ряд типових помилок, яких припускають розробники захищених програм:

- програма захищається лише від засобів статичного дослідження, в результаті вона легко вивчається динамічно, і навпаки;
- захист, який зламується модифікацією одного або декількох байтів (у момент, коли система захисту порівнює контрольну інформацію з еталонною) простою зміною команд переходу направляється по правильному шляху;
- аналогічна ситуація має місце, коли результат роботи функції, що повертає поточну контрольну інформацію, може бути підмінений на еталонне (очікуване) значення (наприклад, за допомогою перехоплення відповідного переривання);
- після розшифрування системою захисту критичного коду він стає доступним і може бути скопійований у інше місце або на диск у момент або після передачі керування на нього.

Щоб уникнути використання таких промахів у системах захистів, у даному навчальному посібнику якраз і розглядалися спеціальні прийоми, які можуть доповнити, посилити захист, підвищити його ефективність.

Звичайно, захистів, які не можна зняти або зламати, немає, але задача розробників захистів від дослідження може полягати у тому, щоб:

- заставити супротивника витратити на зняття захисту час, достатніх для прийняття певних контрольних заходів;
- або гарантувати, що ресурси, витрачені на її злам, можна буде спів ставити з написанням захищеної програми заново.

ЛІТЕРАТУРА

1. Дудатьєв А.В. Захист програмного забезпечення. Ч.1 : навчальний посібник / Андрій Дудатьєв, Валентина Каплун, Василь Семеренко – Вінниця: ВНТУ, 2005. – 140 с.
2. Казарин О.В. Теория и практика защиты программ / Олег Казарин – М.: МГУЛ, 2004. – 450 с.- ISBN 5-93517-178-6.
3. Соколов А. Защита от компьютерного терроризма : [Справочное пособи] / Алексей Соколов, Ольга Степанюк – БХВ-Петербург: Арлит, 2002. – 496 с.
4. Щеглов А.Ю. Защита компьютерной информации от несанкционированного доступа : учебное пособие / Александр Щеглов – Санкт-Петербург: Наука и Техника, 2004. – 384 с. – ISBN 5-318-00244-7.
5. Румянцев П. В. Работа с файлами в Win 32 API. – 2-е изд. доп. / Павел Румянцев – М.: Горячая линия-Телеком, 2002. – 216 с. – ISBN 5-93517-097-3.
6. Румянцев П.В. Исследование программ Win32: до дизассемблера и отладчика – 2-е изд. доп. / Павел Румянцев – М.: Горячая линия-Телеком, 2004. – 367 с. – ISBN 5-93519-178-3.
7. Абашев А.А. Ассемблер в задачах защиты информации / Алексей Абашев, Иван Жуков, Михаил Иванов – М.: Кудиц-Образ, 2004. – 544 с. – ISBN 5-9579-0027-3.
8. Касперски К. Компьютерные вирусы изнутри и снаружи / Крис Касперски – СПб.: Питер, 2006. – 527 с. – ISBN 5-93519-178-3.
9. Касперски К. Техника и философия хакерских атак / Крис Касперски – М.: Солон-Р, 2006. – 272 с. – ISBN 5-93455-015-2.
1. Чернов А.В. Интегрированная среда для исследования "обфускации" программ. Доклад на конференции, посвящённой 90-летию со дня рождения А.А.Ляпунова. Россия, Новосибирск, 8-11 октября 2001 года // Електронний ресурс: http://www.ict.nsc.ru/ws/Lyap2001.
2. Домашев А.В. Программирование алгоритмов защиты информации : Учебное пособие / Алексей Домашев, Михаил Грунтович, Владимир Попов – М.: Нолидж, 2002. – 416 с. – ISBN 5-93519-024-8.

ДОДАТОК А

Лістинг функції визначення процесу

```
PVOID GetInfoTable(ULONG ATableType)
{
  ULONG mSize = 0x4000;
  PVOID mPtr = NULL;
  NTSTATUS St;
  do
  { mPtr = ExAllocatePool(PagedPool, mSize);
    if (mPtr != NULL)
    {
        St = ZwQuerySystemInformation(ATableType,mPtr,mSize,NULL);
    }
    else  return NULL;
    if (St == STATUS_INFO_LENGTH_MISMATCH)
    {
      ExFreePool(mPtr);
      mSize = mSize * 2;
    }
  }
  while (St == STATUS_INFO_LENGTH_MISMATCH);
  if (St == STATUS_SUCCESS)
  { DPRINT("GetInfoTable Success!");
    DPRINT("Info table in memory size - %d", mSize);
    return mPtr;
  }
  else ExFreePool(mPtr);
  DPRINT("Error on GetInfoTable %X", St);
  return NULL;
}

ULONG GetCsrPid()
{    int r;
HANDLE Process, hObject;
NTSTATUS St;
ULONG CsrId = 0;
OBJECT_ATTRIBUTES obj;
CLIENT_ID cid;
POBJECT_NAME_INFORMATION ObjName;
UNICODE_STRING ApiPortName;
      PSYSTEM_HANDLE_INFORMATION_EX Handles;

      RtlInitUnicodeString(&ApiPortName, L"\\Windows\\ApiPort");
      DPRINT("Get handles info");
      Handles = GetInfoTable(SystemHandleInformation);
      if (Handles == NULL) return 0;

    ObjName = ExAllocatePool(PagedPool, 0x2000);
    DPRINT("Number of handles %d", Handles->NumberOfHandles);

      for (r = 0; r != Handles->NumberOfHandles; r++)
      {
        if (Handles->Information[r].ObjectTypeNumber==21)
                                              //Port object
        { InitializeObjectAttributes(&obj,
                              NULL,OBJ_KERNEL_HANDLE,NULL,NULL);
```

```
            cid.UniqueProcess=(HANDLE)Handles->Information[r].ProcessId;
      cid.UniqueThread  = 0;
            if (ZwOpenProcess(&Process,PROCESS_DUP_HANDLE,&obj,&cid) ==
                                        STATUS_SUCCESS)
      {  if (ZwDuplicateObject(Process,
             (HANDLE)Handles->Information[r].Handle,
             NtCurrentProcess(), &hObject, 0, 0,
                    DUPLICATE_SAME_ACCESS) == STATUS_SUCCESS)
                {
              if (ZwQueryObject(hObject, ObjectNameInformation,
                  ObjName, 0x2000, NULL) == STATUS_SUCCESS)
              {
                  if (ObjName->Name.Buffer != NULL)
                  if (wcsncmp(ApiPortName.Buffer,
                              ObjName->Name.Buffer, 20) == 0)
                 {   DPRINT("Csrss %d",
                                      Handles->Information[r].ProcessId);
                     DPRINT("csr port-%ws",ObjName->Name.Buffer);
                     CsrId = Handles->Information[r].ProcessId;
                     ZwClose(Process);
                     ZwClose(hObject);
                     CsrId = Handles->Information[r].ProcessId;
                     ExFreePool(Handles);
                     ExFreePool(ObjName);
                     return CsrId;
                 }
                }
                 else DPRINT("Error in Query Object");
               ZwClose(hObject);
                     }
              else DPRINT("Error on duplicating object");
                     ZwClose(Process);
          }
          else DPRINT("Could not open process");
        }
      }
      ExFreePool(Handles);
      ExFreePool(ObjName);
      return 0;
}
```

ГЛОСАРІЙ

Несанкціонований доступ (НСД) – *unauthorized access* – нелегальні дії щодо використання, зміні та знищенню виконуваних модулів.

Система захисту від несанкціонованого доступу – *system of protection against unauthorized access* – комплекс програмних засобів, що забезпечують утруднення або заборону нелегального розповсюдження, використання і/або зміну програмних продуктів.

Надійність системи захисту – *reliability of protection* – це здатність протистояти спробам проникнення в алгоритм її роботи і обходу механізмів захисту.

Злам програми – *breaking program* – порушення функціональності об'єктів захисту програмного забезпечення (адже їх може бути декілька).

Зламник – *cracker* – користувач обчислювальної системи, який займається незаконним одержанням доступу до захищених ресурсів системи.

Хакер – *hacker* – програміст, спроможний писати програми без попередньої розробки детальних специфікацій і оперативно вносити виправлення в працюючі програми, у тому числі і безпосередньо в машинних кодах, що потребує найвищої кваліфікації.

Патчер – *patcher* – невеликий виконуваний файл, що автоматично вносить зміни в оригінальний файл програми або в код цієї програми безпосередньо в пам'яті.

Disk Space – кількість місця на диску, яке можна використовувати для веб – файлів, пошти, скриптів, баз даних, записів системного журналу і т. і.

Пароль – *password* – набір цифр і літер, котрі разом з логіном слугують ідентифікатором користувача.

CommercialWare – комерційне програмне забезпечення, яке створене для отримання прибутку. Для нього застосовується принцип “гроші – вперед”, тобто користувач отримує програму лише після оплати.

FreeWare – версії програм для вільного розповсюдження зі збереженням прав за автором.

ShareWare – умовно безкоштовне програмне забезпечення, тобто версії програм, які можна 2–4 тижні випробовувати, а потім або не використовувати, або оплатити.

TrialWare – програмне забезпечення, яке можна використовувати на протязі певного терміну, потім продукт стає недієздатним.

CriptWare – програмний продукт має дві версії: демо – версія плюс зашифрована робоча версія.

DemoWare – коли в програмі присутні функціональні обмеження. Наприклад, можна обробляти файли, що не перевищують певного розміру, не можна виконувати їх збереження і т. д. Іноді програми ще називають CrippleWare – “урізане” програмне забезпечення.

Nagware – користувачу постійно пропонуються нагадування про те, що дана версія програми не є повноцінною комерційною версією (у вигляді діалогового вікна, що з'являється з деякою періодичністю під час роботи, або додаткові надписи, що виводяться на екран і т.д.).

AdWare – розробник отримував плату за використання програми не від кінцевого споживача, якому програма надавалася безкоштовно, а від рекламодавців, а користувач повинен був переглядати надані через Інтернет рекламні картинки.

CardWare – кожний користувач програми, що бажає зареєструватися, повинен надіслати автору програми поштову листівку з виглядом місцевості, де він проживає.

MailWare – більш сучасний варіант CardWare, що передбачає відсилання автору електронного листа. Як правило, у відповідь автор надсилає реєстраційний код, що дає можливість працювати з програмою.

DonationWare – коли автор не вимагає ніякої оплати, але пропонує всім, кому сподобалась програма, пожертвувати певну суму для підтримки даної програмної розробки.

GifWare – майже те саме, що і попередній вид розповсюдження, але автор готовий прийняти не лише грошові пожертвування, але й інші подарунки;

BeerWare – подяка за програму приймається у вигляді пива.

VegeWare – автор збирає з користувачів плату за програму у формі рецептів вегетаріанських блюд.

MemorialWare – людина на ім'я Гарі Кремблітт присвятив свою програму пам'яті свого батька. Користувачам цієї програми пропонується допомоги меморіальному фонду Кремблітта – батька.

Промислове шпигунство – *industrial espionage* – незаконне використання алгоритмів, що є інтелектуальною власністю автора, при написанні аналогів продукту.

Крадіжка і копіювання – *stealing and copying* – несанкціоноване використання ПЗ.

Несанкціонована модифікація ПЗ – *unauthorized modification* – модифікація з метою впровадження програмних зловживань.

Піратство – *piracy* – незаконне поширення і збут ПЗ.

BIOS – базова система введення – виведення, яка надає невеликий вбудований стартовий набір для виконання іншого програмного забезпечення на гнучких дисках (FDD), жорстких (HDD), CD або DVD – дисках. BIOS відповідає за завантаження комп'ютера, забезпечуючи базовий набір команд. BIOS виконує всі задачі, що повинні виконуватися під час запуску.

CMOS – *Complementary Metal Oxide Semiconductor* – назва технології, за якою виготовляються ІС із вкрай малим споживанням енергії, тому батарея в комп'ютері майже "не використовується". Фактично на нових системних платах установлена не батарея, а акумулятор.

MBR – *Master Boot Record* – найперший сектор жорсткого диска (сектор 1, доріжка 0, головка 0). Цей запис займає не весь сектор, а тільки його початкову частину.

Partition Table – таблиця розділів диска. Ця таблиця містить чотири елементи, що описують до чотирьох розділів диска. Всі елементи таблиці розділів диска мають однаковий формат – це структура розміром 16 байтів, що відповідає розділу. У структурі знаходиться інформація про розташування і розмір розділу в секторах, а також про призначення розділу

BR – Boot Record. У найпершому секторі активного розділу розташований завантажувальний запис, який не слід плутати з головним завантажувальним записом MBR. Завантажувальний запис зчитується в оперативну пам'ять головним завантажувальним записом, після чого йому передається керування. Задача завантажувального запису – виконати завантаження операційної системи. Кожен тип операційної системи має свій завантажувальний запис.

Електронний ключ захисту – *the electronic security key* – пристрій, що приєднується до комп'ютера через один з можливих портів і запобігає незаконному використанню (експлуатації) програми.

SekretKey – програмно – апаратний спосіб захисту, побудований на тому, що вибраний фрагмент (або декілька фрагментів) зберігаються і виконуються всередині USB – ключа.

Хук – *hook* – механізм перехоплення особливою функцією подій (таких, наприклад, як повідомлення від маніпулятора миші або клавіатури) до того, як вони дійдуть до програмного додатку. Ця функція може потім реагувати на події, а в деяких випадках змінювати або відміняти їх.

Таблиця об'єктів (таблиця розділів файлу) – *object table* – сукупність даних певного призначення: про експортовані та імпортовані функції, про ресурси, про переміщення (relocations) і т. д., які компактно розміщені у виконуваному файлі. Тобто таблиця об'єктів – це просто інформація про зміст розділів.

X – код – частина коду, яку ми збираємося впроваджувати у програмний код з метою реалізації його захисту від несанкціонованого дослідження.

Обфускація – *obfuscation* – це один з методів захисту програмного коду, який дозволяє ускладнити процес реверсивної інженерії коду програмного продукту, що захищається. Суть обфускації полягає в тому, щоб заплутати програмний код і усунути більшість логічних зв'язків в ньому, трансформувати його так, щоб він був важкий для вивчення і модифікації.

Маскування програми – *masking programs* – це таке перетворення її тексту, яке повністю зберігає її функціональність, але робить розуміння, зворотну інженерію і модифікацію тексту програми завданням неприйнятно високій вартості.

Непрозорий предикат – *opaque predicate* **–** предикат, для якого його значення відомо в момент заплутування програми, але важко відновлюване після його завершення.

Недосяжний код – *unreachable code* – фрагмент програми, що ніколи не виконується. Ці коди можуть бути заповнені довільними обчисленнями, які можуть бути схожі на дійсно виконуваний код, наприклад, зібрані із фрагментів тієї ж самої функції. Оскільки недосяжний код ніколи не виконується, дане перетворення впливає тільки на розмір заплутаної програми, але не на швидкість її виконання.

Мертвий код – *dead code* – це такий код, який у програмі виконується, але його виконання ніяк не впливає на результат роботи програми. Це практично означає, що мертвий код не може мати побічного ефекту, навіть у вигляді модифікації глобальних змінних, не може змінювати оточення працюючої програми, не може виконувати ніяких операцій, які можуть викликати виключення в роботі програми.

Надлишковий код – *redundant mode* **–** код, що виконується, і результат його виконання використовується надалі в програмі, але такий код можна спростити або зовсім видалити, оскільки обчислюється або константне значення, або значення, уже обчислене раніше.

Автоматичні дизасемблери – *automatic disassemblers* **–** програмні засоби, аналізують код виконуваного файлу й формують відповідний йому вихідний текст або лістинг у вигляді асемблерного коду.

Інтерактивні дизасемблери – *interactive disassemblers* – формують вихідний текст/лістинг по виконуваному коду програми так само, як це роблять автоматичні дизасемблери, але відрізняються від автоматичних наявністю потужного користувацького інтерфейсу, що значно полегшує аналіз дизасембльованої програми.

Дамп пам'яті – memore dump – це копія вмісту оперативної пам'яті, що знаходиться на жорсткому диску або іншому енергонезалежному пристрої пам'яті.

Навчальне видання

Валентина Аполінаріївна Каплун
Олександр Васильович Дмитришин
Юрій Володимирович Баришев

ЗАХИСТ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
Частина 2

Навчальний посібник

Редактор

Оригінал-макет підготовлено В. Каплун

Підписано до друку
Формат 29,7×42¼. Папір офсетний.
Гарнітура Times New Roman.
Друк різографічний. Ум. друк. арк.
Наклад прим. Зам. № 2014-

Вінницький національний технічний університет,
науково-методичний відділ ВНТУ.
21021, м. Вінниця, Хмельницьке шосе, 95,
ВНТУ к. 2201.
Тел. (0432) 59-87-36.
Свідоцтво суб'єкта видавничої справи
серія ДК № 3516 від 01.07.2009 р.

Віддруковано у Вінницькому національному технічному університеті
в комп'ютерному інформаційно-видавничому центрі.
21021, м. Вінниця, Хмельницьке шосе, 95,
ВНТУ, ГНК, к. 114.
Тел. (0432) 59-81-59.
Свідоцтво суб'єкта видавничої справи
серія ДК № 3516 від 01.07.2009 р.