В. А. Гарнага, В. А. Каплун, В. І. Селезньов

ІНЖИНІРИНГ ЗАХИЩЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Міністерство освіти і науки України
Вінницький національний технічний університет

В. А. Гарнага, В. А. Каплун, В. І. Селезньов

Інжиніринг захищеного програмного забезпечення

Електронний навчальний посібник

Вінниця
ВНТУ
2026

УДК 004.4.056(075.8)
Г20

Рекомендовано до видання Вченою радою Вінницького національного технічного університету Міністерства освіти і науки України (протокол № 5 від 27.11.2025 р.)

Рецензенти:
В. М. Михалевич, доктор технічних наук, професор
Л. І. Тимченко, доктор технічних наук, професор
Ю. В. Баришев, кандидат технічних наук, доцент

Гарнага, В. А.

Г20 Інжиніринг захищеного програмного забезпечення : навчальний посібник [Електронний ресурс] / Гарнага В. А., Каплун В. А., Селезньов В. І. – Вінниця : ВНТУ, 2026. – 155 с.

Посібник присвячний матеріалам лекційного курсу з дисципліни «Інжиніринг захищеного програмного забезпечення» для здобувачів вищої освіти, що навчаються за спеціальністю «Кібербезпека та захист інформації» (освітні програми «Безпека інформаційних і комунікаційних систем», «Кібербезпека критичних систем», «Етичний хакінг і кібербезпека»)

Навчальний посібник містить теоретичні відомості щодо видів уразливостей програмного забезпечення, основних засад створення безпечного програмного забезпечення з урахуванням сучасних світових технологій. рім того, у посібнику розглянуто основні методи захисту програмного забезпечення від несанкціонованого копіювання, використаня і дослідження.

Перелік та зміст тем відповідає програмі вказаної вище дисципліни.

УДК 004.4.056(075.8)

ЗМІСТ

ВСТУП

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

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

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

Таким чином, необхідність внесення в програмне забезпечення захисних функцій протягом всього його життєвого циклу – від етапу виникнення задуму на розробку програм до етапів випробувань, експлуатації, модернізації і супроводу програм – не викликає сумнівів.

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

1 УРАЗЛИВОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Уразливість програмного забезпечення – це будь-яка помилка, недолік або слабке місце в програмному забезпеченні, яке може бути використане зловмисником для порушення його безпеки. Ці уразливості можуть бути використані для несанкціонованого доступу до даних, відмови в обслуговуванні або навіть фізичного пошкодження системи.

1.1 Основні передумови виникнення уразливостей

Бурхливий розвиток інформаційних технологій, стрімке зростання обчислювальної потужності комп'ютерних систем і обсягів оброблюваних даних, а також розширення кола задач, які вирішують інформаційні системи, ускладнюють проведення повного і детального аналізу можливих уразливостей і виключення умов їх появи.

1. *Людський фактор*, звичайно, є однією з передумов виникнення уразливостей. Сюди можна віднести:

- помилки програмістів. Недосвідченість, відсутність належних знань з безпеки, поспіх у розробці можуть призвести до логічних помилок, неправильного використання функцій мови програмування тощо;
- недостатнє тестування. Неповне або поверхневе тестування коду не дозволяє виявити приховані уразливості;
- неправильна конфігурація. Неправильне налаштування програмного забезпечення та систем може відкрити додаткові шляхи для атак;
- використання застарілих програмних платформ і бібліотек може зробити програму вразливою до відомих експлойтів.

2. *Складність та обсяг програм* не можуть не вплинути на існування уразливостей ПЗ, оскільки:

- чим більший код, тим більше можливостей для виникнення помилок;
- сторонні бібліотеки, використані під час розробки ПЗ, можуть містити власні уразливості, які можуть бути успадковані головною програмою;
- складні системи, які є інтеграцією багатьох компонентів, створюють більше точок потенційного втручання;
- використання застарілих програмних платформ і бібліотек може зробити програму вразливою до відомих експлойтів.

3. *Організаційні фактори,* до числа яких входять*:*

- термінові проєкти, оскільки поспіх у розробці може призвести до економії на безпеці;
- конкуренція на ринку – бажання випустити продукт першим може спонукати розробників нехтувати тестуванням на безпеку;

– необхідність додавання нових функцій, тобто зміни у функціоналі можуть порушити існуючу архітектуру і створити нові уразливості.
– неправильне виправлення помилок може призвести до появи нових проблем.

4. *Соціальний інжиніринг:* Зловмисники можуть використовувати соціальні інженерні методи для отримання доступу до систем або виманювання конфіденційної інформації.

Розуміння цих передумов дозволяє розробникам і фахівцям з безпеки вживати заходів для запобігання виникненню уразливостей та мінімізації ризиків.

1.2 Типи уразливостей

Отже, уразливість програмного забезпечення – це, фактично, «дірка» в безпеці програми, яка може бути використана зловмисником для несанкціонованого доступу, модифікації даних або навіть повного контролю над системою. Існує безліч уразливостей. Наведемо деякі з найпоширеніших.

1.2.1 Переповнення буфера

У будь-якому програмному коді програмісти організовують буфери для тимчасового зберігання й оброблення даних. Розмір буфера має бути таким, щоб дані, з одного боку, повністю у ньому вміщались, а з іншого, щоб буфер не був надто великим, оскільки тоді він марно займатиме пам'ять. Для копіювання даних у буфер переважно використовують бібліотечні функції. Якщо робота ведеться з рядками символів, то деякі функції (наприклад, strcat(), strcpy(), sprintf(), gets(), scanf()) не зважають на попереднє обмеження довжини і здійснюють копіювання до кінця рядка, тобто до символу, що є ознакою кінця рядка. Коли довжина рядка перевищує розмір буфера, копіювання триває, і частину рядка, що не вмістилась у буфері, буде записано замість даних, розташованих за його межами (рис. 1.1).

Буфер (8 байтів)								Переповнення (2 байти)	
P	A	S	S	W	O	R	D	1	2
0	1	2	3	4	5	6	7	8	9

Рисунок 1.1 – Приклад переповнення буфера

Отже, коли програма намагається записати більше даних у буфер, ніж він може вмістити, це може призвести до перезапису сусідніх областей пам'яті і, як наслідок, до непередбачуваної поведінки програми або навіть до виконання довільного коду.

Зловмисники використовують проблеми переповнення буфера, перезаписуючи пам’ять програми. Це змінює шлях виконання програми, викликаючи відповідь, яка пошкоджує файли або розкриває конфіденційну інформацію. Наприклад, зловмисник може ввести додатковий код, надсилаючи нові інструкції програмі для отримання доступу до ІТ-систем.

Якщо зловмисники знають структуру пам’яті програми, вони можуть навмисно ввести вхідні дані, які буфер не може зберегти, і перезаписати області, які містять виконуваний код, замінивши його власним кодом. Наприклад, зловмисник може перезаписати вказівник (об’єкт, який вказує на іншу область пам’яті) і вказати його на корисне навантаження експлойту, щоб отримати контроль над програмою.

Конкретні наслідки переповнення буфера залежать від того, яке значення мали втрачені або модифіковані дані та в якій області пам'яті було розміщено буфер: у статичній, динамічній пам'яті чи у стеці. Тобто, переповнення буфера може здійснюватися:

- на основі стека, що є більш поширеним і використовує стекову пам’ять, яка існує лише під час виконання функції;
- на основі динамічної пам’яті. Таку атаку важче здійснити, і вони передбачають переповнення простору пам’яті, виділеного для програми, крім пам’яті, яка використовується для поточних операцій виконання.

Найбільш вразливими до атак переповнення буфера є мови C і C++, оскільки вони не мають вбудованих засобів захисту від перезапису або доступу до даних у своїй пам’яті. Mac OSX, Windows і Linux використовують код, написаний мовами C і C++.

Такі мови, як PERL, Java, JavaScript і C#, використовують вбудовані механізми безпеки, які мінімізують ймовірність переповнення буфера.

Розробники можуть захиститися від уразливості переповнення буфера за допомогою заходів безпеки у своєму коді або за допомогою мов, які пропонують вбудований захист.

Крім того, сучасні операційні системи мають деякі засоби захисту під час виконання, а саме:

- рандомізація адресного простору (ASLR – Address Space Layout Randomization) – випадково переміщується в адресному просторі областей даних. Як правило, атаки переповнення буфера потребують інформації про розташування виконуваного коду, а рандомізація адресних просторів робить це практично неможливим;
- запобігання виконанню даних – позначає певні області пам’яті як невиконувані або виконувані, що зупиняє атаку від виконання коду в невиконуваній області;
- захист від перезапису обробника структурованих винятків (SEHOP – Structured Exception Handler Overwrite Protection) – допомагає зупинити атаку шкідливого коду на структуровану обробку винятків

(SEH – Structured Exception Handling), вбудовану систему для керування апаратними та програмними винятками. Таким чином, зловмисник не зможе використати техніку використання перезапису SEH. На функціональному рівні перезапис SEH досягається за допомогою переповнення буфера на основі стека для перезапису запису реєстрації винятку, що зберігається в стеку потоку.

Захистів коду та захисту операційної системи недостатньо. Коли організація виявляє вразливість переповнення буфера, вона має швидко відреагувати, щоб виправити уражене програмне забезпечення та переконатися, що користувачі програмного забезпечення можуть отримати доступ до виправлення.

1.2.2 SQL-ін'єкції

Ця атака дозволяє зловмиснику вводити шкідливі SQL-команди через веб-форми або інші вхідні точки, що дозволяє отримати несанкціонований доступ до бази даних.

Більшість веб-додатків використовують одну або кілька баз даних для зберігання та обробки інформації в реальному часі.

Дійсно, коли користувач надсилає запити, веб-додаток запитує базу даних, щоб створити відповідь. Однак, коли інформація, надана користувачем, використовується для підробки запиту до бази даних, зловмисник може змінити базу даних, використовуючи її для цілей, відмінних від тих, які передбачав початковий розробник. Це дозволяє зловмиснику запитувати базу даних за допомогою SQL-ін’єкції або SQLi .

SQL-ін’єкція відноситься до атак на реляційні бази даних, такі як MySQL, Oracle Database або Microsoft SQL Server. Навпаки, ін’єкції в нереляційні бази даних, такі як MongoDB або CouchDB, є ін’єкціями NoSQL.

Ін’єкція SQL відбувається, коли зловмисник передає запис, який змінює SQL-запит, надісланий веб-додатком до бази даних. Потім це дозволяє користувачеві виконувати інші небажані SQL-запити безпосередньо в базі даних. Для цього зловмисник має ввести код поза межами очікуваного введення користувача, щоб він не виконувався як стандартне введення. У найпростішому випадку введення одинарних або подвійних лапок достатньо, щоб вставити дані безпосередньо в SQL-запит.

Дійсно, якщо є можливості ін’єкції, зловмисник шукатиме спосіб виконати інший SQL-запит. У більшості випадків вони використовуватимуть код SQL для створення запиту, який виконує як запланований запит SQL, так і новий запит SQL.

Схему здійснення найпростішої SQLi через введення SQL-коду у поле введення наведено на рис. 1.2.

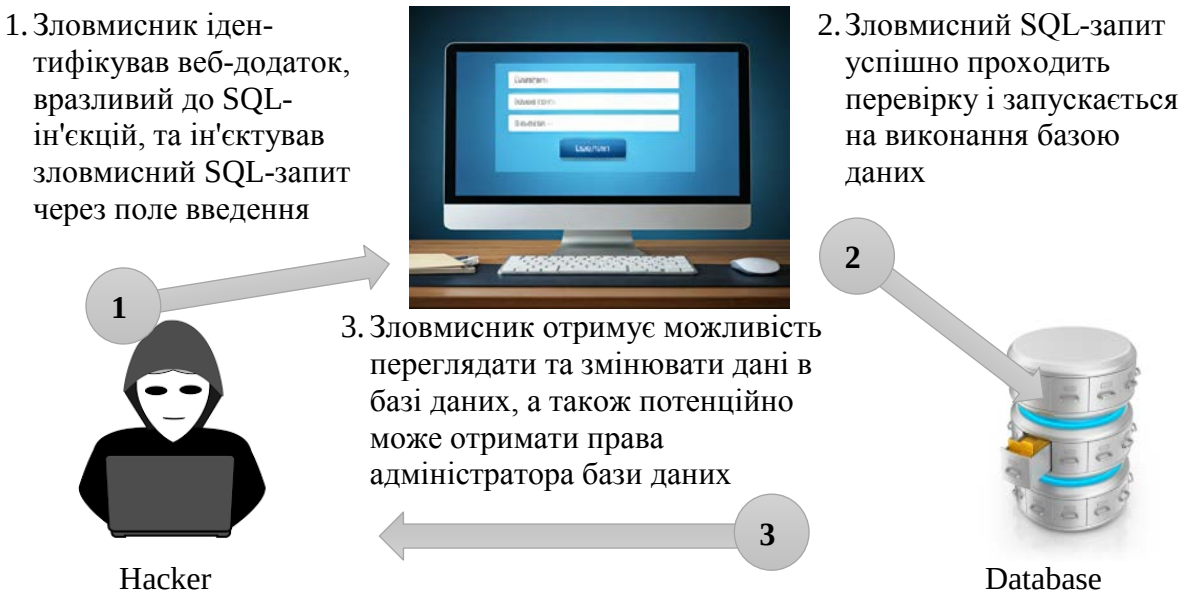


Рисунок 1.2 – Схема здійснення SQL-ін’єкції через введення SQL-коду у поле введення

1.2.3 Міжсайтовий скриптинг (XSS)

Ця атака дозволяє зловмиснику вводити шкідливий JavaScript код на вебсторінку, який виконується у браузері користувача, дозволяючи зловмиснику виконувати дії від імені користувача, такі як крадіжка куків або перенаправлення на шкідливі сайти (рис. 1.3).

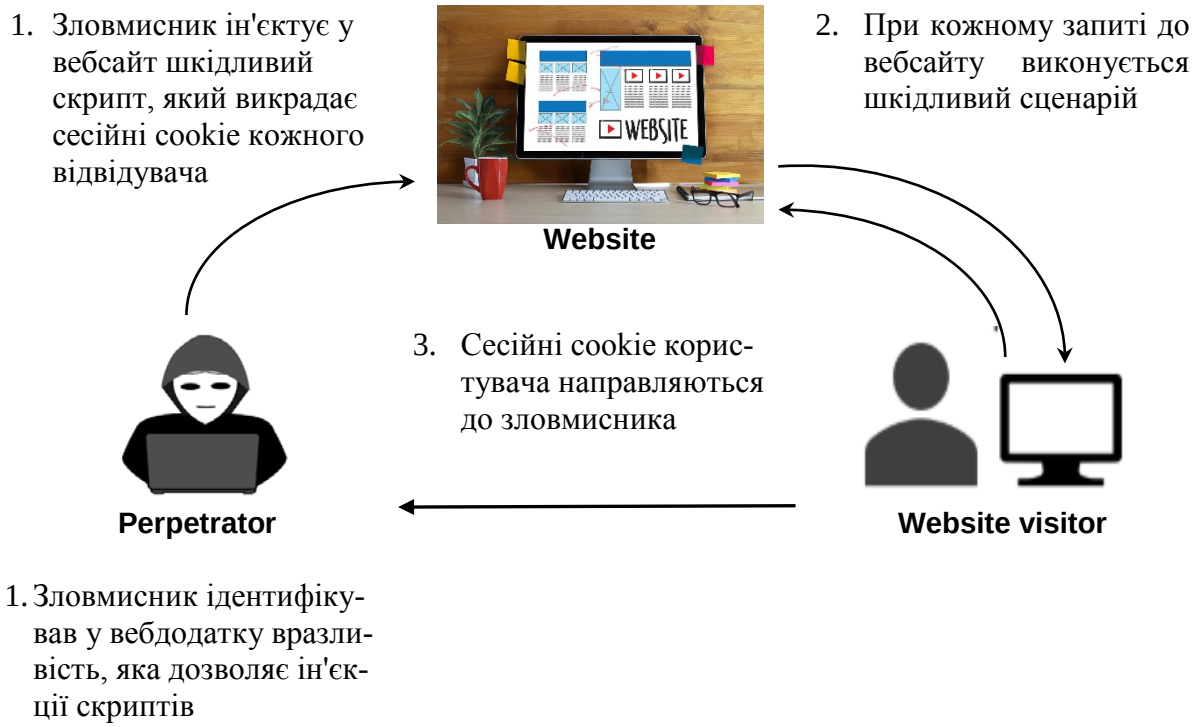


Рисунок 1.3 – Схема здійснення XSS-атаки

Це поширений вектор атаки, який впроваджує шкідливий код у вразливу веб-програму. XSS відрізняється від інших векторів веб-атак (наприклад, ін'єкцій SQL) тим, що він не націлений безпосередньо на саму програму. Натомість користувачі веб-додатку знаходяться під загрозою.

Успішна атака міжсайтового сценарію може мати руйнівні наслідки для репутації онлайн-бізнесу та його відносин з клієнтами.

Залежно від серйозності атаки облікові записи користувачів можуть бути скомпрометовані, троянські програми активовані та вміст сторінки змінено, що спонукає користувачів добровільно надавати свої особисті дані. Нарешті, можуть бути виявлені сеансові файли cookie, що дозволить зловмиснику видавати себе за дійсних користувачів і зловживати їхніми приватними обліковими записами.

Міжсайтові скриптові атаки можуть бути двох типів:

- збережені (постійні XSS), коли шкідливий сценарій впроваджується безпосередньо у вразливу веб-програму;
- відображені, коли передбачається відображення шкідливого сценарію з веб-програми в браузері користувача. Сценарій вбудовано в посилання та активується лише після натискання цього посилання.

1.2.4 Вразливість CSRF

CSRF **(**Cross-Site Request Forgery) – це вразливість веб-безпеки, яка дозволяє зловмисникам змусити автентифікованого користувача виконати небажані дії на веб-сайті.

Зловмисник створює спеціальне посилання або форму, яка вказує на вразливий веб-сайт. Жертва, яка вже авторизована на сайті, випадково клацає на це посилання або відправляє форму. Браузер жертви, який все ще має дійсні сесійні cookie, автоматично відправляє шкідливий запит на веб-сайт. Цей запит може виконувати такі дії: переведення коштів; зміна паролів; видалення акаунтів; публікація повідомлень від імені користувача.

CSRF небезпечний, оскільки зловмисник використовує довіру користувача, тобто атакуючий використовує існуючу автентифікацію користувача для виконання дій, які користувач ніколи б не дозволив.

Фактично, атаки CSRF використовують довіру, яку веб-сайти надають автентифікованим користувачам. Впроваджуючи відповідні заходи безпеки, розробники можуть значно зменшити ризик таких атак.

1.2.5 Відмова в обслуговуванні (DoS)

DoS (Denial of Service) – це тип кібератаки, спрямованої на те, щоб зробити певний сервіс недоступним для його легітимних користувачів. Іншими словами, зловмисник штучно створює таке навантаження на систему, що вона перестає справлятися зі своїми функціями. Ця атака

спрямована на припинення або сповільнення роботи сервера/мережі шляхом перевантаження його ресурсів.

Причини цих атак можуть бути різними: вандалізм (зловмисник просто хоче завдати шкоди), конкуренція (конкурент може здійснити атаку, щоб тимчасово вивести з ладу сайт конкурента), шантаж (зловмисник може вимагати гроші за припинення атаки), а також політичні мотиви (атаки можуть бути спрямовані проти урядів, організацій або окремих осіб).

Наслідки використання уразливостей від DoS-атак можуть містити:

- крадіжку даних, оскільки зловмисники можуть отримати доступ до конфіденційної інформації, такої як номери кредитних карток, паролі та особисті дані;
- несанкціонований доступ, коли зловмисники можуть отримати доступ до систем і мереж, що дозволяє їм здійснювати інші злочинні дії;
- відмову в обслуговуванні, оскільки зловмисники можуть зробити сервіси недоступними для законних користувачів;
- фінансові втрати, адже компанії можуть зазнати значних фінансових збитків внаслідок кібератак;
- пошкодження репутації, оскільки кібератаки можуть негативно вплинути на репутацію компанії.

1.3 Реальні приклади уразливостей та їх наслідки

Одна з необхідних умов створення захищених систем – проведення аналізу успішно здійснених порушень безпеки з метою їх узагальнення і класифікації задля виявлення причини і закономірності появи та існування уразливостей. Це дає змогу в подальшому, під час розроблення систем захисту, спрямовувати зусилля на усунення першопричин появи уразливостей, що допомагає ефективніше протидіяти загрозам.

Злам продуктів компанії Barracuda Networks, 2023 рік

У 2023 році було атаковано компанію Barracuda Networks, зламано її програмне забезпечення для електронної пошти Email Security Gateway (ESG). Хакери, які, ймовірно, були вихідцями з Китаю, скористалися цією вразливістю, щоб націлитися на сотні клієнтів Barracuda по всьому світу. Компанія, що спеціалізується на забезпеченні кібербезпеки, сама стала жертвою масштабної атаки.

Зловмисники експлуатували вразливість нульового дня, яка дозволила їм отримати доступ до систем компанії та її клієнтів. Це означає, що про цю вразливість не було відомо розробникам, і відповідно, не існувало жодних патчів для її усунення. Зловмисники скористалися цією

вразливістю для отримання доступу до внутрішньої мережі компанії та до даних її клієнтів.

Це була масштабна атака, яка зачепила тисячі організацій по всьому світу, включно державні установи та приватні компанії. Зловмисники, ймовірно, експлуатували цю вразливість протягом тривалого часу до того, як вона була виявлена.

Атака мала досить серйозні наслідки:

- вона могла призвести до витоку конфіденційної інформації, такої як листування, паролі та інші дані, що зберігалися на цих пристроях, до переривання роботи електронної пошти та інших сервісів, а також до інфікування систем шкідливим програмним забезпеченням;
- внаслідок атаки були скомпрометовані тисячі пристроїв ESG, встановлених у клієнтів Barracuda Networks по всьому світу. Barracuda Networks порадила своїм клієнтам фізично замінити все скомпрометоване обладнання ESG, оскільки програмні оновлення були неефективними для усунення загрози;
- злам компанії, що спеціалізується на кібербезпеці, серйозно підірвав довіру до її продуктів та послуг, тобто призвів до порушення репутації;
- компанія зазнала значних фінансових збитків, пов’язаних з розслідуванням інциденту, відновленням систем та компенсацією збитків клієнтам.

Barracuda Networks активно співпрацювала з правоохоронними органами для виявлення та затримання зловмисників. Компанія провела масштабне розслідування для визначення масштабів атаки та вжиття заходів для запобігання подібних інцидентів у майбутньому.

Злам MOVEit, 2023 рік

MOVEit – це програмне забезпечення, розроблене компанією Progress Software (раніше Ipswitch), яке призначене для безпечного передавання файлів між підприємствами та їх клієнтами. Воно використовується для управління та автоматизації процесу обміну великими обсягами даних, забезпечуючи при цьому високий рівень безпеки.

MOVEit широко використовується в різних галузях для вирішення таких завдань, як: передавання великих файлів (архіви, відео та зображення), забезпечення безпечного обміну файлами між співробітниками різних відділів, безпечний обмін файлами з зовнішніми партнерами, створення резервних копій важливих даних.

Цей програмний продукт широко використовується корпоративними організаціями. У 2023 році російські хакери виявили та використали вразливість у MOVEit для крадіжки даних таких організацій, як бухгалтерські фірми Big 4, UCLA, Siemens Energy тощо.

Проблема MOVEit стала однією з найбільших кібербезпекових катастроф 2023 року.

У MOVEit була виявлена критична вразливість, яка дозволяла зловмисникам виконати віддалений код. Ця вразливість дозволила хакерам отримати несанкціонований доступ до величезних обсягів даних у багатьох компаніях по всьому світу. Це означало, що хакер міг отримати повний контроль над системою, що використовувала MOVEit, і виконувати на ній будь-які дії, включно:

- викрадення даних, тобто хакери могли завантажувати з системи будь-які файли, включно конфіденційні дані компанії та її клієнтів;
- установлення шкідливого програмного забезпечення для подальшого контролю або для поширення атаки на інші системи;
- використання системи для атак на інші цілі, використовуючи скомпрометовану систему як плацдарм для атак на інші мережі та системи.

Наслідки атаки з використанням уразливості MOVEit були катастрофічними для багатьох компаній. Це і витік конфіденційних даних, і фінансові збитки, і репутаційні втрати. Злам серйозно підірвав довіру до компаній, які використовували MOVEit, оскільки це свідчило про те, що їхні системи безпеки були недостатньо захищені.

Злам SolarWinds, 2020 рік

SolarWinds – це американська компанія, яка розробляє програмне забезпечення для управління IT-інфраструктурою. Їхні продукти широко використовуються організаціями різного масштабу для моніторингу, аналізу та управління мережами, серверами, базами даних та іншими компонентами IT-систем.

У грудні 2020 року стало відомо про одну з найбільших та найскладніших кібератак в історії – злам програмного забезпечення компанії SolarWinds. Зловмисники змогли проникнути до програмного забезпечення SolarWinds Orion, яке використовується тисячами організацій по всьому світу для моніторингу та управління IT-інфраструктурою.

Хоча точна ідентифікація зловмисників є складною, значного поширення набула версія про причетність до атаки російської державної хакерської групи. Однак, остаточні висновки щодо цього питання ще не зроблені.

Зловмисники пробралися в мережу SolarWinds і підмінили легітимне програмне забезпечення на заражену версію. Коли клієнти SolarWinds встановлювали оновлення, вони насправді встановлювали бекдор, який дозволяв хакерам отримати доступ до їхніх систем. Цей бекдор був надзвичайно складним і добре замаскованим, що дозволяло хакерам довгий час залишатися непоміченими. Потім сотні компаній і державних

установ завантажили це заражене програмне забезпечення, надаючи Росії доступ до найвищого рівня Корпоративної Америки та уряду США.

Цей злам став однією з найбільших і найскладніших кібератак в історії. Наслідки атаки SolarWinds були масштабними і далекосяжними:

- масове зараження систем, оскільки тисячі організацій по всьому світу були заражені шкідливим програмним забезпеченням;
- витік конфіденційної інформації, оскільки хакери могли отримати доступ до величезних обсягів конфіденційної інформації, включно урядові та комерційні таємниці;
- підрив довіри до програмного забезпечення та постачальників програмного забезпечення в цілому;
- посилення міжнародної напруженості, оскільки атака стала ще одним приводом для посилення міжнародної напруженості, особливо у відносинах між США та Росією.

Атака вірусу WannaCry, 2017 рік

WannaCry – вірус-шифрувальник, який використовував вразливість у операційній системі Windows для зашифрування файлів на комп'ютерах жертв і вимагав викуп за їх розшифрування. Це був один з найбільших і найруйнівніших вірусів-вимагачів, який атакував комп'ютери по всьому світу у травні 2017 року. Ця кібератака використала вразливість у операційній системі Windows, що дозволило вірусу швидко поширюватися мережами і шифрувати файли на заражених комп'ютерах.

WannaCry використовував вразливість EternalBlue, яка була розроблена Агентством національної безпеки США (АНБ). Ця вразливість дозволяла зловмисникам отримати віддалений доступ до комп'ютера та виконувати на ньому довільний код і дозволяла вірусу самостійно поширюватися по мережі, заражаючи інші вразливі комп'ютери.

Після зараження комп'ютера, WannaCry шифрував файли користувача, роблячи їх недоступними. Потім вірус виводив на екран повідомлення з вимогою викупу в біткоінах для розшифрування файлів.

Наслідки атаки WannaCry були катастрофічними:

- масове шифрування файлів, тобто мільйони комп'ютерів по всьому світу були заражені і їхні файли були зашифровані;
- зупинення роботи підприємств через те, що їхні комп'ютери були заражені і стали недоступними;
- значні фінансові втрати, пов'язані з відновленням систем, втратою даних і виплатою викупу;
- підрив довіри користувачів до комп'ютерних систем та їх безпеки.

Атака вірусу Petya, 2017 рік

Petya (також відоме як NotPetya) – це шкідливе програмне забезпечення, яке використовувало вразливість EternalBlue для шифрування даних

на комп'ютерах жертв. Хоча на перший погляд Petya може здаватися схожим на вимагач WannaCry, який також використовував EternalBlue, ці дві атаки мають значні відмінності.

Як і WannaCry, Petya використовував вразливість EternalBlue для початкового зараження комп'ютерів. Після зараження Petya зашифровував таблицю розділів жорсткого диска, що робило неможливим завантаження операційної системи.

На відміну від WannaCry, Petya мала більш агресивну стратегію саморозповсюдження, намагаючись заразити всі комп'ютери в мережі. WannaCry був спроєктований для шифрування даних і вимагання викупу, тоді як Petya, ймовірно, мав на меті завдати максимальної шкоди інфраструктурі, а не отримати фінансову вигоду. WannaCry шифрував окремі файли, тоді як Petya шифрував всю таблицю розділів диска.

Масова атака вірусу Petya розпочалася 27 червня 2017 року. Цей день запам'ятався як один з наймасштабніших кібернападів в історії, особливо сильно постраждали Україна та інші країни Європи. Хоча перша версія вірусу з'явилася ще в 2016 році, саме в червні 2017 року відбулася його найбільш руйнівна хвиля, яка скористалася вразливістю EternalBlue.

Атака Petya завдала значних збитків багатьом компаніям і організаціям по всьому світу. Особливо сильно постраждали компанії в Україні, де атака почалася з поширення через програмне забезпечення для бухгалтерського обліку M.E.Doc.

Витік даних у компанії Equifax, 2017 рік

Equifax – це одна з трьох найбільших кредитних бюро у Сполучених Штатах Америки (разом з Experian та TransUnion). Ця компанія збирає, аналізує та зберігає інформацію про кредитну історію мільйонів людей, включно імена, адреси, дати народження, номери соціального страхування та історії кредитів.

Саме тому, коли в 2017 році стало відомо про масштабний витік даних у Equifax, це викликало величезний резонанс у всьому світі.

Хакери скористалися вразливістю в веб-додатку Apache Struts, яка була відома і для якої вже існував патч. Equifax не встигла вчасно застосувати цей патч, що дозволило зловмисникам проникнути в їхню систему і викрасти персональні дані понад 147 мільйонів людей.

Наслідки цього інциденту були катастрофічними:

- масштабний витік персональних даних, оскільки мільйони людей стали жертвами витоку своїх найінтимніших даних, що створило для них серйозні ризики для фінансової безпеки та ідентичності;
- Equifax зазнала значних фінансових збитків, пов'язаних з розслідуванням інциденту, юридичними витратами та компенсаціями постраждалим;

– довіра до Equifax, як до компанії, що відповідає за захист персональних даних, була серйозно підірвана.

Вразливість Heartbleed, 2014 рік

Heartbleed – це помилка в реалізації розширення Heartbeat протоколу TLS/DTLS в OpenSSL. Ця вразливість у бібліотеці OpenSSL дозволяла зловмисникам викрадати конфіденційні дані, такі як паролі та ключі шифрування, з великої кількості веб-серверів. Це розширення використовується для підтвердження того, що з'єднання між клієнтом і сервером все ще активне. Однак, через помилку в коді, атакуючий міг попросити сервер відправити більше даних, ніж було запрошено, що дозволяло йому читати частину пам'яті сервера.

Heartbleed – це була одна з найсерйозніших уразливостей в історії Інтернету, яка викликала значний резонанс у 2014 році. Вона вразила фундаментальну бібліотеку OpenSSL, що використовується для шифрування даних в мережі, і дозволяла зловмисникам викрадати конфіденційну інформацію з мільйонів веб-серверів по всьому світу.

Зловмисники могли витягти приватні ключі серверів, які використовуються для шифрування даних. Це дозволяло їм розшифрувати будь-який трафік, що передається між сервером і клієнтом.

Крім того, з пам'яті сервера могли бути витягнуті імена користувачів і паролі, що зберігалися в незашифрованому вигляді.

Звичайно, вплив Heartbleed був катастрофічним:

– мільйони користувачів стали жертвами витоку своїх особистих даних;
– ця подія серйозно підірвала довіру користувачів до безпеки в інтернеті;
– компанії зазнали значних фінансових збитків через необхідність змінювати паролі, сертифікати та проводити інші заходи для усунення наслідків атаки.

Як бачимо, прикладів зламу програмного забезпечення чимало. І всі вони призводять до витоку конфіденційної інформації, компрометації тисяч пристроїв і програм, до значних фінансових збитків та репутаційних втрат.

Всі ці інциденти призводять до посилення вимог до захисту даних у багатьох країнах, а, отже, посилення вимог безпеки до програмного забезпечення інформаційних та комп’ютерних систем.

Отож, враховуючи нескінченну низку кібератак, які вдаються завдяки використанню уразливостей програмного забезпечення, організаціям стало важливо купувати та використовувати лише найбезпечніше програмне забезпечення.

2 ОСНОВНІ ЗАСАДИ СТВОРЕННЯ БЕЗПЕЧНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Необхідність дотримання політики безпеки ПЗ

Безпека програмного забезпечення – це сукупність заходів, спрямованих на захист програмних систем від несанкціонованого доступу, використання, розкриття, зміни, пошкодження або знищення. В сучасному цифровому світі, де інформація є одним з найцінніших активів, забезпечення безпеки програмного забезпечення є критично важливим.

Сукупність практик, що об'єднують розробку (Development) та його обслуговування (Operations) програмного забезпечення називають DevOps. Головна мета DevOps – забезпечити швидку, надійну та безперервну доставку програмних продуктів до користувачів. DevSecOps – це розширення концепції DevOps, яке ставить акцент на безпеці. Якщо DevOps об'єднує розробку та експлуатацію, то DevSecOps додає до цього ще й безпеку.

Іноді розробники розглядають безпеку як перешкоду для інновацій і творчості, що створює затримки у виході продукту на ринок. Таке мислення завдає шкоди прибутку бізнесу, оскільки виправлення помилки під час впровадження і тестування коштує набагато дорожче, ніж виправлення тієї ж помилки під час розробки.

Навряд чи клієнти-замовники будуть задоволені новими функціями програми, якщо продукт містить уразливості, якими можуть скористатися хакери. Сьогодні безпека заслуговує на провідне місце в процесі розробки програмного забезпечення, і організації мають робити це, щоб мати можливість конкурувати.

Отже, головна проблема, з якою ми стикаємось щодня: забезпечення безпеки програмного забезпечення в умовах дедалі складнішого цифрового середовища. І це є надзвичайно важливим, оскільки:

- безпечне ПЗ захищає конфіденційні дані користувачів, такі як особиста інформація, фінансові дані та дані про здоров’я;
- кібератаки можуть призвести до значних фінансових втрат через крадіжку коштів, викуп або репутаційні збитки;
- хакерські атаки можуть порушити роботу критично важливих систем: енергосистем, транспортних мереж, медичних установ тощо;
- порушення безпеки призводить до втрати довіри клієнтів і партнерів.

Безпечна розробка програмного забезпечення – це методологія (часто пов’язана з DevSecOps) для створення програмного забезпечення, яке містить безпеку на кожному етапі життєвого циклу розробки програмного забезпечення (SDLC). Безпека вбудована в код із самого початку, а не розглядається після того, як тестування виявить критичні

недоліки продукту. Безпека стає частиною етапу планування, передбаченою задовго до написання єдиного рядка коду.

2.2 Принципи нульової довіри (Zero-Trust)

Протягом останнього десятиліття компанії почали децентралізувати свої дані, активи, додатки та сервіси (DAAS) у різних середовищах та у постачальників хмарної інфраструктури. Ця децентралізація зробила неефективною традиційну стратегію безпеки «замок і рів», оскільки безпека мережі більше не може обмежуватися одним місцем розташування, набором пристроїв або користувачів. Концепція нульової довіри була розроблена, щоб допомогти сучасним компаніям захистити свої найцінніші активи в цьому розподіленому хмарно-рідному середовищі.

Нульова довіра базується на ідеї, що не існує традиційного периметра мережі, а це потребує розробки системи, яка передбачає, що всі користувачі та сервіси є потенційною загрозою, навіть якщо вони знаходяться у мережі. Система потребуватиме постійного оцінення запитів на доступ перед підключенням до будь-яких додатків та сервісів. Логіни, з'єднання та токени API будуть короткочасними, а користувачі та пристрої постійно підтверджуватимуть свою ідентичність та привілеї (рис. 2.1).

Цей підхід «ніколи не довіряй, завжди перевіряй» дозволяє уважно стежити за доступом до DAAS. У хмарно-рідному світі, де користувачі можуть бути фізично розподілені, використовувати кілька пристроїв або намагатися отримати доступ до DAAS із захищених та незахищених мереж, ваша організація потрібно, щоб мала строгий контроль доступу, постійне оцінення та максимальну спостережливість.

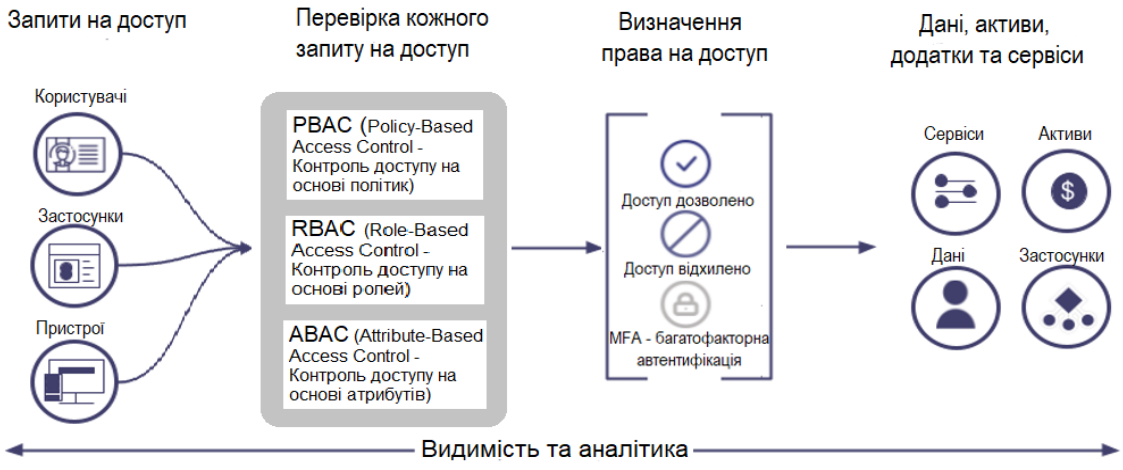


Рисунок 2.1 – Архітектура Zero-Trust (нульова довіра)

Структура Zero-Trust базується на чотирьох основних принципах:

1. *Ніколи не довіряй, завжди перевіряй*. Розроблена система має постійно «вимагати» від користувачів і служб перевірку їхніх ідентифікаторів, пристроїв, місцеположень та інших атрибутів даних, щоб переконатися, що лише привілейовані користувачі та служби мають доступ до конфіденційного ресурсу. Токени, сеанси та з'єднання мають бути короткочасними, а користувачам і службам має бути запропоновано повторно пройти автентифікацію, щоб продовжити доступ до конфіденційних ресурсів.

2. *Безперервний моніторинг і спостереження*. Постійний моніторинг і можливість спостереження дають змогу в реальному часі зрозуміти, які користувачі намагаються отримати доступ до ресурсів, а також результати цього оцінення. Крім того, він надає командам мережі та безпеки інформацію в режимі реального часу про потенційні загрози, аномальну поведінку та активні інциденти безпеки. Це дозволяє їм швидко діяти, щоб усунути будь-які інциденти та обмежити радіус вибуху потенційного порушення.

3. *Найменші привілеї*. Забезпечення того, щоб користувачі мали доступ лише до мінімуму необхідних ресурсів, є основним принципом системи нульової довіри. Важливо точно розуміти, кому з користувачів до яких ресурсів потрібен доступ і що їм потрібно робити з цими ресурсами, щоб обмежити несанкціонований доступ. Це ключовий компонент принципу мікросегментації.

4. *Мікросегментація*. Можна мінімізувати масштаб і радіус вибуху порушення або інциденту безпеки, сегментувавши свій DAAS на менші, більш цілеспрямовані сегменти у своїй мережі. Ці сегменти мережі незалежні один від одного та розроблені, щоб запобігти переміщенню зловмисників по мережі. Кожен сегмент має власний набір користувачів, ролей і політик доступу, які постійно оцінюються та контролюються.

2.3 Топ 10 кращих практик безпечного циклу розробки програмного забезпечення (SSDLC)

У сучасному цифровому середовищі безпека програмного забезпечення важлива як ніколи, оскільки ПЗ використовується для зберігання та обробки величезних обсягів конфіденційної інформації. Якщо програмне забезпечення не розроблено з урахуванням безпеки, воно може бути вразливим до атак, які можуть скомпрометувати конфіденційні дані та завдати шкоди окремим особам і організаціям.

SSDLC (Secure Software Development Lifecycle) – це методологія розробки програмного забезпечення, яка інтегрує безпеку на кожному етапі життєвого циклу, починаючи від концепції та закінчуючи виведенням продукту з експлуатації.

Отже, безпека програмного забезпечення – це постійний процес, який потребує постійної уваги і зусиль. Дотримуючись наведених далі 10 практик, можна значно підвищити рівень захисту систем і даних (рис. 2.2).

Цей перелік практик формувався на основі аналізу попередніх інцидентів (вивчення причин порушень безпеки та розробка заходів для їх запобігання), наукових досліджень (в галузі криптографії, теорії інформації та інших дисциплін) та практичного досвіду розробників та фахівців з безпеки, які стикаються з реальними проблемами.

Рисунок 2.2 – Найкращі практики безпечного циклу розробки ПЗ

2.3.1 Безпека з самого початку

Ще до початку написання коду необхідно планувати, як інтегрувати захист у кожну фазу SDLC. Використовувати потужність автоматизації для тестування та моніторингу уразливостей з першого дня.

Безпека має бути внесена у культуру програмування та код на самій ранній стадії розробки.

Почати варто з визначення потенційних загроз для розроблюваних систем на етапі проєктування. Моделювання загроз – це критично важливий процес, який допомагає передбачити й пом’якшити потенційні уразливості, перш ніж їх можна буде використати у програмному забезпеченні.

2.3.2 Безпечна політика розробки ПЗ

Ця практика спрямована на те, щоб команди розробників використовували добре встановлені та перевірені рішення безпеки. Це важливо, оскільки безпечні рішення потребують міцної основи, і досвід навчив нас, що спроби винайти нові рішення є складною справою та майже завжди призводять до збільшення ризику безпеки та марної втрати часу та зусиль.

Крім того, деякі аспекти проєктування та розробки програмного забезпечення є надто важливими, щоб залишити їх невизначеними, а такі області, як автентифікація та авторизація, а також пов'язане та необхідне журналювання для аудиту є основними засобами контролю, на яких базується багато інших засобів контролю безпеки, і організації мають стандартизувати підхід, що надає чіткі послідовні вказівки з обов'язковими обмеженнями та засобами для перевірки їх виконання відповідно до необхідного стандарту.

Отже, ця практика містить:

- рекомендації для підготовки членів команди, процесів і технологій для виконання безпечної розробки програмного забезпечення;
- конкретні інструкції щодо підходу та інструментування безпеки на кожному етапі SDLC;
- керівні правила й визначення ролей, щоб допомогти членам команди, процесам та інструментам мінімізувати ризик уразливості під час виробництва програмного забезпечення.

2.3.3 Використання безпечної структури розробки ПЗ

Практики безпечного кодування є важливими. Необхідно переконатися, що розробники навчені безпечним стандартам кодування, які стосуються мов програмування та платформ, котрі вони використовують. Регулярне оновлення цих стандартів для відображення нових загроз також має вирішальне значення.

Для цього варто визначити та опублікувати список схвалених інструментів і пов'язаних з ними перевірок безпеки, таких як параметри компілятора і компонувача та попередження.

Необхідно прагнути використовувати найновіші версії схвалених інструментів, наприклад, версії компіляторів, і використовувати переваги нових функцій аналізу безпеки та засобів захисту.

Перевірена структура, як-от NIST SSDF, додають узгодженості зусиллям команди розробників щодо дотримання найкращих практик безпечного програмного забезпечення. Фреймворки можуть допомогти відповісти на питання «Що нам робити далі?» та принести користь для всіх нових розробників програмного забезпечення.

2.3.4 Урахування найкращих практик, щоб відповідати вимогам безпеки

Переконатися, що всі учасники розробки і сторонні постачальники знають про необхідні вимоги безпеки та демонструють відповідність, оскільки вони можуть забезпечити простий шлях для атаки.

Наприклад, вимагати відповідності таким вимогам безпеки, як GDPR, HIPAA, PCI DSS або іншим. GDPR, HIPAA та PCI DSS – це все регулювання, спрямовані на захист конфіденційних даних:

GDPR (Загальний регламент захисту даних). Це регулювання ЄС зосереджено на захисті персональних даних фізичних осіб у межах Європейського Союзу. Воно застосовується до будь-якої організації, яка збирає або обробляє персональні дані резидентів ЄС, незалежно від місцезнаходження організації. GDPR наголошує на конфіденційності даних, згоді та праві на забуття.

HIPAA (Закон про переносимість та підзвітність медичного страхування). Це регулювання США захищає захищену медичну інформацію (PHI). Воно встановлює стандарти для постачальників медичних послуг, планів медичного страхування та клірингових центрів охорони здоров'я для захисту конфіденційності та безпеки інформації про здоров'я пацієнтів.

PCI DSS (Стандарт безпеки даних платіжних карток). Цей стандарт застосовується до будь-якої організації, яка обробляє інформацію про кредитні картки. Він спрямований на захист даних власників карток та запобігання шахрайству. PCI DSS визначає вимоги безпеки для зберігання, обробки та передачі даних власників карток.

2.3.5 Захист цілісності коду

Весь код необхідно зберігати у безпечних сховищах, дозволяючи лише авторизований доступ, щоб запобігти втручанню і строго регулювати всі контакти з кодом, відстежувати зміни та уважно спостерігати за процесом підписання коду, щоб зберегти цілісність.

Сьогодні переважна більшість проєктів програмного забезпечення будується з використанням сторонніх компонентів (як комерційних, так і з відкритим кодом). Вибираючи сторонні компоненти для використання, важливо розуміти вплив на безпеку загальної системи, в яку вони інтегровані; завжди потрібно бути більш вибірковими, використовуючи компоненти високого ризику, і проводити ретельний аналіз перед їх використанням.

Сьогодні існує безліч способів використання OSS (Open Source Software) розробниками: git clone, wget, копіювання та вставлення вихідного коду, реєстрація двійкового файлу в репозиторії, безпосередньо з загальнодоступних менеджерах пакетів, перепакування OSS у .zip, curl,

apt-get, підмодуль git тощо. Захистити ланцюжок постачання OSS майже неможливо, якщо команди розробників не дотримуються єдиного процесу використання OSS.

2.3.6 Перегляд та тестування коду завчасно та часто

Необхідно здійснювати перегляд коду і автоматичне тестування, щоб постійно перевіряти код на недоліки. Виявлення уразливостей на ранніх стадіях життєвого циклу економить гроші та час і запобігає розчаруванню розробників у подальшому.

Впровадження ретельного процесу перевірки коду є одним із найкращих способів завчасного виявлення уразливостей. Варто використовувати автоматизовані інструменти та експертні перевірки, щоб перевірити код з погляду його безпеки.

Тестування безпеки має бути інтегровано у регулярний режим тестування – це тестування на проникнення, сканування уразливостей і використання автоматизованих інструментів для постійного тестування та моніторингу слабких місць.

2.3.7 Готовність швидко пом’якшити вразливі місця

Уразливості є фактом у розробці програмного забезпечення. Справа не в тому, чи трапляються вони, а в тому, коли, тож потрібно бути готовими, маючи в наявності команду, план і процеси для вирішення інцидентів у режимі реального часу. Чим швидше можна буде визначити вразливі місця та відреагувати на них, тим швидше можна скоротити коло можливостей для їх використання. Для цього необхідно:

- підтримувати всі системи в актуальному стані за допомогою останніх виправлень безпеки, регулярно оновлюючи життєво важливі для захисту від відомих уразливостей;
- мати надійний план реагування на інциденти, який дозволяє швидко виявляти, стримувати та пом’якшувати будь-які порушення, що виникають;
- впроваджувати постійний моніторинг своїх систем, щоб виявляти загрози та реагувати на них у реальному часі. Такий підхід може значно зменшити потенційний вплив будь-якого інциденту безпеки.

Визначаючи пом’якшення, варто мати на увазі, що безпека – це не «все або нічого». Часткове пом’якшення, яке підвищує вартість для зловмисника, уповільнює його, щоб дати захисникам час виявити їх, або обмежує масштаб збитку, набагато краще, ніж жодного пом’якшення взагалі.

Необхідно думати про багатошаровий захист. Зловмисники не просто використовують одну вразливість і зупиняються на досягнутому. Вони з’єднують численні уразливості разом, переходячи від однієї цільової

системи до іншої, доки не досягнуть своєї мети (або не будуть спіймані). Кожен рівень захисту збільшує ймовірність того, що зловмисники будуть заблоковані або виявлені.

Крім того, припустимо, що елементи керування безпекою інших рівнів буде обійдено або вимкнено. Це суть філософії «Припустити порушення», яка забезпечує надійний набір багаторівневих засобів захисту, а не покладається виключно на зовнішні засоби захисту, які, якщо їх обійти, призведуть до значного порушення.

2.3.8 Налаштування параметрів безпеки за замовчуванням

Велика кількість клієнтів залишаються вразливими просто через брак знань про функції свого нового програмного забезпечення. Цей додатковий елемент обслуговування клієнтів гарантує, що споживач буде захищений вже на ранніх етапах.

1. Необхідно встановити базовий рівень мінімальної безпеки, який враховує засоби контролю безпеки та відповідності і переконатися, що вони «включені» в процес і конвеєр DevOps. Цей базовий рівень має враховувати принаймні реальні загрози, такі як OWASP Top 10 або SANS Top 25, а також галузеві або нормативні вимоги та проблеми, які, як відомо, існують або можуть бути внесені людською помилкою у вибраний стек технологій.
2. Необхідно підтримувати безпечну базову конфігурацію для всіх середовищ розробки та виробництва, автоматизувати процес керування конфігурацією, щоб зменшити ризик людської помилки.
3. Варто переконатися, що доступ до програмного забезпечення та даних строго контролюється. Для цього можна реалізувати принцип найменших привілеїв, гарантуючи, що люди мають доступ лише до ресурсів, необхідних для виконання їхніх ролей.
4. Потрібно використовувати *підхід нульової довіри* (пп. 2.4). За своєю суттю модель нульової довіри потребує, щоб кожен запит на доступ (користувача, служби чи пристрою) перевірявся так, ніби він походить із ненадійної мережі, незалежно від того, звідки надходить запит або до якого ресурсу він отримує доступ. Ця політика передбачає здійснення автентифікації та авторизації на всіх доступних точках даних, обмеження доступу користувачів, особливо привілейованих користувачів, за допомогою політик Just-In-Time та Just Enough-Access (JIT/JEA) і сегментування доступ, щоб мінімізувати можливу шкоду в разі порушення.

2.3.9 Використання контрольних списків

Під час безпечної розробки програмного забезпечення є багато рухомих частин, які потрібно відстежувати та контролювати. Необхідно,

щоб команда розробників через періодичні проміжки часу, використовуючи контрольні списки дій, проводила, наприклад, щотижневі або щомісячні зустрічі, щоб переконатися, що всі необхідні політики та процедури безпеки актуальні та функціональні.

2.3.10 Залишатися гнучкими та активними

Для того, щоб у сучасному світі бути успішними розробниками програмного забезпечення, необхідно вивчати уразливості: знаходити їх першопричини, виявляти шаблони, запобігати повторним випадкам і оновлювати свій SDLC за допомогою покращених знань. Необхідно також спостерігати за тенденціями та залишатися в курсі найкращих практик.

Простір постійно розвивається; найкращі методи безпеки не стоять на місці. Тому незалежно від того, що з безпекою нині, дуже важливо дивитися вперед, щоб бачити, що буде, продовжувати навчатися та визначати кращі способи захисту процесу розробки програмного забезпечення. Головне – це знати та бути в курсі галузі та найкращих практик

Регулярне навчання та програми підвищення обізнаності для розробників і IT-персоналу мають вирішальне значення, щоб переконатися, що вони обізнані з останніми загрозами безпеці та передовими методами.

2.4 SSDF – безпечна політика розробки програмного забезпечення

SSDF або Рамка безпечної розробки програмного забезпечення – це набір рекомендацій та найкращих практик, розроблений Національним інститутом стандартів і технологій США (NIST), який має на меті забезпечити безпеку програмного забезпечення на всіх етапах його життєвого циклу.

Ознайомившись із типами загроз і уразливостей, з якими стикається сучасне інформаційне суспільство, проаналізувавши методологію розробки програмного забезпечення SSDLC, перейдемо до того, як можна структурно захистити свої проєкти.

Secure Software Development Framework (SSDF) пропонує надійний план для впровадження безпеки протягом життєвого циклу розробки. Ця структура – це не просто набір інструкцій; це проактивний підхід до створення надійної основи для всіх програмних проєктів.

Багато організацій отримують вигоду від узгодження своєї практики з добре встановленою структурою, такою як NIST's SSDF. Такі організації, як OWASP і SAFEcode, створили колекцію основоположних ресурсів для розробки безпечного програмного забезпечення, які детально обговорюють безпеку програмного забезпечення, надаючи необхідні вказівки для змен-

шення кількості, зменшення впливу та запобігання майбутнім уразливостям у випусках програмного забезпечення.

Процес безпечної розробки програмного забезпечення NIST рекомендує організовувати в чотири етапи (рис. 2.3).

1. *Підготовка організації* (PO – Prepare the Organization): необхідно переконатися, що люди, процеси та технології організації підготовлені до безпечної розробки програмного забезпечення на рівні організації та, у деяких випадках, для кожного окремого проєкту.
2. *Захистити програмне забезпечення* (PS – Protect the Software): захистити усі компоненти програмного забезпечення від втручання та несанкціонованого доступу.
3. Розробляти *добре захищене програмне забезпечення* (PW – Produce Well-Secured Software): програмне забезпечення, яке має мінімальну вразливість у своїх випусках.
4. Забезпечити *реагування на уразливості* (RV – Respond to Vulnerabilities): визначити уразливості у випусках програмного забезпечення та належним чином реагувати на ці уразливості та запобігати виникненню подібних уразливостей у майбутньому.

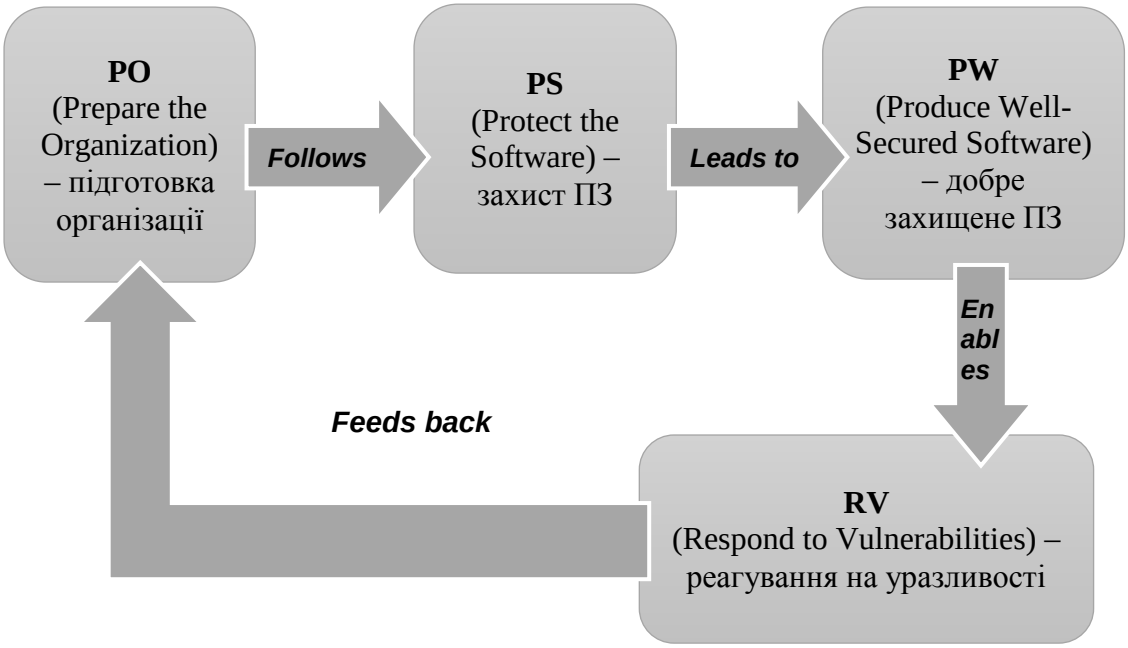


Рисунок 2.3 – Етапи безпечної розробки програмного забезпечення за рекомендаціями NIST

Кожен з наведених етапів визначається такими елементами:

– *практика* – короткий опис практики разом із унікальним ідентифікатором і поясненням того, що таке практика та чому вона корисна;
– *завдання* – окрема дія (або дії), необхідна для виконання практики;

- *приклад реалізації* – заданий сценарій, який можна використати для демонстрації практики;
- *довідка* – встановлений безпечний практичний документ розробки та його відображення для конкретного завдання.

2.4.1 Підготовка організації

Щоб найкраще узгодити організацію з новою політикою, важливо чітко визначити вимоги безпеки. Цей процес містить:

- чітке визначення як внутрішніх (наприклад, політики, стратегії управління ризиками), так і зовнішніх (наприклад, закони, нормативні акти) вимог щодо безпеки розробки ПЗ для організації;
- призначення ролей SSDF та підготовка команд за допомогою спеціального навчання;
- залучення допоміжних інструментів для підвищення швидкості та ефективності в SDLC;
- встановлення перевірок безпеки, щоб переконатися, що програмне забезпечення відповідає стандартам організації.

Завдання містять визначення, передачу та підтримку всіх вимог безпеки протягом тривалого часу: підготовка режимів тренувань; залучення управлінської підтримки; підбір інструментів; визначення контрольних показників для документування досягнення встановлених стандартів безпеки. Наприклад, це можуть бути такі завдання:

- визначення особливостей, як-от методи кодування та вимоги до архітектури для розробників;
- перегляд і оновлення вимог безпеки принаймні раз на рік, особливо після інцидентів;
- призначення планів навчання ролей, пов’язаних із SSDF, а потім встановлення періодичних перевірок і підготовка до оновлення щодо будь-яких змін ролі з часом;
- визначення категорій ланцюга інструментів, визначення інструментів для кожного з них і включення автоматизації для роботи та управління ланцюгом інструментів;
- створення контрольного журналу безпечних дій, пов’язаних із розробкою;
- визначення ключових показників ефективності;
- використання автоматизованого інструментарію для збору відгуків, а також перегляд і документування всіх доказів перевірки безпеки для підтримки визначених стандартів.

2.4.2 Захист програмного забезпечення

Захист коду та забезпечення цілісності програмного забезпечення, доки він не досягне кінцевого споживача, має першочергове значення. Цей процес спрямований на захист коду від несанкціонованого доступу та втручання, перевірку цілісності програмного забезпечення та захист програмного забезпечення після випуску.

Основними завданнями на цьому етапі є збереження коду на основі принципу найменших привілеїв, використання практик для забезпечення лише авторизованого доступу. Крім того, копія кожного випуску з перерахованими компонентами та інформацією про перевірку цілісності надається кожному клієнту.

Цей етап містить, крім того, і багато іншого, наприклад:

- зберігання коду в безпечних сховищах з обмеженим доступом;
- використання контролю версій для відстеження всіх змін коду;
- публікація криптографічних гешів для випущеного програмного забезпечення та використання лише довірених центрів сертифікації для підписання коду.

2.4.3 Створення добре захищеного ПЗ

Як говорилося вище, процес створення безпечного програмного забезпечення складається з багатьох кроків і залучає багатьох учасників і величезну кількість практик.

Конкретні *завдання* цього етапу можуть містити:

- створення списку довірених компонентів;
- використання моделювання загроз для оцінювання ризиків;
- вивчення зовнішніх вимог безпеки,
- передавання стандартів третім особам під час перевірки їх відповідності,
- використання найкращих практик безпечного кодування з використанням найкращих галузевих інструментів та перевірка коду з усіх поглядів шляхом перегляду або аналіз;
- розробка та виконання тестів на уразливості, документування результатів і усунення всіх виявлених проблем;
- встановлення безпечних параметрів за замовчуванням, забезпечення їх узгодженості з іншими функціями безпеки платформи, а потім пояснення їх важливості адміністраторам.

2.4.4 Реагування на уразливості

Організаціям потрібно мати не лише можливість виявляти уразливості, але й ефективно реагувати на них.

Пошук уразливостей є лише частиною роботи професіонала з безпеки, іншим критичним компонентом є виправлення. Цей останній процес зосереджується на виправленні поточних уразливостей і зборі даних для майбутнього запобігання. Після виявлення та підтвердження уразливості необхідно швидко визначити пріоритети та виправити. Швидкість має важливе значення для зменшення вікна можливостей, яке мають суб'єкти загрози для здійснення атак. Крім того, корисно проаналізувати причину уразливості після її пом'якшення, щоб запобігти повторенню в майбутньому.

Завдання в цьому останньому процесі можуть містити:

- збір інформації про клієнта та ретельний перегляд/тестування коду на наявність будь-яких невиявлених недоліків;
- підготовку команди, плану та процесів для швидкого реагування на уразливості та пом'якшення;
- створення та впровадження плану виправлення для кожної виявленої уразливості;
- створення програми звітування про уразливості та реагування;
- використання автоматизації для моніторингу даних про уразливості та проведення автоматизованого аналізу коду;
- вимірювання впливу та ресурсів, необхідних для усунення кожної уразливості, з одночасним визначенням пріоритетів усунення;
- визначення кореня причин для створення бази знань для майбутньої профілактики;
- виявлення та документування першопричин уразливостей під час вдосконалення інструментарію для автоматичного майбутнього виявлення та впровадження відповідних коригувань у SSDF.

Першопричини потрібно аналізувати з часом, щоб виявити закономірності. Потім ці шаблони можна помітити та виправити в іншому програмному забезпеченні. Нарешті, весь SDLC можна періодично оновлювати, щоб усунути подібні проблеми в майбутніх випусках.

2.5 Інструменти для тестування програмного забезпечення

Тестування програмного забезпечення – це важливий етап розробки, який забезпечує якість, функціональність, продуктивність і безпеку програмних продуктів. Існує безліч інструментів, які можна використовувати для різних типів тестування.

Інструменти для тестування важливі, оскільки їх використання:

- звільняє тестувальників від рутинних завдань, дозволяючи їм зосередитися на більш складних сценаріях;
- значно прискорює процес тестування, дозволяючи проводити більше тестів за менший час;
- зменшує кількість людських помилок, що виникають у разі ручного тестування;
- дозволяє легко повторювати тести, що важливо для регресійного тестування.

2.5.1 Класифікація інструментів для тестування

Розглянемо деякі категорії та популярні інструменти, які можна класифікувати за різними критеріями.

За типом тестування:

- функціональне тестування (Selenium, Cypress, TestComplete);
- навантажувальне тестування (JMeter, LoadRunner, Gatling);
- тестування продуктивності (LoadNinja, AppDynamics);
- тестування безпеки (Burp Suite, OWASP ZAP);
- тестування інтерфейсу (Selenium IDE, Appium).

За типом застосування:

- автоматизація тестування (Selenium, Cypress, Robot Framework);
- управління тестами (TestRail, Zephyr);
- аналіз результатів тестування (TestRail, Allure).

За рівнем автоматизації:

- інструменти для ручного тестування;
- інструменти для автоматизованого тестування;
- комбіновані інструменти.

2.5.2 Огляд інструментів для автоматизованого тестування безпеки програмних застосунків

Тестування безпеки розробленого програмного застосунку або програмної системи є найважливішим видом тестування для будь-якої програми. Тестувальник ніби бере на себе роль зловмисника і шукає помилки, пов'язані з безпекою.

Під час вибору інструменту для тестування варто враховувати такі ключові фактори:

- інструмент має *підтримувати технології*, які використовуються у проєкті;
- *простота налаштування* та використання інструменту;
- *мови програмування*, якими володіє команда тестувальників;

- *легкість інтегрування* вибраного інструменту з іншими інструментами (CI/CD, системи управління дефектами тощо);
- *вартість*, тобто чи є інструмент безкоштовним, з відкритим кодом чи комерційним;
- *якість документації* та підтримки інструменту.

Codacy – онлайн-сервіс для статичного аналізу коду та автоматичного рев'ю. Codacy допомагає виявити помилки в коді і проблеми з безпекою використовуваних конструкцій, та надає підказки щодо їх усунення. Крім того, він може дати загальну оцінку якості проєкту.

Сервіс Codacy не складний у налаштуванні, легко вбудовується в існуючий процес (безперервна інтеграція). Для розробників опенсорс проєктів він безкоштовний.

OWASP – відкритий проєкт безпеки web-додатків, призначений для надання інформації про безпеку програми. Він визначає 10 найпоширеніших ризиків безпеки, що роблять програму шкідливою, оскільки дозволяють викрасти дані або повністю захопити вебсервери: функціональний контроль доступу; порушення авторизації/сесії, введення SQL, пряме посилання на об'єкт, неправильне налаштування безпеки, підробка міжсайтових запитів, вразливі компоненти, міжсайтові сценарії або неперевірені переадресації та доступ до даних.

SonarQube – інструмент з відкритим кодом, який використовується для вимірювання якості вихідного коду. Він легко інтегрується з інструментами безперервної інтеграції (сервер Jenkins тощо). Має режими графічного інтерфейсу та командного рядка, що дає змогу оглянути діаграми та списки проблем. SonarQube реалізований мовою Java, але може аналізувати понад 20 різних мов програмування.

Burp suite – інструмент, який найкраще підходить для інтеграції існуючих програм і також використовується для тестування безпеки вебдодатків. Має понад 100 заздалегідь визначених умов уразливості, що допомагають забезпечити повноцінне тестуваня програмного коду.

Інструмент подає результати у вигляді дерева: можна вибрати окрему гілку або вузол, переглянути деталі, а також отримати повні рекомендації щодо усунення уразливостей (опис проблеми, тип довіри, серйозність та шлях до файлу). Звіти з виявленими вразливими місцями можна експортувати у форматі HTML.

MobSF (Mobile Security Framework) – це автоматизована система з відкритим кодом для тестування безпеки мобільних додатків на платформах Android, iOS та Windows. Інструмент виконує як статичний, так і динамічний аналіз і забезпечує:

- тестування безпеки веб-служб, що використовуються більшістю мобільних додатків;
- розміщення в локальному середовищі, завдяки чому конфіденційні дані ніколи не взаємодіють із хмарою;
- підтримку аналізу як двійкового, так і Zipped вихідного коду;
- тестування безпеки веб-API за допомогою API Fuzzer.

Snyk – інструмент для виявлення уразливостей коду ще до розгортання. Його легко інтегрувати в IDE, робочі процеси та звіти.

Snyk використовує принципи логічного програмування для виявлення уразливостей безпеки безпосередньо під час написання коду. Завдяки вбудованому інтелекту, інструмент динамічно регулює частоту сканування, враховуючи різні загальносерверні параметри.

Snyk має готові інтеграції з Jira, Microsoft Visual Studio, GitHub, CircleCI та іншими платформами. Крім того, пропонує кілька цінових планів для задоволення потреб бізнесу різних масштабів.

Отже, інструменти для тестування є незамінними помічниками для забезпечення високої якості програмного забезпечення. Вони допомагають організувати ефективне тестування ПЗ, автоматизувати рутинні завдання, забезпечити безпеку та високу якість продуктів. Правильний вибір інструменту допоможе значно скоротити час на тестування, підвищити надійність програмного продукту і задовольнити потреби користувачів.

3 ЗАГАЛЬНИЙ ОГЛЯД СИСТЕМ ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Мета і доцільність захисту програмного забезпечення

Системи захисту програмного забезпечення (СЗПЗ) широко поширені і знаходяться у постійному розвитку завдяки розширенню ринку ПЗ і телекомунікаційних технологій. Необхідність використання СЗПЗ обумовлена рядом проблем, серед яких варто виділити:

- *промислове шпигунство* – незаконне використання алгоритмів, що є інтелектуальною власністю автора, при написанні аналогів продукту;
- *крадіжка і копіювання* – несанкціоноване використання ПЗ ;
- *несанкціонована модифікація ПЗ* з метою впровадження програмних зловживань;
- *піратство* – незаконне поширення і збут ПЗ.

Таким чином, ***основна мета*** використання систем захисту ПЗ полягає у забезпеченні безпеки, цілісності та доступності програмного забезпечення та даних, які воно обробляє. Конкретніше, системи захисту ПЗ спрямовані на досягнення таких цілей:

1. *Захист від несанкціонованого копіювання та розповсюдження* – запобігання піратству та незаконному використанню ПЗ, що завдає фінансових збитків розробникам та правовласникам.

2. *Захист від модифікації та зворотного інжинірингу*, а саме ускладнення аналізу внутрішньої структури програми з метою запобігання несанкціонованим змінам, крадіжці інтелектуальної власності (алгоритмів, технологій) та створенню шкідливих програм на основі оригінального коду.

3. *Захист від несанкціонованого доступу* – обмеження доступу до функціональності та даних програми лише для авторизованих користувачів.

4. *Забезпечення цілісності даних* – запобігання несанкціонованій зміні, видаленню або пошкодженню даних, що обробляються програмою.

5. *Забезпечення конфіденційності даних* – захист чутливої інформації від несанкціонованого розкриття.

6. *Захист від шкідливого програмного забезпечення*, тобто, мінімізація ризиків зараження вірусами, троянами та іншими видами шкідливого ПЗ через уразливості в програмі.

7. *Забезпечення безперебійної роботи*, що полягає у захисті від атак, спрямованих на відмову в обслуговуванні (DoS), що можуть призвести до зупинки або нестабільної роботи програми.

8. *Дотримання нормативних вимог* – забезпечення відповідності стандартам та законодавству у сфері інформаційної безпеки.

Доцільність використання систем захисту програмного забезпечення визначається низкою факторів, серед яких:

- *цінність ПЗ та даних*, які воно обробляє (чим вища комерційна цінність ПЗ, конфіденційність даних користувачів або критичність програми для бізнес-процесів, тим важливішим є його захист);
- *потенційні загрози* (оцінення ймовірності та потенційного впливу різних загроз – піратство, зловмисні атаки, несанкціонований доступ тощо);
- *репутаційні ризики* (компанія може зазнати значних репутаційних втрат);
- *конкурентна перевага*: захист унікальних технологій та алгоритмів може забезпечити конкурентну перевагу на ринку;
- *юридичні наслідки*, оскільки у багатьох випадках законодавство висуває вимоги щодо вжиття певних заходів для захисту персональних даних та іншої конфіденційної інформації;
- *вартість* впровадження та підтримки (важливо знайти баланс між рівнем захисту та витратами);
- *зручність* використання (важливо забезпечити баланс між безпекою та зручністю).

У підсумку, використання систем захисту програмного забезпечення є доцільним тоді, коли потенційні ризики та збитки від їх відсутності перевищують витрати на впровадження та підтримку адекватних заходів безпеки. Ретельно проаналізувавши всі фактори, можна визначити оптимальний набір інструментів та методів захисту для конкретного програмного продукту та його користувачів.

3.2 Класифікація системи захисту програмного забезпечення

Класифікація систем захисту програмного забезпечення може здійснюватися за різними критеріями, відображаючи їхні цілі, методи дії та рівні застосування.

І. За рівнем впливу на ПЗ системи захисту поділяються на зовнішні і внутрішні.

Зовнішні системи захисту не вносять змін безпосередньо в код програми. Вони, як правило, діють на рівні операційної системи або середовища виконання. До них відносять:

- *апаратні ключі (dongles)* – фізичні пристрої, які підключаються до комп'ютера і містять інформацію про ліцензію. ПЗ перевіряє наявність ключа для запуску;

– *системи керування ліцензіями (License Management Systems)* – централізовані системи для активації, контролю та управління ліцензіями на програмне забезпечення;
– *програмні ліцензії (Software Licensing)*, тобто файли, записи в реєстрі або інші сховища, що містять інформацію про ліцензію та її параметри.

Внутрішні системи захисту інтегруються безпосередньо в код програми, ускладнюючи його аналіз та модифікацію. Серед них такі:
– *обфускування коду (Code Obfuscation*), тобто перетворення коду в форму, яка важка для розуміння, але залишається функціонально еквівалентною;
– *водяні знаки (Watermarking)* – унікальні ідентифікатори у коді або у даних для відстеження джерела нелегального копіювання;
– *віртуалізація коду (Code Virtualization)*, тобто виконання частини коду у віртуальній машині, що ускладнює його аналіз;
– *захист від налагодження (Anti-Debugging)* – техніки, що ускладнюють або унеможливлюють налагодження програми з метою аналізу її роботи;
– *захист від дампінгу пам'яті (Anti-Memory Dumping)* – методи, що ускладнюють отримання копії вмісту пам'яті процесу, де виконується програма.

ІІ. За метою захисту СЗПЗ поділяють на такі категорії:
– системи захисту *від копіювання* (Anti-Piracy), які спрямовані на запобігання нелегальному копіюванню та розповсюдженню ПЗ;
– системи захисту *від реверс-інжинірингу* (Anti-Reverse Engineering), що ускладнюють аналіз внутрішньої структури та алгоритмів програми;
– системи захисту *від модифікації* (Anti-Tampering), що запобігають несанкціонованому внесенню змін до коду програми;
– системи *контролю використання* (Usage Control), що обмежують спосіб та умови використання ПЗ (наприклад, кількість користувачів, термін дії);
– системи захисту *від шкідливого ПЗ* (Malware Protection), які часто розглядаються як окрема галузь, але деякі програмні продукти можуть мати вбудовані механізми для виявлення та блокування відомих загроз.

ІІІ. За технологією реалізації СЗПЗ поділяють на такі, що використовують:
– *криптографічні методи*, які використовують шифрування, цифрові підписи та інші криптографічні техніки для захисту коду, даних та ліцензій;

- *технічні засоби контролю доступу* – механізми автентифікації та авторизації для обмеження доступу до ПЗ та його функцій;
- *програмні засоби захисту*, що реалізують такі різноманітні програмні методи, як, наприклад, обфускування, віртуалізація, водяні знаки тощо;
- *апаратні засоби захисту*, для яких використовується спеціалізоване апаратне забезпечення, таке, наприклад, як апаратні ключі (dongles) або TPM (Trusted Platform Module).

IV. За етапом життєвого циклу ПЗ:

- захист *на етапі розробки*, тобто застосування практик безпечного кодування, аналіз уразливостей та інші заходи для запобігання появі слабких місць у ПЗ;
- захист *на етапі розповсюдження* – використання безпечних каналів розповсюдження, цифрових підписів для перевірки цілісності ПЗ;
- захист *на етапі використання*, що передбачає застосування механізмів ліцензування, захисту від копіювання та модифікації.

V. За складністю та комплексністю СЗПЗ можуть бути:

- *прості* – однорівневі системи, що використовують один або декілька базових механізмів захисту (наприклад, парольний захист);
- *комплексні* – багаторівневі системи, що поєднують різні методи та технології захисту для забезпечення більш високого рівня безпеки (наприклад, використання криптографії, контролю доступу та систем виявлення вторгнень).

Важливо розуміти, що ці класифікації не є взаємовиключними, і конкретна система захисту програмного забезпечення може поєднувати елементи з різних категорій. Вибір конкретних методів захисту залежить від багатьох факторів, включно цінність програмного забезпечення, потенційні загрози та доступні ресурси.

3.3 Показники застосовності та критерії оцінювання СЗПЗ

За результатами дослідження був розроблений набір показників застосовності і критеріїв оцінювання СЗПЗ.

3.3.1 Показники застосовності

Технічні: відповідність СЗПЗ функціональним вимогам виробника ПЗ і вимогам до стійкості, системні вимоги ПЗ і системні вимоги СЗПЗ, обсяг ПЗ й обсяг СЗПЗ, функціональна спрямованість ПЗ, наявність і тип СЗ в аналогах ПЗ-конкурентів.

Економічні: співвідношення втрат від піратства і загального обсягу прибутку, співвідношення втрат від піратства і вартості СЗПЗ та її

впровадження, співвідношення вартості ПЗ і вартості СЗПЗ, відповідність вартості СЗПЗ і її впровадження поставленим цілям.

Організаційні: поширеність і популярність ПЗ, умови поширення і використання, унікальність, наявність загроз, імовірність перетворення користувача в зловмисника, роль документації і підтримки ПЗ.

Загальна картина взаємодії агентів ринку програмного забезпечення може бути подана на схемі, наведеній на рис. 3.1.

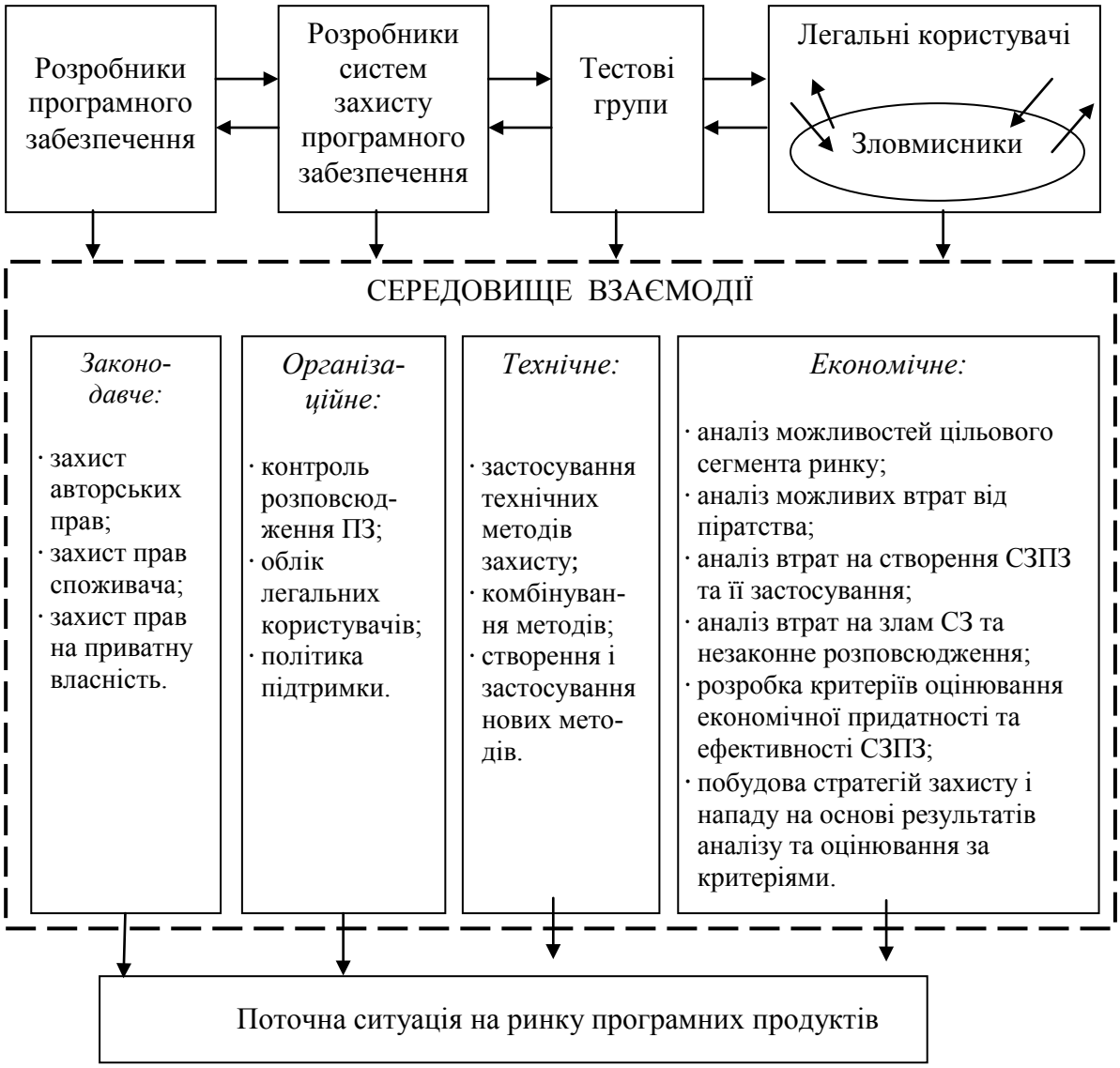


Рисунок 3.1 – Схема взаємодії учасників розробки та розповсюдження програмного забезпечення

З чотирьох зазначених у наведеній схемі видів середовища взаємодії стороні, що захищається, підконтрольні (чи частково підконтрольні) три види – організаційне, технічне й економічне середовище. Очевидно, що найважливішим середовищем взаємодії є економічне середовище, оскільки економічна взаємодія у такому випадку є першопричиною і метою усієї взаємодії.

Під час розробки та аналізування захисту програмного забезпечення необхідно враховувати існуючу законодавчу базу, потрібно проводити докладний економічний аналіз ситуації, застосовуючи різні критерії оцінювання, а потім створювати стратегію захисту, що містить застосування технічних і організаційних заходів для захисту ПЗ.

3.3.2 Критерії оцінювання

Захист як такий: утруднення нелегального копіювання, утруднення нелегального доступу, захист від моніторингу, відсутність логічних проломів і помилок у реалізації системи.

Стійкість до дослідження/зламу: застосування стандартних механізмів, нові/нестандартні механізми.

Відмовостійкість (надійність): імовірність відмови захисту, час напрацювання на відмову, імовірність відмови програми захисту (крах), частота помилкових спрацьовувань.

Незалежність від конкретних реалізацій ОС: використання недокументованих можливостей, «вірусних» технологій і «дір» ОС.

Сумісність. Відсутність конфліктів із системним ПЗ, відсутність конфліктів із прикладним ПЗ, максимальна сумісність з апаратним та програмним забезпеченням.

Незручності для кінцевого користувача ПЗ. Необхідність і складність додаткового настроювання системи захисту, доступність документації, доступність інформації про відновлення модулів системи захисту через помилки/несумісності/нестійкості, доступність сервісних пакетів, безпека мережної передачі пароля/ключа, затримка через фізичне передавання пароля/ключа, порушення прав споживача.

Побічні ефекти: перевантаження трафіка, відмова в обслуговуванні, уповільнення роботи зали захищеного ПЗ, уповільнення роботи ОС, захоплення системних ресурсів, порушення стабільності ОС тощо.

Вартість. Вартість/ефективність, вартість/ціна ПЗ, що захищається, вартість/ліквідовані збитки.

Доброякісність: правдива реклама, доступність результатів незалежної експертизи, доступність інформації про побічні ефекти і т. д.

4 ПРОГРАМНО-АПАРАТНІ СПОСОБИ ЗАХИСТУ

Електронні ключі (ЕК) для захисту програмного забезпечення є інструментами, які використовуються для запобігання несанкціонованому доступу, піратству чи копіюванню програм.

4.1 Поняття електронного ключа

На сьогодні електронний ключ є одним із найбільш надійних та зручних методів захисту тиражованого програмного забезпечення середньої та високої цінової категорії. Він має високу стійкість до зламу і не накладає обмежень на використання легальної копії програми.

Електронними ключами, переважно, захищають так званий «діловий» софт: бухгалтерські і складські програми, правові та корпоративні системи, будівельні кошториси, САПР, електронні довідники, аналітичний софт, екологічні й медичні програми і т. п. Витрати на розроблення таких програм великі, і, відповідно, висока вартість софта, тому збиток від піратського поширення значний. У цьому випадку електронні ключі є оптимальним засобом захисту.

Електронний ключ – це «залізо», яке нам доводиться використовувати без нашого бажання. Електронний ключ звичайно є невеликим пристроєм, що складається з плати з мікросхемами (допоміжні елементи, мікроконтролер і пам'ять), вкладеної в пластиковий корпус. Мікроконтролер містить так звану «математику» – набір команд, що реалізують деяку функцію (або функції), яка слугує для генерації інформаційних блоків обміну ключа і захищеної програми.

Електронні ключі для захисту програмного забезпечення виконують декілька основних задач:

1. *Захист від піратства*. ЕК запобігає нелегальному копіюванню та розповсюдженню програмного забезпечення.
2. *Ліцензування*. ЕК дозволяє реалізувати різні ліцензійні моделі (наприклад, одноразове використання, підписка).
3. *Захист конфіденційних даних*. ЕК забезпечує безпечне зберігання інформації, що обробляється програмним забезпеченням.
4. *Контроль доступу*. ЕК дозволяє обмежити доступ до певних функцій або модулів програми.

Таким чином, електронний ключ можна розглядати як «паспорт» для програмного забезпечення, що запобігає його несанкціонованому копіюванню, аналізу, поширенню та використанню. Сучасні електронні ключі прості у використанні та не потребують спеціальних знань.

Основні види електронних ключів можна поділити на декілька категорій залежно від їхньої реалізації та способу використання (рис. 4.1).

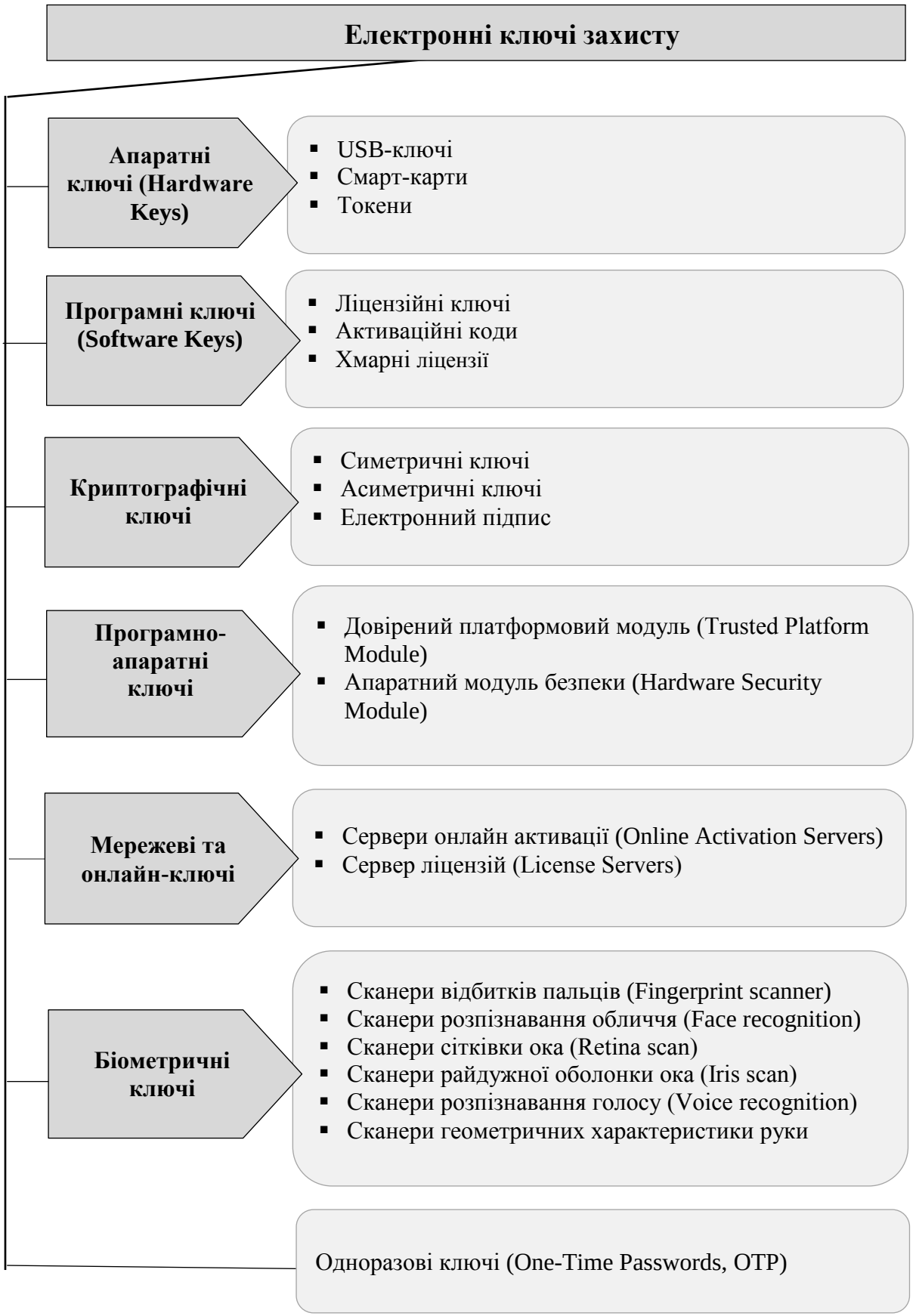


Рисунок 4.1 – Класифікація електронних ключів захисту

4.2 Види електронних ключів

4.2.1 Апаратні ключі (Hardware Keys)

Це фізичні пристрої, які підключаються до комп'ютера та використовуються для перевірки прав доступу до програмного забезпечення.

USB-ключі. Це невеликий пристрій, який підключається до комп'ютера через USB-порт. Містить криптографічний чип для зберігання ліцензійної інформації.

Смарт-карти. Пластикові картки зі вбудованим мікрочипом. Використовуються в більш складних системах захисту. Використовуються з картридерами, часто для підвищеної безпеки.

Токени. Схожі на USB-ключі, але зазвичай мають додаткові функції, наприклад, генерування одноразових паролів. Вони також можуть містити криптографічні ключі.

4.2.2 Програмні ключі (Software Keys)

Програмі ключі – це цифрові ліцензії, які, зазвичай, надсилаються в електронному вигляді.

Ліцензійні ключі, які являють собою комбінацію символів (наприклад, серійний номер), яка активує програму. Вони можуть бути, наприклад, створені на основі апаратних характеристик – використовувати для генерування ліцензійного ключа унікальні характеристики: серійний номер процесора, обсяг оперативної пам'яті тощо.

Активаційні коди, які генеруються сервером розробника на основі даних про систему користувача (апаратного ідентифікатора).

Хмарні ліцензії, які перевіряються через інтернет-сервіси; ключ доступу зберігається в акаунті користувача. Тобто, ліцензійна інформація зберігається на сервері розробника, а доступ до програми здійснюється через інтернет.

4.2.3 Криптографічні ключі

Криптографічні ключі використовуються для шифрування даних або для забезпечення автентичності програм.

Симетричні ключі, коли один ключ використовується для шифрування та розшифрування.

Асиметричні ключі, коли використовуються пари «публічний-приватний» ключів для підписів чи шифрування.

Електронний підпис, який забезпечує цілісність і перевірку джерела програмного забезпечення.

4.2.4 Програмно-апаратні ключі

Це ключі, які поєднують фізичний пристрій і програмне забезпечення.

TPM, або *Trusted Platform Module* (довірений платформовий модуль), – це спеціальний криптографічний чип, який вбудовується безпосередньо в материнську плату комп'ютера. Його основна функція – забезпечення безпеки та довіри до системи. TPM діє як своєрідний цифровий сейф, зберігаючи ключі шифрування, паролі та інші конфіденційні дані.

HSM, або *Hardware Security Module* (апаратний модуль безпеки), – це спеціалізований фізичний пристрій, призначений для захисту та управління криптографічними ключами. Уявити його можна як надбезпечний сейф, де зберігаються найцінніші цифрові активи компанії. HSM використовується для шифрування, створення цифрових підписів та інших криптографічних операцій.

4.2.5 Мережеві та онлайн-ключі

Мережеві та онлайн-ключі працюють через інтернет і прив'язані до серверів розробника.

Online Activation Servers – це спеціально налаштовані сервери, які імітують офіційні сервери активації програмного забезпечення. Вони використовуються для активації програм без необхідності купувати ліцензію. Програма перевіряє ключ через сервери в реальному часі. Використання онлайн-серверів активації дозволяє користувачам користуватися платним програмним забезпеченням без оплати ліцензії, а активація за допомогою онлайн-серверів зазвичай є дуже простою і не потребує спеціальних знань. Онлайн-сервери активації доступні для великої кількості програмного забезпечення.

License Servers. Використовуються в корпоративних рішеннях, коли ключі видаються сервером у мережі. Це спеціальний сервер або програмне забезпечення, яке використовується для управління ліцензіями на програмне забезпечення в організації. Він відстежує кількість використаних ліцензій, контролює доступ користувачів до програмного забезпечення та забезпечує дотримання умов ліцензійної угоди.

4.2.6 Біометричні ключі

Це сучасний підхід, коли доступ до програмного забезпечення надається лише після успішної біометричної автентифікації.

Відбитки пальців (Fingerprint scanner) – один з найстаріших і найпоширеніших методів біометричної ідентифікації.

Розпізнавання обличчя (Face recognition) – технологія, яка аналізує геометричні особливості обличчя для ідентифікації особи.

Сканування сітківки ока (Retina scan) – високоточна технологія, яка аналізує унікальний малюнок кровоносних судин в сітківці ока.

Розпізнавання райдужної оболонки ока (Iris scan) – аналізує унікальний малюнок райдужної оболонки ока, який формується ще в утробі матері.

Розпізнавання голосу (Voice recognition) – аналізує унікальні характеристики голосу людини, такі як тембр, інтонація та ритм мовлення.

Геометричні характеристики руки – аналізує форму руки та розміщення вен.

4.2.7 Одноразові ключі (One-Time Passwords, OTP)

Вони використовуються для одноразового доступу, наприклад, у разі тимчасових ліцензій або захисту облікових записів.

Використання усіх цих ключів залежить від вимог до безпеки, бюджету та зручності для користувача. Зазвичай, для високого рівня захисту використовують комбінацію апаратних і програмних рішень.

4.3 Апаратні ключі від компанії Yubico

4.3.1 Коротка історична довідка

Yubico – технологічна компанія (з офісами в Пало-Альто, Сіетлі і Стокгольмі), яка спеціалізується на створенні безпечних апаратних пристроїв автентифікації для онлайн-безпеки. Компанія була заснована в 2007 році Стіною Еренсвард та Якобом Еренсвардом з метою створення простого, зручного та безпечного способу доступу до персональних облікових записів. Засновники компанії дещо відійшли від традиційних методів автентифікації (паролів та токенів, які були досить громіздкими та ненадійними і часто створювали хибне почуття безпеки). Вони прагнули зробити онлайн-безпеку простою та зручною для всіх.

Як результат з’явилася ідея створення апаратного ключа безпеки, який міг би вирішити цю проблему, і у 2008 році компанія Yubico представила свій перший продукт YubiKey (від *англ. «your ubiquitous key» – ваш всюдисущий ключ*). Це був невеликий пристрій автентифікації, який нагадував звичайну USB-флешку, був стійким до тиску та ударів, був простим у використанні та працював від одного дотику. Технічні параметри цього пристрою зробили його хітом серед користувачів, дозволивши відійти від складних методів автентифікації. Відтоді головним продуктом компанії є серія апаратних ключів безпеки Yubikey, які

протягом багатьох років захищають облікові записи в інтернеті. Багато технологічних компаній (Google, Facebook та ін.), а також низка урядових відомств взяли на озброєння саме ключі Yubico у більш як 160 країнах світу (рис. 4.2).

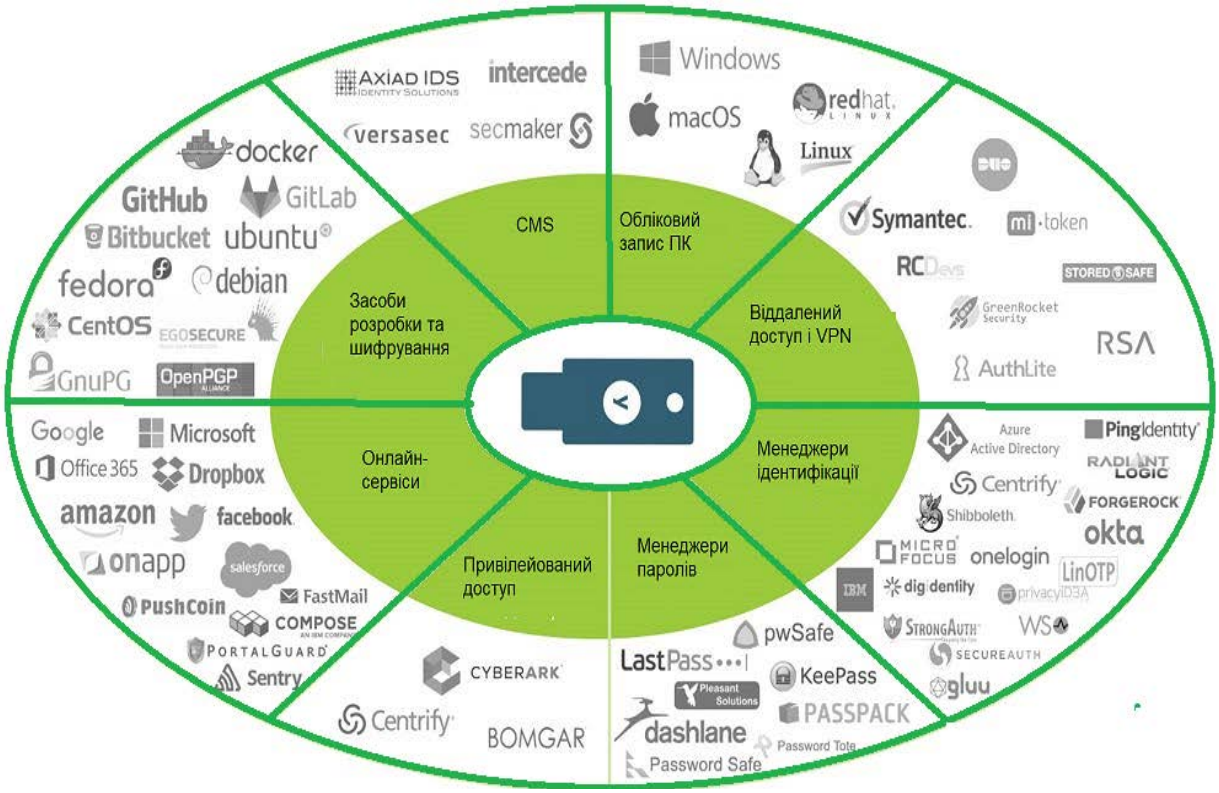


Рисунок 4.2 – Діаграма застосувань ключів Yubico

4.3.2 Огляд продуктів Yubikey та їх характеристики

YubiKey – це фізичний ключ безпеки, який використовується для двофакторної автентифікації. Він є одним з найпопулярніших пристроїв такого типу завдяки своїй надійності та зручності використання.

YubiKey 5 – це серія компактних USB-ключів безпеки для багато-факторної автентифікації. Це пристрій апаратної автентифікації, який забезпечує чудовий захист від фішингу, відповідає сучасним стандартам безпеки та пропонує широкий вибір типів надійної автентифікації.

Взаємодія з YubiKey надзвичайно проста: наприклад, додавши його до облікового запису на GitHub, для входу достатньо ввести логін і пароль та натиснути кнопку YubiKey.

Один YubiKey можна використовувати на багатьох комп’ютерах і мобільних пристроях, водночас він підтримує необмежену кількість облікових записів Google, Microsoft, Dropbox, GitHub та інших сервісів.

Ключі YubiKey практично неруйнівні, їх можна тримати на одній зв'язці з ключами від дому або автомобіля. YubiKey працює з сімействами

операційних систем Windows, macOS та Linux, підтримує популярні менеджери паролів, такі як Lastpass, Dashlane, Keepass та інші.

Ключі YubiKey мають своє втілення у різних моделях, які відрізняються за кількома параметрами (табл. 4.1).

Таблиця 4.1 – Порівняння різних моделей ключів YubiKey

Характеристика	YubiKey 5 NFC	YubiKey 5C NFC	YubiKey 5 Nano	YubiKey 5C Nano	YubiKey Bio
Форм-фактор	USB-A, NFC	USB-C, NFC	USB-A	USB-C	USB-C, NFC, біометричний сканер
Протоколи	FIDO U2F, FIDO2, OTP, PIV, OpenPGP	FIDO U2F, FIDO2, PIV, OTP, OpenPGP	FIDO U2F, FIDO2, OTP, PIV, OpenPGP	FIDO U2F, FIDO2, OTP, PIV, OpenPGP	FIDO U2F, FIDO2, OTP, PIV, OpenPGP
Сертифікація	FIPS (деякі моделі)	FIPS (деякі моделі)	FIPS (деякі моделі)	FIPS (деякі моделі)	FIPS (деякі моделі)
Додаткові функції	-	-	Компактний розмір	Компактний розмір, USB-C	Біометрична автентифікація
Орієнтовна ціна на ринку	Середня	Середня	Нижча	Нижча	Вища

1. Форм-фактор (тип роз'єму та додаткові функції – NFC, біометрика):
   - USB – класичний USB-роз'єм, найпоширеніший;
   - USB-C – сучасніший USB-роз'єм, менший за розміром;
   - NFC – дозволяє з'єднання з пристроями, що підтримують безконтактну передачу даних (смартфони, планшети);
   - Nano – компактні ключі, які можна залишити постійно в порту USB;
   - BIO – додатково оснащені сканером відбитків пальців для біометричної автентифікації.
2. Функціональність:
   - FIDO2 – підтримка сучасного стандарту для створення сильних, унікальних паролів;
   - U2F – попередній стандарт, також широко використовується;
   - OTP – генерація одноразових паролів;
   - PIV – використання як смарт-карти для доступу до корпоративних мереж;
   - OpenPGP – шифрування даних та електронний підпис.
3. Сертифікація FIPS – відповідність вимогам федерального уряду США щодо безпеки інформації.

4.3.3 Модулі безпеки YubiHSM

YubiHSM (*Hardware Security Module*) – це спеціалізований апаратний пристрій, призначений для безпечного зберігання та управління криптографічними ключами. Він є своєрідним сейфом для цифрових ключів, що забезпечує високий рівень захисту.

Хоча обидва пристрої, YubiHSM і YubiKey, є продуктами компанії Yubico та призначені для підвищення рівня безпеки, вони мають різні функції та сфери застосування (табл. 4.2).

Таблиця 4.2 – Основні відмінності ключів YubiHSM і YubiKey

Характеристика	YubiKey	YubiHSM
Призначення	2FA	Захист криптографічних ключів
Принцип роботи	Передбачає підтвердження особистості, фізично підключивши YubiKey до пристрою	Зберігає і обробляє криптографічні ключі в ізольованому середовищі. Ці ключі можуть використовуватися для шифрування даних, цифрового підпису та інших криптографічних операцій
Функції	Генерація OTP, підтримка FIDO2 тощо	Генерація, зберігання та управління ключами, підтримка різних криптографічних алгоритмів
Рівень безпеки	Високий для індивідуального користувача	Дуже високий для організацій з високими вимогами до безпеки (банки, фінансові установи, урядові організації)
Сфера застосування	Захист облікових записів	Захист даних, інфраструктури PKI

Отже, обидва пристрої, YubiKey і YubiHSM, є важливими інструментами для підвищення безпеки. Вибір між ними залежить від конкретних потреб та вимог до безпеки.

4.3.4 YubiKey SDK

Компанія Yubico розробила ряд SDK для мобільних пристроїв (iOS, Android) та настільних комп’ютерів, щоб розробники могли швидко інтегрувати апаратну безпеку у свої програми та сервіси, забезпечуючи високий рівень безпеки на різних пристроях і платформах

Yubico пропонує безкоштовне програмне забезпечення з відкритим кодом для інтеграції надійної автентифікації у власний програмний продукт або послугу, а також рішення для шифрування та захисту інформації на серверах. Серед них такі:

- серверні бібліотеки Java WebAuthn, Python WebAuthn/FIDO2;
- .NET YubiKey SDK;

– Yubico Mobile SDK для iOS;
– Yubico Mobile SDK для Android;
– C-бібліотека FIDO2;
– бібліотека Python YubiHSM;
– Yubico Authenticator для ПК та Android.

4.3.5 Рекомендації щодо вибору ключів YubiKey

Основні серії YubiKey такі (рис. 4.3):

– YubiKey 5 – найпопулярніша серія, що містить різні форм-фактори та функції;
– YubiKey Bio – серія з біометричною автентифікацією;
– YubiKey Nano – більш доступна серія з базовим набором функцій;
– YubiHSM – спеціалізований апаратний пристрій, призначений для безпечного зберігання та управління криптографічними ключами.

Рисунок 4.3 – Зовнішній вигляд ключів YubiKey

Вибір конкретного ключа залежить від потреб. Якщо необхідно використовувати ключ часто, варто звернути увагу на компактні моделі (Nano) або з NFC.

Вибір ключа може залежати від того, чи підтримує він необхідні протоколи та пристрої.

З погляду рівня безпеки для максимального захисту рекомендується вибирати ключі з підтримкою FIDO2 та біометричною автентифікацією.

Якщо ж потрібно захистити велику кількість криптографічних ключів і потрібна автоматизація управління ключами, якщо наявні строгі вимоги до відповідності нормативним актам і є необхідність роботи з чутливими даними, які вимагають високого рівня безпеки, тоді варто вибирати YubiHSM.

Отже, якщо потрібен високий рівень захисту криптографічних ключів, YubiHSM буде кращим вибором. Якщо потрібна проста і зручна двофакторна автентифікація, YubiKey буде більш доречним.

4.4 C100 HOTP – токен з алгоритмом захищеної динамічної автентифікації

Токен C100 HOTP – це компактний пристрій, який використовується для двофакторної автентифікації. Він генерує одноразові паролі (OTP), які змінюються кожні 30 секунд або у разі натискання кнопки. Ці паролі використовуються разом з основним паролем для додаткового рівня безпеки під час входу в онлайн-сервіси (рис. 4.4).

Токени Feitian C100 пропонують алгоритм захищеної динамічної автентифікації з використанням одноразових паролів, які змінюються у разі натискання кнопки на пристрої. Ця модель токенів розроблена на основі алгоритму HOTP (HMAC-based One-Time Password – стандартний алгоритм для генерування одноразових паролів) і є алгоритмом односторонньої автентифікації, а саме: сервер здійснює автентифікацію клієнта (рис. 4.4).

FT-HOTP-C100 Feitian C100 HOTP

Рисунок 4.4 – Зовнішній вигляд токенів C100 HOTP

Токени Feitian C100 підтримуються сервером автентифікації Feitian OTP Authentication System (FOAS). Програмне забезпечення FOAS легко розгортається на сервері і підтримує автентифікацію будь-якої кількості користувачів. Централізована інтеграція і управління здійснюються на стороні сервера, що гарантує безпеку адміністрування та автентифікації користувачів.

У токенах Feitian C100 HOTP (одноразовий пароль на основі події) використовується алгоритм на основі подій, який кожного разу генерується під час натискання кнопки. Оскільки значення на підставі подій можуть бути більш тривалі, ніж за часом, токени спроєктовані так, що після 15 секунд дисплей відключається.

Принци роботи токена C100 HOTP складається з таких кроків:

I. Генерування пароля: за кожного натискання кнопки або через певний інтервал часу токен генерує новий одноразовий пароль.
II. Введення пароля: користувач вводить згенерований пароль разом зі своїм основним паролем під час входу в систему.

III. Перевірка: сервер перевіряє введений пароль і порівнює його зі значенням, яке він розрахував за допомогою того самого алгоритму, що і токен. Якщо паролі збігаються, користувачу надається доступ.

Щодо переваг, то основними *перевагами* токену C100 HOTP є такі:

1. Унікальний пароль генерується кожен раз і не може бути використаний повторно. Одноразовий пароль отримується легко, шляхом натискання однієї кнопки. Не потрібно запам'ятовувати будь-які паролі, або PIN-код.

2. Не потрібне встановлення програмного забезпечення на стороні клієнта. Не залежить від операційних систем і середовища роботи кінцевого користувача.

3. Відповідність відкритого алгоритму OATH з синхронізацією за подією.

4. Легке інтегрування з іншими серверами автентифікації та віддаленого доступу, підходить для ESET Secure Authentication.

5. Доступні seed'и у форматі PSKC. Токени C100 HOTP використовують безпечний та індивідуальний ключ для генерації одноразових паролів. Цей ключ зберігається всередині токена і є основою для високого рівня безпеки системи автентифікації. Seed код знаходиться в захищеній від злому, зашифрованій області пам'яті (має безпечну пам'ять RAM).

6. Підтримує Radius сервер:

- може звертатися до зовнішнього RADIUS-сервера для перевірки автентичності користувачів. Наприклад, Wi-Fi роутер може підтримувати RADIUS для того, щоб дозволяти підключення лише авторизованим користувачам;
- може сам функціонувати як RADIUS-сервер: виконувати роль сервера, зберігаючи інформацію про користувачів і автентифікуючи їх. Це може бути корисно для невеликих мереж або для створення власної системи автентифікації;
- може інтегруватися з існуючою RADIUS-інфраструктурою: може бути налаштований для роботи з вже наявним RADIUS-сервером в мережі, що дозволяє централізовано керувати доступом до різних ресурсів.

7. Інші технічні характеристики:

- легкий (до 15 г) і зручний брелок генератора паролів;
- РК дисплей на 6 символів та таймер зворотного відліку часу (15 с);
- забезпечення графічного інтерфейсу;
- одна вбудована кнопка та вбудований лічильник подій;
- унікальний порядковий номер токена;
- тривалість життя батареї 5-7 років;
- кількість циклів натискання – приблизно 16000.

4.5 Токен BioPass FIDO U2F FIDO2 USB

BioPass FIDO U2F FIDO2 USB – сертифікований FIDO Alliance U2F-автентифікатор, який забезпечує безпечну, фішинг-стійку схему двофакторної автентифікації, а також підтримує стандарт FIDO2 для безпарольної верифікації (рис. 4.5). Токен дозволяє користувачам безпечно входити у свої облікові записи, не потребує драйверів, визначається HID, CTAP2.

BioPass FIDO U2F FIDO2 USB-C K26

BioPass FIDO U2F FIDO2 USB-A K27

Рисунок 4.5 – Зовнішній вигляд токенів BioPass FIDO U2F FIDO2 USB

Токени BioPass FIDO U2F FIDO2 USB підтримують:

- такі програми, як Microsoft, Microsoft Azure, Microsoft Azure AD, Google, GMail, Google Drive, Facebook, Twitter, Dropbox, Github, GitLab, Salesforce, Bitbucket, Dashlane, Duo, Digientity, BITFINEX, FastMail, Gandi.net, Keeper, Sentry і багато інших;
- всі сучасні браузери Chrome, Edge, Firefox, Safari, Opera;
- macOS, Linux через USB;
- хмарні рішення від Microsoft.

Вбудований чип безпеки BioPass містить вдосконалену архітектуру безпеки, яка була розроблена для шифрування (підтримує криптоалгоритми: ECDSA, SHA256, AES, HMAC, ECDH), зберігання та захисту відбитків пальців (відбитки пальців зберігаються в спеціальному захищеному чипі).

Для підключення за стандартом FIDO2 цей токен використовує WebAuthn-JS API для менеджменту облікових записів на публічних ключах та CTAP2 (Client-to-Authenticator Protocol version 2) – стандарт, який описує протокол для спілкування ОС з автентифікатором по USB.

Інші характеристики:

- вага – 1 г;
- розмір – 51x18x6,5 мм ;
- захист від пилу і води – IP68;
- кількість циклів перезапису (зчитування/запису) інформації – 100000.

4.6 Апаратні ключі безпеки ATKey.Pro

Компанія AuthenTrend була заснована у 2016 році групою інженерів-біометрологів, які понад 15 років працюють із біометричними технологіями. Зараз вона має найбільшу кількість сертифікатів FIDO2 у

Тайвані та є першою компанією, яка розробила ключі безпеки із зовнішніми датчиками відбитків пальців. Сьогодні AuthenTrend користується довірою з боку асоціації інтелектуальної безпеки Microsoft (MISA), FIDO та RSA.

Офіційним представником компанії Authentrend.com в Україні є Authentrend.com.ua.

ATKey.Pro побудований на базі мікроконтролера BCM5810X компанії Broadcom®, сертифікованого за стандартом FIPS 140-2 третього рівня, що створений для додатків з високим рівнем безпеки. ATKey.Pro використовує функції Secure Boot та Secure XIP для захисту від хакерських атак на апаратне та мікропрограмне забезпечення.

Принцип роботи ATKey.Pro можна описати такими кроками:

1. Реєстрація відбитка пальця: під час першого використання ключа реєструється відбиток пальця.
2. Автентифікація: для входу в онлайн-сервіс потрібно підключити ключ до пристрою та прикласти палець до сканера.
3. Генерування ключа: ключ генерує криптографічний ключ, який використовується для автентифікації.
4. Підтвердження: сервіс перевіряє отриманий ключ і, якщо він відповідає збереженому, надає доступ.

ATKey Type A (рис. 4.6) – це найтонший, компактний і легкий USB-ключ безпеки для 2FA (двофакторної автентифікації), сертифікований FIDO, має бічний сканер відбитків пальців. До того ж може визначати відбитки пальців під будь-яким кутом і зчитувати їх менше ніж за секунду. Він дозволяє проводити просту та надійну автентифікацію в онлайн-сервісах на мобільних та стаціонарних пристроях. Тип роз'єму – USB Type A.

ATKey.Card (рис. 3.6) – це смарт-карта безпеки, яка може зв'язуватися з хостом через USB/BLE/NFC та підтримує технології FIDO2 та FIDO U2F.

ATKey Type A

ATKey.Card

Рисунок 4.6 – Зовнішній вигляд апаратних ключів ATKey.Pro

Смарт-карта ATKey.Card може використовуватися як пропуск і як ключ безпеки, що робить її незамінною для бізнесів, де важливий контроль співробітників: реєстрація початку та закінчення часу роботи або

навчання, час знаходження в певному приміщенні або відстеження їх місця розташування на підприємстві.

Карта може використовуватись замість майстра-ключа, отже, можна не хвилюватись про безпеку бізнесу, навіть якщо адміністратор втратить її!

Унікальність смарт-карти полягає в тому, що:

- вона має вбудований сканер відбитків пальців Egis. Це дає можливість не лише захистити картку, а й ідентифікувати її власника. Тому вона є найкращою перепусткою, що дозволяє визначити, чи не зловживає співробітник службовими обов'язками;
- карта має подвійний захист: PIN-код та відбиток пальця. Тому ніхто, крім власника картки, не зможе її використовувати;
- алгоритми шифрування в криптографії базується на FIDO2. Стандарт підтримує безпечне зберігання та захист даних відбитків пальців.

4.7 Ключі безпеки Google Titan Key

Google Titan Security Key – це апаратний автентифікатор, розроблений компанією Google для забезпечення безпечної багатофакторної автентифікації (MFA). Titan Key допомагає захистити облікові записи від фішингових атак та несанкціонованого доступу, оскільки ці ключі забезпечують криптографічну перевірку сервера, що унеможливлює викрадення даних через підроблені веб-сайти. Titan Key має спеціальний чип із криптографічним модулем, який генерує унікальні ключі для автентифікації. Чип захищає ключі від фізичного злому та програмного доступу.

Google Titan Key мають надійний корпус із міцного пластику або металу, компактні за розміром для зручності носіння, мають кільце або отвір для кріплення до ключів чи брелока (рис. 4.7). Активація ключа займає лише декілька секунд, і немає необхідності вводити складні паролі. Апаратний ключ не залежить від інтернету або батареї (окрім Bluetooth-версії).

Рисунок 4.7 – Зовнішній вигляд ключів Google Titan Security Key

Підтримувані стандарти: протоколи FIDO U2F (використовується для двофакторної автентифікації) та FIDO2, зокрема CTAP2, що дозволяє повністю відмовитися від паролів, використовуючи апаратний ключ як основний метод входу.

Типи підключення:

- USB-A або USB-C – для підключення до комп'ютерів, ноутбуків чи інших пристроїв;
- Bluetooth – безпровідне підключення до смартфонів або інших пристроїв, сумісних із Bluetooth;
- NFC – безконтактне підключення для сучасних смартфонів (наприклад, Android або iOS-пристроїв з підтримкою NFC).

Щодо сумісності, то Titan Key працює з обліковими записами Google, а також із будь-якими сервісами, які підтримують FIDO (GitHub, Facebook, Dropbox тощо), сумісний із сучасними операційними системами: Windows, macOS, ChromeOS, Linux, Android, iOS.

Ключі Google Titan Key ідеально підходять для людей, які потребують високого рівня безпеки, таких як журналісти, бізнесмени, урядовці чи IT-фахівці.

5 ЗАХИСТ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ВІД НЕСАНКЦІОНОВАНОГО КОПІЮВАННЯ

Захист програмного забезпечення від копіювання – це комплекс заходів, які розробники застосовують для запобігання несанкціонованому розповсюдженню та використанню їхнього продукту. Існує декілька основних методів такого захисту.

5.1 Апаратна прив’язка (Hardware Binding)

Прив'язка програмного забезпечення до апаратного забезпечення є одним із методів захисту від несанкціонованого копіювання та використання. Реалізується цей метод таким чином:

- під час активації програмне забезпечення збирає унікальні ідентифікатори певних апаратних компонентів комп'ютера;
- на основі зібраних ідентифікаторів генерується унікальний ключ активації або ліцензійний файл, який прив'язується до конкретної конфігурації «заліза»;
- інформація про прив'язану конфігурацію зберігається на комп'ютері користувача (наприклад, у файлі ліцензії, реєстрі) або на сервері розробника.

Результатом такого методу захисту є те, що ПЗ активується та працює лише на конкретному комп'ютері або пристрої, ідентифікованому за унікальними характеристиками його апаратного забезпечення. Кожного разу під час запуску програмне забезпечення перевіряє поточну конфігурацію апаратного забезпечення та порівнює її зі збереженою інформацією про прив'язку. Якщо конфігурація змінилася (наприклад, у разі спроби запустити ПЗ на іншому комп'ютері), програмне забезпечення може відмовитися працювати або потребувати повторної активації.

Прив'язка може здійснюватися до унікальних ідентифікаторів компонентів, серед яких можуть бути такі:

- MAC-адреса мережевої карти – унікальний фізичний ідентифікатор мережевого адаптера;
- серійний номер жорсткого диска;
- серійний номер центрального процесора (CPU), який є унікальним;
- ID материнської плати;
- інші.

Крім того, як було зазначено у розділі 4, для захисту від несанкціонованого копіювання і використання можуть бути застосовані:

- апаратні ключі (Dongle) – окремі фізичні пристрої, які містять унікальні ідентифікатори і підключаються до комп'ютера. ПЗ перевіряє його наявність;

– апаратні криптографічні модулі (Trusted Platform Module), вбудовані у деякі материнські плати, які можуть використовуватися для безпечного зберігання криптографічних ключів та виконання криптографічних операцій, включно прив'язку ліцензій.

Розробники мають ретельно вибирати компоненти, до яких буде здійснюватися прив'язка. Занадто часті зміни таких компонентів користувачами можуть призвести до проблем з активацією. З іншого боку, прив'язка лише до одного компонента може бути менш надійною. Часто використовується комбінація декількох ідентифікаторів.

Прив'язка ПЗ до апаратного забезпечення є ефективним методом захисту від несанкціонованого копіювання, але вона також має свої недоліки, особливо з погляду зручності для легальних користувачів (табл. 5.1). В процесі прийняття рішення про використання цього методу розробникам необхідно зважити всі «за» і «проти» та врахувати потреби своєї цільової аудиторії. Часто оптимальним є комбінування прив'язки до «заліза» з іншими методами захисту та гнучкою політикою ліцензування.

Таблиця 5.1 – Переваги і недоліки прив'язки ПЗ до апаратного забезпечення

Переваги	Недоліки
Високий рівень захисту від копіювання, оскільки значно ускладнює нелегальне копіювання та використання ПЗ на декількох комп'ютерах одночасно.	Незручність для легальних користувачів, оскільки заміна апаратних компонентів може призвести до необхідності повторної активації або навіть втрати ліцензії, якщо система ліцензування не передбачає механізмів перенесення ліцензії.
Обмеження використання відповідно до ліцензії, оскільки дозволяє чітко контролювати кількість установлених ПЗ.	Проблеми з віртуальними машинами, оскільки ПЗ, прив'язане до фізичного «заліза», може некоректно працювати або взагалі не запускатися у віртуальних середовищах.
Ускладнення обходу захисту, оскільки зміна апаратних компонентів для обходу прив'язки є складним і часто невигідним для зловмисників.	Можливі помилкові спрацювання: незначні зміни в конфігурації (наприклад, оновлення драйверів) іноді можуть спричинити помилкову деактивацію ПЗ.
	Складність реалізації та підтримки (розробка та підтримка системи прив'язки до "заліза" може бути технічно складною та вимагати значних ресурсів).

Прикладом ПЗ, захист якого здійснюється за допомогою апаратних прив’язок, можуть слугувати такі програми, як AutoCAD (Autodesk) (застосовується прив’язка до комп’ютера, а у разі зміни обладнання потрібна повторна активація), Matlab (ліцензія залежить від апаратного ID або MAC-адреси).

5.2 Активація за ліцензійним ключем

Активація програмного забезпечення за ліцензійним ключем є одним із найпоширеніших методів контролю використання та захисту від несанкціонованого копіювання. Суть цього процесу полягає в тому, що користувач, який придбав програмний продукт, отримує унікальний ліцензійний ключ (серійний номер, код активації тощо), який необхідно ввести під час встановлення або першого запуску програми для підтвердження легальності використання. Процес активації за ліцензійним ключем зазвичай містить такі кроки.

1. *Отримання* ліцензійного ключа якимось способом: надсилання на електронну пошту після онлайн-покупки, у друкованому форматі на пакуванні або картці, в особистому кабінеті на сайті розробника і т. п.

2. *Введення і перевірка* ліцензійного ключа. Під час встановлення або першого запуску ПЗ користувачеві пропонується ввести отриманий ліцензійний ключ, а спеціальний модуль активації здійснює перевірку введеного ключа. Ця перевірка може бути локальною (на комп'ютері користувача) або віддаленою (через підключення до сервера розробника).

3. Якщо ліцензійний ключ є дійсним і відповідає програмному продукту, відбувається *активація*, яка може містити запис інформації про активацію на комп'ютері користувача (наприклад, у файлі ліцензії, реєстрі), надсилання інформації про успішну активацію на сервер розробника з прив'язкою до унікальних ідентифікаторів комп'ютера (якщо використовується онлайн-активація) тощо. Активація може бути здійснена по-різному:

- *локальна активація* – без підключення до Інтернету, коли програма може використовувати певні алгоритми для валідації ключа (метод менш надійний, оскільки алгоритми перевірки можуть бути зламані;
- *онлайн-активація*, коли для перевірки ліцензійного ключа потрібно підключення до сервера розробника. Сервер перевіряє дійсність ключа, може реєструвати кількість активацій та прив'язувати ліцензію до унікальних ідентифікаторів комп'ютера (дещо вищий рівень захисту);
- *офлайн-активація* (через запит-відповідь), коли користувач генерує код запиту на своєму комп'ютері, передає його розробнику (наприклад, через веб-сайт або телефон), а у відповідь отримує код активації, який вводить у програму.

Хоча активація програмного забезпечення за ліцензійним ключем є важливим і широко використовуваним методом захисту від піратства та контролю ліцензійного використання, розробникам необхідно враховувати потенційні незручності для легальних користувачів та забезпечувати зручні механізми повторної активації або перенесення ліцензій у випадках зміни апаратного забезпечення (табл. 5.2).

Таблиця 5.2 – Переваги і недоліки активації програмного забезпечення за ліцензійним ключем

Переваги	Недоліки
Простий процес для легальних користувачів: зазвичай, активація є одноразовою та нескладною процедурою.	Ризик втрати ключа, що може призвести до проблем з повторною активацією.
Ефективний метод контролю: дозволяє обмежити використання програмного забезпечення лише тими користувачами, які придбали ліцензію.	Незручності у разі зміни апаратного забезпечення (особливо під час онлайн-активації з прив'язкою до «заліза») знадобиться повторна активація або перенесення ліцензії.
Можливість контролю кількості установлень: під час онлайн-активації розробник може обмежувати кількість комп'ютерів, на яких може бути активований один ліцензійний ключ.	Можливість нелегального розповсюдження ключів: ліцензійні ключі можуть бути скомпрометовані та нелегально розповсюджені.
Підтримка різних моделей ліцензування: ліцензійні ключі можуть бути прив'язані до різних типів ліцензій (наприклад, персональна, корпоративна, пробна).	Залежність від інфраструктури розробника (за онлайн-активації): для онлайн-активації потрібна стабільна робота серверів розробника.

5.3 Ліцензування у хмарі (Cloud Licensing)

Ліцензії у хмарі являють собою сучасний підхід до управління та надання доступу до програмного забезпечення, який тісно інтегрований з хмарними технологіями. Замість традиційного придбання ліцензії, прив'язаної до конкретного пристрою або користувача, ліцензування в хмарі передбачає використання програмного забезпечення як послуги (SaaS) або керування ліцензіями через хмарну інфраструктуру.

Ліцензування у хмарі може використовувати різні моделі:

- *підписка на користувача (Per-User Subscription)*, коли кожен окремий користувач має свій обліковий запис та може отримувати доступ до програмного забезпечення з різних пристроїв;
- *підписка на пристрій (Per-Device Subscription)*, коли ліцензія прив'язується до конкретного пристрою, а кількість користувачів, які можуть використовувати ПЗ на цьому пристрої, може бути необмеженою;
- *ліцензування за обсягом (Volume Licensing)* у випадку надання знижок у разі придбання великої кількості ліцензій для організації, а керування цими ліцензіями також може здійснюватися через хмарну платформу;

- *ліцензування за функціональністю (Feature-Based Licensing),* коли користувачі оплачують доступ до певних функцій або модулів ПЗ;
- *ліцензування на основі використання (Usage-Based Licensing),* коли оплата залежить від фактичного використання ресурсів (обчислювальна потужність, обсяг збережених даних або кількість транзакцій);
- *конкурентне ліцензування (Concurrent Licensing / Floating Licensing),* коли визначається максимальна кількість одночасних користувачів ПЗ. Ліцензії розподіляються динамічно між користувачами, які в цей момент працюють з програмою, а керування здійснюється хмарним сервером.

Ліцензування у хмарі є прогресивним та зручним підходом до надання доступу до ПЗ. Воно пропонує значні переваги в плані гнучкості, масштабованості та управління. Однак, користувачам потрібно враховувати залежність від інтернет-з'єднання, періодичні платежі та питання безпеки під час вибору цієї моделі ліцензування (табл. 5.3). Ліцензії у хмарі стають все більш популярними, особливо для бізнес-застосунків та онлайн-сервісів.

Таблиця 5.3 – Переваги і недоліки ліцензування у хмарі

Переваги	Недоліки
Зниження початкових витрат: замість великих одноразових платежів користувачі сплачують періодичні внески, що може бути більш доступним.	Залежність від інтернет-з'єднання: для доступу до ПЗ потрібне стабільне підключення до Інтернету.
Гнучкість та масштабованість: легко адаптувати кількість ліцензій до поточних потреб бізнесу.	Періодичні платежі: загальна вартість використання ПЗ протягом тривалого періоду може перевищити вартість одноразової ліцензії.
Просте управління: централізоване керування ліцензіями спрощує адміністрування для організацій.	Питання безпеки та конфіденційності: дані користувачів зберігаються на хмарних серверах, що може викликати занепокоєння щодо безпеки та конфіденційності.
Автоматичні оновлення: користувачі завжди мають доступ до останньої версії ПЗ без додаткових зусиль.	Ризик припинення роботи сервісу: у випадку проблем у постачальника хмарних послуг користувачі можуть втратити доступ до програмного забезпечення.
Покращений контроль за використанням: розробники мають краще розуміння того, як використовується їхнє ПЗ.	Обмеження у налаштуванні: SaaS-рішення часто мають обмежені можливості для кастомізації порівняно з локальним ПЗ.
Доступність: можливість роботи з ПЗ з будь-якого пристрою та місця за наявності інтернет-з'єднання.	

Прикладами програм, захист яких реалізовано за допомогою ліцензування у хмарі, є Microsoft 365 (Office) (працює через хмару, ліцензії зберігаються в обліковому записі Microsoft), Adobe Creative Cloud (постійне підключення до Інтернету та перевірка підписки).

5.4 Захист прив'язкою до файлової системи

Захист програмного забезпечення шляхом прив'язки до файлової системи (ФС) є менш поширеним і менш надійним методом порівняно з прив'язкою до апаратного забезпечення. Проте, все ж існують підходи, які використовують характеристики файлової системи для обмеження використання ПЗ.

Ідея полягає в тому, щоб програмне забезпечення під час активації або першого запуску ідентифікувало певні унікальні характеристики файлової системи, на якій воно встановлено. Потім ця інформація використовується для генерування ліцензії або для подальшої перевірки під час кожного запуску. Якщо характеристики файлової системи змінюються (наприклад, під час копіювання на інший диск або іншу файлову систему), програмне забезпечення може відмовитися працювати.

Для реалізації такого захисту необхідно виконати декілька кроків.

1. Збір інформації про файлову систему, тобто під час активації ПЗ збирає одну або декілька унікальних характеристик поточної ФС.
2. Генерування ліцензії/ключа, коли на основі зібраної інформації генерується ліцензійний ключ або файл, який прив'язується до цих характеристик.
3. Збереження інформації, яке здійснюється, як правило, локально (наприклад, у файлі ліцензії).
4. Перевірка під час запуску, тобто під час кожного запуску ПЗ знову збирає характеристики поточної файлової системи та порівнює їх зі збереженою інформацією. У разі невідповідності ПЗ може заблокуватися.

Для прив’язок можна використати такі характеристики файлової системи:

- серійний номер то́му (Volume Serial Number), оскільки кожен логічний диск (том) у файловій системі Windows має унікальний серійний номер;
- ідентифікатор файлової системи (Filesystem ID) – унікальний ідентифікатор, що присвоюється файловій системі під час її створення (може бути специфічним для різних операційних систем та файлових систем);
- унікальні ідентифікатори файлів або каталогів. В процесі захисту програмне забезпечення може створювати або шукати певні файли чи каталоги з унікальними атрибутами (наприклад, часом створення, розміром) та використовувати їх як частину ідентифікації;

– інформація про структуру файлової системи – ПЗ може аналізувати певну структуру каталогів або наявність конкретних системних файлів.

Поза деякими позитивними характеристиками цього методу захисту, кількість недоліків прив'язки ПЗ до файлової системи перевищує:

– низька надійність, оскільки характеристики файлової системи можуть змінюватися в процесі різних операцій, таких як зміна розділів, копіювання файлів на інший носій з іншою файловою системою;
– проблеми для легальних користувачів, оскільки заміна жорсткого диска, переустановлення операційної системи або навіть просте копіювання програми на інший логічний диск може призвести до втрати активації;
– легкість обходу, тому що зловмисники можуть емулювати або змінювати характеристики ФС на рівні операційної системи або використовувати інші методи для обходу такого захисту;
– залежність від операційної системи та файлової системи, оскільки методи отримання та перевірки характеристик можуть відрізнятися в різних операційних та файлових системах (NTFS, FAT, ext4 тощо), що ускладнює розробку кросплатформного захисту;
– обмеженість унікальних ідентифікаторів – кількість дійсно унікальних та стабільних ідентифікаторів ФС може бути обмеженою.

Через вищезазначені недоліки прив'язка до файлової системи рідко використовується як основний або єдиний метод захисту ПЗ від копіювання. Вона може бути використана як один з елементів багаторівневої системи захисту, але покладатися лише на неї не рекомендується через її низьку надійність та зручність для користувачів. Тому цей метод не рекомендується як основний спосіб захисту ПЗ від несанкціонованого копіювання. Звичайно, більш ефективними є методи прив'язки до апаратного забезпечення, ліцензування та комбінація різних технік захисту, але й прив’язка до файлової системи в деяких випадках може використовуватись у комбінації з іншими методами або як додатковий параметр прив’язки.

Хоча жоден метод не є абсолютно надійним, комбінація декількох з них значно ускладнює копіювання та злам програмних застосунків і нелегальний доступ.

6 ІНСТРУМЕНТИ ЗЛАМНИКА ДЛЯ НЕСАКЦІОНОВАНОГО ДОСЛІДЖЕННЯ ПРОГРАМ

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

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

Скільки б рівнів захисту не було передбачено, програма може бути зламана – це тільки питання часу і витрачених зусиль. Але, за відсутності дієвих правових регуляторів захисту інтелектуальної власності, розробникам часто доводиться покладатися на стійкість свого захисту.

Існує думка, що якщо витрати на нейтралізацію захисного механізму будуть не нижчими за вартість легальної копії, її ніхто не зламуватиме. Це неправильно: матеріальний стимул – не єдине, що рухає хакером. Набагато сильнішою мотивацією є інтелектуальна боротьба з автором захисту, спортивний азарт, цікавість, підвищення свого професіоналізму, та й просто цікаве проведення часу. Деякі люди можуть тижнями працювати над налагоджувачем, знімаючи захист з програми вартістю лише декілька доларів.

Внаслідок цього першочерговою задачею зламника під час зламу практично будь-якого захисту є можливість отримання твердої копії досліджуваної програми, тобто можливість статичного дослідження програми.

Відсутність початкових текстів зовсім не є непереборною перешкодою для вивчення і модифікації коду програмного застосунка. Методики зворотного проєктування дозволяють автоматично розпізнавати бібліотечні функції, локальні змінні, типи даних, розгалуження, цикли тощо.

Будь-який, навіть самий надійний захист від копіювання, що забезпечує майже 100 %-ву гарантію легальності копії, є марним, якщо код програмного продукту доступний для вивчення й аналізу. У ньому завжди знайдуться місця, злегка змінивши які, можна якщо не цілком відключити захист, то, принаймні, отримати певну інформацію, яка має деяку цінність.

6.1 Основні класи і категорії засобів для дослідження програм

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

Всі засоби дослідження роботи програмних продуктів, зокрема і захищених, можна розбити на чотири класи:

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

Для несанкціонованого дослідження ПЗ (реверс-інжинірингу, аналізу, зламу чи модифікації) використовуються різноманітні інструменти та техніки, які допомагають аналізувати, розуміти та модифікувати програмне забезпечення без дозволу власника або розробника.

Основні категорії засобів для НСД такі:

Дизасемблери – інструменти для статичного дослідження, які видають лише «чистий код», хоча сучасні дизасемблери здатні також розпізнати виклики стандартних функцій, виділити локальні змінні в процедурах і надати інший подібний сервіс, а також виконувати функції декомпіляторів та налагоджувачів.

Декомпілятори – інструменти для статичного дослідження, які намагаються виконати зворотний процес до компіляції. Компілятор бере вихідний код, написаний людиною, і перетворює його на машинний код, який може безпосередньо виконуватися процесором. Декомпілятор намагається зробити навпаки – взяти цей машинний код і згенерувати більш зрозумілий код високого рівня.

Налагоджувачі – інструменти для динамічного дослідження – виконують принципово інші функції, вони дозволяють аналізувати код в процесі його роботи, відстежувати і змінювати стан регістрів і стека, правити код «на льоту» – загалом, спостерігати за роботою програми і навіть активно в неї втручатися.

Утиліти для дампінгу процесів – це інструменти для динамічного дослідження, які дозволяють зробити знімок стану пам'яті (дамп) запущеного процесу в певний момент часу та зберегти його у файл. Цей дамп містить вміст віртуального адресного простору процесу, включно код програми, дані, стек, купу та інформацію про потоки виконання.

Редактори ресурсів – інструменти для статичного дослідження і модифікації, які спеціалізуються на роботі з ресурсами, вбудованими у виконувані файли (наприклад, EXE, DLL) та інші файли (наприклад, RC-файли). Ресурси можуть містити в собі іконки, курсори, рядки, діалогові вікна, меню, зображення, звуки та інші невиконувані дані програми.

Шістнадцяткові редактори (Hex editors) – інструменти для статичного дослідження і модифікації, які дозволяють користувачам переглядати та редагувати вміст файлів на рівні окремих байтів. Замість інтерпретації даних як тексту, зображень або інших форматів, вони відображають кожен байт у його шістнадцятковому поданні (від 00 до FF) разом із його ASCII-еквівалентом (якщо він є).

Утиліти для розпакування (unpackers) використовуються для вилучення оригінального вмісту файлу, який був попередньо запакований або заархівований спеціальним програмним забезпеченням (пакувальником або архіватором).

Утиліти моніторингу – інструменти для динамічного дослідження. Використовуються, коли необхідно знати, які саме дії виконує та або інша програма, звідки зчитує і куди записує дані, які стандартні функції вона викликає і з якими параметрами. Одержати ці відомості якраз і допомагають утиліти моніторингу.

6.2 Дизасемблери

Ці інструменти перетворюють виконуваний код у зрозумілішу форму (машинний код або псевдокод). Користуючись дизасемблером, можна лише здогадуватися про те, які дані одержує та або інша функція як параметри і що вони означають. Щоб з'ясувати це, найчастіше потрібним є вивчення якщо не всієї програми, то досить значної її частини.

6.2.1 Складові типового дизасемблера

Звичайно, кожен дизасемблер має свої характеристики і функціональні можливості, свої переваги і недоліки. Вигляд типового дизасемблера і його основних частин наведено на рисунку 6.1.

Як правило, головне вікно типового дизасемблера, містить, як мінімум, такі складові:

- вікно дизасемблера (Disassembler window). Тут розташовуються код, який виконується (асемблерні інструкції), адреса інструкцій у пам’яті та машинний код (опкоди);

– вікно регістрів (Registers window), яке показує значення регістрів процесора (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP, EFLAGS). Тут можна бачити, як змінюються дані під час виконання програми. Також зручно стежити за прапорцями (Zero, Carry, Sign, Overflow);
– вікно дампа пам’яті (Dump window), яке показує дані пам’яті у шістнадцятковому вигляді (Hex) та ASCII. Воно використовується для аналізу даних, які зберігаються в оперативній пам’яті (рядки, масиви, ключі). В ньому зазвичай можна переходити за будь-якою адресою;
– вікно стека (Stack window), яке показує вміст стеку – адреси повернення, локальні змінні, параметри функцій. Це іноді дуже зручно для аналізу викликів функцій та роботи з підпрограмами.

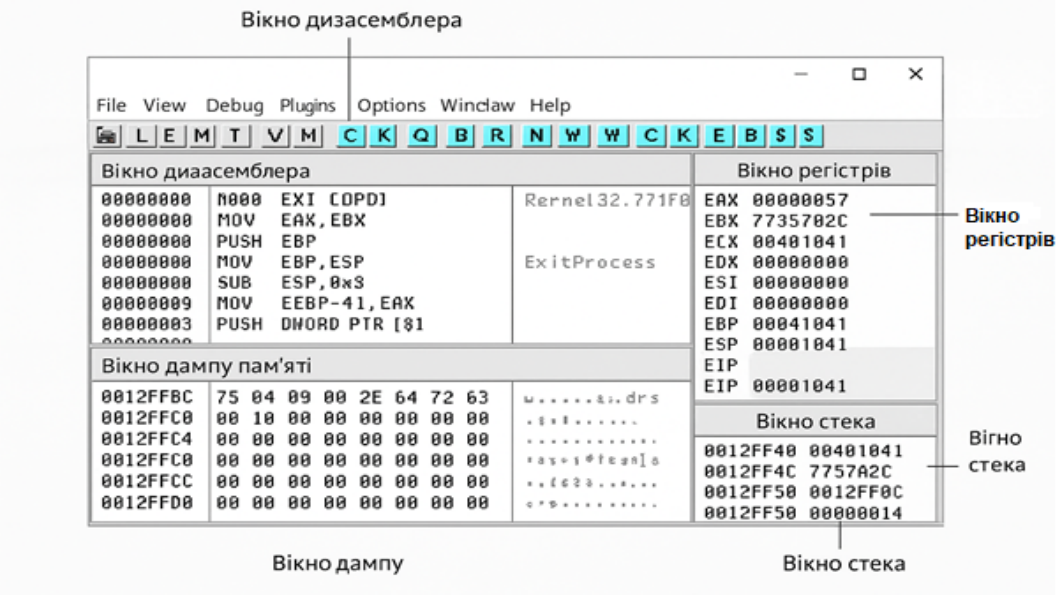


Рисунок 6.1 – Вигляд головного вікна типового дизасемблера

6.2.2 Огляд деяких сучасних дизасемблерів

IDA Pro. IDA Pro (Interactive Disassembler Professional) – інструмент, розроблений бельгійською приватною компанією Hex-Rays, яка спеціалізується на розробці інструментів для аналізу бінарного програмного забезпечення в галузі інформаційної безпеки.

Це один з найпотужніших та найпоширеніших комерційних дизасемблерів, який перетворює машинний код виконуваних файлів у зрозумілу форму асемблерного коду. IDA Pro підтримує велику кількість архітектур (x86, x64, ARM, MIPS, PowerPC та ін.) і форматів виконуваних файлів та має безліч плагінів для розширення функціональності. Так, окрім можливостей дизасемблювання коду, він може виконувати функції декомпілятора (перетворює машинний код у псевдокод, схожий на мови високого рівня, наприклад, C++) або налагоджувача (дозволяє покроково виконувати код,

встановлювати точки зупину, переглядати значення регістрів та пам'яті для аналізу поведінки програми під час виконання).

IDA Pro має зручний графічний інтерфейс користувача, що полегшує навігацію та аналіз коду. Це інтерактивний інструмент, який дозволяє аналітику (або зламнику) перейменовувати змінні та функції, додавати коментарі, визначати структури даних, що значно покращує розуміння коду.

IDA Pro є комерційним програмним забезпеченням і має різні варіанти ліцензування:

- базова версія (IDA Pro Essential) з обмеженою кількістю декомпіляторів на вибір (хмарні);
- версії (IDA Pro Expert) з різною кількістю локальних декомпіляторів на вибір;
- повна версія (IDA Pro Ultimate) з усіма доступними декомпіляторами.

Але існують і безкоштовні та комерційні альтернативи IDA Pro.

Ghidra. Це безкоштовний інструмент для реверс-інжинірингу, розроблений Агентством національної безпеки США (NSA). Він був представлений на конференції RSA у березні 2019 року.

Ghidra має графічний інтерфейс і такі можливості:

- під час дизасемблювання перетворює машинний код у зрозумілий код асемблера, полегшуючи аналіз низькорівневої логіки програми;
- підтримує безліч архітектур (x86, x64, ARM, MIPS, PowerPC та ін.);
- має вбудований декомпілятор, який перетворює асемблерний код у псевдокод, схожий на мову C (це спрощує розуміння високорівневої логіки програми без необхідності глибокого знання асемблера);
- надає графічне відображення потоку керування (Function Graph), що допомагає візуалізувати структуру функцій та їх взаємодію;
- дозволяє аналізувати дані в різних форматах, виявляти рядки, константи та інші важливі елементи;
- може ідентифікувати відомі бібліотечні функції, що допомагає відрізнити стандартний код від кастомного;
- починаючи з версії 11.0, має вбудований налагоджувач, що дозволяє аналізувати програми під час їх виконання.

Крім того, Ghidra підтримує можливість спільної роботи над проєктами, дозволяючи кільком аналітикам (або зламникам) працювати з одним і тим самим набором даних одночасно, а, завдяки плагін-орієнтованій архітектурі, користувачі можуть розробляти власні плагіни для додавання нової функціональності.

Radare2. Це безкоштовний та потужний фреймворк для реверс-інжинірингу з відкритим вихідним кодом. Працює через командний рядок, що, з одного боку, робить його дуже гнучким та придатним для автоматизації і

надає широкі можливості для аналізу, емуляції, налагодження та модифікації бінарних файлів, а, з іншого боку, інтерфейс командного рядка може бути складним для новачків, особливо порівняно з інструментами, які мають графічний інтерфейс.

Щодо дизасемблювання, то Radare2 здійснює детальне виведення дизасемблерного коду з можливістю навігації, анотування та аналізу. Серед його можливостей:

- підтримує багато процесорних архітектур (x86, x64, ARM, MIPS, RISC-V й багато інших) та форматів файлів (ELF, PE, Mach-O, Java DEX, Android APK, iOS Mach-O та ін.);
- може бути використаний як декомпілятор – існують плагіни (наприклад, r2ghidra) для інтеграції з іншими декомпіляторами;
- має вбудований налагоджувач, який дозволяє покроково виконувати програми, встановлювати точки зупину, переглядати регістри та пам'ять, здійснюючи локальне та віддалене налагодження;
- може використовуватись для аналізу та дослідження образів файлових систем та для аналізу мережевого трафіка;
- містить інструменти для криптографічного аналізу та виконання базових криптографічних операцій.

Cutter. Це безкоштовний графічний інтерфейс користувача, побудований на основі Radare2 (спочатку Cutter розроблявся як графічний інтерфейс для Radare2) з відкритою платформою для реверс-інжинірингу. Важливою особливістю Cutter є те, що він використовує Rizin (це вільний фреймворк з відкритим вихідним кодом для реверс-інжинірингу та аналізу бінарних файлів) як свій основний рушій, і продовжує розвиватися у зв'язці з ним.

Основна мета Cutter – зробити потужні можливості Rizin/Radare2 доступними та зручними для використання, особливо для новачків, зберігаючи гнучкість для досвідчених користувачів:

- надає інтуїтивно зрозумілий інтерфейс для виконання багатьох складних команд Rizin, використовуючи вікна, меню та інтерактивні віджети тощо, підтримує графічне відображення потоку керування (Control Flow Graph) для кращого розуміння структури функцій;
- працює на Linux, macOS та Windows, тобто є кросплатформним;
- під час дизасемблювання відображає дизасемблерний код з навігацією, підсвічуванням синтаксису та можливістю коментування;
- підтримує плагіни для інтеграції декомпіляторів, включно Ghidra (через плагін rz-ghidra), RetDec та інші. Це дозволяє отримувати псевдокод високого рівня для кращого розуміння логіки програми;

– має вбудований налагоджувач, що дозволяє покроково виконувати код, встановлювати точки зупину та аналізувати стан програми під час виконання.

6.3 Декомпілятори

Основна мета декомпіляторів полягає в тому, щоб відтворити вихідний код програми високого рівня (наприклад, C, C++, Java, Python, .NET C#, Go) з її машинно-орієнтованого подання (наприклад, асемблерний код, байт-код).

Як було зазначено вище, майже усі дизасемблери, розглянуті вище (IDA Pro, Ghidra, Radare2, Cutter та інші), містять в собі функціональність декомпілятора для багатьох мов та архітектур. Але є багато програм, призначених саме для цієї цілі.

Hex-Rays. Hex-Rays (плагін до IDA Pro). Це один з найкращих комерційних декомпіляторів, який перетворює машинний код назад у псевдокод, схожий на мови високого рівня (наприклад, C) (рис. 6.2).

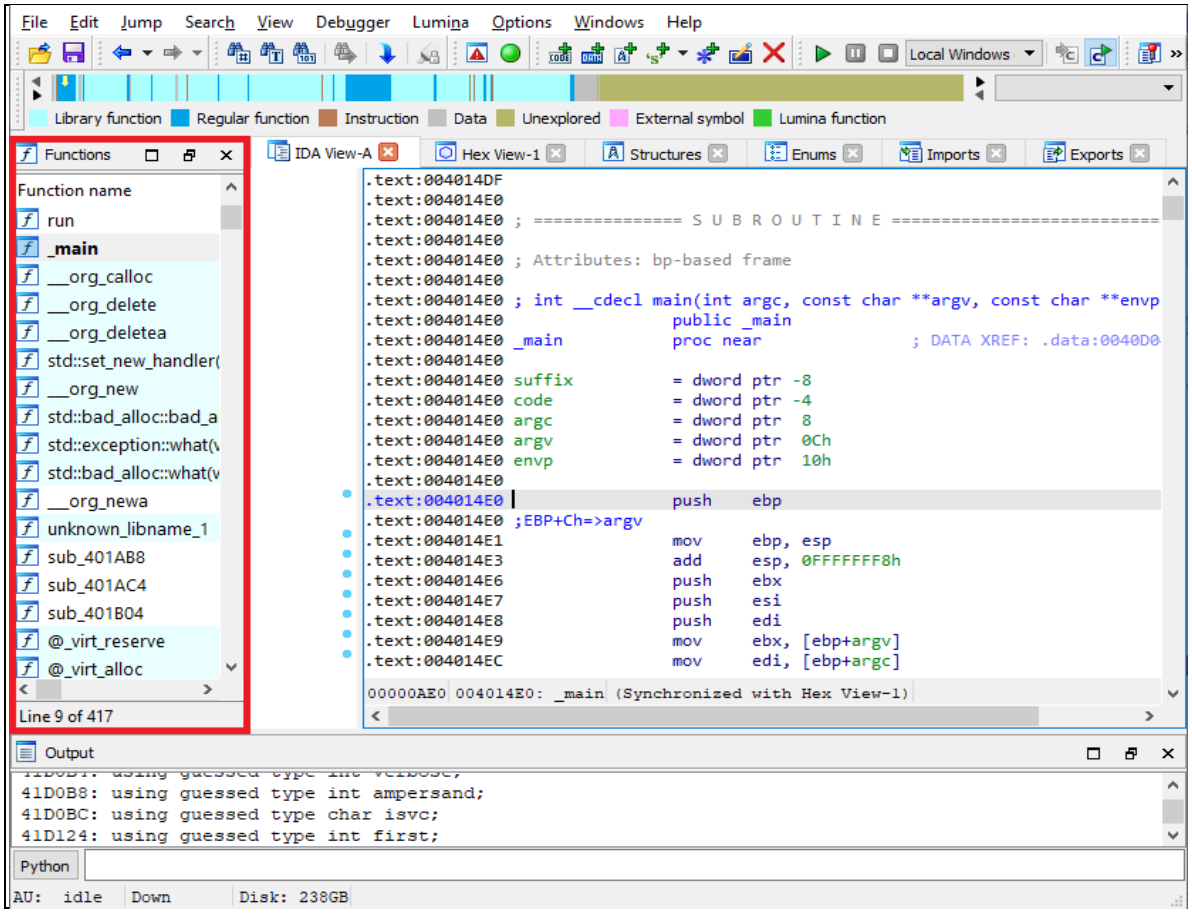


Рисунок 6.2 – Вигляд одного з вікон декомпілятора Hex-Rays Decompiler

Основні характеристики Hex-Rays Decompiler такі:

- декомпілятор підтримує широкий спектр архітектур, включно x86, x64, ARM (32-bit та 64-bit), PowerPC та інші;
- вміє розпізнавати типові патерни, які генеруються різними компіляторами, що допомагає у відтворенні більш читабельного псевдокоду;
- тісна інтеграція з IDA Pro забезпечує зручний робочий процес для аналізу, переходу між асемблерним кодом та декомпільованим псевдокодом, а також використання інших функцій IDA Pro;
- забезпечує читабельність коду, оскільки декомпілятор намагається генерувати псевдокод, який є максимально зрозумілим, використовуючи змінні, структури та інші елементи, що нагадують вихідний код високого рівня;
- має програмний інтерфейс (API), який дозволяє розробникам писати власні плагіни для розширення його функціональності та автоматизації певних завдань.

Через вказані можливості Hex-Rays Decompiler є одним з лідерів на ринку декомпіляторів завдяки своїй якості, широкій підтримці архітектур та тісній інтеграції з IDA Pro. Він є незамінним інструментом для професійних аналітиків шкідливого ПЗ, дослідників безпеки та реверс-інженерів та, звичайно, хакерів-зламників захисних механізмів ПЗ.

JD-GUI. JD-GUI є самостійною утилітою з графічним інтерфейсом (GUI), розробленою для декомпіляції та відображення вихідного коду Java з файлів .class. Це частина проєкту «Java Decompiler», який займається розробкою інструментів для декомпіляції та аналізу байт-коду Java і доступний для Windows, Linux та macOS. Для реконструкції вихідного коду Java JD-GUI використовує бібліотеку JD-Core. Основні характеристики JD-GUI такі:

- має простий у використанні графічний інтерфейс, дозволяє легко відкривати файли .class, .jar, .war, .ear, .aar, .kar, .jmod та .zip і переглядати відновлений вихідний код Java;
- надає зручну навігацію по відновленому вихідному коду з миттєвим доступом до методів та полів класів й дозволяє переглядати ієрархію класів та модулів Java у файлах;
- допомагає аналізу коду підсвічуванням синтаксису для кращої читабельності, дозволяє перетягувати файли безпосередньо у вікно JD-GUI для їх декомпіляції, підтримує фільтрований пошук у відновленому вихідному коді Java та ін. (рис. 6.3).

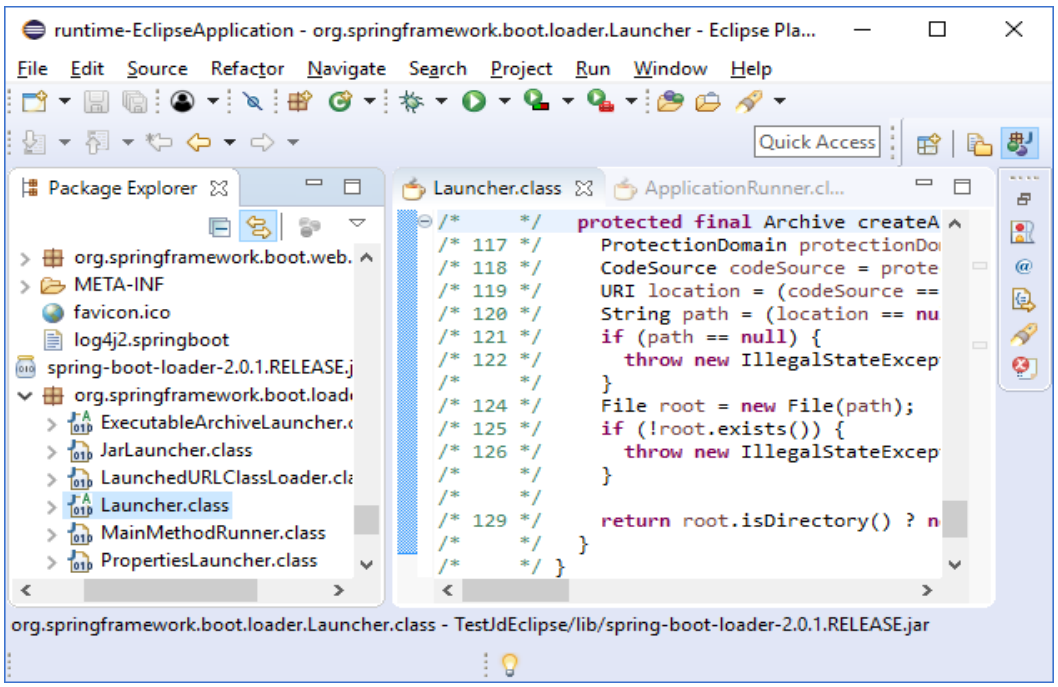


Рисунок 6.3 – Вигляд головного вікна декомпілятора Java Decompiler

Procyon. Procyon – це набір інструментів для метапрограмування Java, серед яких є потужний декомпілятор Java. Він вирізняється тим, що підтримує розширення мови Java, починаючи з версії 5 і вище, які часто не обробляються іншими декомпіляторами. Procyon добре справляється з декомпіляцією коду, написаного з використанням нововведень Java, включно такі, як Generics (узагальнення), Annotations (анотації), Enums (перелічення), Varargs (аргументи змінної довжини), Lambdas (лямбда-вирази) та method references (посилання на методи), String switch statements (оператори switch для рядків), Local та анонімні класи, Try-with-resources (оператор try з ресурсами), динамічні константи (Dynamic Constants) та ін.

Procyon розроблено як набір бібліотек, що дозволяє інтегрувати його функціональність в інші Java-програми. Проєкт активно розвивається, і регулярно виходять нові версії з виправленнями помилок та підтримкою нових можливостей Java.

Luyten. Luyten є безкоштовною утилітою з відкритим вихідним кодом з графічним інтерфейсом для декомпілятора Procyon. Luyten доступний у вигляді JAR-файлу, що робить його потенційно кросплатформним (за наявності встановленої Java). Також є виконувані файли для Windows.

Крім того, варто відзначити такі його характеристики: підтримка вкладок для одночасного перегляду у вкладках; можливість налаштування різних опцій декомпіляції; підтримка тем, що дозволяє користувачам налаштувати зовнішній вигляд; можливість зберігання декомпільованого коду у файли та інші функції.

Jadx. Jadx (Dex to Java decompiler) – це безкоштовний інструмент з відкритим вихідним кодом для декомпіляції виконуваних файлів Android (APK, DEX, AAR, AAB та JAR) у Java. Працює на Windows, Linux та macOS (для Java 11 і новіші). Він надає режим роботи за допомогою командного рядка (що дозволяє автоматизувати процес декомпіляції та інтегрувати Jadx в інші інструменти) і режим графічного інтерфейсу (підсвічування синтаксису, перехід до оголошень, пошук використання, повнотекстовий пошук тощо).

Jadx має вбудований деобфускатор, принаймні містить базові можливості для часткового деобфускування коду, що може бути корисним під час аналізування обфускованих додатків, а також може потенційно використовуватися зловмисниками для обходу захисту ПЗ. Але деобфускатор Jadx не є всеосяжним, Для складних технік обфускування можуть знадобитися спеціалізовані інструменти.

Отже, Jadx є потужним та зручним інструментом для декомпіляції Android-додатків, а безкоштовність, простота використання та наявність графічного інтерфейсу роблять його популярним вибором серед розробників, дослідників безпеки та ентузіастів Android. Хоча він не завжди може ідеально відтворити вихідний код, він є цінним інструментом для розуміння внутрішньої роботи Android-додатків.

dnSpy (.NET). dnSpy (.NET) – це безкоштовний інструмент з відкритим вихідним кодом для реверс-інжинірингу .NET-збірок. Він дозволяє переглядати вміст .NET-збірок (DLL, EXE), включно код на C#, VB.NET та інших .NET-мовах, а також редагувати їх та налагоджувати, причому якість декомпіляції зазвичай дуже висока.

Привабливим є і його графічний інтерфейс, який дозволяє використовувати підсвічування синтаксису, здійснювати навігацію по коду, переходити до оголошення символів, переглядати ієрархії типів та інші функції, зберігати декомпільований код у файли тощо.

Одна з ключових особливостей dnSpy – можливість редагувати існуючі .NET-збірки, змінювати код методів, додавати нові члени, редагувати метадані та ресурси. Крім того, dnSpy має вбудований потужний налагоджувач для .NET-застосунків, який дозволяє встановлювати точки зупину, покроково виконувати код, переглядати значення змінних, стектрейс та іншу інформацію під час виконання програми.

Разом з тим, dnSpy, як і будь-який декомпілятор, може мати проблеми з сильно обфускованим кодом, а редагування збірок може призвести до нестабільної роботи програми або викликати неочікувану поведінку та збої в роботі програми.

Важливо розуміти, що декомпілювання не є ідеальним процесом. Під час компіляції програм інформація втрачається (наприклад, імена змінних, коментарі, деякі структури керування. Тому декомпільований код зазвичай не є точною копією оригінального вихідного коду. Він може бути менш

читабельним і потребувати додаткового аналізу. Якість декомпіляції залежить від багатьох факторів: від складності програми, від використання оптимізації компілятора, від мови програмування та якості самого декомпілятора.

6.4 Налагоджувачі

Налагоджувачі є потужними інструментами для динамічного дослідження програмного забезпечення. На відміну від статичного аналізу, який передбачає вивчення коду без його виконання, динамічний аналіз зосереджується на поведінці програми під час роботи. Налагоджувачі дозволяють дослідникам контролювати виконання програми в реальному часі, спостерігати за її внутрішнім станом та виявляти потенційні проблеми.

6.4.1 Налагоджувачі для розробки програм під Windows

Visual Studio Debugger – потужний налагоджувач, інтегрований в середовище розробки Visual Studio, який підтримує налагодження програм, написаних мовами C#, C++, VB.NET, F# та іншими (рис. 6.4). Володіє широким спектром можливостей, включно: розширені точки зупину, перегляд об'єктів та значень змінних, багатопотокове налагодження, редагування коду під час налагодження, аналіз пам'яті та профілювання. Загалом, Visual Studio Debugger є потужним інструментом, який значно спрощує процес пошуку та усунення помилок у коді. Водночас він може бути успішно використаний для зламу систем захисту програм.

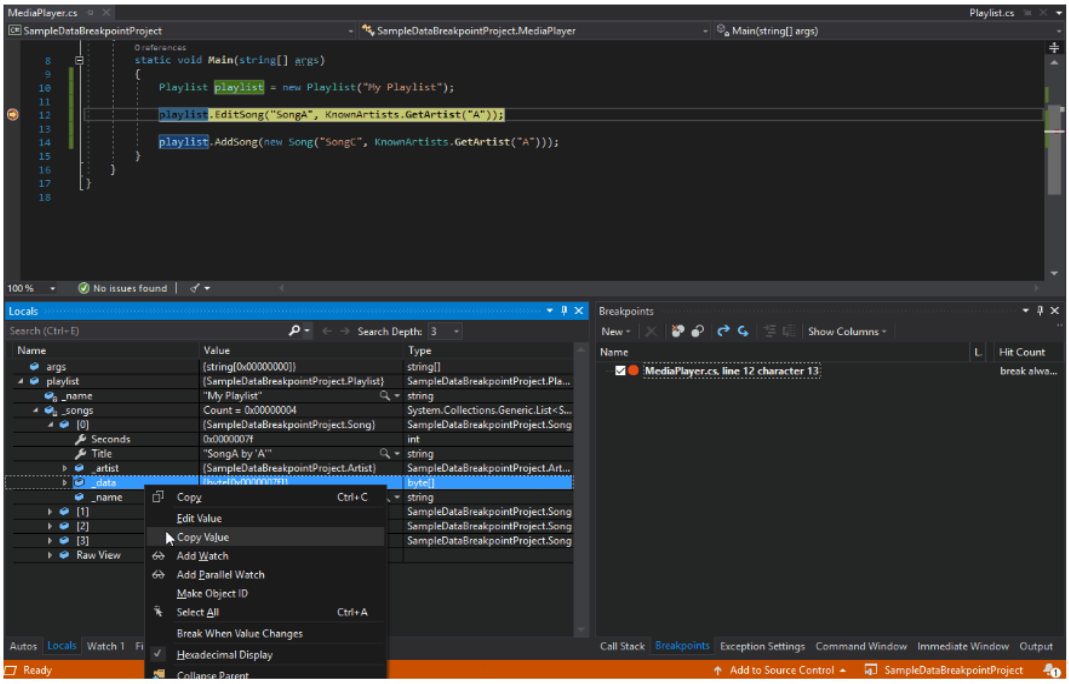


Рисунок 6.4 – Вигляд вікна налагоджувача Visual Studio Debugger

WinDbg є сучасним низькорівневим налагоджувачем від Microsoft, що розвиває можливості класичних інструментів для налагодження Windows. Він має оновлений інтерфейс, підвищену продуктивність та розширені функції для налагодження застосунків у режимі користувача та ядра, драйверів пристроїв і самої операційної системи. Хоча останні версії оснащені графічним інтерфейсом, WinDbg традиційно використовує командний інтерфейс, що забезпечує високу гнучкість роботи.

Крім того, він добре підходить для аналізу дампів пам'яті (crash dumps), що виникають після BSOD (синього екрана смерті) або аварій застосунків, допомагаючи визначити причину збою.

WinDbg є незамінним інструментом для розробників драйверів, системних адміністраторів та фахівців з безпеки, яким необхідно глибоко розуміти роботу операційної системи та налагоджувати складні проблеми.

OllyDbg – це 32-бітний налагоджувач рівня асемблера для операційних систем Windows. Він спеціалізується на аналізі двійкового коду, що робить його корисним у випадках, коли вихідний код програми недоступний. Цей налагоджувач використовується для багатьох цілей:

- для аналізу шкідливого програмного забезпечення, оскільки дозволяє досліджувати його поведінку без доступу до вихідного коду;
- для зворотної розробки (Reverse Engineering), особливо коли вихідний код недоступний;
- для пошуку та виправлення помилок у програмах;
- для дослідження захисту ПЗ (Software Cracking), для обходу механізмів захисту, хоча це є контроверсійним застосуванням.

Хоча остання стабільна версія OllyDbg 1.10 вийшла досить давно (у 2004 році), але вона залишається популярною. Розробляється також 64-бітна альфа-версія OllyDbg.

JetBrains Rider Debugger – налагоджувач, інтегрований в кросплатформне IDE Rider від JetBrains, спеціалізоване для розробки на .NET: ефективно працює з консольними застосунками, вебзастосунками (ASP.NET Core), юніт-тестами, Xamarin-застосунками та іншими типами застосунків. Він пропонує інтелектуальні функції налагодження, зручний інтерфейс та підтримку складних сценаріїв, має функцію редагування значень змінних під час налагодження. Загалом, JetBrains Rider Debugger є потужним, зручним та інтегрованим інструментом, який значно полегшує процес налагодження .NET-застосунків для розробників, що використовують Rider.

6.4.2 Налагоджувачі для розробки програм під macOS та iOS

Xcode Debugger – це потужний інструмент, інтегрований в середовище розробки Xcode, який використовується для налагодження

застосунків під macOS, iOS, watchOS та tvOS. Підтримує налагодження Objective-C та Swift, має інструменти для візуалізації інтерфейсу, аналізу продуктивності та пам'яті. Серед його можливостей: зручні інспектори для перегляду значень змінних, властивостей об'єктів та вмісту колекцій; стандартні функції покрокового виконання доповнені можливостями для низькорівневого налагодження; відображення історії викликів функцій, що дозволяє відстежувати шлях виконання програми; можливість перегляду вмісту пам’яті процесу та дизасембльованого коду для глибшого розуміння роботи програми на низькому рівні.

Xcode використовує LLDB як свій основний налагоджувач, надаючи потужну командну консоль для виконання складних запитів та команд.

LLDB *(Low Level Debugger)* **–** це високопродуктивний налагоджувач відкритого коду, який є основним налагоджувачем для Xcode на платформах Apple (macOS, iOS, watchOS, tvOS) та за замовчуванням використовується в багатьох інших середовищах розробки на Unix-подібних системах. Він розроблений з урахуванням модульності та розширюваності, що робить його гнучким та легким для інтеграції в різні інструменти. Він є основою для графічного інтерфейсу Xcode Debugger та багатьох інших UNIX-подібних налагоджувальних систем. Також може використовуватися через командний рядок. LLDB є дуже потужним та розширюваним.

6.4.3 Кросплатформні налагоджувачі

GNU Debugger (GDB) – класичний та дуже поширений налагоджувач командного рядка, який працює на багатьох UNIX-подібних системах (Linux, macOS тощо) та підтримує безліч мов програмування (C, C++, Fortran, Ada, Python та інші). Має такі можливості, як перегляд пам’яті, стека викликів, має можливість перегляду машинного коду програми (дизасемблювання), дозволяє зупиняти виконання програми у разі зміни значення певної змінної або виразу, може використовуватися для налагодження як запущених процесів, так і дампів пам'яті після аварійного завершення програми та багато інших можливостей.

Хоча GDB є інструментом командного рядка, існують графічні інтерфейси (фронтенди) для GDB, такі як KDbg або інтеграція в IDE (наприклад, IntelliJ IDEA, VS Code), які полегшують його використання. Він може здаватися менш інтуїтивним порівняно з графічними налагоджувачами, але його гнучкість та широкі можливості роблять його незамінним інструментом для багатьох розробників, особливо в Unix-подібних середовищах.

Visual Studio Code Debugger – це інтегрований налагоджувач у популярному крос-платформному редакторі коду Visual Studio Code. Він

підтримує налагодження широкого спектра мов програмування через розширення (наприклад, C++, Python, Java, JavaScript/Node.js). Має зручний графічний інтерфейс та базові функції налагодження. Завдяки розширенням, можливості налагодження VS Code можуть бути значно розширені для підтримки специфічних фреймворків та інструментів. Його інтеграція з редактором та підтримка широкого спектра мов роблять його одним з найпопулярніших налагоджувачів серед розробників.

IntelliJ IDEA Debugger – потужний налагоджувач, інтегрований в IDE IntelliJ IDEA від JetBrains. Підтримує налагодження Java, Kotlin, Groovy та інших JVM-мов, а також має розширення для налагодження JavaScript, Python та інших мов. Пропонує розширені можливості: оцінювання виразів, «гаряча заміна» коду та інтегрування з профілювальниками, спеціалізовані подання даних для колекцій, масивів та інших складних структур, що полегшують їх аналіз під час налагодження, зручні інструменти для аналізу та налагодження багатопотокових застосунків і цілу низку інших.

PyCharm Debugger. Цей налагоджувач, інтегрований в IDE PyCharm від JetBrains, спеціалізований для розробки на Python. Він пропонує зручні інструменти для налагодження Python-коду, включно точки зупину, покрокове виконання, перегляд змінних, віддалене налагодження, відображає історію викликів функцій, що призвели до поточної точки виконання, допомагаючи зрозуміти потік виконання програми, надає інструменти для налагодження багатопроцесних та багатопотокових застосунків Python. PyCharm пропонує спеціалізовані інструменти для налагодження коду, що використовує такі бібліотеки, як NumPy, Pandas та Matplotlib, включно зручний перегляд структур даних. Інтеграція з іншими функціями PyCharm робить його одним з найкращих налагоджувачів для Python-розробників.

6.4.4 Налагодження веб-сторінок у браузері

Chrome DevTools – це набір вбудованих інструментів розробника безпосередньо у веббраузері Google Chrome. Вони надають веброзробникам глибокий контроль над вебсторінками та їхніми застосунками, дозволяючи діагностувати проблеми, оптимізувати продуктивність та експериментувати зі змінами. Основні його можливості такі:

- перегляд та редагування DOM (Document Object Model) та CSS вебсторінки в реальному часі, тобто, можливість змінювати структуру HTML, стилі, атрибути та миттєво бачити результати;
- використання консолі (Console) для реєстрації повідомлень (помилок, попереджень, інформації) з JavaScript, виконання JavaScript-коду в контексті сторінки та перегляду об'єктів JavaScript;

- доступ до вихідного коду HTML, CSS та JavaScript, а також до інших ресурсів сторінки;
- включення налагоджувача JavaScript з можливістю встановлення точок зупину, покрокового виконання та перегляду значень змінних;
- показ інформації про всі HTTP-запити та відповіді, що відбуваються під час завантаження сторінки або виконання дій користувача;
- аналіз продуктивності веб-сторінки, із записом та візуалізацією активності браузера, що дозволяє виявляти вузькі місця, довгі завдання JavaScript, проблеми з рендерингом тощо;
- наявність інструментів для роботи з локальним сховищем (Local Storage), сесійним сховищем (SessionStorage), файлами cookie, базами даних IndexedDB та Web SQL, а також сервісними службами (Service Workers) та маніфестом веб-застосунку (Manifest);
- аналіз безпеки веб-сторінки – перевірка сертифікатів, політики безпеки контенту (CSP) та ін.;
- наявність автоматизованого інструменту для аудиту якості вебсторінки, включно продуктивність, доступність, SEO та найкращі практики для прогресивних веб-застосунків (PWA);
- наявність інструментів рендерингу (Rendering) для аналізу та оптимізації процесу рендерингу вебсторінки, виявлення проблем зі зсувами макета тощо.

Отже, Chrome DevTools є надзвичайно важливим інструментом для будь-якого веброзробника, оскільки він надає все необхідне для розуміння, налагодження та оптимізації вебзастосунків безпосередньо в браузері. Його постійно оновлюють та додають нові корисні функції.

6.5 Редактори ресурсів

Редактори ресурсів використовуються для керування некомпілювальними частинами застосунків, такими як графічні інтерфейси, зображення, звуки, рядки тексту, діалоги тощо. Вони спрощують процес розробки графічних інтерфейсів та керування іншими ресурсами, дозволяючи розробникам зосередитися на логіці застосунку, а не на ручному написанні коду для створення кожного елемента інтерфейсу. Вибір редактора часто залежить від використовуваної платформи, фреймворку розробки та особистих вподобань.

6.5.1 Редактори ресурсів для Windows

Visual Studio Resource Editor – редактор ресурсів інтегрований безпосередньо в середовище розробки Visual Studio. Підтримує візуальне редагування діалогів, меню, панелей інструментів, іконок, курсорів,

рядкових таблиць, версій ресурсів та інших типів ресурсів Windows. Зручний для розробників, які вже використовують Visual Studio.

PE Explorer – комерційний інструмент для аналізу та редагування файлів PE. Містить потужний редактор ресурсів з підтримкою візуального редагування діалогів, меню, зображень та інших типів ресурсів. Також надає розширені можливості для аналізу структури PE-файлів.

Resource Hacker – це безкоштовний, самостійний редактор ресурсів для файлів Win32 PE (Portable Executable), таких як .exe, .dll, .cpl тощо. Дозволяє переглядати, редагувати, додавати, видаляти та компілювати ресурси. Особливо корисний для модифікації існуючих застосунків або аналізу їхніх ресурсів (рис. 6.5).

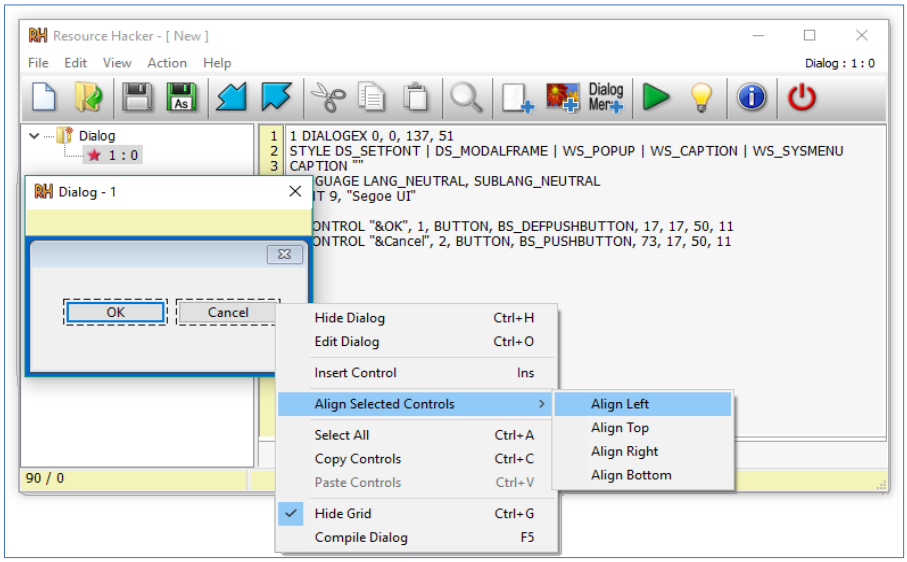


Рисунок 6.5 – Вигляд вікна редактора ресурсів Resource Hacker

6.5.2 Редактори ресурсів для macOS та iOS

Xcode Interface Builder (Storyboard та XIB Files) – редактор ресурсів, інтегрований в Xcode. Він використовується для візуального створення та редагування інтерфейсів користувача для macOS, iOS, watchOS та tvOS. Дозволяє розміщувати елементи керування, налаштовувати їх властивості та встановлювати зв'язки (outlets та actions) з кодом.

IBInspectable та IBDesignable. Хоча це не окремі редактори ресурсів для Xcode, вони разом дозволяють робити власні компоненти інтерфейсу візуально редагованими безпосередньо в Interface Builder, відображаючи зміни в реальному часі.

6.5.3 Кросплатформні редактори ресурсів

Qt Designer – частина Qt Framework – потужний візуальний редактор для створення інтерфейсів користувача на основі віджетів Qt. Він підтримує створення складних макетів, діалогів, головних вікон. Збережені інтерфейси використовуються з кодом на C++ або Python (PyQt, PySide).

wxFormBuilder – це безкоштовний кросплатформний редактор форм для wxWidgets, який дозволяє візуально створювати діалоги та фрейми, які потім можна використовувати в застосунках на C++, Python, Perl та інших мовах, що підтримують wxWidgets.

Glade – інструмент для швидкого розроблення інтерфейсів користувача для GTK+. Він дозволяє візуально створювати вікна, діалоги та інші елементи інтерфейсу. Збережені інтерфейси можуть бути використані з різними мовами програмування: C, Python (PyGTK, PyGObject), Vala тощо.

Використання редакторів ресурсів хакерами може мати кілька аспектів, пов'язаних з аналізом, модифікацією та розумінням існуючого ПЗ, а не зі створенням нових шкідливих програм з нуля (хоча це також можливо):

- аналіз інтерфейсу та функціональності: перегляд діалогів та форм, дослідження рядкових ресурсів, перегляд меню та панелей інструментів, що допомагає зрозуміти доступні функції програми та їх організацію;
- модифікація існуючого ПЗ: зміна інтерфейсу з метою обману користувача або приховання шкідливої функціональності, доданої іншим способом, маніпуляція повідомленнями для відображення фальшивих повідомлень, підроблених вікон авторизації або іншої дезінформації з метою фішингу або соціальної інженерії, заміна ресурсів, що може бути частиною атаки для маскування шкідливої програми під легітимну;
- розуміння внутрішньої структури: виявлення нестандартних ресурсів, за допомогою яких розробники можуть зберігати нестандартні дані, зв'язок з кодом, оскільки ресурси часто пов'язані з певними частинами коду тощо.

6.6 Шістнадцяткові редактори

Шістнадцяткові редактори (або hex-редактори) – це програми, які, на відміну від інших програм, дозволяють переглядати та редагувати вміст файлів на рівні байтів, тобто інтерпретують вміст файлу на вищому рівні, відображаючи кожен байт у шістнадцятковому (hex) та ASCII форматах.

Вони особливо корисні для:

- аналізу бінарних файлів (EXE, DLL, BIN, ISO, тощо);
- реверс-інжинірингу;

– пошуку та модифікації сигнатур;
– виправлення файлів, які пошкоджені або зашифровані;
– створення патчів до програмного забезпечення.

Зазвичай hex-редактори відображають вміст файлу у трьох областях:

1. Адресна область, де вказується зміщення (адреса) кожного рядка байтів у файлі.
2. Шістнадцяткова область, в якій значення кожного байта відображається у шістнадцятковому форматі.
3. Символьна область, яка показує спробу інтерпретації кожного байта як символу ASCII або іншого кодування.

Вигляд одного з вікон типового шістнадцяткового редактора наведено на рисунку 6.6.

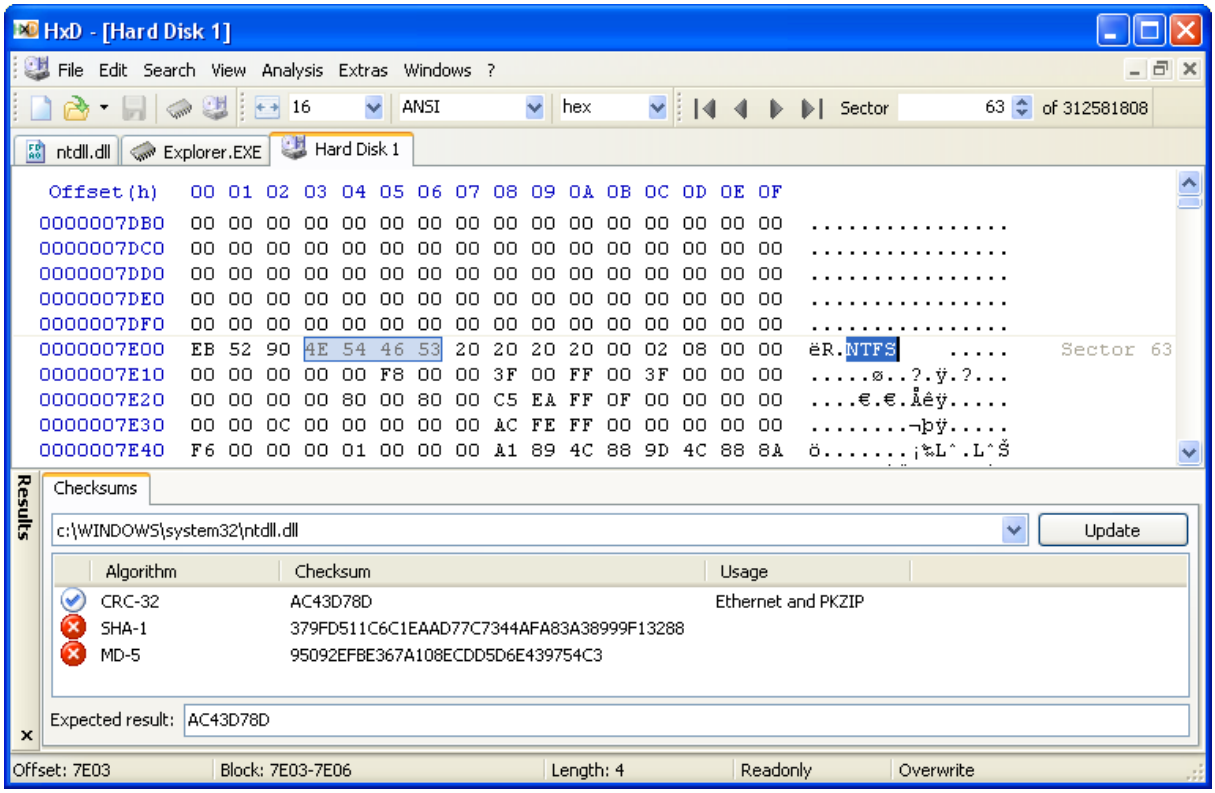


Рисунок 6.6 – Вигляд головного вікна шістнадцяткового редактора HxD

Багато інструментів, розглянутих вище (дизасемблери, налагоджувачі), мають певні можливості hex-редакторів коду.

Порівняльний огляд шістнадцяткових редакторів, використовуваних для аналізу коду, наведено у таблиці 6.1.

Таблиця 6.1 – Порівняльний аналіз шістнадцяткових редакторів

Назва	Особливості	Функціо-нальність
Для Windows		
Hex Editor Neo	Комерційний, з широким набором функцій, включно дизасемблер та підтримку скриптів (є безкоштовна версія з обмеженою функціональністю).	Розширений, професійний
010 Editor	Комерційний редактор, відомий своєю потужною системою шаблонів для розбору двійкових форматів	Розширений, професійний
Frhed	Портативний, безкоштовний, невеликий та швидкий двійковий файловий редактор, який дозволяє перегляда-ти та редагувати необроблені дані будь-якого файлу, незалежно від його розміру (обмежено лише системною пам'яттю).	Базовий
WinHex	Потужний інструмент з функціями для криміналіс-тичної експертизи.	Розширений, професійний
Для Linux		
Ghex	Стандартний графічний шістнадцятковий редактор для середовища GNOME	Базовий
Bless	Просунутий hex-редактор з можливістю редагування великих файлів та іншими корисними функціями	Розширений, професійний
Okteta	Безкоштовний редактор, є частиною KDE Applica-tions, з різними режимами відображення та інструментами	Базовий
wxHexEditor	Потужний редактор, здатний працювати з дуже великими файлами	Розширений, професійний
Для macOS		
Hex Fiend	Швидкий, безкоштовний, оптимізований для macOS, добре працює з великими файлами	Базовий
Кросплатформні		
ImHex	Редактор з відкритим кодом, розроблений для зворотної інженерії та програмістів, з підтримкою шаблонів та аналізу даних	Розширений, професійний
HxD	Безкоштовний, швидкий та зручний редактор з багатьма корисними функціями, включно редагування RAM та дисків	Базовий
wxHexEditor	Безкоштовний шістнадцятковий редактор з відкритим вихідним кодом	Розширений, професійний
UltraEdit	Комерційний, потужний текстовий редактор для Windows, Linux та macOS. Хоча він відомий як редактор коду та тексту з широкими можливостями, він також має вбудований шістнадцятковий редактор. Його шістнадцятковий режим пропонує розширені функції для аналізу та редагування двійкових файлів, інтегровані в загальне середовище редактора.	Розширений, професійний

У наведеній таблиці до базових редакторів віднесено ті, які надають основні функції перегляду та редагування шістнадцяткових даних, а до

розширених та професійних – ті, що мають додаткові інструменти для аналізу даних, розбору форматів файлів, скриптування тощо.

6.7 Програми для отримання дампу

Мета програм-дамперів полягає у створенні копії (дампу) пам'яті процесу або цілої операційної системи на певний момент часу. Ця копія містить у собі поточний стан оперативної пам'яті, включно код програми, її дані, значення змінних, вміст регістрів процесора та іншу важливу інформацію. Саме ця інформація може цікавити зламника. До того ж, більшість програм-дамперів дозволяють отримати готовий виконуваний файл, з якого вже було знято захист.

Отже, програми-дампери використовуються для отримання (знімання) всього вмісту оперативної пам'яті комп'ютера чи пристрою. Зазвичай вони використовуються для:

- аналізу роботи програм (зокрема для відновлення інформації після збою);
- виявлення уразливостей в системах безпеки;
- виконання аналізу пам'яті для зламу програм чи захисту від таких атак.

Програми для дампінгу можуть бути різними за своїм функціоналом, оскільки дампінг застосовується для отримання даних з пам'яті або даних з інтерфейсів.

6.7.1 Інструменти для дампінгу програм

Операційна система ***Windows*** має вбудовану функціональність для створення дампів пам'яті. Найпростіший спосіб створити дамп пам'яті окремого процесу, що працює – використати «Диспетчер завдань», в якому, клацнувши правою кнопкою миші на потрібному процесі, вибрати опцію «Створити файл дампа».

Крім того, Windows автоматично налаштований на створення дампів пам'яті у випадку критичних помилок системи (BSOD – «синій екран смерті») або збоїв програм.

Хоча ці вбудовані засоби корисні для базового дампінгу та системних збоїв, для більш гнучкого та контрольованого створення дампів процесів часто використовують утиліти сторонніх розробників, такі як ProcDump (частина Sysinternals Suite від Microsoft).

ProcDump – це потужна утиліта командного рядка від Microsoft Sysinternals, яка використовується для моніторингу процесів у Windows та створення дампів пам'яті на основі різних тригерів (наприклад, високе використання ЦП, помилки). Вона є значно гнучкішою та функціональ-

нішою за вбудований Диспетчер завдань у плані створення дампів, має більше опцій конфігурації.

ProcDump може відстежувати та створювати дампи не лише для вказаного процесу, а й для всіх його дочірніх процесів. Дампи пам'яті зберігаються у файли, які потім можна аналізувати за допомогою інструментів налагодження, таких як WinDbg.

WinDbg не є програмою для безпосереднього створення дампів у тому сенсі, як ProcDump або Диспетчер завдань. Її основна функція – це потужний інструмент налагодження (дебагінгу), який використовується для аналізу дампів пам'яті, а також для налагодження живих процесів та ядра операційної системи. Хоча це і не є її першочерговим завданням, WinDbg залишається незамінним інструментом, коли потрібно глибоко проаналізувати вже створений дамп для виявлення причин збоїв, помилок або дослідження шкідливого програмного забезпечення.

Volatility – набір інструментів для аналізу дампів пам'яті з комп'ютерів і мобільних пристроїв. Це потужний open-source фреймворк для аналізу оперативної пам'яті (RAM), написаний на Python. Він використовується в цифровій криміналістиці та реагуванні на інциденти для вилучення цифрових артефактів з дампів пам'яті.

На відміну від традиційних методів криміналістичного аналізу, які зосереджуються на статичних даних на жорсткому диску, Volatility досліджує динамічні та тимчасові дані, що знаходяться в оперативній пам'яті на момент її зняття. Це дозволяє отримати унікальне розуміння стану системи під час інциденту, виявити активні процеси, мережеві з'єднання, завантажені драйвери, шкідливе програмне забезпечення та багато іншого, що може бути відсутнім або приховано на диску.

Volatility має велику кількість плагінів, кожен з яких призначений для вилучення конкретного типу інформації з дампа:

- список запущених процесів, їхні батьківські процеси, час створення та завершення, ідентифікатори процесів (PID);
- активні мережеві з'єднання, відкриті порти, прослуховувальні сокети;
- список завантажених DLL-бібліотек та драйверів ядра;
- виявлення руткітів, інжектованого коду, підозрілих процесів та інших ознак шкідливої активності;
- розподіл пам'яті, виявлення прихованих викликів;
- доступ до кешованих даних реєстру;
- доступ до буфера командного рядка та історію команд;
- список відкритих файлів, м'ютексів, семафорів тощо;
- ключі шифрування та паролі, що зберігаються в пам'яті.

LordPE – утиліта для системних програмістів, яка потребує ручного редагування виконуваних файлів (Portable Executable). За допомогою цієї

утиліти можна редагувати exe-файли, аналізувати їх, оптимізувати, розбивати їх на частини та збирати назад. Утиліта також дозволяє знімати дамп процесу повністю або частково, оптимізувати як сам дамп, так і скомпільований виконуваний файл. Крім того, можливе редагування інформації в заголовку файлу, а також перегляд та редагування багатьох інших секцій виконуваного файлу.

kdump – це механізм дампів ядра, вбудований в ядро Linux. Його основне призначення – зберегти вміст системної пам'яті (RAM) у файл у випадку краху ядра (kernel panic), зависання системи або інших критичних помилок. Створений дамп пам’яті (часто називається vmcore) потім може бути проаналізований розробниками та системними адміністраторами для визначення причини збою.

lldb – це потужний, високопродуктивний налагоджувач, який є основним для операційних систем macOS, iOS, watchOS та tvOS, а також широко використовується на Linux та інших Unix-подібних системах. Він входить до складу проєкту LLVM (Low Level Virtual Machine), до якого також належить компілятор Clang та інші інструменти розробки.

6.7.2 Програми для дампінгу баз даних

Дампінг бази – це процес створення резервної копії або вилучення даних із СУБД. Його можна виконати через командний рядок або за допомогою спеціалізованих утиліт. Отриманий дамп може бути використаний для міграції, резервного копіювання або аналізу, зокрема і для несанкціонованого.

MySQLdump – це консольна утиліта, яка входить до складу клієнтського пакета MySQL. Її основне призначення – створення логічних бекапів баз даних MySQL. Це означає, що mysqldump генерує набір SQL-інструкцій, які під час виконання на сервері MySQL можуть відтворити структуру бази даних (таблиці, подання, тригери тощо) та її дані.

pg_dump – це консольна утиліта, яка входить до складу PostgreSQL. Її основне призначення – створення резервних копій (бекапів) баз даних PostgreSQL. Вона створює резервні копії у вигляді SQL-скриптів, здатних відновити структуру та дані бази, або у власному форматі архіву PostgreSQL, який є більш гнучким для відновлення та підтримує стиснення.

MongoDB Dump – утиліта для створення резервних копій баз даних MongoDB. Вона робить знімок даних і зберігає його у двійковому форматі BSON (Binary JSON) , що дозволяє надалі виконати точне відновлення.

Програми для дампінгу сайтів (наприклад, вебскрейпінг). Ці програми використовуються для збору великих обсягів даних з вебсайтів. Крім того, вони мають такі можливості:

- агрегування інформації з інтернет-магазинів;
- парсинг контенту для дослідження ринку;
- збір статистичних даних або аналіз цін.

Як приклади можна розглянути такі програми.

Scrapy – це потужний фреймворк з відкритим вихідним кодом, написаний мовою програмування Python, який призначений для вебскрейпінгу (web scraping) та вебкраулінгу (web crawling). Він надає структурований та ефективний спосіб вилучення даних з вебсайтів.

BeautifulSoup – це бібліотека Python, призначена для парсингу HTML та XML документів. Вона створює дерево розібраних об'єктів з HTML або XML, що дозволяє легко навігуватися по структурі документа, шукати та вилучати потрібні дані. Beautiful Soup часто використовується для вебскрейпінгу, коли потрібно витягувати інформацію з вебсторінок.

У деяких випадках дампи пам'яті можуть використовуватися в цифровій криміналістиці для аналізу стану системи під час інциденту безпеки або іншої події.

6.8 Утиліти для моніторингу ресурсів комп’ютера

Існують спеціалізовані програми для моніторингу роботи з файлами, реєстром та мережею в операційній системі Windows.

6.8.1 Моніторинг роботи з файловою системою

Ці програми відстежують створення, видалення, перейменування, читання та запис файлів і каталогів. Це може бути корисно для виявлення шкідливої активності, відстеження змін, аудиту доступу до файлів тощо. Але також ці утиліти можуть бути використані для відстежування файлів, де можуть зберігатися конфіденційні дані (паролі, ключі тощо).

Sysmon (System Monitor) – це безкоштовна утиліта від Microsoft Sysinternals. Це потужний системний сервіс і драйвер пристрою, який після встановлення постійно відстежує системну активність і записує її в журнал подій Windows. Sysmon може фіксувати створення процесів, мережеві з'єднання, зміни часу створення файлів, створення та видалення процесів, завантаження драйверів та DLL, а також зміни у файловій системі.

ProcMon (Process Monitor) – це ще одна безкоштовна утиліта від Microsoft Sysinternals. ProcMon в реальному часі відображає активність файлової системи, реєстру та процесів. Вона дозволяє фільтрувати за операціями (CreateFile, ReadFile, WriteFile, DeleteFile, Rename тощо), шляхом до файлу, процесом, результатом операції та іншими параметрами.

File Auditing – вбудовані засоби Windows для аудиту файлової системи. Можна налаштовувати аудит для певних файлів або каталогів, щоб від-

стежувати успішні та/або невдалі спроби доступу (читання, записування, виконання, зміна дозволів тощо). Налаштування аудиту здійснюється через групові політики (gpedit.msc) або через редактор локальної групової політики (secpol.msc).

Third-party File Monitoring Software – це категорія програмного забезпечення (розробленого не Microsoft, а сторонніми компаніями) для відстежування та контролю активності файлової системи на комп'ютері або сервері. Це комерційні програми (наприклад, SolarWinds Security Event Manager, Netwrix Auditor та ін.) для моніторингу файлової системи з розширеними можливостями, такими як сповіщення в реальному часі, звіти, централізоване керування тощо.

6.8.2 Моніторинг роботи з реєстром

Програми цієї групи відстежують зміни, які вносяться до системного реєстру Windows, що часто використовується для зберігання конфіденційної інформації (ключі, налаштування, ліцензії тощо). Також це може бути корисно для виявлення змін, спричинених шкідливим ПЗ, відстеження налаштувань програм тощо.

Дуже часто у складі програм для моніторингу ресурсів операційної системи є можливість фільтрувати події реєстру за операціями, шляхом до ключа реєстру, процесом, результатом операції та іншими параметрами. Це такі утиліти, як *ProcMon, Sysmon та інші.*

Regshot – безкоштовна утиліта з відкритим кодом, яка дозволяє робити «знімки» реєстру в різні моменти часу та порівнювати їх для виявлення змін. Вона робить два знімки реєстру і показує всі додані, видалені та змінені ключі та значення.

Registry Auditing (вбудована функція Windows). Подібно до аудиту файлової системи, Windows дозволяє налаштувати аудит певних ключів реєстру для відстеження спроб доступу та змін. Налаштування здійснюється через групові політики або редактор реєстру (regedit.exe). Журнали аудиту зберігаються в журналі безпеки Windows.

Third-party Registry Monitoring Software. Існують комерційні рішення, які пропонують розширені можливості моніторингу реєстру, включаючи сповіщення, звіти та інтегрування з іншими системами безпеки.

6.8.3 Моніторинг мережевої активності

Ці програми відстежують мережеві з'єднання, трафік, використовувані протоколи, DNS-запити тощо. Це корисно для виявлення шкідливої активності, аналізу мережевих проблем, моніторингу використання мережі програмами тощо.

Згадані вище програми *ProcMon* і *Sysmon* також фіксують мережеві події, такі як TCP/UDP-з'єднання, дозволяють фільтрувати за процесом, адресою, портом та іншими параметрами.

Wireshark – безкоштовний та потужний аналізатор мережевих протоколів. Він дозволяє перехоплювати та детально аналізувати мережевий трафік у реальному часі. Wireshark підтримує безліч протоколів та має потужні фільтри для аналізу.

TCPView – безкоштовна утиліта від Microsoft Sysinternals, яка відображає всі активні TCP та UDP-з'єднання на досліджуваному комп'ютері, а також процеси, які їх встановили.

Resource Monitor (resmon.exe) – вбудована у Windows утиліта, яка відображає використання ресурсів системи, включно й мережу, дозволяє бачити активні мережеві з'єднання, процеси, що використовують мережу, обсяг переданих даних тощо.

Third-party Network Monitoring Software. Існують численні комерційні та безкоштовні програми для моніторингу мережі з розширеними можливостями, такими як аналіз трафіка в реальному часі, виявлення аномалій, звіти, візуалізація мережі тощо. Наприклад, це програми SolarWinds Network Performance Monitor, PRTG Network Monitor, GlassWire та інші.

Вибір програми залежить від ваших конкретних потреб та рівня технічної підготовки.

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

7 ЗАХИСТ ВІД НЕСАКЦІОНОВАНОГО ДОСЛІДЖЕННЯ ПРОГРАМ

7.1 Необхідність і доцільність захисту від дослідження

Захист ПЗ від несанкціонованого дослідження чи зворотного інжинірингу є важливою складовою у підтримці безпеки, конфіденційності та забезпеченні прав власності на програмні продукти.

Необхідність захисту може бути обґрунтована такими факторами:

- *захист інтелектуальної власності*, оскільки програмне забезпечення, особливо комерційне, є результатом значних інтелектуальних і фінансових вкладень, і якщо код доступний для зворотного інжинірингу, це може призвести до його копіювання або несанкціонованого використання конкурентами. Тому необхідно мінімізувати ризик крадіжки технологій або бізнес-інформації за допомогою захисту, наприклад, через обфускування або шифрування коду;
- *захист від уразливостей* також є причиною запобігати зворотному інжинірингу, оскільки, якщо зловмисники мають доступ до коду, вони можуть знайти помилки, уразливості або недосконалості в реалізації безпеки. Отже, захист ПЗ від дослідження допомагає зменшити можливість таких атак різного виду;
- *забезпечення конфіденційності* (алгоритмів, протоколів або даних, що не мають ставати відомими стороннім особам) також є критично важливим;
- *підвищення довіри користувачів*, оскільки належний рівень захисту від несанкціонованого доступу до коду та серйозний підхід розробників до безпеки та конфіденційності, може забезпечити схильність замовників ПЗ;
- *забезпечення стабільності і сумісності* ПЗ дозволяє утримувати контроль над програмою та гарантувати її належну роботу, а тому дослідження коду не має призвести до змін або модифікацій;
- *юридичні та регуляторні вимоги* у багатьох країнах передбачають, щоб компанії захищали свої програми від зворотного інжинірингу, особливо коли мова йде про програмне забезпечення, яке працює з чутливими даними (фінанси, алгоритми, плани тощо).

Щодо ***доцільності*** захисту, то, з одного боку, застосування засобів захисту програми (як-от обфускування коду, криптографія, ліцензування) є важливим кроком у забезпеченні безпеки та захисту бізнес-інтересів. Однак, з іншого боку, надмірний захист може ускладнити розробку, тестування та оновлення ПЗ, створюючи додаткові витрати для розробників.

Отже, захист має бути збалансованим, інакше надмірна складність чи обмеження можуть негативно вплинути на зручність використання і розвитку програм.

7.2 Класифікація методів захисту від несанкціонованого дослідження

Захист програмного забезпечення від несанкціонованого дослідження (аналізу, зворотної інженерії, модифікації) містить в собі низку технічних та організаційних методів, які розробники використовують для захисту своїх програм від несанкціонованого дослідження та аналізу.

Вбудовування захисних механізмів у захищувані програми може бути виконано різними способами:

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

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

До процесу і результату вбудовування захисних механізмів можна висунути такі ***вимоги***:

- висока трудомісткість виявлення захисного фрагмента під час статичного дослідження (особливо актуальна в процесі вбудовування в початковий код програмного продукту);
- висока трудомісткість виявлення захисного фрагмента під час динамічного дослідження (налагоджування і трасування за зовнішніми подіями);
- висока трудомісткість обходу або редагування захисного фрагмента.

Загальні підходи захисту програм від несанкціонованого дослідження наведено у таблиці 7.1.

Таблиця 7.1 – Основні підходи до захисту від несанкціонованого дослідження

Метод захисту	**Різновиди, деталізація, приклади**	**Механізм реалізації**
Використання спеціалізованих інструментів	– Themida, VMProtect, Enigma Protector – захист виконуваних файлів (комерційні); – ProGuard, DexGuard – захист для Android; – Obfuscator-LLVM (для C/C++) та інші.	Навісний
Ліцензування та активація	Механізми перевірки ліцензії: – прив'язка до апаратного забезпечення (HWID); – онлайн-перевірка ключа або токена; – тимчасові ключі (trial) та інші.	Навісний (програмно-апаратний)
Пакування (packing)	Упаковування виконуваного файлу в оболонку, яка розпаковується лише під час виконання. Використовується для ускладнення статичного аналізу. Часто містить шифрування чи стиск коду.	Навісний
Архітектурні рішення	– рознесення критичної логіки на сервер: перенесення важливих обчислень та обробки даних на сервер (у клієнтської програми обмежений доступ); – використання захищених анклавів (Secure Enclaves): запуск чутливого коду в ізольованому апаратно-програмному середовищі, до якого навіть операційна система не має прямого доступу.	Вбудований Комбінований
Шифрування коду	– шифрування рядків та констант; – шифрування окремих частин коду; – використання криптографічних геш-функцій.	Навісний Комбінований
Обфускування коду	– перейменування ідентифікаторів; – ускладнення потоку керування; – перемішування порядку виконання; – вставка «мертвого» коду; – додавання фрагментів «сміттєвого коду»; – зміна структури коду; – використання непрозорих предикатів і т. ін.	Вбудований
Техніки протидії налагодженню та декомпіляції	– виявлення налагоджувача (перевірка наявності процесів типу OllyDbg, x64dbg, IDA); – анти-дампінг; – використання інструкцій, що викликають помилки під час налагодження; – визначення часу затримки між командами (затримки можуть свідчити про налагодження); – віртуалізація (використання віртуальних машин, що виконують код у захищеному середовищі); – виявлення переривань або змін у потоках тощо.	Вбудований

Метод захисту	Різновиди, деталізація, приклади	Механізм реалізації
Захист цілісності програми	– контрольні суми та підписи коду; – водяні знаки (watermarking); – перевірка, чи не були змін критичних частин програми (гешування/підпис файлів, самоперевірка коду під час виконання) тощо.	Вбудований
Захист від зворотної інженерії	– використання нативного коду (C/C++) для чутливих ділянок у програмах на Java/.NET; – винесення логіки на сервер (SaaS) для мінімізації клієнтської реалізації; – вбудовані механізми захисту у фреймворках (наприклад, SecureString, Windows DPAPI); – інші антиналагоджувальні методи.	Вбудований Комбінова–ний

7.3 Техніки протидії статичному аналізу та декомпілюванню

Техніки протидії декомпіляції та статичному аналізу (Anti-Decompilation & Static Analysis) спрямовані на ускладнення автоматичного перетворення бінарного коду назад у вихідний код (декомпіляція) або ручного аналізу (дизасемблювання).

Усі методи протидії несанкціонованому статичному дослідженню можна поділити на декілька основних груп.

7.3.1 Обфускування

Обфускування (Obfuscation) – це одна з основних технік захисту програмних продуктів від несанкціонованого дослідження (як статичного, так і динамічного). Детальніше про цей метод і способи його реалізації йтиметься далі. Суть полягає у тому, що у разі його застосування змінюється структура та вигляд коду, але функціональність його зберігається.

Кожна мова програмування, використана під час створенні програмного застосунку, має свої власні особливості і притаманні лише їй семантичні і синтаксичні конструкції. Тому інструменти для обфускування розробляються, зазвичай, під певну мову (або групу мов) програмування. Але майже всі вони, переважно, мають такі функціональні можливості:

- Control Flow Flattening – сплющення потоку управління програм. Тобто, проста послідовність інструкцій програми перетворюється на складний, заплутаний граф з безліччю непотрібних переходів;
- Bogus Control Flow – хибний потік управління – це додавання штучних умовних переходів, які завжди ведуть до однієї й тієї ж гілки, але декомпілятор може сприйняти їх як значущі;

– Instruction Substitution – підстановка інструкцій, яка полягає у заміні простих інструкцій на більш складні і розгорнуті, але функціонально еквівалентні послідовності, використовуючи у цьому випадку певні алебраїчні перетворення;
– Constant Blinding – засліплення констант, коли замість прямого використання констант, вони обчислюються динамічно під час виконання або зберігаються у коді програми у зашифрованому вигляді, а розшифровуються лише під час роботи програми;
– String Encoding – кодування рядків, коли рядкові літерали зашифровуються або закодовуються у бінарному файлі та розшифровуються лише під час виконання;
– поліморфізм та метаморфізм, суть якого в тому, що код змінюється кожного разу, коли програма компілюється або навіть коли запускається. А це, звичайно, ускладнює сигнатурний аналіз.

7.3.2 Віртуалізація коду та нестандартні архітектури (Code Virtualization)

Віртуалізація коду є однією з передових технік у сфері захисту ПЗ від реверс-інжинірингу та несанкціонованого дослідження. Її основна ідея полягає у перетворенні нативного коду програми на інструкції власної, кастомної спеціальної віртуальної машини (ВМ), яка потім інтерпретує ці інструкції під час виконання. Це створює додатковий рівень абстракції, значно ускладнюючи аналіз і розуміння логіки програми зловмисниками, робить бінарний код незрозумілим для стандартних декомпіляторів, оскільки вони не знають архітектури цієї віртуальної машини.

Віртуалізація коду може бути виконана у декілька етапів.

І етап – компілювання програми у віртуальний байт-код. Тобто, замість того, щоб компілювати вихідний код безпосередньо в машинний код, віртуалізатор перетворює його на унікальний байт-код, зрозумілий лише спеціально розробленій ВМ. Цей байт-код зазвичай сильно відрізняється від стандартних наборів інструкцій (наприклад, x86 або ARM).

ІІ етап – генерування індивідуальної віртуальної машини. На цьому етапі для кожної захищеної програми (або навіть для окремих її частин) може бути згенерована унікальна ВМ. Розробники можуть створювати унікальні набори інструкцій та архітектури ВМ, роблячи кожну реалізацію віртуалізації унікальною та менш вразливою до загальних атак. Це означає, що зловмисники не зможуть просто використати стандартні інструменти для декомпілювання, оскільки вони не знають архітектури цієї конкретної віртуальної машини.

ІІІ етап – динамічна інтерпретація. Під час виконання програми ВМ інтерпретує віртуальний байт-код, перетворюючи його на нативні інструкції «на льоту». Цей процес може містити динамічне

переупорядкування інструкцій, обфускування потоку виконання та інші техніки, що ускладнюють зловмисникам відстеження.

Отже, аналіз віртуалізованого коду потребує не тільки розуміння логіки програми, а й реверс-інжинірингу самої віртуальної машини, що є надзвичайно складним завданням. Оскільки основна логіка програми прихована за інтерпретатором ВМ, більшість статичних аналізаторів не можуть ефективно витягти значущу інформацію. Стандартні декомпілятори та налагоджувачі часто безсилі проти віртуалізованого коду, оскільки вони не розраховані на роботу з кастомними віртуальними архітектурами.

У сучасному світі, де загрози безпеці ПЗ постійно зростають, віртуалізація коду є потужним інструментом у арсеналі розробників, які прагнуть захистити свої інтелектуальні надбання та забезпечити цілісність програм.

Разом з тим, застосування техніки віртуалізації коду має свої виклики та обмеження:

- по-перше, зниження продуктивності, оскільки процес інтерпретування коду віртуальною машиною зазвичай повільніший, ніж пряме виконання нативного коду, а це часто може бути критичним для високопродуктивних програм;
- по-друге, збільшення розміру файлу, оскільки віртуалізований код та сама ВМ додають до розміру виконуваного файлу;
- по-третє, звичайно, складність реалізації, оскільки ефективна віртуалізація коду потребує глибоких знань в архітектурі комп'ютерів та безпеці, а також значних зусиль для її впровадження.

Варто зазначити, що хоча віртуалізація значно ускладнює реверс-інжиніринг, вона не робить його повністю неможливим. Досвідчені зловмисники можуть витратити значний час та ресурси на девіртуалізацію.

Обхід такого захисту – це одна з найскладніших задач у реверс-інжинірингу, що називається *девіртуалізацією*. Її мета – перетворити кастомний байт-код назад у зрозумілий x86-код. Спрощено процес девіртуалізації можна подати так:

Локалізувати ВМ, тобто, знайти в коді програми сам інтерпретатор та його обробники (код, що виконує інструкції байт-коду).

1. Здійснити аналіз обробників, щоб зрозуміти, яку саме дію він виконує. Наприклад, з'ясувати, що байт 0x15 приводить до виклику блока коду, який виконує додавання.
2. Здійснити реконструкцію, тобто, написати скрипти або використати спеціальні інструменти для трасування виконання байт-коду. Трасувальник крок за кроком виконує віртуальну програму і записує, які семантичні дії були виконані.
3. На основі зібраної інформації створити транслятор, який переводить байт-код у еквівалентний йому код мовою високого рівня або в асемблер x86.

Це надзвичайно трудомісткий процес, який потребує глибоких знань та значних витрат часу. Саме тому ВМ-захист вважається вершиною обфускування коду у разі протидії статичному аналізу.

7.3.3 Пакувальники та протектори (Packers and Protectors)

Для захисту ПЗ від НСД часто використовуються пакувальники та протектори. Хоча ці терміни іноді вживаються як синоніми, вони мають певні відмінності у своїх цілях та реалізації.

Пакувальники – це утиліти, основною метою яких є зменшення розміру виконуваного файлу програми шляхом його ущільнення та/або зашифрування. Коли запакована програма запускається, невеликий код-завантажувач (stub), доданий пакувальником під час пакування, спочатку розпаковує (і, якщо потрібно, розшифровує) основний код програми в пам'ять, а потім передає йому управління для виконання безпосередньо програми. Тобто, основними функціями пакувальників є:

- зменшення розміру файлу, що особливо актуально в умовах повільного Інтернету та обмеженого дискового простору;
- базове приховування коду, внаслідок якого ущільнений або зашифрований код набагато складніше аналізувати статичними методами (наприклад, дизасемблером), оскільки оригінальна структура програми невидима до моменту розпакування в пам'яті;
- ускладнення сигнатурного аналізу, оскільки антивіруси, які у своїй роботі використовують сигнатури для виявлення шкідливого ПЗ, можуть не розпізнати запаковану програму, якщо її сигнатура змінилася після пакування.

Прикладом програм-пакувальників може слугувати програма UPX (Ultimate Packer for eXecutables) – безкоштовний, портативний, розширюваний, високопродуктивний пакувальник для різних форматів виконуваних файлів. UPX є програмою з відкритим вихідним кодом, що розповсюджується під ліцензією GNU General Public License. Основне призначення – зменшення розміру програмних файлів без втрати їхньої функціональності.

Зазвичай, пакувальники надають лише базовий рівень захисту. Зламник з досвідом може відносно легко розпакувати програму, використовуючи спеціалізовані інструменти або відстежуючи процес розпакування в пам'яті за допомогою налагоджувача (табл. 7.2).

Протектори – це більш складні та досить потужні системи, призначені для активного захисту ПЗ від реверс-інжинірингу, налагодження, модифікації та нелегального копіювання. Вони використовують широкий спектр технік, щоб максимально ускладнити аналіз програми (табл. 7.2). Як правило, програми-протектори пропонують користувачам такі функції:

- анти-налагоджувальні техніки (Anti-Debugging);

- анти-дизасемблювання (Anti-Disassembly), тобто, використання технік, що заплутують логіку дизасемблерів, шифрування частин коду;
- обфускування коду (Code Obfuscation);
- віртуалізація коду (Code Virtualization);
- шифрування важливих частин коду, секцій або даних, які розшифровуються лише під час виконання за певних умов;
- перевірка цілісності коду (Integrity Checks);
- захист від «дампів» пам'яті (Anti-Memory Dumping);
- різноманітні прив'язки до «заліза» (Hardware Locking) та управління ліцензіями.

Таким чином, протектори забезпечують значно вищий рівень захисту порівняно з пакувальниками. Обхід захисту, реалізованого сучасними протекторами, потребує глибоких знань, спеціалізованих інструментів та значних часових витрат. Прикладами відомих протекторів є такі: VMProtect, Themida, Enigma Protector, Denuvo та інші.

Таблиця 7.2 – Порівняльна таблиця пакувальників і протекторів

Характеристика	Пакувальник	Протектор
Основна мета	Зменшення розміру, базове приховування коду	Активний захист від реверс-інжинірингу та несанкціонованого використання
Складність	Відносно прості	Дуже складні, багатошарові системи
Використовувані техніки	Ущільнення, просте шифрування	Антиналагодження, обфускування, віртуалізація, прив’язки, шифрування, анти-дампінг тощо.
Рівень захисту	Низький або середній	Високий або дуже високий
Вплив на розмір	Зазвичай зменшує	Може збільшувати розмір через додавання захисного коду
Вплив на швидкодію	Незначний (час на розпакування)	Може суттєво впливати через складні перевірки та обфускування

7.3.4 Зміни в таблиці імпорту (Import Table Obfuscation):

Щоб зрозуміти цей метод, спершу розглянемо, як працює програма в звичайних умовах.

Кожен виконуваний файл (.exe) у Windows містить спеціальну структуру, яка називається Таблицею Адрес Імпорту (Import Address Table – IAT). Ця таблиця є, по суті, «списком замовлень» програми до операційної системи, тобто, список усіх зовнішніх функцій, які потрібні програмі для роботи, та перелік тих DLL-бібліотек, в яких ці функції знаходяться. Коли програма запускається на виконання, завантажувач Windows проходить по IAT, знаходить вказані DLL, завантажує їх у

пам'ять і записує реальні адреси потрібних функцій у таблицю. Після цього програма може викликати ці функції за їхніми адресами.

Під час статичного аналізу дослідник, антивірус або зламник може відкрити файл і, навіть не запускаючи його, переглянути IAT. Список імпортованих функцій (CreateFile, WriteFile, Socket, URLDownloadToFile та ін.) миттєво дає зрозуміти, що програма може робити: працювати з файлами, виходити в інтернет тощо.

Суть методу зміни або обфускування таблиці імпорту полягає у навмисному приховуванні, спотворенні або видаленні інформації про функції, які програма викликає із зовнішніх системних DLL-бібліотек, що ускладнює розуміння того, які системні виклики робить програма. Це популярна техніка захисту програмного забезпечення (і, водночас, поширений прийом у шкідливих програмах) від дослідження, зворотного інжинірингу (реверс-інжинірингу) та статичного аналізу.

7.3.5 Анти-дампінг (Anti-Dumping)

Якщо головна мета зміни IAT – змусити дослідника запустити програму для аналізу, то мета анти-дампінгу – не дати йому зробити якісний «знімок» (дамп) процесу з пам'яті після запуску.

Якщо програма розпаковується в пам'яті під час виконання, зловмисник може спробувати зробити дамп, щоб отримати доступ до розпакованого коду. Головна мета – зруйнувати ключовий інструмент реверс-інженера. Без роботоздатного дампа аналізувати складний, упакований код стає вкрай важко. Тобто, анти-дампінг – це наступний логічний крок у захисті програми після обфускування таблиці імпорту.

Суть анти-дампінгу – це набір технік, які програма використовує для того, щоб:

- виявити спробу створення свого дампа;
- перешкодити цьому процесу або спотворити результат так, щоб отриманий дамп був пошкодженим, неповним або непрацездатним.

Анти-дампінг може містити перевірку цілісності пам'яті, використання нестандартних секцій пам'яті тощо. Детальніше про техніки запобігання дампінгу процесів та програм йтиметься далі.

7.3.6 Водяні знаки (Watermarking)

Суть цього методу полягає у вбудовуванні в бінарний файл програми унікальних, невидимих ідентифікаторів, які можуть допомогти відстежити джерело витоку або ідентифікувати копії.

На відміну від видимих водяних знаків на зображеннях чи відео, програмні водяні знаки є таємними та інтегрованими в саму логіку або дані програми. Їхня головна мета – не перешкодити запуску програми, а

залишити стійкий слід, який важко виявити та видалити без пошкодження функціональності.

Найефективніший захист досягається шляхом комбінування кількох технік. Наприклад, обфускування коду, його віртуалізація, а потім пакування протекторами. Це створить багатошаровий захист, який потребує від зловмисника значних зусиль, знань і часу.

7.4 Техніки протидії динамічному дослідженню

Динамічне дослідження програм, що передбачає аналіз їхньої поведінки під час виконання, є ключовим методом для виявлення уразливостей, розуміння логіки роботи шкідливого програмного забезпечення та забезпечення якості коду. Однак техніки протидії динамічному аналізу активно застосовуються як розробниками для захисту програм від піратства, так і авторами шкідливого ПЗ. Ці методи спрямовані на ускладнення або повне унеможливлення роботи аналітичних інструментів, таких як налагоджувачі (debuggers), емулятори та пісочниці (sandboxes)

Техніки протидії несанкціонованому налагодженню (Anti-Debugging) спрямовані на виявлення факту роботи програми під контролем налагоджувача (debugger) і зміну її поведінки або повний збій у разі виявлення такого налагоджувача. Це ускладнює покроковий аналіз коду, а отже, запобігає динамічному дослідженню програм.

Захиститися від несанкціонованого дослідження програм під налагоджувачем можна декількома шляхами:

- тим або іншим методом *виявити налагоджувач* і передати керування на деяку гілку реакції на налагоджувач;
- *виявлення віртуалізованого середовища*, якщо налагоджувач працює у деякій віртуальній машині;
- *«забруднення» програми* фрагментами коду, які нормально виконуються без налагоджувача, але під налагоджувачем призводять до аварійного завершення, зависання комп'ютера або перекручування ходу виконання програми.

Розглянемо, яким чином можна реалізувати такі техніки.

7.4.1 Виявлення присутності налагоджувача

Це найпоширеніша категорія методів, спрямованих на пряме виявлення присутності налагоджувача в оперативній пам’яті.

Прямі виклики API або перевірка прапорів. Кожна операційна система має в наявності певні функції, які вказують, чи запущений процес під налагоджувачем. Наприклад, в Windows – це функція IsDebuggerPresent(), яку мовою C++ можна використати так:

```
if (IsDebuggerPresent()) {
    std::cout << "Увага: Програма запущена під дебагером!" << std::endl;
    // Тут можна додати специфічну логіку для випадку, коли дебагер присутній
}
else {  std::cout << "Програма запущена без дебагера." << std::endl }
```

В Unix-подібних системах це прапор PTRACE_TRACEME:

```
int is_debugger_present() {
    errno = 0;
    // Спроба дозволити налагодження самого себе
    if (ptrace(PTRACE_TRACEME, 0, NULL, NULL) == -1) {
        if (errno == EPERM) {
            return 1;  // Виявлено: вже налагоджується
    }    }
    return 0;  // Не налагоджується
}
```

Цей метод легко обійти досвідченим реверсерами – наприклад, через підміни функцій, патчинг, LD_PRELOAD тощо. Але для базового рівня антидебагу – він ефективний.

Перевірка батьківського процесу полягає у тому, що програма може перевіряти, чи її батьківський процес є якимось відомим налагоджувачем, який у цей час аналізує її.

Наприклад, в Linux кожен процес може дізнатися PID свого батька (PPID), а далі прочитати інформацію про нього – зокрема, його ім’я (ім’я процесу), використавши для цього виклик команд:

`/proc/self/status` або `/proc/self/stat.`

У Windows також можна перевірити, чи є батьківський процес відомим налагоджувачем, хоча методи відрізняються від Linux. Для цього можна використовувати функції Windows API для отримання інформації про процеси, зокрема через функції OpenProcess(), GetParentProcessId(), і QueryFullProcessImageName(), щоб дізнатися ім'я батьківського процесу та порівняти його з відомими дебагерами, такими як gdb, ollydbg і т. д.

Наприклад, для отримання PID батьківського процесу можна використати таку функцію (мова C++):

```
DWORD GetParentProcessId(DWORD pid) {
    DWORD parent_pid = 0;
    HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, pid);
    if (hProcess) {
        PROCESS_BASIC_INFORMATION pbi;
        ZeroMemory(&pbi, sizeof(PROCESS_BASIC_INFORMATION));
        if (NT_SUCCESS(NtQueryInformationProcess(hProcess,
     ProcessBasicInformation, &pbi, sizeof(pbi), NULL))) {
              parent_pid = pbi.InheritedFromUniqueProcessId;
        }
        CloseHandle(hProcess);
    }
    return parent_pid;
}
```

Для перевірки імені батьківського процесу можна застосувати функцію:

```
int IsParentDebugger(DWORD pid) {
    DWORD parent_pid = GetParentProcessId(pid);
    if (parent_pid == 0) { return 0;} // Не вдалося отримати PID батька
    HANDLE hParentProcess = OpenProcess(PROCESS_QUERY_INFORMATION |
                                PROCESS_VM_READ, FALSE, parent_pid);
    if (hParentProcess==NULL){return 0;}// Не вдалося відкрити батьківський процес
    CHAR procName[MAX_PATH_LEN];
    DWORD NameLen = MAX_PATH_LEN;
    if (QueryFullProcessImageNameA(hParentProcess, 0, procName, &NameLen)) {
        // Порівнюємо ім'я процесу батька з відомими дебагерами
        if (strstr(procName, "gdb") || strstr(procName, "ollydbg")  {
            CloseHandle(hParentProcess);
            return 1; // Виявлено дебагер
        }
    }
  CloseHandle(hParentProcess);
        return 0; // Не виявлено дебагера
}
```

Аналіз структур процесу. Суть цього методу полягає у тому, що програми мають можливість перевіряти внутрішні структури даних процесу, які змінюються під час підключення налагоджувача. Наприклад, в операційних системах сімейства Windows NT є фундаментальна структура даних PEB (Process Environment Block), яка містить детальну інформацію про те, що розташовується в адресному просторі кожного процесу і яка доступна в режимі користувача (user-mode). В структурі PEB є поле BeingDebugged, яке встановлюється в 1, коли процес працює під налагоджувачем. Саме його можна перевіряти, наприклад, так:

```
PEB* peb = (PEB*)__readfsdword(PEB_OFFSET);
if (peb->BeingDebugged)
  std::cout << "Процес налагоджується (BeingDebugged=TRUE)"<<std::endl;
else
  std::cout<<"Процес НЕ налагоджується (BeingDebugged=FALSE)"<<std::endl;
```

Використання специфічних інструкцій. Цей метод базується на тому, що деякі інструкції процесора (наприклад, INT 1 – трасувальне переривання або переривання покрокової роботи, INT 3 – переривання контрольної точки, прапор трасування TF) використовуються налагоджувачами. Програма може перевіряти власний код на наявність таких інструкцій або їхніх модифікацій (наприклад, командою SCASB).

У програмах на C++ перевірити, чи є інструкція INT 3 (її код 0xCC) в певній області пам'яті (наприклад, в адресі початку функції), можна так:

```
bool containsInt3(BYTE * address, SIZE_T length) {
      for (SIZE_T i = 0; i < length; ++i) {
            if (address[i] == 0xCC) { return true; }
      }
      return false;
}
...
BYTE* funcAddress = (BYTE*)&<тут назва функції>;
```

```
if (containsInt3(funcAddress, 32))
        std::cout << "Знайдено INT 3 (0xCC) у функції!" << std::endl;
```

Перевірка регістрів налагоджувача. Деякі налагоджувачі використовують специфічні регістри CPU (наприклад, регістри DR0-DR7 для апаратних точок зупинки), наявність яких може бути перевірена у захищуваних програмах. Наприклад, у мові C++ це можна робити, використавши функцію GetThreadContext():

```
bool checkDebugRegisters() {
       CONTEXT ctx = {};
       ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
       HANDLE hThread = GetCurrentThread();
        if (GetThreadContext(hThread, &ctx))
             if (ctx.Dr0 || ctx.Dr1 || ctx.Dr2 || ctx.Dr3)
                   return true; // Дебагер, якщо будь-який DR встановлений
        return false;
}
...
if (checkDebugRegisters())
     std::cout << "Виявлено активні debug-регістри." << std::endl;
else
     std::cout << "Debug-регістри чисті." << std::endl;
```

Перевірка часу виконання інструкцій. Інструкції, які виконуються дуже швидко в нормальному режимі, можуть виконуватися повільніше, якщо встановлено точки зупинки або якщо налагоджувач покроково виконує код. Виконання коду під налагоджувачем відбувається значно повільніше через зупинки та аналіз стану. Програма може вимірювати час виконання критичної ділянки коду за допомогою певних інструкцій, і, якщо він перевищує очікуване значення, це може свідчити про наявність налагоджувача.

Наприклад, у програмах на C++ для цього може бути використана функція QueryPerformanceCounter():

```
bool detectDebuggerByTiming() {
     LARGE_INTEGER start, end, freq;
     QueryPerformanceFrequency(&freq);
     QueryPerformanceCounter(&start);
     // Швидка інструкція (можна зробити цикл або порожню функцію)
     for (volatile int i = 0; i < 1000; ++i);
     QueryPerformanceCounter(&end);
     LONGLONG elapsed = end.QuadPart - start.QuadPart;
     double elapsedTimeMs = (double)elapsed * 1000.0 / freq.QuadPart;
     std::cout << "Час виконання: " << elapsedTimeMs << " ms\n";
     // Якщо надто довго – можливо, дебагер
     return elapsedTimeMs > 5.0; // поріг залежить від системи
    }
```

або функція GetTickCount() (менш точно, але простіше):

```
DWORD start = GetTickCount();
    // якась швидка операція
    Sleep(10); // або цикл
    DWORD end = GetTickCount();
    if ((end - start) > 20) { // підозріло довго  }
```

Перевірка цілісності коду, тобто, перевірка контрольних сум або гешів власних ділянок коду. Якщо налагоджувач модифікував код (наприклад, вставив INT 3 (0xCC), змінив умовні переходи тощо), контрольна сума зміниться. Якщо мати оригінальну контрольну суму (CRC32, MD5, SHA1) і порівняти її під час виконання, можна виявити підозрілі зміни.

Наприклад, якщо використовувати для отримання контрольної суми алгоритм CRC32 (бібліотека zlib.h), процес перевірки контрольної суми деякої функції func() спрощено можна подати так:

```
DWORD calculateFunctionCRC(BYTE* funcPtr, SIZE_T length) {
        return crc32(0, funcPtr, (uInt)length);
    }
...
    BYTE* funcStart = (BYTE*)&<тут назва функції>;
    SIZE_T length = 100; // або визначити точно (є способи дл цього)
    DWORD actualCRC = calculateFunctionCRC(funcStart, length);
    DWORD expectedCRC = 0xDEADBEEF; // <- значення, яке обчислюється заздалегідь
    std::cout << "Actual CRC32:   0x" << std::hex << actualCRC << std::endl;
    std::cout << "Expected CRC32: 0x" << std::hex << expectedCRC << std::endl;
    if (actualCRC != expectedCRC)
        std::cout << "Код змінено! Можливий дебагер або тамперінг." << std::endl;
```

Перевірка кількості ядер процесора – це непряма антидебаг-техніка, яка використовується для виявлення віртуальних машин, емуляторів, або ізольованих середовищ, які часто конфігуруються з мінімальними ресурсами, наприклад, лише з одним ядром CPU. Хоча сама по собі наявність одного ядра не означає, що працює дебагер, але це може викликати підозру, особливо в комбінації з іншими ознаками (одне ядро, багато затримок, ім'я процесора типу «Virtual» тощо). Програма може відмовитися працювати, якщо значення NumberOfProcessors ≤ 2. Це, наприклад, можна виявити, використавши функцію GetSystemInfo():

```
bool isSuspiciousCpuCoreCount() {
    SYSTEM_INFO sysInfo;
    GetSystemInfo(&sysInfo);

    std::cout << "Кількість логічних процесорів: " <<
            sysInfo.dwNumberOfProcessors << std::endl;

    return sysInfo.dwNumberOfProcessors <= 1; // Підозріло, якщо ≤ 1
}
```

Інший варіант – використати функцію GetLogicalProcessorInformation(). Це більш точна API-функція, що дає змогу перевірити фізичні ядра, логічні потоки (HyperThreading) тощо.

```
int getLogicalCoreCount() {
    DWORD len = 0;
    GetLogicalProcessorInformation(NULL, &len); // Отримати потрібний розмір буфера
    SYSTEM_LOGICAL_PROCESSOR_INFORMATION* buffer =
                    (SYSTEM_LOGICAL_PROCESSOR_INFORMATION*)malloc(len);
    if (!GetLogicalProcessorInformation(buffer, &len)) {
```

```
    free(buffer);
    return -1;
}
int count = 0;
DWORD entryCount = len / sizeof(SYSTEM_LOGICAL_PROCESSOR_INFORMATION);
for (DWORD i = 0; i < entryCount; ++i) {
    if (buffer[i].Relationship == RelationProcessorCore)
        ++count;
}
free(buffer);
return count;
}
```

7.4.2 Виявлення віртуалізованого середовища та пісочниць

Виявлення віртуалізованих середовищ та пісочниць (Anti-VM & Anti-Sandbox) – це ключова тема в галузі кібербезпеки, оскільки шкідливі програми використовують ці техніки, щоб уникнути виявлення та аналізу, а легітимне ПЗ (наприклад, ігри або програми із захистом від копіювання) – для захисту від злому та шахрайства. У цьому випадку намагаються визначити, чи не працюють вони у віртуальній машині (VMWare, VirtualBox) або пісочниці. Ці техніки можна поділити на декілька категорій.

1. Перевірка артефактів. Програма може шукати специфічні файли, драйвери, процеси, ключі реєстру або ідентифікатори обладнання, які є унікальними для таких конкретних віртуальних машин, як VirtualBox, VMware, Hyper-V, QEMU, або пісочниць:

а) Пошук *специфічних файлів і драйверів*, що належать до інструментів віртуалізації:
- VMware – vmtoolsd.exe, VMwareUser.exe, vmmouse.sys;
- VirtualBox – VBoxService.exe, VBoxGuestAdditions.exe, VBoxMouse.sys;
- QEMU – наявність драйверів qemupciserial.sys.

б) Перевірка наявності *гілок і ключів реєстру*, створених під час встановлення віртуальних машин:
- HKEY_LOCAL_MACHINE\SOFTWARE\VMware, Inc.\VMware Tools
- HKEY_LOCAL_MACHINE\SOFTWARE\Oracle\VirtualBox Guest Additions
- HKEY_LOCAL_MACHINE\HARDWARE\Description\System\SystemBiosVersion (часто містить рядок "VBOX", "VMWARE").

в) Перевірка *MAC-адреси*. Віртуальні машини часто використовують MAC-адреси з певних діапазонів, що належать виробнику певної віртуальної машини:
- для VMware – 00:05:69, 00:0C:29, 00:50:56;
- для VirtualBox – 08:00:27;
- для Hyper-V – 00:03:FF.

г) Пошук *ідентифікаторів пристроїв* (Hardware IDs), оскільки обладнання у віртуальних машинах має специфічні назви:

– диски можуть містити підрядки VMware, VBOX, QEMU;
– відеоадаптери – VirtualBox Graphics Adapter, VMware SVGA II.

д) Пошук *імен процесів та вікон*, пов'язаних з інструментами аналізу (наприклад, wireshark.exe, procexp.exe, ollydbg.exe).

2. Техніки, основані на інструкціях процесора, використовують відмінності у роботі апаратного забезпечення та емуляції інструкцій.

а) *інструкція CPUID* – найвідоміший спосіб. Процесори надають інформацію про себе через інструкцію CPUID. Гіпервізори (програми, що керують віртуальними машинами) часто залишають свої «підписи» у відповіді на певні запити CPUID. Наприклад, запит зі значення регістру EAX, що дорівнює 0x40000000, може повернути назву виробника гіпервізора. Наприклад, віртуальна машина VMware поверне «VMwareVMware», а KVM – «KVMKVMKVM»;

б) *«червона пігулка»* (Red Pill) **–** класична техніка, що використовує інструкцію SIDT (Store Interrupt Descriptor Table Register). Адреса таблиці переривань у гостьовій ОС зазвичай знаходиться у вищих адресах пам'яті, ніж у хостовій ОС;

в) *інші системні інструкції*, оскільки значення, що повертаються цими інструкціями, можуть відрізнятися у ВМ:

– SLDT (Store Local Descriptor Table Register),
– SGDT (Store Global Descriptor Table Register),
– STR (Store Task Register).

3. Перевірки конфігурації системи:

– *розмір диска та RAM*, оскільки занадто малий розмір диска (менший за 80 ГБ) або обсяг оперативної пам'яті (менший за 4 ГБ) може бути ознакою ВМ;
– *роздільна здатність екрана*, оскільки типові або низькі роздільні здатності екрана (800×600, 1024×768) можуть свідчити про ВМ;
– *кількість останніх документів*. Якщо список нещодавно відкритих файлів порожній, це може бути ознакою «чистої» системи пісочниці.

7.4.3 Введення в оману налагоджувача і ускладнення аналізу коду

Самомодифікування коду. Це потужна техніка, що використовується розробниками ПЗ для захисту своїх творінь від несанкціонованого аналізу та зворотної розробки (реверс-інжинірингу). Налагоджувачі часто не можуть коректно обробляти код, який динамічно змінює сам себе під час виконання. Ця техніка полягає у здатності програми змінювати свій власний код під час виконання, що робить її

вкрай складною для розуміння та дослідження. Основна ідея самомодифікації полягає у створенні «рухомої мішені» для аналітиків. Замість статичного та передбачуваного коду, дослідник стикається з програмою, яка постійно змінює свою структуру та логіку. Це досягається за допомогою різноманітних технік, що працюють на різних рівнях. Для досягнення мети захисту від дослідження, самомодифікування коду використовує низку витончених методів, серед яких такі, як:

- обфускування, яке, звичайно, захищає не тільки від статичного дослідження, але й від динамічного (перейменування змінних та функцій, вставлення «мертвого» коду, зміна потоку управління тощо);
- генерування коду в реальному часі, коли програма генерує новий виконуваний код безпосередньо під час своєї роботи, базуючись на певних умовах або даних. Цей новостворений код виконується, а потім може бути відкинутий, що робить поведінку програми непередбачуваною для аналітика, який не може заздалегідь знати, який код буде згенеровано.

Вбудовування «мертвих точок зупинки». Вставлення інструкцій, які імітують точку зупинки (наприклад, INT 3 на x86/x64), але програма може обробляти їх сама, збиваючи налагоджувач. Термін «мертва точка зупинки» є дещо метафоричним. Він означає не просто «мертвий» (зайвий) код, а фрагменти коду, які виглядають як важливі логічні вузли (наприклад, перевірки ліцензії, виклики важливих функцій), але насправді є пастками для аналітика.

Для прикладу можна уявити собі програму, яка перевіряє ліцензійний ключ. Справжня перевірка може бути простою та швидкою. Однак, розробник додає великий блок коду, який виглядає як складна криптографічна перевірка. Цей блок містить цикли, умовні переходи та маніпуляції з даними. Аналітик, натрапивши на цей блок, припустить, що саме тут відбувається перевірка, і почне його детально вивчати. Насправді ж, цей блок є «мертвою точкою зупинки» – він ніколи не виконується до кінця або його результат ні на що не впливає. Справжня ж перевірка захована в іншому, непомітному місці.

Нескінченні цикли або збої. Це більш агресивні та руйнівні методи захисту від динамічного аналізу, що діють за принципом «зупинити аналітика за будь-яку ціну». Суть такого захисту полягає у тому, щоб експлуатувати фундаментальну слабкість динамічного аналізу: необхідність виконувати код для його дослідження. Якщо виконання коду заходить у глухий кут (цикл) або раптово припиняється (збій), процес аналізу також зупиняється. Нескінченний цикл (Infinite Loops) – це пастка, яка заморожує програму, а разом з нею і сесію налагодження.

Коли аналітик (зламник) крок за кроком виконує програму під налагоджувачем (наприклад, командами «step over» або «step into»), він очікує,

що кожна функція чи блок коду завершить свою роботу. Нескінченний цикл ніколи не завершується, тому налагоджувач «зависає» на цій ділянці, не дозволяючи аналітику просунутися далі до важливих частин логіки програми. Цикл починає інтенсивно споживати ресурси центрального процесора, що може сповільнити всю систему, включно й сам інструмент для налагодження. Щоб вийти з такого циклу, аналітик змушений зупинити виконання вручну, проаналізувати код циклу, щоб зрозуміти умову виходу (якої немає або яка замаскована), вручну змінити стан програми (наприклад, змінити значення лічильника в пам'яті або покажчик інструкції EIP/RIP), щоб «перестрибнути» через цикл. Це потребує часу, зусиль та глибокого розуміння асемблера.

Щодо збоїв та аварійного завершення (Crashes & Aborts), то це так звана тактика «випаленої землі». Замість того, щоб просто зупинити аналітика, програма «вбиває» сама себе, знищуючи сесію налагодження.

Коли в програмі стається критична помилка (наприклад, спроба доступу до недійсної пам'яті), операційна система її примусово завершує. Це автоматично завершує і сесію налагодження. Аналітик або зламник втрачає весь поточний стан аналізу: точки зупинки, значення змінних у пам'яті, історію викликів. Йому доводиться починати все спочатку. Дослідник запускає програму під налагоджувачем, доходить до певного місця, програма знову «падає». Він перезапускає її, намагаючись обійти це місце, але може натрапити на іншу пастку. Це перетворює процес аналізу на виснажливу гру, що демотивує і забирає багато часу.

7.4.4 Аналіз поведінки та взаємодії з користувачем

Це сучасний підхід до захисту програмного забезпечення, що переносить фокус з аналізу самого коду на аналіз того, хто і як цим кодом користується. Суть цього методу полягає у відстежуванні та оцінюванні патернів взаємодії людини з програмою, щоб відрізнити легітимного користувача від автоматизованого інструмента чи дослідника, який намагається її проаналізувати. В основі цього підходу лежить принцип: «машина поводиться не так, як людина». Головна мета – виявити аномалії у взаємодії, які вказують на процес дослідження або налагодження програми. Ці аномалії слугують тригерами для активації захисних механізмів. Такий аналіз поведінки у випадку несанкціонованого дослідження може виявити досить багато, а саме:

– виявити наявність налагоджувачів та емуляторів, оскільки робота програми під налагоджувачем значно сповільнює її виконання. Аналізуючи часові інтервали між діями користувача (натискання клавіш, рух миші) та реакцією інтерфейсу, програма може виявити нетипові затримки, характерні для середовища налагодження;

- виявити автоматизовані скрипти та боти, що намагаються автоматизувати взаємодію з програмою (наприклад, для брутфорсу), оскільки вони зазвичай виконують дії з надлюдською швидкістю, точністю та монотонністю;
- ідентифікувати людську присутність, тобто, перевірити, чи дійсно за комп'ютером сидить людина.

Програма, яка реалізує цей вид захисту, має збирати та аналізувати дані про поведінкові біометричні показники користувача.

1. Аналіз рухів миші:

- *траєкторія руху*, оскільки людські рухи мишею зазвичай плавні, дещо вигнуті та ніколи не є ідеально прямими. Рухи, згенеровані скриптом, часто є миттєвими або абсолютно прямими лініями від точки А до точки Б;
- *швидкість та прискорення*, оскільки людина змінює швидкість руху миші; сповільнює її перед натисканням на кнопку. А от у ботів швидкість часто є сталою;
- *мікрорухи та тремтіння*, оскільки навіть коли курсор нерухомий, людина здійснює ледь помітні мікрорухи;
- *час простою*, оскільки періоди бездіяльності миші є природними для людини, але можуть бути підозріло регулярними або відсутніми у скриптів.

2. Аналіз взаємодії користувача з клавіатурою:

- *швидкість набору*, коли аналізується швидкість та ритм набору тексту. Занадто швидке або дещо монотонне введення може вказувати на автоматизацію;
- *час між натисканнями* (Keystroke Dynamics). Унікальні часові інтервали між натисканням різних клавіш (flight time) та час утримання клавіші (dwell time) є стабільними для кожної людини. Відхилення від очікуваного патерну є підозрілим;
- *шаблон введення*, оскільки дослідник, який вводить команди в консоль налагоджувача, використовує, як правило, інші комбінації клавіш, ніж звичайний користувач.

3. Аналіз загальної взаємодії з інтерфейсом:

- *послідовність дій*, оскільки легітимний користувач зазвичай дотримується логічної послідовності дій в інтерфейсі. Дослідник може хаотично перемикатися між вікнами, намагаючись викликати певні функції в нетиповий спосіб;
- *час реакції* – вимірюється час між появою діалогового вікна та реакцією на нього. Надто швидка або, навпаки, занадто довга реакція (характерна для покрокового налагодження) є підозрілою.

На практиці програма може певний час збирати дані про поведінку конкретного легітимного користувача, щоб створити його «поведінковий

профіль». Далі під час роботи програми здійснювати постійний моніторинг – у фоновому режимі безперервно збирати та аналізувати дані про взаємодію. Якщо аналізатор виявляє значні відхилення від «нормальної» людської поведінки, він активує захисні механізми. Це можуть бути ті ж самі контрзаходи, що й у разі виявлення налагоджувачів: аварійне завершення роботи, вхід у нескінченний цикл, приховування або зміна критично важливої функціональності, повідомлення розробника про підозрілу активність.

Таким чином, аналіз поведінки перетворює самого користувача (або його відсутність) на ключ до захисту, роблячи традиційні методи злому, що ігнорують людський фактор, значно менш ефективними.

7.4.5 Мережеві техніки

Мережеві техніки захищають програми від дослідження, переносячи ключові компоненти, дані або логіку з комп'ютера користувача на віддалений сервер. Це робить локальний аналіз програми неповним і часто зовсім марним, оскільки найважливіші частини програми просто відсутні в доступному для дослідження коді.

Онлайн-активація та перевірка ліцензії, коли під час першого запуску або періодично під час роботи програма з'єднується з сервером для перевірки справжності ліцензійного ключа. Цей метод захисту ПЗ вже розглядався раніше у сенсі протидії піратству та несанкціонованому копіюванню, оскільки унеможливлює використання одного ключа на багатьох пристроях одночасно.

Цей самий спосіб захисту має своє місце і під час захисту від несанкціонованого дослідження, оскільки може досить суттєво ускладнити досліднику-зламнику процес створення «кряків». Зловмиснику потрібно не просто обійти перевірку, а й емулювати відповіді сервера, що складніше, ніж модифікувати локальний файл.

Виконання критичної логіки на сервері (Server-Side Logic). Це один з найпотужніших мережевих методів. Замість того, щоб весь код програми виконувався на комп'ютері користувача, найважливіші алгоритми переносяться на сторону сервера. Локальний клієнт лише надсилає дані на сервер, отримує результат і відображає його. Наслідками такого захисту є:

- по-перше, приховування коду, оскільки **к**ритична логіка (наприклад, алгоритм шифрування, перевірка ліцензії, інші унікальні функції програми) фізично відсутня на комп'ютері користувача. Її неможливо декомпілювати або здійснити налагодження;
- по-друге, захист даних, через те, що секретні дані (ключі шифрування, бази даних тощо), ніколи не залишають сервер;

– по-третє, контроль над використанням, оскільки розробник повністю контролює, як і коли виконується критична функція.

Для прикладу візьмемо програму для конвертування відео. Користувацький інтерфейс дозволяє лише вибрати файл, але сам процес конвертування відбувається на сервері розробника (серверна частина програми). У цьому випадку дослідник може аналізувати лише код, що відповідає за передавання файлу та відображення прогресу.

Перевірка «серцебиття» (Heartbeat Checks), коли програма періодично надсилає невеликі сигнали («пульс») на сервер, щоб підтвердити, що вона все ще працює в легітимному режимі та має активне з'єднання. Неважко зрозуміти, що це може дати у сенсі захисту програми:

– по-перше, постійний контроль, оскільки, якщо сервер перестає отримувати «пульс» або отримує його з нетиповими параметрами, він може зрозуміти, що програму закрито, від'єднано від мережі (можливо, для обходу захисту) або щось втручається в її роботу.
– по-друге, протидія фаєрволам, оскільки змушує користувача тримати відкритим з'єднання з сервером, інакше програма взагалі перестане працювати.

Віддалена атестація (Remote Attestation). Це просунута техніка, де клієнтська програма збирає докази про стан свого середовища (геші власних файлів, інформацію про запущені процеси, версію ОС) і надсилає їх на сервер для перевірки. У сенсі захисту це дозволяє:

– по-перше, виявити модифікацій, а саме: сервер перевіряє, чи не було змінено код програми на клієнті (наприклад, з метою відключення захисту). Якщо геш файлу не збігається з еталонним, сервер відмовить в обслуговуванні;
– по-друге, виявити інструменти аналізу, оскільки сервер може перевірити, чи не запущені в системі налагоджувачі (Debugger), аналізатори (IDA Pro) або віртуальні машини, і на основі цього заблокувати роботу програми.

Отже, мережеві техніки захисту ефективно протидіють локальному статичному та динамічному аналізу, переносячи поле бою на територію, яку повністю контролює розробник, – на свій сервер. Досліднику стає недостатньо аналізувати лише програму; йому потрібно аналізувати мережевий протокол, намагатися емулювати сервер і долати перевірки, які відбуваються поза його контролем.

7.5 Техніки обфускування, рівні та види обфускування

Обфускування – це один з методів захисту програмних застосунків, який дозволяє ускладнити процес реверсивної інженерії початкового коду захищуваної програми. Це процес свідомого ускладнення, затемнення або заплутування вихідного або скомпільованого програмного коду, щоб

зробити його менш зрозумілим для людини або комп'ютерного аналізатора. Це робиться з метою захисту інтелектуальної власності, запобігання зворотного проєктування (reverse engineering), або для ускладнення аналізу шкідливого програмного забезпечення.

Можна виділити мінімум два аспекти, які дозволяють визначити, що робота обфускатора була виконана недаремно:

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

Розглядаючи процес обфускування з погляду захисту ПЗ і трансформування початкового коду програми Prog1 у вихідний код Prog2 без можливості повернення до його початкового вигляду (трансформування в «один бік»), можна сказати, що процес трансформування буде вважатися процесом обфускування, якщо будуть задовольнятися такі ***вимоги***:

1. Код програми Prog2 внаслідок трансформування буде суттєво відрізнятися від коду Prog1, але водночас буде дієздатним і виконуватиме ті самі функції.
2. Вивчення принципу роботи, тобто процес реверсивної інженерії програми Prog2 буде більш складним, трудомістким і займатиме більше часу, ніж Prog1.
3. Під час кожного процесу трансформування одного і того самого коду програми Prog1 код програми Prog2 буде різним.
4. Створення програми, яка детрансформує програму Prog2 в її найбільш схожий початковий вигляд, буде неефективним.

Процес обфускування може бути здійснений на різних ***рівнях***:

- *низький рівень*, коли процес обфускування здійснюється над асемблерним кодом програми або навіть безпосередньо над двійковим файлом програми, який зберігає машинний код;
- *високий рівень*, коли процес обфускування здійснюється над вихідним кодом програми, написаної мовою високого рівня.

Здійснення обфускування на низькому рівні вважається менш комплексним процесом, але його важче реалізувати за рядом причин. Одна з таких причин полягає в тому, що мають бути враховані особливості роботи більшості процесорів, оскільки спосіб обфускування, прийнятний на одній архітектурі, може виявитись неприйнятним на іншій.

Потрібно зауважити, що може бути не зовсім ефективно піддавати обфускуванню весь код програми (наприклад, через те, що, наприклад, може збільшитись час виконання програми), в таких випадках доцільно здійснювати обфускування лише на найбільш важливих ділянках коду.

Обфускування буває декількох ***видів***:

- лексичне (символьне) обфускування;

– обфускування даних;
– обфускування потоку керування.

7.6 Лексичне обфускування (Lexical obfuscation)

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

Обфускування такого виду може містити певні складові.

1. *Видалення необов'язкових конструкцій* мови програмування:
– видалення усіх коментарів в коді або зміна їх на дезінформаційні;
– видалення різноманітних пробілів, відступів, які, зазвичай, використовуються для кращого візуального сприйняття коду програми.

2. *Перейменування методів, змінних і т. д. в набір безглуздих символів.* Наприклад, був метод класу GetPassword(), після обфускування цей метод матиме ім'я KJHS92DSLKf(). Це значно ускладнює аналіз коду та затримує процес його реверс-інжинірингу. Проте, багато декомпіляторів, зустрічаючи на своєму шляху подібного роду імена, замінюють їх на більш читабельні (method_1, method_2), тим самим зводячи всю роботу обфускатора нанівець. У таблиці 7.2 наведено декілька прикладів лексичного обфускування програмного коду.

3. *Перейменування найменувань змінних і методів у коротші*. Проходячи по всіх класах, методах, параметрах, вони замінюють імена на їх порядкові номери. Наприклад, був метод GetConnectionString(), а став _0(). До того ж, у зв'язку з тим, що символьної інформації тепер в збірці зберігатиметься менше, розмір самої збірки, відповідно, теж значно скоротиться. Подібні обфускатори можна також використовувати як оптимізувальні компілятори. Також подібне рішення добре тим, що існує вірогідність того, що одне і те саме ім'я буде використане для іменування класу, методів класу (що, наприклад, відрізняються тільки типом повернення) (табл. 7.2). Це дозволить також заблокувати роботу деяких дизасемблерів.

4. *Використання для імен змінних нечитабельних символів*. Частина обфускаторів вставляють в імена нечитабельні символи, наприклад, символи японської мови. Хоча .Net і працює з кодуванням UTF-8, не всі декомпілятори коректно обробляють такі символи: одні замінюють імена з такими символами на зрозуміліші, інші – просто відмовляються працювати з ними.

5. *Використання ключових слів мови програмування*. Цей вид символьного обфускування дозволить захиститися від найпримітивніших декомпіляторів, які, побачивши як ім'я зарезервоване слово, вважають, що збірка не валідна і відмовляються з нею працювати.

Таблиця 7.2 – Приклади лексичного обфускування коду

Оригінальний код	Код після обфускування
Python (заміна змінних)	

def calculate_total_price(item_price, quantity):
    tax_rate = 0.15
    subtotal = item_price * quantity
    total_price = subtotal * (1 + tax_rate)
    return total_price
 	
price = 100
qty = 5
final_cost = calculate_total_price(price, qty)
print(f"Final cost: {final_cost}")
	def f(a, b):
	    c = 0.15
	    d = a * b
	    e = d * (1 + c)
	    return e
	
	_1 = 100
	_2 = 5
	_3 = f(_1, _2)
	print(f"Final cost: {_3}")

Java (заміна класів і властивостей)	

public class UserProfile {
    private String userName;
    private int userId;
    public UserProfile(String userName,int userId)
{
        this.userName = userName;
        this.userId = userId;
    }
    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
}
 	
UserProfile user = new UserProfile("JohnDoe", 123);
System.out.println(user.getUserName());
	public class A {
	    private String a;
	    private int b;
	    public A(String a, int b) {
	        this.a = a;
	        this.b = b;
	    }
	    public String a() {
	        return a;
	    }
	    public void a(String a) {
	        this.a = a;
	    }
	}
	
	A a_obj = new A("JohnDoe", 123);
	System.out.println(a_obj.a());

Perl (декілька видів одночасно)	

my $filter;
     if (@pod)
    { my
($buffd,$buffer)=File::Temp::tempfile(UNLINK=>1);
       print $buffd "";
       print $buffd @pod or die "";
       print $buffd
       close $buffd   or die "";
       @found = $buffer;
       $filter = 1;
     } exit;
     sub is_taINT ed
     { my $arg = shift;
       my $nada = substr($arg, 0, 0); #zero-length
       local $@; # preserve caller's version
       eval { eval "#" };
       return length($@)!= 0;
     }
     sub am_taint _checking
     { my($k,$v)= each %ENV;
              return is_taINT ed($v);
     }
	sub z109276e1f2{(my $z4fe8df46b1 =
	shift(@_)); (my $zf6f94df7a7=
	substr($z4fe8df46b1, (0x1eb9+765-
	0x21b6),(0x0849+1465-0x0e02)));local $@;
	eval {eval(""));};return((length($@)!=
	(0x262+59-0x270d)));} my($z9e5935eea4);
	if(@z6a70320a) {(my($z5a5fa8125d,
	$zcc158ad3e0)= File::Temp::tempfile("",
	(0x196a+130-
	0x19eb)));print($z5a5a85d"");
	(print($z5a5fa8125d @z6a703c020a) or
	die((("".$zc158ae0)."\x3a\x20").$!)));
	print ($z5a5fa25d."");(close($z5aa85d)
	or die((("")));(@z83cc86e= $zcc13e0);
	($z9e5935eea4=(0x1209+1039-0x1617));}
	exit;sub z021c43d5f3{(my($z0f1649f7b5,
	$z9e1f91fa38)= each(%ENV));
	return(z109276e1f2($z9e1f91fa38));}

6. *Використання імен, що змінюють суть*. Припустимо, клас SecurityInfo з методом GetInformation() став класом Car з методом Wash(). Звичайно, це може заплутати недосвідченого зламника, але процесу декомпіляції ніяк не зашкодить. Тут швидше діє психологічний чинник.

Лексичне обфускування є фундаментальною частиною більшості комерційних обфускаторів коду, оскільки воно значно підвищує трудомісткість ручного аналізу коду. Однак, воно не робить код абсолютно невразливим, а лише значно збільшує час і ресурси, необхідні для його аналізу, та є лише першим кроком і часто поєднується з більш складними методами обфускування для досягнення вищого рівня захисту.

7.7 Обфускування даних (Data Obfuscaion)

Обфускування даних пов'язане із трансформацією структур даних і стосується змін у тому, як дані зберігаються та обробляються. Таке заплутування вважається більш складним, є більш сучасним й часто використовуваним. Його прийнято ділити на три основні групи, які описані нижче.

1. Обфускування зберіганням полягає в трансформації сховищ даних, а також самих типів даних (наприклад, створення й використання незвичайних типів даних, зміна подання існуючих і т. д.). Нижче наведено деякі підходи, що дозволяють здійснити таке обфускування.

а) Зміна інтерпретації даних певного типу. Як відомо, збереження даних у сховищах (змінних, масивах і т. д.) певного типу (ціле число, символ) у процесі роботи програми є дуже поширеним явищем. Наприклад, для переміщення по елементах масиву дуже часто використовують змінну типу «ціле число», що виступає в ролі індексу. Використання в цьому випадку змінних іншого типу можливе, але це буде нетривіально й може бути менш ефективно. Інтерпретація комбінацій розрядів даних, що містяться у сховищі, здійснюється залежно від його типу. Так, наприклад, можна сказати, що 16-розрядна змінна цілого типу, яка містить 0000000000001100, подає ціле число 12, але ж дані в такій змінній можна інтерпретувати по-різному (необов'язково як 12, а, наприклад, як 1100).

б) Зміна терміну використання сховищ даних, наприклад, перехід від локального їх використання до глобального і навпаки. Як приклад можна навести дві різні функції:

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

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

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

в) Перетворення статичних (незмінних) даних у процедурні. Більшість програм у процесі роботи виводять різну інформацію, що найчастіше в коді програми подається у вигляді статичних даних, таких як рядки, які дозволяють візуально орієнтуватися в її коді й визначати виконувані

операції. Такі рядки також бажано піддати обфускуванню, це можна зробити просто записуючи кожен символ рядка, використовуючи його ASCII-код, наприклад символ «A» можна записати як шістнадцяткове число «0x41», але такий метод занадто простий. Ефективнішим є метод, коли в код програми в процесі здійснення обфускування додається функція, що генерує необхідний рядок відповідно до переданих їй аргументів, після чого рядки в цьому коді видаляються, і на їх місце записується виклик цієї функції з відповідними аргументами.

г) Розділення змінних. Змінні фіксованого діапазону можуть бути розділені на дві й більше змінних. Для цього змінну *V*, що має тип *x* розділяють на *k* змінних $v_1,...,v_k$ типу *y*, тобто $V=v_1,...,v_k$. Потім створюється набір функцій (або виразів), що дозволяють витягати змінну типу *x* зі змінних типу *y* і записувати змінну типу *x* у змінні типу *y*.

Як приклад розділення змінних можна розглянути спосіб подання однієї змінної *B* логічного типу (*boolean*) двома змінними b_1 і b_2 типу короткого цілого (*short*), що набувають значень 1 або 0. Тоді наведений фрагмент коду (мова C/C++) буде перетворено у такий:

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

д) Зміна подання (або кодування). Наприклад, цілочисельну змінну *i* у наведеному нижче фрагменті коду можна замінити виразом $i=c_1 \cdot i+c_2$, де c_1, c_2 – константи. Результат – наведений код зміниться й буде складніший для сприйняття:

```
int  i=1;                         int  i=11, c1=8, c2=3 ;
...                               ...
while (i<1000){                   while (i<8003) {
   ... A[i] ...                      ... A[(i-3)/8] ...
       i++ ;                         i+=8 ;
}                                 }
```

2. Обфускування з'єднанням. Один з важливих етапів у процесі реверсивної інженерії програм базується на вивченні структур даних. Тому важливо намагатися у процесі обфускування ускладнити подання використовуваних програмою структур даних. Наприклад, у разі використання обфускування з'єднання заплутування досягається об'єднанням незалежних даних чи розділенням залежних. Далі наведено основні методи, що дозволяють здійснити таке обфускування.

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

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

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

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

Тоді

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

Тепер, знаючи *Z*, для знаходження *Y* можна використати формулу:

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

Під час здійснення арифметичних операцій над значеннями змінних X, Y, що зберігаються в Z, потрібно враховувати наведену вище формулу:

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

Результат – код до обфускування (мова C/C++):

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

трансформується в такий:

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

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

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

можна замінити на:

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

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

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

можна змінити на такий:

```
int  A[2][3] = { {1,2,3}, {4,5,6} };
for (int  i = 0 ; i < 2 ; i++){
        for (int  ii = 0 ; ii < 3 ; ii++){
```

```
                    A[i][ii] = A[i][ii] + 1 ;
                    print f("%d\n", A[i][ii]) ;
                }
        }
```

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

3. Обфускування переупорядкуванням полягає у зміні послідовності оголошення змінних, внутрішнього розташування сховищ даних, а також переупорядкування методів, масивів (використання нетривіального подання багатовимірних масивів), певних полів у структурах і т. д.

4. Модифікація області дії змінних у коді програми

а) Локалізація змінних у базовому блоці. Це перетворення локалізує використання змінних одним базовим блоком. Для кожного базового блока функції, що заплутується, створюється свій набір змінних. Всі використання локальних і глобальних змінних у вихідному базовому блоці замінюються на використання відповідних нових змінних. Щоб забезпечити правильну роботу програми між базовими блоками вставляються так звані сполучні (connective) базові блоки, завдання яких скопіювати вихідні змінні попереднього базового блока у вхідні змінні наступного базового блока. Застосування такого заплутуваного перетворення приводить до появи у функції великої кількості нових змінних, які, однак, використовуються тільки в одному-двох базових блоках, що заплутує людину, яка аналізує програму.

б) Розширення області дії змінних. Це перетворення за змістом обернене попередньому. Воно намагається збільшити час «життя» змінних настільки, наскільки можливо. Наприклад, виносячи блокову змінну на рівень функції або виносячи локальну змінну на статичний рівень, розширюється область її дії й ускладнюється аналіз програми. Тут використовується те, що глобальні методи аналізу добре обробляють локальні змінні, але для роботи зі статичними змінними потрібні більш складні методи міжпроцедурного аналізу.

7.8 Обфускування потоку керування (Control Flow Obfuscation)

Найскладнішим в плані реалізації, але найстійкішим до спроб зламу є обфускування графу потоку управління. Суть такого обфускування полягає у модифікації нормальної послідовності виконання інструкцій у програмі, щоб зробити її нелінійною та важкою для трасування. Це може бути досягнуто такими способами:

– *перетворенням обчислень* (наприклад, вбудовування в алгоритми помилкових умов);

– *видаленням або додаванням абстракцій коду* (наприклад, заміна виклику деякої функції безпосередньо тілом функції, або навпаки заміна однієї функції на декілька маленьких функцій);
– *перемішуванням* випадковим чином лінійних ділянок.

7.8.1 Маніпулювання функціями

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

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

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

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

д) Клонування функцій (*function cloning*). Під час зворотної інженерії функцій насамперед вивчається її сигнатура, а також те, як ця функція використовується, у яких місцях програми, з якими параметрами і як викликається. Аналіз контексту використання функції можна ускладнити, якщо кожен виклик деякої функції буде виглядати як виклик якоїсь іншої,

щоразу нової функції. Може бути створено декілька клонів і до кожного з них буде застосований різний набір заплутувальних перетворень.

7.8.2 Використання непрозорих предикатів

Непрозорі предикати є однією з технік обфускування коду, що використовується для ускладнення аналізу та реверс-інжинірингу програм. Суть цієї техніки полягає у впровадженні до коду умовних виразів (предикатів P^{false} або P^{true}), істинність яких завжди відома розробнику, але які надзвичайно важко або практично неможливо визначити статично для дослідника. Змінна є *непрозорою*, якщо існує деяка властивість щодо цієї змінної, яка відома в момент заплутування програми, але важко відновлюється після того, як заплутування завершене. Це створює заплутані шляхи виконання, що значно ускладнюють розуміння логіки програми.

Арифметичні вирази є одним з найпоширеніших і найефективніших способів створення непрозорих предикатів та загалом обфускування коду. Вони дозволяють маскувати прості умови або константи за допомогою складних, але завжди детермінованих обчислень. Наприклад:

```
(x * 0 == 0);
(x % 2 == 0) || (x % 2 != 0);
x / x завжди дорівнює 1 (для x != 0);
x + 0 або x * 1;
x % 2 == 0 або x % 2 != 0;
(A % N) * B + C, …
```

Така арифметика може бути використана для створення умов (непрозорих предикатів), які залежать від парності/непарності чисел, або ж для створення складних виразів, результат яких передбачуваний, але виглядає складним.

Алгебраїчні тотожності. Ці вирази використовують математичні властивості для маскування простих чисел або умов. Наприклад:

```
log_b(b^x) = x;
(A + B) – B = A;
A×B)/B = A (для B!=0);
A(modN)=(A+k×N)(modN) (для будь-якого цілого k);
sin²(x)+cos²(x)=1;
sin(x)=cos(п/2–x) або sin(x)=sin(п–x) та ін.
```

Використання цих тотожностей додає значний рівень складності для тих, хто намагається провести реверс-інжиніринг коду, оскільки вони змушують дослідника не лише розібратися у синтаксисі, але й зрозуміти математичну логіку, що стоїть за обфускованими виразами.

Складні булеві вирази, побудовані за допомогою еквівалентних перетворень із формули *true*. Наприклад:

```
x ^ x завжди дорівнює 0;
```

```
x | x завжди дорівнює x;
x & (~x) завжди дорівнює 0, …
```

Або ж можна взяти k довільних булевих змінних $x_1 ... x_k$ і за допомогою еквівалентних алгебраїчних перетворень, коли частина дужок або всі дужки розкриваються, побудувати з них тотожності, наприклад:

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

Це можна використати у багатьох випадках. Так, у наступному прикладі фрагмент коду після розширення умов циклу матиме зовсім інший вигляд:

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

⇒

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

Комбінаторні тотожності. Програмування комбінаторних тотожностей може бути використане для ускладнення логіки програми всюди: і у разі вставляння функцій, і під час клонування функцій, і у випадку розгортання циклів, і у разі вставляння мертвого та недосяжного кодів тощо. Наприклад:

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

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

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

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

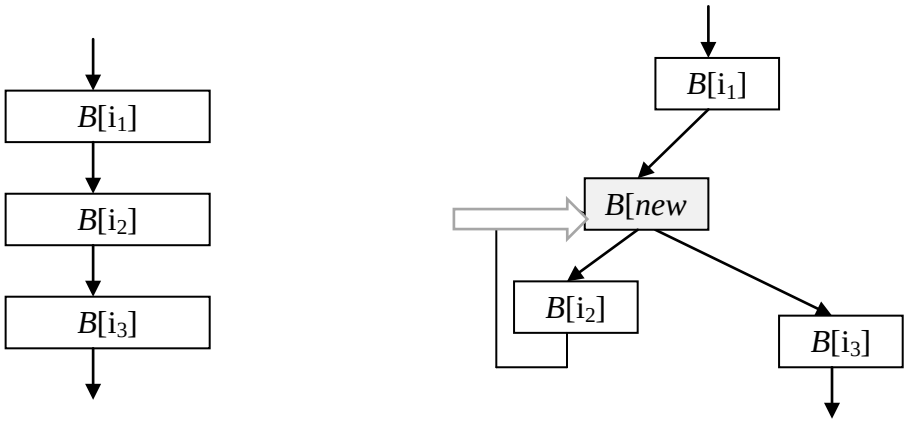
$$n = \sum_{k=0}^{n} (-1)^k 4^{n-k} C_{2n-k+1}^k - 1 .$$

Таким чином, непрозорі предикати є цінним інструментом в арсеналі розробників, які прагнуть захистити свій програмний код від дослідження. Вони ефективно збільшують складність аналізу, змушуючи дослідників витрачати значно більше часу та зусиль на розуміння логіки програми. Однак, як і будь-яка техніка обфускування, вони не забезпечують абсолютного захисту і мають розглядатися як частина комплексного підходу до безпеки програмного забезпечення.

7.8.3 Використання різноманітних перетворень циклів

Створення псевдоциклів. Перетворення полягає у внесенні в граф потоку управління функції зворотної дуги. У цьому випадку контролюється, щоб тіло отриманого циклу виконувалося лише один раз.

Типову схему такого перетворення показано на рисунку 7.1. Тут поєднуються дуги $B[i_1]–B[i_2]$ та $B[i_2]–B[i_3]$. Предикат, що знаходиться в кінці базового блока $B[new]$, має забезпечити одноразове виконання базового блока $B[i_2]$.



а) (початковий граф) б) (перетворений граф)

Рисунок 7.1 – Схема побудови псевдоциклу

Розгортання циклів. Цей метод застосовується в оптимізувальних компіляторах для прискорення роботи циклів та їх розпаралелювання.

Розгортання циклів полягає в тому, що тіло циклу копіюється два або більше разів, умова виходу з циклу і оператор збільшення лічильника відповідним чином модифікуються. Якщо кількість повторень циклу відома в момент компіляції, цикл може бути розгорнутий повністю. Наприклад, розгортання циклу може так змінити його:

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

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

Зниження розмірності індексного простору. На рисунку 7.2 наведено приклад застосування перетворення зниження розмірності індексного простору для функції перемножування двох матриць. Перетворення дозволило перейти від триразового вкладеного циклу до єдиного циклу.

```
enum {N=128};
typedef double matr_t[N][N];

void mm(matr_t m1,matr_t m2,
        matr_t r) {
     int  i,j,k;

     for(i=0;i<N;i++)
        for(j=0;j<N;j++)
           r[i][j]=0;
     for(i=0;i<N;i++)
        for(j=0;j<N;j++)
           for(k=0;k<N;k++)
              r[i][j]+=
              m1[i][k]*m2[k][j];
     . . . .
    }
```

```
enum {N=128};
typedef double matr_t[N][N];

void mm (matr_t m1,matr_t m2,
         matr_t r)  {
     int  i,j,k;

     for(i=0;i<N*N;i++)
        r[i/N][i%N]=0;
     for(i=0;i<N*N*N;i++)
        r[i/(N*N)][i%(N*N)/N]+=
        m1[i/(N*N)][i%N]*
        m2[i%N][i%(N*N)/N];
     . . . .
    }
```

а) (функція до перетворення) б) (функція після перетворення)

Рисунок 7.2 – Приклад перетворення зниженням розмірності

Розкладання циклів. Розкладання циклів полягає в тому, що цикл зі складним тілом розбивається на декілька окремих циклів із простими тілами й з тим самим простором ітерування. Наприклад, наведений цикл після розкладання може мати інший вигляд:

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

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

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

7.8.4 Внесення недосяжного, мертвого або надлишкового коду

Якщо в програму внесено деякі непрозорі предикати видів P^{false} або P^{true}, гілки умови, які відповідають умові «true» у першому випадку й умові «false» у другому випадку, ніколи не будуть виконуватися.

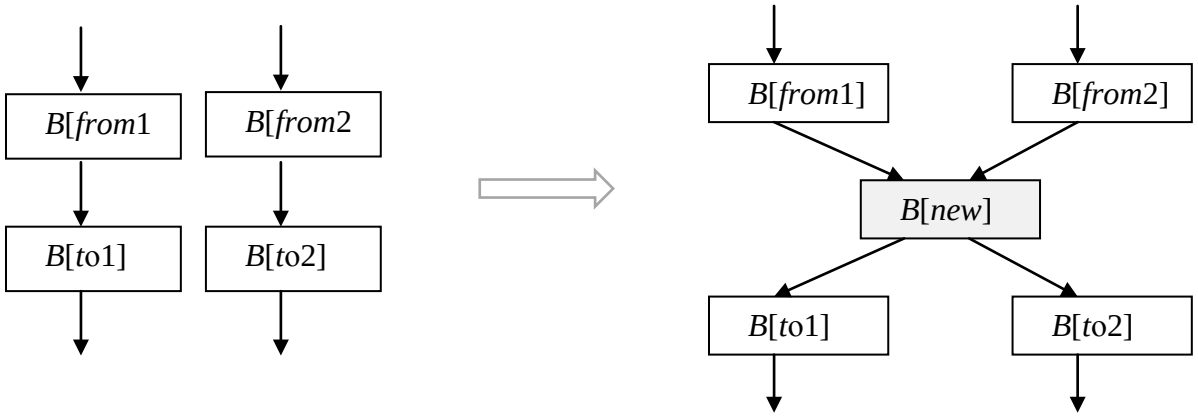
Фрагмент коду програми, що ніколи не виконується, називається ***недосяжним кодом*** (*unreachable code*). Ці гілки можуть бути заповнені довільними обчисленнями, які можуть бути схожі на дійсно виконуваний код, наприклад, зібрані із фрагментів тієї ж самої функції. Оскільки недосяжний код ніколи не виконується, таке перетворення впливає тільки на розмір заплутаної програми, але не на швидкість її виконання. Загальне завдання виявлення недосяжного коду, як відомо, алгоритмічно нерозв'язне. Це означає, що для виявлення недосяжного коду мають застосовуватися різні евристичні методи, наприклад, основані на статистичному аналізі програми.

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

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

Маскувальний код (мертвий, надлишковий, недосяжний) може бути використаний під час обфускування коду програми, наприклад, у разі використання зчеплення дуг графу коду управління програми, під час клонування блоків програми тощо.

Щодо ***зчеплення дуг***, то схему такого перетворення показано на рисунку 7.3. Для перетворення вибираються дві випадкові дуги графу потоку управління функції. Перевага віддається «далеким» одна від одної дугам, де відстань вимірюється як мінімальна з довжин двох найкоротших шляхів уздовж графу від кінця однієї дуги до початку іншої. Потрібно, щоб дві вибрані дуги не мали загального початку або загального кінця. Ключовим для забезпечення надійності зачеплення дуг є так званий предикат *P*, який в кінці виконання нового базового блока *B[new]* гарантує повернення на «правильний» шлях виконання – так званий повертальний предикат.



а) (початковий граф)

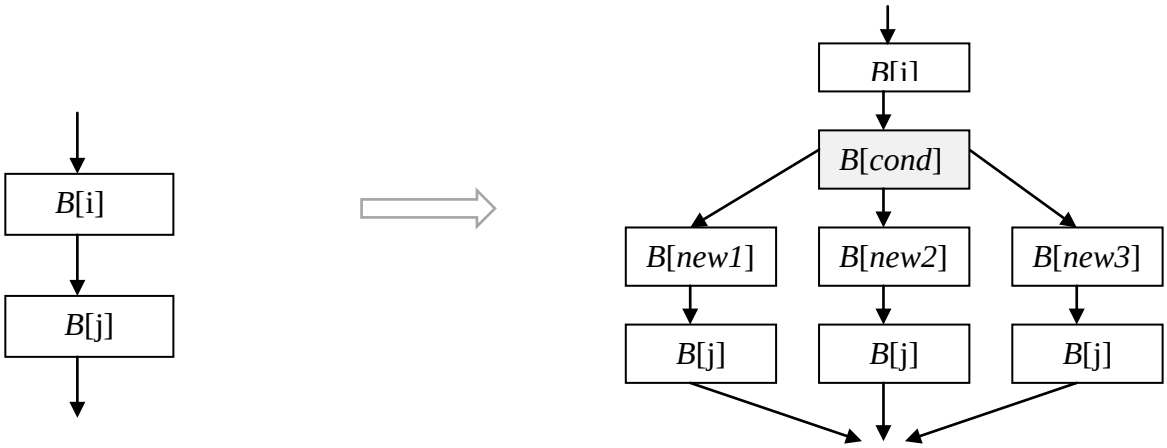
б) (перетворений граф)

Рисунок 7.3 – Схема перетворення зачеплення дуг

Найпростіший повертальний предикат – це звичайна булева змінна, яка може бути оголошена на рівні локальних або глобальних змінних. Повертальні предикати можуть бути побудовані на основі геш-функцій.

Перетворення ***клонування базових блоків*** полягає в заміні послідовного виконання двох базових блоків (наприклад, *B[i]* і *B[j]*) на оператор розгалуження з виконанням копії базового блока *B[j]* на кожній з гілок. Для цього базовий блок *B[j]* має бути скопійований необхідну кількість разів.

На рисунку 7.4 наведено відповідну схему перетворення.



а) (початковий граф) б) (перетворений граф)

Рисунок 7.4 – Схема перетворення клонування базових блоків

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

7.9 Інструменти для обфускування

Багато мов програмування та платформ мають власні інструменти обфускування.

7.9.1 Обфускатори програм Java

Оскільки Java-код компілюється в байт-код (файли `.class`), який відносно легко декомпілювати назад у вихідний код, обфускування є важливою технікою для захисту інтелектуальної власності та ускладнення реверс-інжинірингу. Існує багато інструментів для обфускування Java, які пропонують різні рівні захисту та функціональності. Вони можуть бути як комерційними, так і з відкритим вихідним кодом.

ProGuard – один з найвідоміших інструментів для Java-обфускування. Він є безкоштовним та з відкритим вихідним кодом (GPL ліцензія), але також доступні комерційні версії з розширеними функціями та підтримкою (ProGuardCORE, DexGuard для Android). ProGuard був інтегрований у Android Studio за замовчуванням для зменшення розміру APK та базового обфускування. ProGuard дуже ефективний для зменшення розміру, широко використовується, добре документований, гнучкий у налаштуванні через «keep rules».

Разом з тим, закладене в ньому базове лексичне обфускування може бути досить легко обійдено досвідченими реверс-інженерами. Для серйозного захисту потрібні додаткові техніки або більш просунуті обфускатори.

DashO (Commercial) – один з лідерів серед комерційних обфускаторів для Java та Android (від PreEmptive Solutions). Пропонує набагато глибший рівень захисту, ніж ProGuard. Крім базового лексичного обфускування, має дуже високий рівень захисту та комплексний набір функцій: заплутування логіки виконання коду (впровадження непрозорих предикатів, зациклень, зміна порядку блоків); шифрування літеральних рядків у байт-коді, щоб їх не було видно під час декомпіляції; шифрування цілих класів; впровадження прихованих міток для ідентифікації копій (водяних знаків); механізми виявлення модифікацій коду; запобігання налагодженню.

Allatori – ще один висококласний комерційний Java-обфускатор, відомий своєю ефективністю, який позиціонується як «друге покоління» обфускаторів. Активно використовується для захисту комерційного ПЗ. За ключовими функціями схожий на DashO, включно перейменування, обфускування потоку управління, шифрування рядків, водяні знаки, захист від налагодження та аналізу.

Zelix KlassMaster – потужний комерційний обфускатор, що пропонує широкий спектр технік захисту. Відомий своїми можливостями з обфускування потоку управління та захисту від реверс-інжинірингу.

yGuard – спочатку комерційний інструмент від yWorks, який з часом став відкритим вихідним кодом. Фокусується на перейменуванні та зменшенні розміру (перейменування класів, полів та методів, видалення невикористовуваного коду).

Stringer Java Obfuscation Toolkit – інструмент, зосереджений, переважно, на шифруванні рядків, що є однією з найважливіших технік для приховування конфіденційних даних (ключів, URL-адрес, повідомлень тощо). Здійснює ефективне шифрування рядків з динамічними ключами та ін'єкцією функцій дешифрування.

Skidfuscator (FOSS / Enterprise Version). Це відносно новий обфускатор з відкритим вихідним кодом, який активно розвивається і пропонує деякі цікаві та сучасні техніки обфускування, включно просунуті трансформації потоку управління. Має платну версію з розширеними функціями.

7.9.2 Обфускатори .NET-мов

Як і у випадку з Java, програми, написані на .NET мовах (C#, VB.NET, F# тощо), компілюються в проміжну мову (MSIL – Microsoft Intermediate Language). Цей MSIL, разом з метаданими, вбудований в .NET

збірки (файли `.exe` або `.dll`) і є досить легким для декомпіляції назад у вихідний код за допомогою таких інструментів, як Telerik JustDecompile, JetBrains dotPeek або ILSpy. Тому обфускування є критично важливим для захисту інтелектуальної власності та ускладнення реверс-інжинірингу .NET-додатків.

Dotfuscator (PreEmptive Solutions) – один з найбільш відомих і давно існуючих комерційних обфускаторів для .NET. Microsoft свого часу навіть вносила обмежену версію Dotfuscator Community Edition до Visual Studio. Він пропонує широкий спектр функцій захисту: перейменування, маскування потоку, шифрування, прив'язка до збірки (може об'єднувати кілька збірок в одну, ускладнюючи аналіз залежностей), вбудовування водяних знаків, захист від підробки та від налагоджування. Автоматично інтегрується з Visual Studio та CI/CD.

Eazfuscator.NET (Gapotchenko) – ще один дуже популярний комерційний обфускатор, відомий своєю простотою використання та ефективністю. Часто рекомендується розробниками за його надійність та автоматизацію. Має повний набір функцій, схожих на Dotfuscator, включно перейменування, обфускування потоку управління (з перетворенням на віртуальний код у просунутих версіях), шифрування рядків, захист від налагодження, захист від підробки. Особливо відомий функцією «Code Virtualization», яка перетворює частини IL-коду на власний віртуальний машинний код, що надзвичайно ускладнює декомпіляцію.

Babel Obfuscator (Babel for .NET). Це потужний інструмент захисту для .NET Framework та .NET Core/5+/8+, відомий своїми глибокими трансформаціями MSIL. Здійснює такі ключові функції: перевантажене перейменування (дозволяє перейменовувати методи з різними типами повернення, що робить декомпіляцію неможливою до C#/VB.NET), обфускування потоку управління, шифрування рядків, захист ресурсів, XAML/BAML перейменування (що дуже важливо для WPF/MAUI додатків), водяні знаки.

.NET Reactor (Eziriz) – комплексне рішення для захисту .NET, яке виходить за рамки простого обфускування. Він пропонує різні шари захисту, включно такі функції: обфускування (перейменування, потік управління, шифрування), техніку NecroBit (яка перетворює IL-код у нативний x86-код, що надзвичайно ускладнює декомпіляцію), віртуалізація коду, анти-тамперінг, анти-налагодження, шифрування збірок, вбудовані можливості ліцензування. Цей інструмент вважається одним з найбільш агресивних і ефективних.

Skater .NET Obfuscator (Rustemsoft) – потужний комерційний обфускатор, який забезпечує різні рівні захисту .NET збірок. Його ключові функції такі: перейменування, шифрування рядків (з можливістю

зберігання їх у «хмарному сховищі»), обфускування потоку управління, захист ресурсів, водяні знаки, анти-декомпіляція.

Хоча комерційні інструменти пропонують найвищий рівень захисту, існують також безкоштовні та open-source альтернативи, які можуть бути корисними для базового обфускування або для вивчення принципів.

ConfuserEx (Open Source) – дуже популярний і добре підтримуваний обфускатор з відкритим вихідним кодом. Він пропонує досить сильний захист для безкоштовного інструменту. Здійснює такі функції, як перейменування, обфускування потоку управління, шифрування рядків, захист від дебагінгу/дампінгу, «Packer» для ущільнення збірок, підтримка плагінів для розширення функціональності. Має активну спільноту та відносно сильний захист для вільного інструменту.

Obfuscar (Open Source) – простий, але ефективний безкоштовний обфускатор з відкритим вихідним кодом. Він більше зосереджений на перейменуванні та ущільненні рядкових констант. Легкий у використанні, добре підходить для базового обфускування, хоча менш комплексний захист порівняно з комерційними інструментами або ConfuserEx.

7.9.3 Обфускатори для JavaScript

Javascript-obfuscator – один з найпопулярніших та найпотужніших безкоштовних обфускаторів для JavaScript. Він пропонує широкий спектр опцій для захисту коду: перейменування змінних, функцій та параметрів, вилучення та шифрування рядків, впровадження «мертвого» коду, модифікація потоку та інші різноманітні кодові трансформації. Доступний як npm-пакет, може бути інтегрований у процеси збірки (наприклад, з Webpack).

Jscrambler (комерційний) – професійне рішення для обфускування та захисту JavaScript, орієнтоване на корпоративний сегмент. Jscrambler пропонує дуже високий рівень захисту та розширені можливості: надзвичайно потужні техніки обфускування, що містять мутації коду, поліморфізм, метаморфізм, виявлення та попередження атак у реальному часі, блокування доменів та інші. Використовує хмарний сервіс та API.

Terser та ***UglifyJS*** – сучасні та потужні мініфікатори/обфускатори, які виконують базовий рівень обфускування та оптимізації JavaScript. Підтримують сучасний синтаксис ES6+. Ці інструменти, насамперед, призначені для мініфікації коду (зменшення розміру файлу), але також виконують деякі базові форми обфускування, такі як перейменування змінних, видалення пробілів, коментарів, крапок з комою, скорочення імен змінних, функцій та властивостей і т. д.

Google Closure Compiler – розроблений Google інструмент для оптимізації JavaScript. Має три рівні оптимізації: SIMPLE_OPTIMIZATIONS,

WHITESPACE_ONLY та ADVANCED_OPTIMIZATIONS. Останній рівень надає дуже агресивну оптимізацію, яка може також слугувати формою обфускування: дуже агресивне перейменування глобальних змінних, функцій та властивостей, інтенсивне видалення «мертвого» коду, вбудовування функцій (function inlining) та констант. Він може значно зменшити розмір коду та ускладнити його розуміння. Доступний як JAR-файл (Java-додаток) або як npm-пакет, або через API.

Таким чином, обфускатори JavaScript – це корисний інструмент для підвищення безпеки клієнтського коду та захисту інтелектуальної власності, проте вони мають використовуватися як частина комплексної стратегії безпеки, а не як єдиний засіб захисту.

7.9.4 Обфускатори коду Python

Обфускування коду Python має свої особливості, оскільки Python є інтерпретованою мовою. На відміну від компільованих мов, де код перетворюється безпосередньо в машинні інструкції, Python-код виконується віртуальною машиною, яка спочатку компілює його в байт-код (файли `.pyc`). Цей байт-код порівняно легко декомпілювати назад у схожий на вихідний Python-код. Тому «обфускування» Python-коду часто є радше «заплутуванням» або «ускладненням декомпіляції», ніж повним унеможливленням зворотної розробки. Найкращий захист для чутливої Python-логіки – це розміщення її на сервері, щоб клієнти не мали до неї прямого доступу.

Проте, існують інструменти та підходи, які допомагають ускладнити аналіз Python-коду

Pyarmor – один з найпопулярніших та найпотужніших інструментів для обфускування Python. Це інструмент командного рядка. Він працює на рівні байт-коду, перетворюючи його таким чином, щоб декомпіляція була значно складнішою. Pyarmor може прив'язувати обфускований код до конкретних машин або встановлювати термін дії. Крім основних способів обфускування (перейменування функцій, методів, класів, змінних та аргументів), він має ще деякі ключові особливості: перетворення деяких Python-функцій у функції C (що потім компілюються в машинні інструкції) для незворотного обфускування, прив'язка коду до конкретного пристрою, встановлення терміну дії для коду, підтримка Themida (для Windows) для додаткового захисту.

PyObfuscator – це бібліотека Python, призначена для обфускування Python-коду. Пропонує вбудовані методи для реалізації більшості технік обфускування, включає різні рівні обфускування, перейменування, шифрування рядків, можливе використання антиналагоджувальних технік. Може використовуватись як бібліотека Python (імпорт та виклик функцій) або через командний рядок.

Pyminifier – відкритий (GPL-3.0) мініфікатор, обфускатор і компресор коду Python, який працює у форматі командного рядка. Він виконує базові операції обфускування, такі як видалення коментарів, скорочення імен змінних, мініфікацію імен змінних, функцій та класів, ущільнення коду.

Obfupy – обфускатор вихідного коду Python, що має на меті створення коректного та функціонального, але заплутаного коду і використовується як бібліотека Python. Серед його функціональних особливостей переписування умовних операторів, перейменування локальних змінних, виділення функцій, обфускування числових та рядкових констант, заміна вбудованих функцій, додавання зайвих операторів керування потоком, видалення docstrings та коментарів, кодування цілого файлу (base64, zip, bz2). Дуже кастомізований.

Package-obfuscator – інструмент, робота якого фокусується на компіляції коду в бінарний вигляд в межах пакетів. Код перетворюється на бінарні файли, що ускладнює пряме читання та зворотну розробку.

7.9.5 Обфускатори коду C/C++

Обфускування коду C/C++ є складнішим завданням, ніж для інтерпретованих мов або мов з байт-кодом (наприклад, Java або .NET), оскільки C/C++ компілюється безпосередньо в машинні інструкції. Це означає, що немає «проміжної мови», яку можна було б легко декомпілювати назад у вихідний код.

Однак, хоча пряма декомпіляція у читабельний вихідний код C/C++ є надзвичайно складною (практично неможливою з повним збереженням логіки та імен змінних), зловмисники можуть використовувати інструменти реверс-інжинірингу (дизасемблери, декомпілятори у псевдокод, налагоджувачі) для аналізу бінарного файлу.

Обфускатори C/C++ зазвичай працюють:

- на рівні вихідного коду. Ці інструменти приймають вихідний код C/C++ і застосовують трансформації, перш ніж код компілюється стандартним компілятором;
- на рівні LLVM. Ці обфускатори модифікують проміжне подання коду, яке генерується компілятором LLVM, що дозволяє застосовувати більш глибокі трансформації, які потім компілюються в машинний код;
- на рівні бінарного файлу. Ці інструменти модифікують вже скомпільований бінарний файл, додаючи захист від налагодження, аналізу та модифікації. Вони часто використовуються разом з обфускуванням вихідного коду для багатошарового захисту.

Stunnix C and C++ Obfuscator. Це комерційний крос-платформний обфускатор, що працює з вихідним кодом. Він зосереджений на заплутуванні імен змінних, функцій, макросів, класів та інших ідентифікаторів, заплутуванні тіл макросів, вставлянні «мертвого» коду та ін.

Pelles C/C++ Obfuscator – вбудований обфускатор, що постачається з Pelles C (IDE для розробки на C/C++ для Windows). Це більш базовий рівень обфускування на рівні вихідного коду, зосереджений на перейменуванні та видаленні пробілів.

StarForce C++ Obfuscator – комерційне рішення, яке фокусується на захисті вихідного коду від аналізу та модифікації, змінюючи логіку програми для ускладнення зворотної розробки, маскуючи доступ до змінних, змішуючи граф виконання коду (вставлення фіктивних зв'язків у граф виконання, дублювання гілок графу, динамічне розгалуження графу тощо).

Obfuscator-LLVM (O-LLVM). Це проєкт з відкритим вихідним кодом, що є форком LLVM та додає до нього механізми обфускування. Це один із найвідоміших і найуживаніших інструментів для обфускування C/C++ на цьому рівні. O-LLVM виконує такі трансформації:

- перетворює лінійний потік виконання на складний, заплутаний граф, що ускладнює розуміння логіки;
- додає незрозумілі умовні переходи та мертвий код, які ніколи не виконуються, але заплутують аналіз;
- замінює прості інструкції на більш складні, еквівалентні послідовності;
- шифрує рядкові літерали у бінарному файлі, розшифровуючи їх лише під час виконання.

Важливо розуміти, що жоден метод обфускування і жоден обфускатор не є 100 % надійним. Обфускування – це не панацея, а лише ускладнення процесу зворотного проєктування та аналізу. Воно має використовуватись у поєднанні з іншими заходами безпеки, такими як належна архітектура безпеки, криптографічні практики, аутентифікація, авторизація та регулярні аудити безпеки.

7.10 Спеціалізовані інструментів для захисту програм

Існує ціла низка спеціалізованих інструментів та технологій, які розробники використовують для захисту своїх програм від несанкціонованого дослідження (зворотної розробки) та копіювання. Ці інструменти часто поєднують різні методи для створення багаторівневого захисту.

ASProtect – це система захисту програмного забезпечення, розроблена компанією ASPack Software. Вона призначена для швидкого впровадження функцій захисту в програми, орієнтована насамперед на

розробників програмного забезпечення. Основні можливості ASProtect містять:

- ущільнення програми та шифрування програм;
- протидія дампінгу пам'яті та захист від модифікації даних у пам'яті (Memory Patching);
- протидія налагоджувачам і дизасемблерам:
- перевірка цілісності програми та перевірка реєстраційних ключів.

ASProtect все ще використовується (ліцензії на продукт все ще пропонуються для придбання), але його популярність, можливо, зменшилася з часом через появу більш складних та ефективних рішень для захисту програмного забезпечення, які пропонують більш просунуті методи.

Themida – це потужна комерційна система захисту ПЗ, розроблена компанією Oreans Technologies. Її основна мета – захистити додатки від просунутої зворотної розробки та несанкціонованого доступу, що є ключовим для запобігання піратству та захисту інтелектуальної власності (див. рис. 7.1).

Themida вважається одним з найбільш надійних комерційних інструментів захисту програмного забезпечення, і обхід її захисту зазвичай потребує значних зусиль та знань у галузі зворотної розробки. Вона містить у собі майже всі антиналагоджувальні та антидизасемблерні техніки, перелічені у табл. 7.1. В основі Themida лежить інноваційна технологія SecureEngine®. Вона складається з набору складних антикрекінгових заходів, які додаються до виконуваних файлів і дозволяють програмі запускатися лише в захищеному середовищі.

Хоча Themida призначена для законного захисту ПЗ, її складні техніки захисту також іноді використовуються розробниками шкідливого ПЗ для уникнення виявлення антивірусами та ускладнення аналізу дослідниками безпеки. Через це антивіруси можуть позначати файли, захищені Themida, як «Themida Trojan» або подібне, якщо вони містять шкідливий контент.

Enigma Protector – це комплексна система захисту програмного забезпечення та ліцензування, розроблена для захисту виконуваних файлів Windows (як 32-бітних, так і 64-бітних) та .NET-застосунків. Її головна мета – запобігти несанкціонованому доступу, зворотному інжинірингу, модифікації та незаконному копіюванню програмного забезпечення.

Основні можливості і методи, що їх застосовує Enigma Protector, такі:

- віртуалізація коду;
- мутація коду або поліморфізм (оригінальний код різними способами змінюється, створюючи різні його версії кожного разу під час застосування захисту);

- техніки проти налагодження та захист від дампінгу пам'яті;
- захист таблиці імпорту, рядків та ресурсів виконуваних програм;
- перевірка цілісності за допомогою контрольних сум;
- методи для введення в оману інструментів аналізу файлів.
- перевірка процесів та служб і виявлення віртуальних машин;
- цілу низку функцій системи ліцензування (генерування реєстраційних ключів, прив'язка до обладнання, керування ліцензіями та інші).

VMProtect (VMP) є одним із найефективніших методів ускладнення зворотної розробки програмного забезпечення. Він використовує передову технологію віртуалізації коду, яка значно ускладнює аналіз та розуміння внутрішньої роботи захищеної програми.

VMProtect забезпечує захист від дослідження, використовуючи такі техніки і технології:

- віртуалізація коду, тобто перетворення оригінальних x86/x64 інструкцій захищених ділянок коду в інструкції спеціально створеної віртуальної машини, архітектура якої є унікальною та невідомою для аналітиків. Звичайні дизасемблери не можуть безпосередньо інтерпретувати інструкції віртуальної машини. Водночас VMProtect може використовувати різні архітектури віртуальних машин для різних частин коду, що ще більше ускладнює аналіз;
- мутації коду (поліморфізм), що ускладнює створення статичних сигнатур для виявлення захищених ділянок коду;
- антиналагоджувальні техніки: виявлення налагоджувачів, протидія трасуванню, виявлення точок зупину, зміна поведінки у разі виявлення налагоджувача;
- захист від дампінгу пам'яті шляхом шифрування коду та даних;
- захист цілісності коду шляхом перевірки контрольних сум.
- приховування API, що ускладнює розуміння взаємодії програми з операційною системою.

CodeVirtualizer – це комерційний протектор програмного забезпечення, розроблений Oreans Technologies (тією ж компанією, що й Themida). Його основна мета – захистити програми від зворотної розробки та аналізу шляхом віртуалізації коду. CodeVirtualizer перетворює вибрані частини оригінального коду програми в інструкції для спеціально розробленої віртуальної машини. Тобто, він використовує ті самі технології, що і інші інструменти, але, звичайно, має і деякі особливості.

По-перше, інтеграція з IDE. CodeVirtualizer зазвичай пропонує інтеграцію з популярними середовищами розробки (IDE), такими як Visual Studio, що спрощує процес захисту програми.

По-друге, підтримка різних платформ забезпечує захист як для 32-бітних, так і для 64-бітних Windows-застосунків.

По-третє, гнучкі налаштування, оскільки розробники мають можливість налаштовувати рівень захисту та вибирати конкретні методи віртуалізації та антиналагоджувальні техніки. Крім того, CodeVirtualizer може добре поєднуватися з іншими продуктами Oreans, такими як Themida, для створення багаторівневого захисту.

Окрім наведених у цьому підрозділі інструментів для захисту від несанкціонованого дослідження, існує багато інших, які фокусуються на окремих способах захисту, таких як захист обфускуванням, захист від налагодження і дизасемблювання тощо. Деякі з цих інструментів будуть розглянуті далі.

ScyllaHide – це потужний інструмент з відкритим вихідним кодом, призначений для фахівців з реверс-інжинірингу та аналізу програмного забезпечення. Його головна функція – «анти-анти-налагодження» (anti-anti-debugging), тобто, коли програма намагається виявити, що її аналізують за допомогою налагоджувача-дебагера, ScyllaHide втручається і приховує присутність цього налагоджувача, дозволяючи досліднику продовжувати свою роботу безперешкодно. ScyllaHide працює як «щит» для налагоджувача. Він перехоплює запити від програми і підміняє відповіді так, ніби ніякого налагодження не відбувається. Основний механізм – перехоплення функцій. ScyllaHide вбудовується в процес, що аналізується, і ставить свої «заглушки» на системні функції, які зазвичай використовуються для виявлення налагодження. ScyllaHide зазвичай використовується як плагін для популярних налагоджувачів x64dbg, OllyDbg, IDA Pro.

Отже, у підсумку можна сказати, що ScyllaHide – це інструмент в арсеналі реверс-інженера, який дозволяє «зрівняти шанси» у боротьбі з програмами, що активно опираються аналізу, роблячи процес налагодження значно плавнішим та ефективнішим.

TitanHide – це драйвер режиму ядра (kernel-mode driver), розроблений для приховування процесів налагодження від найпросунутіших технік виявлення. Це також інструмент для фахівців з реверс-інжинірингу, що працює на значно нижчому рівні, ніж як ScyllaHide. Головне завдання TitanHide – зробити процес налагодження невидимим для програми, що аналізується, протидіючи навіть тим методам виявлення, які працюють на рівні ядра операційної системи.

На відміну від ScyllaHide, який працює в режимі користувача (user-mode), TitanHide є драйвером, що працює в режимі ядра. Це надає йому значно більші повноваження та контроль над системою. Тому TitanHide може перехоплювати та модифікувати результати системних викликів безпосередньо на їхньому джерелі, а також напряму змінювати структури даних ядра Windows, які містять інформацію про процеси та їхній стан налагодження. Програми в режимі користувача не мають такого доступу.

TitanHide спеціалізується на обході низькорівневих технік, які не піддаються інструментам типу ScyllaHide.

Таким чином, TitanHide використовують тоді, коли програма або шкідливий код застосовує просунуті методи антиналагодження, що оперують на рівні ядра, і звичайних інструментів, що працюють у режимі користувача, стає недостатньо.

Denuvo Anti-Tamper – це технологія захисту від несанкціонованого доступу та копіювання (DRM – Digital Rights Management), розроблена австрійською компанією Denuvo Software Solutions GmbH. Наразі Denuvo належить компанії Irdeto. Основна мета Denuvo – запобігти піратству відеоігор, особливо в перші, найважливіші для продажів, тижні та місяці після релізу. Технологія не прагне зробити гру «незламною» назавжди, а, скоріше, максимально відтермінувати появу зламаних версій.

Для розробників та видавців гри ця технологія є привабливою з погляду захисту продажів, захисту інвестицій (розробка ігор є дуже дорогою), збільшення «вікна продажів» (навіть якщо гру врешті-решт зламають, затримка в декілька тижнів або місяців може суттєво вплинути на загальні продажі).

Точні деталі роботи Denuvo є комерційною таємницею, але загальний перелік технік такий: інтегрування у код гри, прив'язка до «заліза» та регулярні перевірки унікального «ліцензійного файлу» або «токена», різні методи обфускування коду, шифрування та віртуалізації.

Незважаючи на заявлені цілі, Denuvo часто стає об'єктом критики з боку ігрової спільноти, оскільки робота Denuvo суттєво впливає на продуктивність, потребує онлайн-активації (потрібне інтернет-з'єднання), накладає обмеження на зміну «заліза», створює незручності для чесних покупців, тоді як пірати рано чи пізно отримують зламану версію, часто без проблем, пов'язаних із DRM. Ефективність Denuvo є предметом постійних дискусій.

8 ЗАХИСТ ВІД ПРОГРАМ-ДАМПЕРІВ

Дампінг (Dumping) – це процес створення «знімка» (дампа) оперативної пам'яті, що належить певній програмі. Мета зловмисників – витягнути з пам'яті розпакований код, секретні дані, ключі шифрування, ігрові ресурси та іншу цінну інформацію, яка стає доступною лише під час виконання програми, в динаміці. Тому захист програм від дампінгу передбачає сукупність методів, спрямованих на ускладнення або унеможливлення створення повного дампу процесу в пам’яті з метою аналізу або злому.

8.1 Мета і принципи роботи програм-дамперів

Програми-дампери (Dumper Programs) – це спеціалізовані утиліти, головне завдання яких – створювати дамп (dump), тобто, точний «знімок» (або «зліпок») вмісту оперативної пам'яті, що належить певному процесу, або всього стану системи на конкретний момент часу. Будь-яка програма під час своєї роботи зберігає в оперативній пам'яті безліч тимчасової, але важливої інформації: розпакований код (якщо вона була захищена за допомогою пакування), дані користувача або/та ключі шифрування (навіть якщо вони були у коді зашифровані, то під час виконання вони розшифровуються), налаштування тощо. Дампер підключається до цільового процесу (або до ядра системи), що дозволяє «витягнути» всю цю інформацію з пам'яті та зберегти її в один або декілька файлів для подальшого аналізу.

Сфера застосування програм-дамперів дуже широка і містить як легітимні, так і зловмисні цілі.

Щодо *легітимного використання*, то тут дампери використовують:

- для налагодження, коли розробники створюють дампи аварійних програм (crash dumps), щоб проаналізувати їх стан у момент збою, знайти помилки та виправити їх (часто для цього використовують програму ProcDump від Microsoft або вбудовані функції диспетчера завдань);
- у цифровій криміналістиці (Forensics), коли фахівці з безпеки аналізують дампи пам'яті заражених систем, щоб знайти сліди шкідливого ПЗ, виявити мережеві з'єднання вірусу, витягнути ключі шифрування, які використав зловмисник тощо;
- для аналізу продуктивності, оскільки дослідження дампу дозволяє зрозуміти, як програма використовує пам'ять, і знайти можливі витоки (memory leaks).

Зловмисне або «сіре» використання (реверс-інжиніринг) проявляється у тому, що програми-дампери використовують:

- для зламу програм (Cracking), що є основним застосуванням дамперів у цій сфері. Якщо програма захищена пакувальником

(наприклад, UPX) або протектором (наприклад, VMProtect), її оригінальний код розпаковується в пам'ять лише під час виконання. Дампер дозволяє «зловити» цей розпакований код і зберегти його для подальшого аналізу та модифікації (наприклад, для видалення перевірки ліцензії);
- для видобування ресурсів, коли з пам'яті ігор або мультимедійних програм за допомогою дамперів витягують 3D моделі, текстури, звуки та інші ресурси, захищені від прямого копіювання з оригінальних файлів гри;
- для аналізу шкідливого ПЗ. Це випадок, коли дослідники вірусів використовують дампери для отримання «чистого» коду вірусу, який також часто запакований для ускладнення аналізу.

8.2 Приклади популярних програм-дамперів

Як правило, програми-дампери – це потужні менеджери процесів, які мають вбудовану функцію для створення дампів пам'яті будь-якого процесу. Вони часто використовуються як альтернативи стандартному диспетчеру задач і дозволяють отримати докладну інформацію про процеси, бібліотеки, дескриптори, мережеву активність тощо.

Process Explorer від Microsoft (раніше Sysinternals, створена Марком Руссіновичем) є потужним інструментом для моніторингу, аналізу та діагностики процесів у Windows (рис. 8.1). Програма часто використовується розробниками та фахівцями з кібербезпеки. Основні її можливості:
- детальний перегляд усіх запущених процесів і потоків і відображення дерева процесів;
- аналіз відкритих дескрипторів (файлів, ключів реєстру, об'єктів синхронізації) та пошук по дескрипторах;
- виведення інформації про завантажені DLL-бібліотеки та модулі;
- створення дампу процесу (пам'яті) та збереження його у форматі .dmp, який можна аналізувати у WinDbg, Visual Studio або інших дебагерах.

Process Hacker – це потужна утиліта з відкритим кодом для моніторингу системи, діагностики, аналізу і втручання в процеси (рис. 8.2). Функціонал її подібний до функціонала Process Explorer, але з більшою гнучкістю. Основні можливості такі:
- можливість завершення або зупинки системних/захищених процесів;
- аналіз ін’єкцій DLL, налагодження;
- перехоплення обробників та аналіз API-хуків;
- моніторинг мережі, диск I/O, GPU;
- редагування пам’яті процесу;

– збереження дампа – дозволяє створювати повні або вибіркові дампи пам’яті процесу.

Перевагою Process Hacker порівняно з Process Explorer є більш просунуті функції для реверс-інженерії та аналізу шкідливого ПЗ, а також те, що утиліта має відкритий код, тож може бути модифікована під власні потреби.

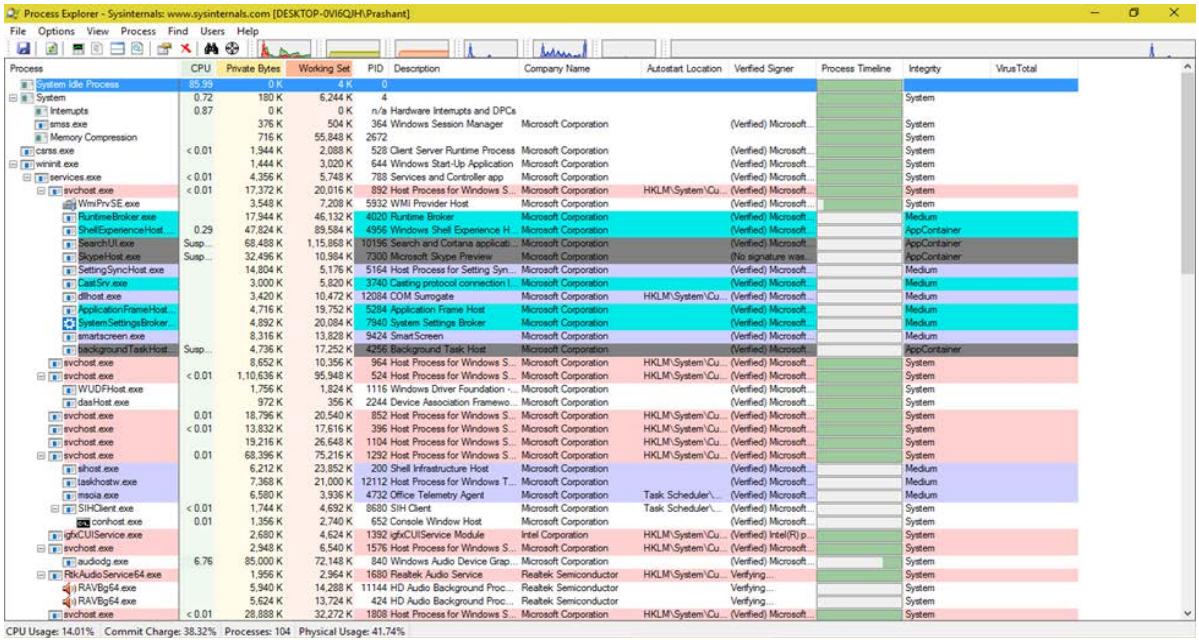


Рисунок 8.1 – Вигляд одного з основних вікон Process Explorer

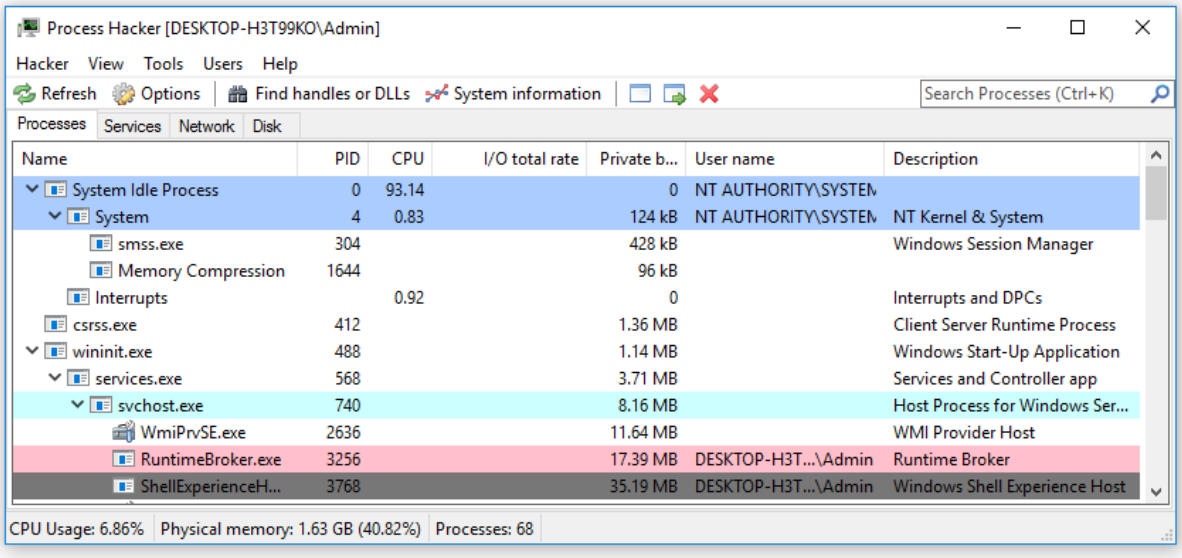


Рисунок 8.2 – Вигляд одного з основних вікон Process Hacker

Отже, обидві програми, Process Explorer і Process Hacker, можуть використовуватись для аналізу шкідливого ПЗ, знімання дампів для реверс-інжинірингу або для налагодження додатків. Але у Process Hacker

більше можливостей для інженерів з безпеки та досвідчених користувачів, хоча він може сприйматися антивірусами як потенційно небажане ПЗ. Обидві утиліти не потребують встановлення – це портативні програми.

ProcDump. Це утиліта командного рядка від Microsoft для створення креш-дампів за певними умовами (наприклад, за різкого зростання навантаження на CPU).

Scylla – спеціалізований дампер, що інтегрується з налагоджувачами (як x64dbg) і призначений саме для зняття дампів захищених програм, здатний автоматично знаходити та виправляти таблицю імпорту розпакованого файлу.

PC Hunter / ExtremeDumper. Це потужні інструменти, що працюють на низькому рівні (часто з використанням драйверів) і здатні робити дампи навіть захищених системних процесів або процесів, що опираються дампінгу.

Підводячи підсумки, можна сказати, що дампер – це двосічний меч. В руках розробника чи криміналіста це потужний інструмент для діагностики та аналізу, а в руках реверс-інженера – ключовий засіб для обходу захисту та вивчення внутрішньої роботи програм. Тому захист від дампінгу є критично важливим для протидії реверс-інжинірингу та злому. Розглянемо деякі техніки та методи, що для цього використовуються.

8.3 Основні способи захисту від дампінгу

8.3.1 Антидампінгові захисні механізми (Anti-Attach Techniques)

Найпростіший спосіб зробити дамп – приєднати до процесу налагоджувач (debugger). Тому перше, що можна зробити, – не дозволити цього зробити, для чого можна використати таке:

- *прямі виклики API*, тобто постійна перевірка наявності налагоджувача за допомогою спеціальних функцій: IsDebuggerPresent() або CheckRemoteDebuggerPresent();
- *низькорівневі перевірки* – аналіз прапорців у структурах даних процесу (наприклад, PEB.BeingDebugged у Windows), які система встановлює при підключенні налагоджувача;
- *створення «пасток»,* а саме, використання специфічних винятків або системних викликів, які по-різному обробляються у звичайному режимі та під налагоджувачем. Якщо програма виявляє налагоджувач, вона може негайно завершити роботу, щоб не дати зробити дамп.

8.3.2 Шифрування секцій та даних у пам'яті (Memory Encryption)

Навіть у випадку, якщо зловмисник зможе зробити дамп, дані в ньому будуть марними, якщо вони зашифровані. Тому можна для захисту від дампінгу використати таке:

- *шифрування секцій PE-файлу*, коли критично важливі секції виконуваного файлу (наприклад, `.text`, де міститься код програми) зашифровані. Вони розшифровуються в пам'яті лише на короткий момент перед виконанням, а потім одразу ж зашифровуються знову. Дампінг, в будь-який інший момент часу, дасть лише зашифровані дані;
- *шифрування змінних та рядків*, тобто, коли усі важливі дані, такі як ліцензійні ключі, паролі, IP-адреси серверів, зберігаються в пам'яті не у відкритому вигляді, а в зашифрованому. Програма розшифровує їх «на льоту» тільки перед безпосереднім використанням, а після цього одразу ж видаляє розшифровану копію з пам'яті.

8.3.3 Перевірка цілісності коду (Code Integrity Checks)

Цей метод спрямований на виявлення втручання в код програми, зокрема спроб модифікувати його для створення дампу:

- *обчислення контрольних сум (CRC32).* Це означає, що програма періодично обчислює контрольну суму для власних фрагментів коду в пам'яті. Якщо зловмисник поставить точку зупинки (breakpoint), він змінить код, і контрольна сума не зійдеться. Це буде сигналом для активації захисту;
- *«нано-міти».* Це означає створення безлічі дрібних, взаємопов'язаних функцій, які постійно перевіряють одна одну. Якщо одна з них буде змінена, інші це виявлять.

8.3.4 Моніторинг та очищення пам'яті (Memory Monitoring&Wiping)

Програма може активно стежити за власною пам'яттю та зачищати важливу інформацію.

- *зачистка ключів.* Після того як криптографічний ключ був використаний, область пам'яті, де він зберігався, заповнюється випадковими даними («сміттям»), щоб його неможливо було відновити з дампа;
- *виявлення підозрілих сторінок пам'яті.* Це базується на тому, що деякі інструменти для дампінгу створюють додаткові сторінки пам'яті або змінюють права доступу до них (наприклад, роблять секцію з кодом доступною для записування). Програма може моніторити таблицю власних сторінок пам'яті та, виявивши такі аномалії, завершити роботу.

8.3.5 Використання технік пакувальників та віртуальних машин

Сучасні протектори (пакувальники) є найефективнішим засобом проти дампінгу, оскільки вони використовують такі техніки:

- *розпакування «на льоту»,* коли пакувальник розпаковує лише невеликі блоки коду безпосередньо перед їх виконанням. Весь інший код залишається запакованим. Зробити повний та робочий дамп стає вкрай складно, оскільки потрібно «зловити» та зібрати всі розпаковані шматочки коду;
- *віртуалізація коду* (так працюють VMProtect, Themida). Це найвищий рівень захисту. Оригінальний машинний код програми перетворюється на байт-код для унікальної віртуальної машини, вбудованої в саму програму. В пам'яті знаходиться не звичний x86-код, а цей специфічний байт-код та інтерпретатор для нього. Зробити дамп такого процесу можна, але отриманий результат буде абсолютно незрозумілим без повного відновлення архітектури цієї унікальної віртуальної машини, що є надзвичайно складним завданням.

Отже, ефективний захист від дампінгу – це завжди комбінація декількох методів. Покладатися лише на один спосіб недостатньо. Сучасні протектори використовують багатошаровий підхід, поєднуючи анти-налагодження, шифрування, перевірки цілісності та віртуалізацію, щоб максимально ускладнити життя зловмисникам.

9 ЗАХИСТ АВТОРСЬКИХ ПРАВ НА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

9.1 Загальні положення

Захист авторських прав на ПЗ є важливою складовою правової охорони інтелектуальної власності. Робота розробника, інвестиції часу та зусиль заслуговують на належний захист.

Захист авторських прав на програмне забезпечення є надзвичайно важливим у сучасному цифровому світі, оскільки:

- демонструє серйозність та професіоналізм, що підвищує довіру клієнтів та партнерів;
- гарантує отримання належної винагороди за працю та інвестиції;
- допомагає зберегти унікальну пропозицію розробника або команди та відрізнятися від конкурентів;
- перешкоджає несанкціонованому копіюванню, розповсюдженню та використанню програмного продукту.

Для розробників існує декілька ефективних способів захистити свою інтелектуальну власність.

1. *Реєстрація авторського права на програмний продукт*. Хоча реєстрація не є обов'язковою, вона може слугувати доказом авторства в разі суперечок. В Україні з 8 листопада 2022 року існує державний орган для реєстрації авторських прав – Український національний офіс інтелектуальної власності та інновацій (УКРНОІВІ), який є суб’єктом, що виконує функції Національного органу інтелектуальної власності (НОІВ).
2. *Ліцензії та угоди на використання ПЗ.* Шляхом ліцензування можна чітко визначити, як інші можуть використовувати розроблене ПЗ. Існують різні типи ліцензій (відкриті, закриті, комерційні тощо), які дозволяють вибрати оптимальний варіант для конкретного програмного продукту.

Складання чітких договорів з клієнтами та партнерами, які визначають права та обов'язки сторін щодо використання ПЗ дозволить, у разі порушення авторських прав, звернутися до суду для захисту своїх інтересів.

3. *Технічні способи захисту ПЗ*, до яких можна віднести:
   - *ліцензійні ключі*. Використання ліцензійних ключів обмежує доступ до програми лише авторизованим користувачам;
   - *шифрування*. Захист вихідного коду за допомогою шифрування ускладнює його копіювання та модифікацію;
   - *встановлення справжності коду – tamper-proofing* (з англ. «захист від підробки або втручання») – сукупність технологій та методів, спрямованих на забезпечення цілісності системи або продукту, запобігання несанкціонованому втручанню, модифікації чи підробці. Щодо програмного забезпечення – це можуть бути *водяні знаки* або *електронний цифровий підпис*.

9.2 Захист програм за допомогою електронного цифрового підпису (Electronic Digital Signature)

Захист програмного забезпечення за допомогою електронного цифрового підпису (ЕЦП) є ефективним способом забезпечення цілісності, автентичності та захисту від підробок. ЕЦП дозволяє переконатися, що програмний продукт або його компонент не був змінений після створення автором і походить саме від заявленого розробника.

Наприклад, усі драйвери для Windows мають бути підписані сертифікованим ЕЦП. Установчі пакети та інсталятори (наприклад, .exe, .msi) часто підписуються, щоб запобігти встановленню підроблених програм. Мобільні додатки iOS та Android потребують підписання додатків для їх розповсюдження через App Store або Google Play.

Вбудовування електронного цифрового підпису у програмне забезпечення – це процес інтеграції криптографічних механізмів, які дозволяють підтвердити автентичність та цілісність програмного продукту.

Для цього може знадобитися бібліотека для роботи з ЕЦП, яка підтримує алгоритми цифрового підпису (наприклад, RSA, ECDSA) і криптографічні протоколи. Наприклад,

- для мов C/C++ – бібліотеки OpenSSL, Crypto++;
- для мови Java – Java Cryptography Architecture (JCA), BouncyCastle;
- для Python – PyCryptodome, cryptography;
- для C#/.NET – System.Security.Cryptography.

Наведемо приклад використання ЕЦП для програми мовою Python з використанням бібліотеки `cryptography`.

Для створення цифрового підпису потрібні приватний (закритий) і публічний (відкритий) ключі.

```
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
         # Генерування приватного ключа
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
)
         # Збереження приватного ключа у файл
with open("private_key.pem", "wb") as f:
    f.write(private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()
    ))
         # Отримання публічного ключа
public_key = private_key.public_key()
         # Збереження публічного ключа у файл
with open("public_key.pem", "wb") as f:
    f.write(public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    ))
```

Програмне забезпечення або його компоненти (наприклад, файли або рядки) підписуються закритим ключем.

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
          # Дані для підписання
data = b"Це приклад даних для підпису."
          # Підписання даних
signature = private_key.sign(
    data, padding.PSS(
                  mgf=padding.MGF1(hashes.SHA256()),
                  salt_length=padding.PSS.MAX_LENGTH,
          ),
    hashes.SHA256(),
)
          # Збереження підпису у файл
with open("signature.sig", "wb") as f:
    f.write(signature)
```

Програма має вміти перевіряти, чи відповідає цифровий підпис даним і чи його створив заявлений автор.

```
          # Завантаження підпису
with open("signature.sig", "rb") as f:
    signature = f.read()
          # Перевірка підпису
try:
    public_key.verify(
        signature, data, padding.PSS(
                                  mgf=padding.MGF1(hashes.SHA256()),
                                  salt_length=padding.PSS.MAX_LENGTH,
                          ),
        hashes.SHA256(),
    )
    print("Підпис дійсний.")
except Exception as e:
    print("Підпис недійсний або дані змінено:", e)
```

Інтегрування ЕЦП може бути виконано різними способами залежно від мети:

- для перевірки підпису під час запуску програми (наприклад, з використанням геш-суми інсталяторів, файлів або бібліотек);
- для підписання конфіденційних даних, коли програма підписує файли, які вона створює, щоб гарантувати їхню цілісність;
- для оновлення, коли підпис перевіряється перед встановленням оновлення.

Закритий ключ має бути збережений у захищеному місці (бажано уникати зберігання закритого ключа безпосередньо в програмі), наприклад, з використанням апаратних модулів безпеки (HSM).

Щоб гарантувати довіру користувачів, варто сертифікувати публічний ключ у деякому авторитетному сертифікаційному центрі.

9.3 Водяні знаки на програмному забезпеченні

Водяні знаки (Watermarking) на програмному забезпеченні – це приховані або явні мітки, інтегровані в програмний код або вихідні дані для підтвердження авторства чи забезпечення захисту від несанкціонованого використання.

Ця технологія є важливим інструментом, оскільки:

- може бути використана у боротьбі з піратством (зокрема, використання водяних знаків може знизити мотивацію до розповсюдження неліцензійного ПЗ), нелегальним копіюванням і несанкціонованим розповсюдженням;
- дозволяє підтвердити авторство або право власності на ПЗ та слугувати доказом у разі судового спору;
- можуть допомогти ідентифікувати нелегальні копії.

9.3.1 Типи водяних знаків

Водяні знаки можуть бути використані у різних типах програмного забезпечення:

- у комерційному ПЗ вони можуть бути пов’язані з ліцензією конкретного користувача;
- у вебдодатках можна додавати мітки до результатів роботи, наприклад, до згенерованих зображень чи файлів;
- у мультимедійному ПЗ, наприклад, у відео- чи фотообробці водяні знаки можуть бути додані до створених користувачем матеріалів.

Водяні знаки з погляду їх видимості у програмному забезпеченні можуть бути прихованими та явними.

Приховані водяні знаки (Invisible Watermarks). Вони вбудовуються в код або дані таким чином, щоб не впливати на роботу програми, але залишатися доступними для виявлення. Вони можуть бути виконані у вигляді:

- вбудовування безпосередньо у програмний код. Наприклад, інтеграція певного унікального ідентифікатора (номер ліцензії або ID користувача) у змінні, функції або бібліотеки;
- метаданих – додавання специфічної інформації у метадані файлів, які генерує або обробляє програма;
- послідовності коду, тобто використання унікальних шаблонів у структурі коду або коментарях;
- маркування алгоритмів, тобто внесення унікальних змін у реалізацію алгоритмів, які не впливають на функціональність.

Явні водяні знаки (Visible Watermarks), які чітко видимі для користувача можуть проявляти себе, як:

- відображення інформації в інтерфейсі. Наприклад, у демоверсіях ПЗ може відображатися логотип або назва компанії.
- генерування міток у результатах роботи програми. Наприклад, у програмах для роботи з PDF файли можуть містити підпис «Створено у [Назва програми]»;
- виведення даних із зазначенням користувацької ліцензії, тобто, у звітах або документах, створених програмою, можна додати інформацію про ліцензійного користувача.

Впровадити водяні знаки можна по-різному:

- приховати певні мітки у машинному коді (наприклад, унікальні послідовності байтів у скомпільованих файлах);
- згенерувати знаки на етапі розгортання (наприклад, у вигляді унікального маркера, який додається під час активації ліцензії користувачем);
- за допомогою обфускування коду (інтегрувати водяні знаки у заплутаний код, щоб ускладнити реверс-інжиніринг);
- згенерувати унікальні версії для кожного користувача, коли програмний засіб генерується індивідуально з унікальним водяним знаком для кожного ліцензійного користувача.

9.3.2 Способи вбудовування водяних знаків у програми

Існує декілька підходів до вбудовування водяних знаків у програмне забезпечення, які можна умовно поділити на три основні категорії:

- статичні;
- динамічні;
- апаратні.

Статичні водяні знаки впроваджуються в програмний код або дані до етапу виконання програми. Вони є частиною виконуваного файлу і можуть бути виявлені шляхом аналізу цього файлу. Тут можуть бути використані основні техніки обфускування, а саме:

- *вбудовування у дані*, коли водяний знак закодований у значеннях констант, рядкових літералах або інших статичних даних у програмі (наприклад, послідовність байтів, що подає водяний знак, може бути прихована всередині великого масиву даних);
- *вбудовування в код*, що передбачає модифікацію самого коду програми. Наприклад, перемішування інструкцій (зміна порядку незалежних одна від одної інструкцій, яка не впливає на загальний результат, але може кодувати біти водяного знака); використання еквівалентних інструкцій (заміна одних інструкцій процесора на інші, функціонально ідентичні, але з іншим двійковим поданням);

вставлення «мертвого» коду (додавання блоків коду, які містять у собі закодовану інформацію);

– *вбудовування у структуру*, коли водяний знак закодований у структурі графу потоку управління програми (наприклад, додавання фіктивних розгалужень, які ускладнюють аналіз і водночас несуть у собі приховану інформацію).

Отже, використання статичних знаків відносно просте у реалізації та не впливає на продуктивність під час виконання програми. Разом з тим, вони більш вразливі до атак, оскільки аналіз виконуваного файлу може виявити аномалії в коді чи даних.

Динамічні водяні знаки проявляються лише під час виконання програми. Вони вбудовуються в динамічні структури даних або в поведінку програми, що робить їх виявлення значно складнішим завданням. Реалізувати це можна такими способами:

– вбудувати водяний знак *у динамічні структури даних* – закодувати у структурі динамічно створюваних об'єктів у пам'яті (деревах, списках або графах), коли сама послідовність створення та зв'язки між цими об'єктами може нести приховану інформацію;

– використати *часові характеристики*, коли водяний знак буде проявлятися через специфічні затримки у виконанні певних операцій, які важко відрізнити від звичайних коливань продуктивності;

– вбудувати *через потік виконання*, коли водяний знак буде поданий у вигляді унікальної послідовності викликів функцій або переходів по графу потоку управління, а сама послідовність буде активуватися за певних умов.

Динамічні водяні знаки більш стійкі до статичного аналізу і значно ускладнюють реверс-інжиніринг, оскільки потребують аналізу програми в динаміці. Але, як недолік, потрібно зазначити, що вони можуть дещо впливати на продуктивність програми, та в реалізації й тестуванні вони складніші.

Апаратно-залежні водяні знаки використовують унікальні характеристики апаратного забезпечення для генерування або перевірки водяного знака. Наприклад, програма може зчитувати унікальний ідентифікатор процесора (CPUID) або інші апаратні параметри та використовувати їх як частину водяного знака. Це ускладнює перенесення зламаної копії програми на інший комп'ютер, тобто захищає від несанкціонованого копіювання.

9.3.3 Стійкість водяних знаків до атак та обмеження

Для досягнення максимального рівня захисту водяні знаки рідко використовуються ізольовано. Їхня ефективність значно зростає в

поєднанні з іншими технологіями захисту, такими як обфускування коду, анти-тампериng та іншими.

Ефективність програмних водяних знаків оцінюється за їхньою стійкістю до різноманітних атак, спрямованих на їх видалення або спотворення, а саме:

- *атаки на видалення* (Subtractive Attacks), коли зловмисник намагається локалізувати та видалити водяний знак;
- *спотворювальні атаки* (Distortive Attacks), які полягають у спробах пошкодити водяний знак шляхом трансформацій коду, таких як оптимізація, перекомпіляція або подальше обфускування;
- *адитивні атаки* (Additive Attacks), коли зловмисник додає власні водяні знаки, щоб створити плутанину щодо справжнього власника.

Жоден метод водяних знаків не є абсолютно невразливим. Винахідливий та мотивований зловмисник з достатньою кількістю часу та ресурсів теоретично може зламати будь-який захист. Однак головна мета водяних знаків – не створити нездоланну перешкоду, а значно підвищити вартість та складність злому, роблячи його економічно невигідним.

9.4 Антитампериng (Tamper-proofing)

Антитампериng, або захист від втручання, – це набір методів та технологій, спрямованих на те, щоб унеможливити або значно ускладнити несанкціоновану модифікацію програмного забезпечення. Звичайно, для цього можна використати всі наведені методи захисту від нелегального копіювання, несанкціонованого статичного та динамічного дослідження.

Основними цілями антитамперингу є:

- захист від злому (Cracking), тобто запобігання видаленню перевірок ліцензії, обходу обмежень пробних версій (trial) або активації платних функцій безкоштовно;
- захист інтелектуальної власності – ускладнення аналізу коду (реверс-інжинірингу) для крадіжки унікальних алгоритмів;
- запобігання чітерству в іграх, що означає не дати гравцям модифікувати клієнт гри для отримання нечесних переваг (наприклад, «wallhack» або «aimbot»);
- захист від шкідливих модифікацій, тобто запобігання вбудовуванню в програму шпигунського коду, вірусів чи іншого шкідливого функціоналу;
- захист водяних знаків шляхом забезпечення їх цілісності та неможливості видалення.

Антитампериng тісно пов'язаний з обфускуванням та антиналагодженням, часто працюючи разом з ними для створення багатошарової системи захисту.

Реалізація захисту від втручання передбачає вбудовування в програму механізмів самоперевірки і залежить від платформи (наприклад, Windows, Android, iOS), мови програмування (C++, C#, Java тощо), а також від типу загроз (злом, ін’єкції, реверс-інжиніринг).

Розглянемо деякі основні підходи.

1. *Перевірка цілісності коду за допомогою геш-сум*. Це найпоширеніший та фундаментальний метод. Суть його полягає у тому, щоб обчислити криптографічну геш-суму (наприклад, SHA-256, MD5, …) для критичних частин коду програми або для всього виконуваного файлу і зберегти цей «еталонний» геш. Під час запуску або в процесі роботи програма перераховує геш і порівнює його з еталонним. Якщо геш не збігається – вихід або запуск помилкової поведінки.
2. *Шифрування коду або даних*. Цей підхід полягає у тому, що частини програми (особливо логіка ліцензування чи авторизації) можуть бути зашифровані, а розшифруватись лише під час виконання. Такий підхід у поєднанні з обфускуванням дає гарний результат.
3. *Обфускування (заплутування) коду*. Воно змінює структуру коду так, що його важко зрозуміти або модифікувати без виконання. Наприклад, у .NET – використання Dotfuscator, ConfuserEx.
4. *Детектування дебагерів або реверс-інструментів*. Цей підхід базується на виявленні запуску в середовищі дебагера (наприклад, через API `IsDebuggerPresent()`) або на виявленні специфічних процесів (OllyDbg, IDA, x64dbg тощо).
5. *Захист від ін'єкцій* шляхом виконання:
   – перевірки пам’яті на предмет сторонніх DLL або змінених функцій;
   – підпису та перевірки цілісності важливих функцій.
6. *Runtime Self-Checks,* коли програма сама перевіряє свій стан у процесі виконання:
   – CRC/геш важливих блоків пам’яті;
   – випадкові перевірки послідовностей байтів.

Таким чином, основна мета тамперингу – зробити так, щоб програма могла самостійно виявляти спроби зміни свого коду чи поведінки та реагувати на них.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Атака переповнення буфера. URL: https://www.imperva.com/learn/application-security/buffer-overflow/ (дата звернення: 16.03.2025).
2. SQL Injection Attacks (SQLi) – Web-based Application Security. URL: https://www.spanning.com/blog/sql-injection-attacks-web-based-application-security-part-4/ (дата звернення: 14.04.2025).
3. Міжсайтові сценарії (XSS) атаки. URL: https://www.imperva.com/learn/application-security/cross-site-scripting-xss-attacks/ (дата звернення: 16.05.2025).
4. Secure software development: best practices, frameworks, and resources. URL: https://hyperproof.io/resource/secure-software-development-best-practices (дата звернення: 22.03.2025).
5. Мітнік К. Мистецтво бути невидимим. Як зберегти приватність в епоху Big Data. Харків : Фабула, 2019. 432 с.
6. Еріксон Д. Хакінг. Мистецтво експлойту. Санкт-Петербург : Пітер, 2014. 608 с.
7. Білтон Н. Кіберзлочинець №1. Харків : Клуб сімейного дозвілля, 2018. 448 с.
8. Технології захисту інформації : підручник. М. М. Браіловський та ін. Київ : ЦП «Компринт», 2021. 296 с.
9. Системи захисту інформації : підручник. Ю. В. Костюк та ін. Київ : Київський столичний університет імені Бориса Грінченка, 2025. 887 с.
10. Новітні технології захисту інформації : підручник / М. Г. Луцький та ін. Київ : НАУ, 2023. 312 с.
11. Building and designing secure software: best practices and development framework/ URL: https://codewave.com/insights/building-designing-secure-software/ (дата звернення: 22.05.2025).
12. Security Development Lifecycle (SDL) Practices. URL: https://www.microsoft.com/en-us/securityengineering/sdl/practices?oneroute=true (дата звернення: 16.05.2025).
13. What Is the SSDLC (Secure Software Development Life Cycle)? URL: https://www.hackerone.com/knowledge-center/what-ssdlc-secure-software-development-life-cycle (дата звернення: 16.0.2025).
14. Mitigating the Risk of Software Vulnerabilities by Adopting a Secure Software Development Framework (SSDF). URL: https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.04232020.pdf
15. Технічна документація yubikey. URL : https://yubikey.com.ua/en/documentation (дата звернення: 12.04.2025).
16. Програма розробника. URL: https://developers.yubico.com/ (дата звернення: 8.07.2025).

17. Токен C100 HOTP. URL : https://lockers.com.ua/token-c100-hotp/ (дата звернення: 12.04.2025).
18. Токен Biopass Fdo U2F FIDO2 USB-A K27. URL: https://lockers.com.ua/token-biopass-fido-u2f-fido2-usb-a-k27/ (дата звернення: 12.04.2025).
19. Про компанію Authentrend. URL: https://authentrend.com.ua/pro-nas/ (дата звернення: 12.04.2025).
20. Аппаратні ключі безпеки ATKEY.Pro. URL: https://lockers.com.ua/token-c100-hotp/ (дата звернення: 12.04.2025).
21. Державна система правової охорони інтелектуальної власності. URL: https://ukrpatent.org/uk/articles/copyright (дата звернення: 8.08.2025).
22. 5 кращих інструментів для тестування безпеки. URL: https://qagroup.com.ua/publications/5-krashchykh-instrumentiv-dlia-testuvannia-bezpeky/ (дата звернення: 8.08.2025).
23. Найкращі інструменти тестування безпеки з відкритим кодом. URL: https://www.guru99.com/uk/security-testing-tools.html (дата звернення: 8.08.2025).
24. Системний інформатор. URL: https://systeminformer.sourceforge.io/ (дата звернення: 10.06.2025).
25. Провідник процесів версії 17.06. URL: https://learn.microsoft.com/en-us/sysinternals/downloads/process-explorer (дата звернення: 10.06.2025).

ГЛОСАРІЙ

ASLR (Address Space Layout Randomization) – це технологія, що використовується в операційних системах для підвищення рівня безпеки програм. Вона працює шляхом рандомізації розташування важливих структур даних у пам'яті процесу кожного разу, коли він запускається. Це робить значно складнішим для зловмисників експлуатувати такі уразливості, як переповнення буфера, оскільки вони не можуть передбачити точну адресу, куди потрібно помістити свій шкідливий код.

SEH (Structured Exception Handling) – це механізм, вбудований в операційні системи Windows, який дозволяє програмам ефективно обробляти непередбачувані події, що можуть виникнути під час їх виконання. Такі події називаються виключними ситуаціями.

SEHOP (Structured Exception Handler Overwrite Protection) – це технологія безпеки, розроблена для підвищення захисту операційних систем від одного з найпоширеніших типів уразливостей – переповнення буфера обробника виключних ситуацій.

SQL Injection (SQLi) – це одна з найпоширеніших і найнебезпечніших уразливостей вебдодатків. Вона дозволяє зловмисникам вводити шкідливі SQL-команди безпосередньо в поля введення веб-форм, в URL-параметри, cookies, HTTP-заголовки тощо, що дає їм можливість маніпулювати базою даних і отримувати доступ до конфіденційної інформації.

XSS (Cross-Site Scripting) – це поширена вразливість вебдодатків, яка дозволяє зловмисникам вводити свій JavaScript код на вебсторінку, яку переглядає користувач. Цей код може виконуватися в браузері жертви, даючи зловмиснику можливість виконувати різні шкідливі дії.

CSRF (Cross-Site Request Forgery) – це вразливість веббезпеки, яка дозволяє зловмисникам змусити автентифікованого користувача неусвідомлено виконати запит на веб-сайті, який сервер розцінює як легітимний.

DoS (Denial of Service) – це тип кібератаки, спрямованої на те, щоб зробити певний сервіс недоступним для його легітимних користувачів. Іншими словами, зловмисник штучно створює таке навантаження на систему, що вона перестає справлятися зі своїми функціями.

Вразливість нульового дня. Це означає, що про цю вразливість не було відомо розробникам до моменту атаки, тобто захиститись від неї заздалегідь було неможливо.

SSDF (Secure Software Development Framework) – це сукупність рекомендацій та кращих практик, розроблена Національним інститутом

стандартів і технологій США (NIST) для створення, розробки та захисту програмного забезпечення, одночасно забезпечуючи захист конфіденційної інформації.

SSDLC (Secure Software Development Lifecycle) – це методологія розробки програмного забезпечення, яка інтегрує контроль безпеки на кожному етапі життєвого циклу: від концепції та проєктування до тестування, супроводу й виведення продукту з експлуатації.

DevOps – це сукупність практик, що об'єднують розробку програмного забезпечення (Development) та його обслуговування (Operations). Головна мета DevOps – забезпечити швидку, надійну та безперервну доставку програмних продуктів до користувачів, використовуючи автоматизацію процесів і тісну взаємодію команд.

DevSecOps – це розширення концепції DevOps, яке ставить акцент на безпеці. Якщо DevOps об'єднує розробку та експлуатацію, то DevSecOps додає до цього рівняння ще й безпеку.

OWASP (Open Web Application Security Project) – це міжнародна некомерційна організація, яка об'єднує фахівців з безпеки вебдодатків з усього світу. Її головна мета – зробити вебдодатки безпечнішими шляхом розробки відкритих, вільних від роялті матеріалів та інструментів, які допомагають розробникам і організаціям у всьому світі створювати більш безпечне програмне забезпечення. OWASP:

- регулярно публікує OWASP Top 10 – список найпоширеніших і найнебезпечніших уразливостей вебдодатків;
- розробляє стандарти та методики для оцінення та підвищення рівня безпеки вебдодатків;
- має велику та активну спільноту, де фахівці з різних країн обмінюються досвідом, знаннями та розробляють нові інструменти;
- контент OWASP доступний безкоштовно, що робить його доступним для широкого кола користувачів.

SANS Institute – це світовий лідер у галузі кібербезпеки. Це не просто організація, а ціла екосистема, яка об'єднує професіоналів з усього світу для підвищення рівня кібербезпеки. SANS:

- пропонує широкий спектр навчальних програм – від базових до дуже спеціалізованих курсів, що охоплюють різні аспекти кібербезпеки: етичний хакінг, цифрове розслідування, захист мереж, аналіз загроз тощо;
- після успішного завершення курсів видає престижні сертифікати, що підтверджують високий рівень знань і навичок у галузі кібербезпеки і які визнаються в усьому світі;
- постійно проводить дослідження в галузі кібербезпеки, виявляючи нові загрози та розробляючи методи їх протидії.

SAFEcode – це глобальна некомерційна організація, яка об'єднує лідерів бізнесу та технічних експертів для обміну ідеями та досвідом у створенні, вдосконаленні та просуванні масштабованих і ефективних програм безпеки програмного забезпечення.

OSS (Open Source Software) або програмне забезпечення з відкритим кодом – це тип програмного забезпечення, вихідний код якого доступний для загального використання і модифікації. Це означає, що будь-хто може:

- вивчати код: розуміти, як працює програма, і використовувати ці знання для навчання;
- змінювати код: адаптувати програму до своїх потреб, виправляти помилки або додавати нові функції;
- поширювати код: розповсюджувати змінений або незмінений код під тією ж ліцензією.

Just-In-Time (JIT) компіляція – це техніка, що використовується в багатьох сучасних мовах програмування та віртуальних машинах, яка дозволяє значно підвищити продуктивність виконання коду. На відміну від традиційної компіляції, яка перетворює весь вихідний код у машинний код перед виконанням програми, JIT-компіляція робить це «на льоту», тобто під час виконання програми.

Just Enough Access (JEA), або **принцип мінімально необхідного доступу**, є важливим поняттям в інформаційній безпеці та управлінні доступом. JEA означає, що кожен користувач або процес отримує лише ті дозволи, які строго необхідні для виконання своїх завдань. Іншими словами, доступ обмежується «саме тим, що необхідно».

Web Scraping – це автоматичне отримання даних із вебсторінок відповідно до заданих параметрів. Спеціальна програма сканує сайт та копіює його дані: тексти, зображення, аудіофайли тощо. Потім систематизує їх і зберігає, наприклад, таблицю формату CSV. Таким чином можна вивантажити цілий каталог інтернет-магазину, бібліотеку або будь-яку іншу базу даних.

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

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

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

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

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

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

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

Промислове шпигунство – *industrial espionage* – незаконне використання алгоритмів, що є інтелектуальною власністю автора, під час написання аналогів продукту.

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

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

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

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

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

Обфускування – *obfuscation* – це один з методів захисту програмного коду, який дозволяє ускладнити процес реверсивної інженерії коду програмного продукту, що захищається. Суть обфускування полягає в тому, щоб заплутати програмний код і усунути більшість логічних зв'язків в ньому, трансформувати його так, щоб вивчити і модифікувати його було складно.

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

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

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

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

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

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

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

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

Водяні знаки (Watermarking) на програмному забезпеченні – це приховані або явні мітки, інтегровані в програмний код або вихідні дані для підтвердження авторства чи забезпечення захисту від несанкціонованого використання.

Анти-тампериг (Tamper-proofing), або захист від втручання, – це набір методів та технологій, спрямованих на те, щоб унеможливити або значно ускладнити несанкціоновану модифікацію програмного забезпечення.

Електронне навчальне видання

Володимир Анатолієвич Гарнага,
Валентина Аполінаріївна Каплун,
Віталій Ігорович Селезньов

Інжиніринг захищеного програмного забезпечення

Навчальний посібник

Рукопис оформила *В. Каплун*
Редактор *Т. Старічек*
Оригінал-макет виготовила *Т. Старічек*

Підписано до видання 03.06.2026 р.
Гарнітура Times New Roman.
Зам. № Р2026-072.
Видавець та виготовлювач
Вінницький національний технічний університет,
Редакційно-видавничий відділ.
ВНТУ, ГНК, к. 114.
Хмельницьке шосе, 95,
м. Вінниця, 21021.
press.vntu.edu.ua;
Email: irvc.vntu@gmail.com
Свідоцтво суб'єкта видавничої справи
серія ДК № 3516 від 01.07.2009 р.