МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНА СЛУЖБА УКРАЇНИ З НАДЗВИЧАЙНИХ СИТУАЦІЙ
ЛЬВІВСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ БЕЗПЕКИ ЖИТТЄДІЯЛЬНОСТІ

Наталія МАСЛОВА, Максим НИКОЛАЙЧУК,
Валерія БАЛАЦЬКА, Євген СЄДІН

БЕЗПЕКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ У ПІДРОЗДІЛАХ ЦИВІЛЬНОГО ЗАХИСТУ

Навчальний посібник

Львів
2026

УДК 004.056(075.8)

Рецензенти: **Олег ГАРАСИМЧУК,** кандидат технічних наук, доцент, доцент кафедри захисту інформації Національного університету “Львівська політехніка”

Назарій БУРАК, кандидат технічних наук, доцент, начальник кафедри інформаційних технологій та систем електронних комунікацій Львівського державного університету безпеки життєдіяльності

Маслова Н.О., Николайчук М.І., Балацька В.С., Сєдін Є.О. *Безпека програмного забезпечення у підрозділах цивільного захисту: навч. посібник* / Львівський державний університет безпеки життєдіяльності. Львів : ЛДУБЖД, 2026. 456 с.

Навчальний посібник висвітлює ключові аспекти безпеки програмного забезпечення, що використовується у підрозділах цивільного захисту в умовах сучасної цифровізації та підвищених кіберзагроз. У ньому розглянуто життєвий цикл програмного забезпечення з позицій безпеки, загрози програмним реалізаціям, методи протидії реверс-інжинірингу, несанкціонованій модифікації коду, атакам на виконуване середовище, а також підходи Secure SDLC і DevSecOps.

Особливу увагу приділено практичним аспектам захисту програмного коду й даних, зокрема застосуванню маскуючих перетворень, аналізу виконуваних файлів, використанню інструментів статичного та динамічного аналізу, а також забезпеченню цілісності й доступності програмних систем у критичних умовах функціонування цивільного захисту.

Матеріал подано у доступній та систематизованій формі з використанням прикладів, схем і ілюстрацій, що сприяє кращому засвоєнню навчального матеріалу. Контрольні запитання, завдання для самоперевірки та практичні вправи дозволяють закріпити теоретичні знання та сформувати практичні навички аналізу й захисту програмного забезпечення.

Посібник призначений для студентів спеціальності F5 «Кібербезпека та захист інформації», суміжних ІТ-спеціальностей, а також для фахівців підрозділів цивільного захисту, які залучені до експлуатації, розробки та супроводу програмних систем і зацікавлені у підвищенні рівня їх захищеності.

ISBN 978-966-919-596-8

Рекомендовано рішенням Вченої ради Львівського державного університету безпеки життєдіяльності,

(Протокол № 5 від 17.12.2025)

© Маслова Н.О., 2026
© Николайчук М.І., 2026
© Балацька В.С., 2026
© Сєдін Є.О., 2026

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА ТЕРМІНІВ 9
ВСТУП 14
1. СТРАТЕГІЇ ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 16
1.1. АКТУАЛЬНІСТЬ АНАЛІЗУ ПРОГРАМ 16
1.2. ЗАКОНОДАВЧІ ОСНОВИ ЗАХИСТУ ПРОГРАМ 17
1.2.1. Основні терміни та визначення 17
1.2.2. Загрози, вразливості й ризики програмного забезпечення 21
1.2.3. Захист програм та даних як складова інформаційної безпеки 23
1.2.4. Модель інтеграції інформаційної безпеки у життєвий цикл програм 28
1.3. ЕТАПИ АНАЛІЗУ ПРОГРАМ 30
1.4. ПІДХОДИ ДО ДОСЛІДЖЕННЯ АЛГОРИТМІВ 33
1.5. ПРОЦЕС СТВОРЕННЯ ЗАХИЩЕНИХ РЕАЛІЗАЦІЙ 36
1.6. ПРИНЦИПИ ПОБУДОВИ СИСТЕМ ЗАХИСТУ 39
1.6.1. Основні підходи до створення захисних механізмів 42
1.6.2. Криптографічний і системний захист даних 45
1.6.3. Інтеграція вимог національного законодавства та практичних підходів до захисту програмного забезпечення у підрозділах цивільного захисту 46
1.7. СТРУКТУРА ПРОГРАМ, ЗАХИЩЕНИХ ВІД ДОСЛІДЖЕННЯ 47
1.8. МЕХАНІЗМИ ЗАХИСТУ ПРОГРАМНИХ РЕАЛІЗАЦІЙ 51
ПИТАННЯ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ ЗА ТЕМОЮ 1 57
ПРАКТИЧНІ ЗАВДАННЯ ДО ТЕМИ 1 58
ЗАВДАННЯ 1. Побудова моделі загроз для програмного забезпечення підрозділу цивільного захисту. 58
ЗАВДАННЯ 2. Аналіз підозрілого файлу за допомогою онлайн-інструментів безпеки. 60
ЗАВДАННЯ 3. Порівняльний аналіз підходів до захисту програмного забезпечення. 62
2. СУЧАСНИЙ СТАН ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 65
2.1. ОСОБЛИВОСТІ СУЧАСНИХ СИСТЕМ ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 65
2.1.1. Механізми перевірки цілісності програмних реалізацій 67
2.1.2. Контроль доступу та управління ідентифікацією 69
2.1.3. Програмні механізми захисту виконуваного коду 70
2.1.4. Захист програмного забезпечення під час виконання (runtime security) 73
2.1.5. Захист ланцюга постачання програмного забезпечення (supply chain security) 75
2.1.6. DRM у сучасних системах захисту програмного забезпечення 77
2.2. СИСТЕМА ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 80
2.3. DRM ТА СУЧАСНІ МЕХАНІЗМИ КІБЕРЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 84
2.4. ОСНОВНІ АЛГОРИТМИ ТА МЕТОДИ ЗАХИСТУ 87

2.5. ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ ВИМОГИ ДО ПОБУДОВИ СИСТЕМ ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 90
ПИТАННЯ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ ЗА ТЕМОЮ 2 94
ПРАКТИЧНІ ЗАВДАННЯ ДО ТЕМИ 2 95
ЗАВДАННЯ 1. Оцінювання комерційної системи захисту програмного забезпечення. ..95
ЗАВДАННЯ 2. Порівняльний аналіз алгоритмів та методів захисту програмного забезпечення. 97
ЗАВДАННЯ 3. Проєктування системи захисту програмного забезпечення для програмного комплексу цивільного захисту. 100
3. ЗАГАЛЬНІ ОЗНАКИ ВРАЗЛИВОСТІ ПРОГРАМ ТА ОЗНАКИ НЕСАНКЦІОНОВАНОГО ВТРУЧАННЯ 103
3.1 ВРАЗЛИВОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 103
3.1.1. Основні типи вразливостей та потенційні ризики безпеки 105
3.1.2. Ознаки несанкціонованого втручання 108
3.2. ПРОГРАМНІ ЗАКЛАДКИ 110
3.2.1. Визначення і класифікація програмних закладок 111
3.2.2. Типи програмних закладок 114
3.3. МОДЕЛІ ВЗАЄМОДІЇ ЗАКЛАДКИ З СИСТЕМОЮ 116
3.3.1. Модель «спостерігач» 117
3.3.2. Модель «перехоплювач» 120
3.3.3. Модель «спотворення» 121
3.3.4. Приклади відомих закладок 126
3.3.5. Методи виявлення та протидії 131
ПИТАННЯ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ ЗА ТЕМОЮ 3 133
ПРАКТИЧНІ ЗАВДАННЯ ДО ТЕМИ 3 134
ЗАВДАННЯ 1. Аналіз ознак несанкціонованого втручання в програмне забезпечення системи цивільного захисту 134
ЗАВДАННЯ 2. Порівняльний аналіз моделей програмних закладок 136
ЗАВДАННЯ 3. Аналіз вразливостей і методів впровадження програмних закладок. 139
4. ПЕРВИННИЙ АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 142
4.1. СУТНІСТЬ ПЕРВИННОГО АНАЛІЗУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 143
4.2. ІНСТРУМЕНТАЛЬНІ ЗАСОБИ ТА ТЕХНІКИ ПЕРВИННОГО АНАЛІЗУ 144
4.2.1. Можливості й приклади Hex-редакторів 146
4.2.2 Особливості й приклади PE-viewers 153
4.2.3 Спеціалізований інструмент CFF Explorer 158
4.3. ПЕРВИННИЙ АНАЛІЗ СТРУКТУРИ PE-ФАЙЛУ 160
4.4. ПРИКЛАДИ ЕКСПЕРИМЕНТІВ 165
4.4.1. Аналіз формату заголовка бінарного файлу даних. 165
4.4.2. Виявлення випадкового параметра у криптосистемі 168
4.4.3. Приклад проведення первинного аналізу виконуваного файлу 169

ПИТАННЯ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ ЗА ТЕМОЮ 4 175
ПРАКТИЧНІ ЗАВДАННЯ ДО ТЕМИ 4 175
ЗАВДАННЯ 1. ДОСЛІДЖЕННЯ СТРУКТУРИ БІНАРНОГО ФАЙЛУ МЕТОДОМ ПЕРВИННОГО АГАЛІЗУ 175
ЗАВДАННЯМ 2. ВИЗНАЧЕННЯ ХАРАКТЕРИСТИК КРИПТОСИСТЕМИ МЕТОДАМИ ПЕРВИННОГО АНАЛІЗУ 178
ЗАВДАННЯ 3. ПОРІВНЯЛЬНИЙ АНАЛІЗ HEX-РЕДАКТОРІВ ДЛЯ ЗАДАЧ ПЕРВИННОГО АНАЛІЗУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 181
5. СТАТИЧНИЙ АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 185
5.1. СУТНІСТЬ І ПРИНЦИПИ СТАТИЧНОГО АНАЛІЗУ 185
5.2. УЗАГАЛЬНЕНА МОДЕЛЬ ПРОЦЕСУ СТАТИЧНОГО АНАЛІЗУ 186
5.3. АНАЛІЗ СТРУКТУРИ ВИКОНУВАНИХ ФАЙЛІВ (PE) ЯК СКЛАДОВА СТАТИЧНОГО АНАЛІЗУ 189
5.3.1. DOS Header 192
5.3.2. PE Header (NT Headers) 196
5.3.3. Section Headers 202
5.4. ПОГЛИБЛЕНИЙ АНАЛІЗ PE-ФАЙЛІВ 206
5.4.1. Аналіз аномалій у структурі PE-файлів 206
5.4.2. Маніпулювання параметрами PE-файлу як метод аналізу 209
5.4.3. Дослідження впливу змін заголовка на поведінку програми 211
5.5. ІНСТРУМЕНТАЛЬНІ ЗАСОБИ СТАТИЧНОГО АНАЛІЗУ 214
5.5.1. Інструменти аналізу структури PE-файлів 214
5.5.2. Інструменти поглибленого аналізу PE-файлів 215
5.5.3. Дизасемблери 217
5.5.4. Статичні аналізатори коду 217
5.5.5. Сигнатурний аналіз 218
5.5.6. Довідкові ресурси 219
5.6. ПРИКЛАД ЗАСТОСУВАННЯ СТАТИЧНОГО МЕТОДУ 220
ПИТАННЯ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ ЗА ТЕМОЮ 5 227
ПРАКТИЧНІ ЗАВДАННЯ ДО ТЕМИ 5 227
ЗАВДАННЯ 1. СТАТИЧНИЙ АНАЛІЗ PE-ЗАГОЛОВКУ виконуваного файлу. 227
ЗАВДАННЯ 2. ВИЯВЛЕННЯ АНОМАЛІЙ У СТРУКТУРІ PE-файлу 230
ЗАВДАННЯ 3. ПОРІВНЯЛЬНИЙ АНАЛІЗ ІНСТРУМЕНТІВ СТАТИЧНОГО АНАЛІЗУ PE-ФАЙЛІВ. 233
6. ДОСЛІДЖЕННЯ ПРОГРАМНИХ ПРОДУКТІВ В ПРОЦЕСІ ВИКОНАННЯ 238
6.1. СУТНІСТЬ ДИНАМІЧНОГО АНАЛІЗУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 239
6.2. МЕТОДИКА ДОСЛІДЖЕННЯ ПРОГРАМ ДИНАМІЧНИМ МЕТОДОМ 241
6.2.1. Основні принципи й поняття 241
6.2.2. I етап – Метод маяків та Step-Trace першого етапу 244
6.2.3. II етап – Визначення функціональності функцій 248

6.2.4. III етап – Аналіз й дизасемблювання функцій 253
6.3. ІНСТРУМЕНТАЛЬНІ ЗАСОБИ ДИНАМІЧНОГО АНАЛІЗУ 257
6.4. ПЕРЕВАГИ ТА ОБМЕЖЕННЯ ДИНАМІЧНОГО АНАЛІЗУ 262
6.5. ПРАКТИЧНІ АСПЕКТИ ЗАСТОСУВАННЯ ДИНАМІЧНОГО АНАЛІЗУ 264
6.6. ПРИКЛАД ЗАСТОСУВАННЯ ДИНАМІЧНОГО МЕТОДУ 266
6.6.1. Постановка задачі та програмне середовище 266
6.6.2. Застосування методу маяків 269
6.6.3. Застосування методу Step-Trace першого етапу 272
6.6.4. Аналіз криптографічного алгоритму 273
6.6.5. Особливості застосування апаратних точок зупину 275
6.6.6. Висновки за результатами дослідження 275
ПИТАННЯ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ ЗА ТЕМОЮ 6 276
ПРАКТИЧНІ ЗАВДАННЯ ДО ТЕМИ 6 277
ЗАВДАННЯ 1. ТРИСТАДІЙНИЙ ДИНАМІЧНИЙ АНАЛІЗ МЕХАНИЗМУ ПЕРЕВІРКИ ЛІЦЕНЗІЇ. 277
ЗАВДАННЯ 2.ПОРІВНЯЛЬНИЙ АНАЛІЗ ІНСТРУМЕНТІВ ДИНАМІЧНОГО АНАЛІЗУ. 282
ЗАВДАННЯ 3. ДИНАМІЧНИЙ АНАЛІЗ ПІДОЗРІЛОГО КОМПОНЕНТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ У СЕРЕДОВИЩІ УМОВНОЇ СИСТЕМИ ЦИВІЛЬНОГО ЗАХИСТУ. 287
7. МЕТОДИ ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ВІД ЗАГРОЗ ТА НЕСАНКЦІОНОВАНОГО ДОСЛІДЖЕННЯ 292
7.1. ДИЗАСЕМБЛЮВАННЯ ТА АНАЛІЗ ВИКОНУВАНОГО КОДУ 292
7.1.1 Методи протидії дизасемблюванню (обфускація та anti-reversing) 294
7.1.2. Методи проти автоматичних дизасемблерів 295
7.1.3. Методи проти інтерактивних дизасемблерів 300
7.2. X-КОД ТА МЕХАНІЗМИ ЙОГО ВИКОРИСТАННЯ 303
7.2.1 X-код та вимоги до нього 303
7.2.2. Механізми функціонування X-коду 306
7.2.3. Методи впровадження X-коду 309
7.2.4. Запобіжні заходи та обмеження 315
7.3. ДИНАМІЧНІ ПРИЙОМИ ПРОТИДІЇ АНАЛІЗУ КОДУ 318
7.3.1. Динамічне саморозгортання та розшифрування коду 320
7.3.2. Anti-debugging-техніки та перевірка середовища виконання 321
7.3.3. Емуляційні та віртуалізаційні механізми 322
7.4. ОБФУСКАЦІЯ ПРОГРАМНОГО КОДУ 323
7.4.1. Основні поняття та класифікація методів обфускації 324
7.4.2. Обфускація даних 325
7.4.3. Обфускація та маскування графа керування програмою 331
7.4.4. Структурна та модульна обфускація 338

7.4.5. Динамічна обфускація 339
7.4.6. Оцінювання ефективності обфускації 340
7.4.7. Інструменти та приклади обфускаторів 341
7.5. КОМПЛЕКСНІ МЕТОДИ МАСКУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 344
7.5.1 Маніпулювання функціями 346
7.5.2. Недосяжний, мертвий та надлишковий код 348
7.5.3. Практичні рекомендації щодо застосування методів маскування 349
7.6. RUNTIME-МЕХАНІЗМИ ПРОТИДІЇ НАЛАГОДЖЕННЮ ТА АНАЛІЗУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 351
7.6.1. Основні механізми роботи visual studio debugger 352
7.6.2. Методи виявлення налагоджувача 354
7.6.3. Реакції програм на виявлення налагоджувача 358
7.6.4. Обмеження та практичні рекомендації щодо anti-debugging 360
ПИТАННЯ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ ЗА ТЕМОЮ 7 362
ПРАКТИЧНІ ЗАВДАННЯ ДО ТЕМИ 7 364
ЗАВДАННЯ 1. ЗАСТОСУВАННЯ МЕТОДІВ ОБФУСКАЦІЇ ДО ПРОГРАМНОГО КОДУ. 364
ЗАВДАННЯ 2. ОБФУСКАЦІЯ ГРАФА ПОТОКУ КЕРУВАННЯ ТА МАСКУЮЧІ ПЕРЕТВОРЕННЯ. 368
ЗАВДАННЯ 3. ПРОЄКТУВАННЯ АНТИДЕБАГ-ЗАХИСТУ ДЛЯ КОМПОНЕНТА СИСТЕМИ ЦИВІЛЬНОГО ЗАХИСТУ 374
ЗАВДАННЯ 4. КЛАСИФІКАЦІЯ ТА АНАЛІЗ МЕХАНІЗМІВ ВПРОВАДЖЕННЯ ЗАХИСНОГО КОДУ У PE-ФАЙЛ. 379
РОЗДІЛ 8. АПАРАТНО-ПРОГРАМНИЙ ЗАХИСТ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 383
8.1. АПАРАТНО-ПРОГРАМНІ МЕХАНІЗМИ ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 383
8.1.1. Особливості апаратно-програмного захисту програмного забезпечення 383
8.1.2. Trusted Platform Module у захисті програмного забезпечення 385
8.1.3. Secure Boot та формування ланцюга довіри 387
8.1.4. Trusted Execution Environment та захист середовища виконання 388
8.1.5. Апаратний захист від несанкціонованого копіювання 389
8.2. ПОТОКОВІ МЕХАНІЗМИ ТА СКРЕМБЛЕРИ У ЗАХИСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 391
8.2.1. Особливості використання скремблерів 392
8.2.2. LFSR як механізм формування псевдовипадкових послідовностей 397
8.2.3. Самосинхронізуючі скремблери 399
8.3. ПРОГРАМНА РЕАЛІЗАЦІЯ ПОТОКОВИХ МЕХАНІЗМІВ ЗАХИСТУ 400
8.3.1. Загальні принципи реалізації 401
8.3.2. Програмна реалізація LFSR-скремблера 403
8.3.3. Програмна реалізація scramble()/descramble() 405

8.3.4. Аналіз безпеки реалізації 407
8.4. ВИКОРИСТАННЯ АПАРАТНО-ПРОГРАМНИХ МЕХАНІЗМІВ У СИСТЕМАХ ЦИВІЛЬНОГО ЗАХИСТУ 409
8.5. ПРАКТИЧНІ ПРИКЛАДИ ЗАСТОСУВАННЯ АПАРАТНО-ПРОГРАМНОГО ЗАХИСТУ У СИСТЕМАХ ЦИВІЛЬНОГО ЗАХИСТУ 415
8.5.1. Захист систем оповіщення населення 415
8.5.2. Захист embedded-систем моніторингу надзвичайних ситуацій 418
8.5.3. Захист телеметричних і SCADA-систем 420
8.5.4 Захист ситуаційних центрів та систем підтримки прийняття рішень 423
8.5.5. Захист мобільних комплексів управління та реагування 426
8.5.6. Використання скремблерів у системах службового зв’язку та телеметрії 429
ПИТАННЯ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ за ТЕМОЮ 8 431
ПРАКТИЧНІ ЗАВДАННЯ ДО ТЕМИ 8 432
Практичне завдання 1. Проєктування лінійного регістра зсуву зі зворотними зв’язками для захисту каналу передачі даних ЦЗ-системи. 432
Практичне завдання 2. Реалізація та дослідження самосинхронізуючого скремблера для захисту потоку даних підрозділу ЦЗ. 436
ЛІТЕРАТУРА 446

ПЕРЕЛІК СКОРОЧЕНЬ ТА ТЕРМІНІВ

Скорочення	Значення
ACL	*Access Control List* – список контролю доступу
AES	*Advanced Encryption Standard* – стандарт симетричного шифрування
AES-256	варіант алгоритму AES із довжиною ключа 256 біт
API	*Application Programming Interface* – інтерфейс програмування застосунків
ASC	*Application Security Controls* – контрольні заходи безпеки застосунків
ASCII	*American Standard Code for Information Interchange* – стандарт кодування символів
ASLR	*Address Space Layout Randomization* – рандомізація розміщення адресного простору
ASMP	*Application Security Management Process* – процес управління безпекою застосунків
CBC	*Cipher Block Chaining* – режим зчеплення блоків шифротексту
C2	*Command and Control* – інфраструктура командування та керування
CFG	*Control Flow Graph* – граф потоку керування програми
CIA	*Confidentiality, Integrity, Availability* – конфіденційність, цілісність, доступність
CIA+AA	*Confidentiality, Integrity, Availability, Authenticity, Accountability* – конфіденційність, цілісність, доступність, автентичність і підзвітність
CI/CD	*Continuous Integration / Continuous Delivery (Deployment)* – безперервна інтеграція та доставка/розгортання
CIL	*Common Intermediate Language* – спільна проміжна мова платформи .NET
CLR	*Common Language Runtime* – середовище виконання .NET
CRC	*Cyclic Redundancy Check* – циклічний надлишковий код контролю цілісності
CRC-32	32-бітний алгоритм циклічного надлишкового контролю
CSRF	*Cross-Site Request Forgery* – міжсайтова підробка запиту
CTR	*Counter Mode* – лічильниковий режим роботи блокового шифру
CVE	*Common Vulnerabilities and Exposures* – система ідентифікації відомих вразливостей
CWE	*Common Weakness Enumeration* – класифікація типових слабких місць програмного забезпечення
DAST	*Dynamic Application Security Testing* – динамічне тестування безпеки застосунків

DEP	*Data Execution Prevention* – запобігання виконанню коду з областей даних
DLL	*Dynamic-Link Library* – динамічно підключувана бібліотека
DNS	*Domain Name System* – система доменних імен
DOS	*Disk Operating System*; у структурі PE – DOS Header
DRM	*Digital Rights Management* – керування цифровими правами
ECB	*Electronic Codebook* – режим електронної кодової книги
EDR	*Endpoint Detection and Response* – виявлення загроз і реагування на кінцевих пристроях
ENISA	*European Union Agency for Cybersecurity* – Агентство Європейського Союзу з кібербезпеки
FPGA	*Field-Programmable Gate Array* – програмована користувачем вентильна матриця
GCM	*Galois/Counter Mode* – режим автентифікованого шифрування
GDPR	*General Data Protection Regulation* – Загальний регламент ЄС про захист даних
GIS	*Geographic Information System* – геоінформаційна система
GUI	*Graphical User Interface* – графічний інтерфейс користувача
HMAC	*Hash-based Message Authentication Code* – код автентифікації повідомлення на основі хеш-функції
HSM	*Hardware Security Module* – апаратний модуль безпеки
HTML	*HyperText Markup Language* – мова гіпертекстової розмітки
HTTP	*HyperText Transfer Protocol* – протокол передавання гіпертексту
HTTPS	*HyperText Transfer Protocol Secure* – захищена версія HTTP
HWID	*Hardware Identifier* – апаратний ідентифікатор
IAT	*Import Address Table* – таблиця адрес імпорту PE-файлу
IDA	*Interactive DisAssembler* – програмний засіб дизасемблювання та аналізу виконуваного коду
IDS/IPS	*Intrusion Detection System / Intrusion Prevention System* – системи виявлення та запобігання вторгненням
IL	*Intermediate Language* – проміжна мова програмного коду
IOC	*Indicator of Compromise* – індикатор компрометації
ISMS	*Information Security Management System* – система управління інформаційною безпекою
ISO/IEC	*International Organization for Standardization / International Electrotechnical Commission* – Міжнародна організація зі стандартизації / Міжнародна електротехнічна комісія
ITSM	*Information Technology Service Management* – управління IT-послугами
IV	*Initialization Vector* – вектор ініціалізації
JTAG	*Joint Test Action Group* – стандартний апаратний інтерфейс тестування та налагодження

LFSR	*Linear Feedback Shift Register* – лінійний регістр зсуву зі зворотним зв’язком
MAC	*Mandatory Access Control* – мандатне керування доступом; у контексті мереж – *Media Access Control*
MD5	*Message Digest 5* – криптографічна хеш-функція, що вважається застарілою для задач безпеки
MFA	*Multi-Factor Authentication* – багатофакторна автентифікація
NIST	*National Institute of Standards and Technology* – Національний інститут стандартів і технологій США
NT	*New Technology*; NT Headers – заголовки формату PE
O-SSF	*Organization Normative Framework* – організаційна база політик і вимог
OTA	*Over-the-Air Update* – оновлення програмного забезпечення через бездротовий канал
OWASP	*Open Worldwide Application Security Project* – міжнародний проєкт у сфері безпеки застосунків
PE	*Portable Executable* – формат виконуваних файлів Windows
PE32 / PE64	32- та 64-розрядні варіанти формату Portable Executable
PEB	*Process Environment Block* – блок середовища процесу Windows
PKI	*Public Key Infrastructure* – інфраструктура відкритих ключів
PLC	*Programmable Logic Controller* – програмований логічний контролер
PPTP	*Point-to-Point Tunneling Protocol* – протокол тунелювання «точка-точка»
RASP	*Runtime Application Self-Protection* – самозахист застосунку під час виконання
RBAC	*Role-Based Access Control* – рольова модель керування доступом
RCE	*Remote Code Execution* – віддалене виконання коду
RSA	*Rivest–Shamir–Adleman* – асиметричний криптографічний алгоритм
RTF	*Rich Text Format* – формат збереження форматованого тексту
RTU	*Remote Terminal Unit* – віддалений термінальний пристрій
RVA	*Relative Virtual Address* – відносна віртуальна адреса у PE-файлі
SAST	*Static Application Security Testing* – статичне тестування безпеки застосунків
SBOM	*Software Bill of Materials* – перелік програмних компонентів і залежностей
SCA	*Software Composition Analysis* – аналіз складу програмного забезпечення
SCADA	*Supervisory Control and Data Acquisition* – система диспетчерського управління та збору даних

SDK	*Software Development Kit* – комплект засобів розробки програмного забезпечення
SDLC	*Software Development Life Cycle* – життєвий цикл розробки програмного забезпечення
SDR	*Software-Defined Radio* – програмно-визначувана радіосистема
SEH	*Structured Exception Handling* – структурована обробка винятків
SEV	*Secure Encrypted Virtualization* – технологія захищеної зашифрованої віртуалізації AMD
SGX	*Software Guard Extensions* – технологія ізольованого виконання Intel
SHA	*Secure Hash Algorithm* – сімейство криптографічних хеш-функцій
SHA-1	алгоритм сімейства SHA першого покоління
SHA-256	256-бітна хеш-функція сімейства SHA-2
SHA-3	сімейство криптографічних хеш-функцій стандарту SHA-3
SIEM	*Security Information and Event Management* – система управління інформацією та подіями безпеки
SOAR	*Security Orchestration, Automation and Response* – оркестрація, автоматизація та реагування на інциденти безпеки
SOC	*Security Operations Center* – центр операцій кібербезпеки
SQL	*Structured Query Language* – мова структурованих запитів
SSL/TLS	*Secure Sockets Layer / Transport Layer Security* – криптографічні протоколи захисту мережевого обміну
SWD	*Serial Wire Debug* – апаратний інтерфейс налагодження
TEE	*Trusted Execution Environment* – довірене ізольоване середовище виконання
TPM	*Trusted Platform Module* – довірений платформний модуль
USB	*Universal Serial Bus* – універсальна послідовна шина
UPX	*Ultimate Packer for eXecutables* – пакувальник виконуваних файлів
VPN	*Virtual Private Network* – віртуальна приватна мережа
XOR	*Exclusive OR* – логічна операція «виключне АБО»
XSS	*Cross-Site Scripting* – міжсайтове впровадження сценаріїв
YARA	інструмент і мова правил для ідентифікації шкідливого програмного забезпечення та ознак компрометації
ZAP	*Zed Attack Proxy* – інструмент OWASP для тестування безпеки вебзастосунків
АРМ	автоматизоване робоче місце
БД	база даних
ДСНС	Державна служба України з надзвичайних ситуацій
ДСТУ	державний стандарт України
ЖЦ	життєвий цикл
ІБ	інформаційна безпека

ІТ	інформаційні технології
КСЗІ	комплексна система захисту інформації
НД ТЗІ	нормативний документ системи технічного захисту інформації
НС	надзвичайна ситуація
НСД	несанкціонований доступ
ОС	операційна система
ПР	програмна реалізація
ПСП	псевдовипадкова послідовність
ПЗ	програмне забезпечення
ТЗІ	технічний захист інформації
ЦЗ	цивільний захист

ВСТУП

Цифровізація процесів управління, моніторингу та реагування в підрозділах цивільного захисту зумовлює широке використання програмного забезпечення різного рівня складності – від локальних прикладних систем до розподілених інформаційно-аналітичних платформ. Таке програмне забезпечення безпосередньо впливає на прийняття управлінських рішень у кризових ситуаціях, що висуває підвищені вимоги до його надійності, стійкості та захищеності. Порушення безпеки програмних реалізацій у цій сфері може призвести не лише до втрати або спотворення даних, але й до збоїв у функціонуванні критично важливих служб цивільного захисту.

Сучасні загрози інформаційній безпеці дедалі частіше орієнтовані не на теоретичні слабкі місця алгоритмів, а на вразливості їх практичної реалізації – помилки програмування, некоректну роботу з пам’яттю, недостатній контроль доступу, можливості аналізу, модифікації та втручання в процес виконання коду. У зв’язку з цим особливої актуальності набуває системний підхід до захисту програмних реалізацій, що поєднує організаційні, технічні та програмні методи, інтегровані в усі етапи життєвого циклу програмного забезпечення.

Метою посібника є формування системного уявлення про загрози програмним реалізаціям та набуття практичних навичок застосування сучасних методів захисту програмного забезпечення, зокрема в умовах функціонування підрозділів цивільного захисту. У посібнику розглядаються основні підходи до аналізу програмного забезпечення, принципи побудови захищених реалізацій, а також сучасні методи протидії атакам на програмний код і середовище його виконання.

Програмні реалізації, що використовуються в системах цивільного захисту, піддаються впливу широкого спектра загроз як навмисного, так і ненавмисного характеру. До основних загроз належать вразливості, спричинені помилками програмування, некоректною обробкою вхідних даних, відсутністю належної перевірки прав доступу та недостатнім контролем цілісності програмного коду. Окрему небезпеку становлять атаки, спрямовані на аналіз і

модифікацію виконуваних програм, зокрема статичний і динамічний аналіз, реверс-інжиніринг, підміна бібліотек, ін’єкції коду та несанкціоноване втручання в пам’ять процесу.

Для підрозділів цивільного захисту особливо критичними є також загрози, пов’язані з порушенням доступності програмних систем унаслідок збоїв, навмисних атак або експлуатації вразливостей, що може унеможливити своєчасне реагування на надзвичайні ситуації. Таким чином, захист програмних реалізацій у цій сфері має враховувати не лише конфіденційність і цілісність інформації, але й безперервність функціонування програмного забезпечення.

Особливістю захисту програмного забезпечення в підрозділах цивільного захисту є поєднання високих вимог до безпеки з необхідністю оперативності, надійності та простоти експлуатації. Програмні системи часто функціонують у гетерогенних середовищах, взаємодіють із зовнішніми інформаційними ресурсами та використовуються персоналом з різним рівнем підготовки, що підвищує ризик виникнення реалізаційних вразливостей.

У таких умовах доцільним є застосування підходів Secure SDLC та DevSecOps, які передбачають інтеграцію безпеки на всіх етапах життєвого циклу програмного забезпечення – від формування вимог і проєктування до розгортання та супроводу. Особлива увага приділяється захисту програмного коду від аналізу та модифікації, контролю цілісності, безпечному управлінню конфігураціями й оперативному реагуванню на виявлені вразливості. Застосування зазначених підходів дозволяє підвищити стійкість програмних систем цивільного захисту до сучасних кіберзагроз та забезпечити їх надійну роботу в умовах кризових ситуацій.

1. СТРАТЕГІЇ ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1. АКТУАЛЬНІСТЬ АНАЛІЗУ ПРОГРАМ

У цифрову епоху аналіз програм і даних є центральним елементом забезпечення інформаційної безпеки, надійності та ефективності програмного забезпечення [1; 3].

В умовах швидкого розвитку технологій, постійного оновлення коду та використання відкритих бібліотек, необхідно не лише створювати програмне забезпечення, але й контролювати його поведінку, оцінювати якість даних і виявляти потенційні загрози. Сучасні підходи до безпечної розробки передбачають інтеграцію механізмів безпеки на всіх етапах життєвого циклу програмного забезпечення – від формування вимог до супроводу та оновлення програмних продуктів [2; 5].

З розвитком цифрових технологій кількість атак на програмне забезпечення та інформаційні системи постійно зростає. Безпека даних і програм є ключовим елементом інформаційної безпеки держави, бізнесу та особистості. Вивчення стратегій і методів захисту дозволяє створювати надійні рішення, запобігати загрозам, адаптуватися до нових викликів і зменшувати ризики, пов’язані з експлуатацією програмного забезпечення [3; 4; 8].

Актуальними викликами є:

− зростання обсягу коду – сучасні програми можуть містити мільйони рядків, що ускладнює ручний контроль і тестування;

− інтеграція сторонніх бібліотек і API – відкриті компоненти часто містять приховані вразливості або шпигунські функції та можуть бути джерелом атак на ланцюги постачання програмного забезпечення (software supply chain attacks);

− кіберзагрози нового покоління – експлойти, zero-day атаки, supply-chain атаки використовують неочевидні недоліки в коді;

− ненадійність вхідних даних – уразливості часто виникають не в коді, а в обробці користувацьких чи мережевих даних;

− автоматизація злому – сучасні інструменти (BurpSuite, Metasploit, AI-based fuzzers) швидко виявляють слабкі місця у програмному забезпеченні.

Наведені виклики підтверджуються сучасними дослідженнями у сфері кібербезпеки, результатами аналізу актуальних вразливостей програмного забезпечення та тенденціями розвитку кіберзагроз [5; 6; 7].

Отже, своєчасний аналіз програм і даних – це не факультативна опція, а обов'язковий етап життєвого циклу будь-якого безпечного програмного продукту. Аналіз програмного забезпечення – це не тільки технічна перевірка, а й методологічна основа управління ризиками, спрямована на запобігання атакам, витоків даних і внутрішнім помилкам, що можуть призвести до серйозних порушень у роботі інформаційних систем [2; 4; 5].

1.2. ЗАКОНОДАВЧІ ОСНОВИ ЗАХИСТУ ПРОГРАМ

1.2.1. Основні терміни та визначення

Захист програмного забезпечення в Україні регламентується рядом ключових документів:

– Кодекс цивільного захисту України – визначає вимоги до функціонування систем управління у надзвичайних ситуаціях, де ПЗ є критичним елементом;

– Закон України «Про основні засади забезпечення кібербезпеки України» – встановлює принципи кіберзахисту державних інформаційних ресурсів;

– Закон України «Про захист інформації в інформаційно-комунікаційних системах» – регламентує вимоги до КСЗІ;

– НД ТЗІ 3.7-003-2023 – визначає порядок створення комплексної системи захисту інформації;

– ДСТУ ISO/IEC 27001:2023 – встановлює вимоги до систем управління інформаційною безпекою;

– ISO/IEC 27034 – Application Security, стандарт безпеки програмних систем.

Зазначені нормативні документи визначають вимоги до забезпечення безпеки програмного забезпечення, у тому числі в частині виявлення вразливостей, проведення аналізу програм та запобігання несанкціонованому втручанню [9-14].

Відповідно до цих документів, програмне забезпечення, що використовується у підрозділах цивільного захисту, повинно:

– проходити оцінку відповідності (атестацію КСЗІ);

– забезпечувати безперервність функціонування (особливо для систем реального часу);

– підтримувати механізми аудиту та журналювання;

– відповідати вимогам щодо захисту критичної інфраструктури.

Зазначені вимоги є складовою сучасного підходу до забезпечення інформаційної безпеки державних інформаційно-комунікаційних систем та об'єктів критичної інфраструктури [10-13].

Розглянемо більш детально особливості захисту програмного забезпечення за ISO/IEC 27034. Цей стандарт визначає модель управління безпекою програмних застосунків (ASMP), організаційну базу політик і вимог (O-SSF), контрольні заходи безпеки (ASC) й включає:

1) Визначення вимог безпеки до застосунку.

2) Впровадження компонентів безпеки у процес SDLC.

3) Аналіз уразливостей та перевірка відповідності.

4) Валідація під час розгортання.

Контрзаходи: контроль автентифікації, валідація вхідних даних, захист API, обмеження доступу до баз даних, журналювання, криптографічні протоколи.

Перелічені заходи відповідають рекомендаціям міжнародних стандартів безпеки застосунків та сучасним практикам безпечної розробки програмного забезпечення [14; 5; 6].

Для систематизації підходів до забезпечення безпеки програмного забезпечення на всіх етапах життєвого циклу розробки (SDLC) доцільно узагальнити вимоги стандарту ISO/IEC 27034, які визначають цілі безпеки, типові дії та відповідні інструменти їх реалізації (таблиця 1.1).

Таблиця 1.1

Основні засади ISO/IEC 27034

Етап SDLC	**Мета безпеки**	**Типові дії**	**Інструменти / стандарти**
Аналіз вимог	Ідентифікація ризиків і потреб у захисті	Моделювання загроз	ISO/IEC 27005, OWASP Threat Dragon
Проєктування	Формування безпечної архітектури	Визначення політик доступу, шифрування	ISO/IEC 27034
Розробка	Безпечне кодування	Перевірка вхідних даних, обмеження привілеїв	Secure Coding Guidelines
Тестування	Виявлення вразливостей	SAST, DAST, fuzzing	OWASP ZAP, SonarQube
Впровадження	Контроль доступу, цілісність	Сертифікація, цифровий підпис	ISMS, ISO/IEC 27001
Експлуатація	Підтримка, моніторинг	IDS/IPS, журналювання	SIEM, SOC
Вилучення	Безпечне видалення даних	Знищення ключів, очищення носіїв	GDPR, ISO/IEC 27040

Таблиця демонструє інтеграцію вимог безпеки в життєвий цикл програмного забезпечення відповідно до концепції SDLC та міжнародних стандартів управління безпекою застосунків [2; 14; 5].

Інформаційна безпека (ІБ) – це стан захищеності інформації, програм і систем, при якому забезпечуються основні властивості – CIA-тріада, де кожна буква має значення, пояснене нижче.

C – Confidentiality (конфіденційність) – дані та програмні компоненти доступні лише уповноваженим особам.

I – Integrity (цілісність) – програмний код і дані не можуть бути змінені без авторизації.

A – Availability (доступність) – системи працюють без збоїв, дані доступні у потрібний момент.

Концепція CIA-тріади є базовою моделлю забезпечення інформаційної безпеки та використовується в міжнародних стандартах і сучасних навчальних курсах з кібербезпеки [3; 8; 13].

Загальні принципи інформаційної безпеки:

– мінімальні привілеї – користувач має доступ лише до необхідних ресурсів.

– глибока оборона – багаторівнева система захисту (мережевий, прикладний, криптографічний рівні).

– безпека за замовчуванням – програма без явно дозволених дій вважає їх забороненими.

Об’єкт захисту – інформація, інформаційна система або ресурс, що потребує захисту.

Суб’єкт загрози – особа, група або програмний засіб, який може вплинути на об’єкт безпеки.

Інцидент безпеки – будь-яка подія, яка порушує політику або може призвести до порушення безпеки інформації.

Програмна реалізація (ПР) – це практичне втілення алгоритмів, моделей або функціональних вимог у вигляді програмного коду та пов’язаних із ним структур даних, конфігурацій і механізмів виконання, які забезпечують функціонування програмного забезпечення в конкретному апаратному та програмному середовищі.

Захист програм і даних – це комплекс організаційних, технічних і криптографічних заходів, спрямованих на запобігання, виявлення й усунення несанкціонованого доступу, модифікації або знищення програмного коду чи інформації.

Несанкціонований доступ до програмого продукту – це отримання можливості взаємодії з програмним забезпеченням, його даними або внутрішніми механізмами виконання з порушенням установлених політик безпеки, прав доступу або процедур автентифікації та авторизації.

Реверс-інжиніринг програмного забезпечення – це процес аналізу програмної реалізації з метою відновлення її логіки, структури, алгоритмів або протоколів функціонування без доступу до початкового вихідного коду, зазвичай шляхом статичного або динамічного аналізу виконуваних файлів.

Модифікація програмного коду – це цілеспрямована зміна програмної реалізації на рівні вихідного, проміжного або машинного коду, яка може призводити до порушення її цілісності, зміни функціональних властивостей або обходу вбудованих механізмів захисту.

Атака на програмну реалізацію – це сукупність дій, спрямованих на використання вразливостей практичної реалізації програмного забезпечення з метою порушення його конфіденційності, цілісності або доступності, незалежно від криптографічної чи алгоритмічної стійкості застосованих методів.

Наведені визначення узгоджуються з міжнародними підходами до управління безпекою застосунків та аналізу ризиків інформаційних систем [14; 15; 5].

Таким чином, нормативно-правова база формує основу для організації процесів аналізу програмного забезпечення як складової забезпечення інформаційної безпеки [9-15].

1.2.2. Загрози, вразливості й ризики програмного забезпечення

Загроза – потенційна подія або дія, яка може завдати шкоди системі чи даним. Приклад загрози – шкідливе ПЗ, SQL-ін’єкція, помилка конфігурації.

Вразливість – слабке місце в системі або програмі, яке може бути використане загрозою. Приклад вразливості – незахищені API, відкриті порти, відсутність оновлень.

Ризик – ймовірність реалізації загрози через вразливість і рівень потенційних збитків.

Аналіз загроз, вразливостей і ризиків є основою ефективного захисту програм та даних, оскільки дозволяє своєчасно виявляти слабкі місця, прогнозувати можливі атаки й обґрунтовано обирати заходи безпеки для мінімізації потенційних втрат [15; 16; 4].

Загрози програмного забезпечення (ПЗ) підлягають класифікації. Виділяють декілька типів класифікаторів, основними типами класифікаторів ПЗ є три.

1) За джерелом:

- внутрішні (інсайдери, помилки користувачів)
- зовнішні (хакери, конкуренти, бот-мережі).

2) За характером дії:

- технічні (віруси, експлойти, DoS-атаки)
- програмно-логічні (обхід автентифікації, SQL-ін’єкції)
- соціоінженерні (фішинг, підміна сторінок).

3) За наслідками:

- порушення конфіденційності;
- порушення цілісності;
- втрата доступності.

Загрози виникають, якщо є вразливі місця у програмному забезпеченні. Чотири основних типи вразливостей програмного забезпечення, це:

1) вразливості коду:

- буферні переповнення (Buffer Overflow);
- некоректна валідація вхідних даних;
- використання застарілих бібліотек.

2) вразливості конфігурацій:

- відкриті порти й облікові записи за замовчуванням;
- відсутність HTTPS;
- помилки в правах доступу.

3) вразливості архітектури:

- відсутність багаторівневого контролю доступу;
- неправильна логіка сесій;
- відсутність механізмів шифрування.

4) вразливості середовища виконання:

- відсутність оновлень ОС;
- невиправлені патчі;
- використання неліцензованого ПЗ.

Перелічені типи вразливостей є типовими категоріями слабких місць програмного забезпечення та регулярно фігурують у звітах щодо кіберзагроз і вразливостей застосунків [6; 7].

Нормативні підходи до визначення загроз і вразливостей безпосередньо використовуються при проведенні аналізу програмного забезпечення та оцінюванні його захищеності [15; 16].

1.2.3. Захист програм та даних як складова інформаційної безпеки

Програмне забезпечення є ключовим інструментом цифровізації сучасних галузей, забезпечуючи автоматизацію процесів управління, обробку даних, підтримку прийняття рішень та інтеграцію інформаційних ресурсів у єдине цифрове середовище. У різних сферах діяльності – від промисловості й транспорту до охорони здоров'я та цивільної безпеки – програмні системи виконують критично важливі функції, від коректності й безперервності роботи яких залежить ефективність функціонування організацій і безпека людей. У підрозділах цивільного захисту програмне забезпечення використовується для моніторингу обстановки, координації дій, аналізу ризиків і управління ресурсами в умовах надзвичайних ситуацій, що зумовлює підвищені вимоги до його надійності та захищеності. При цьому програмне забезпечення розглядається не як статичний продукт, а як динамічний об'єкт, що проходить повний життєвий цикл – від формування вимог і проєктування до розробки, тестування, розгортання та супроводу. На кожному з етапів життєвого циклу

програмного забезпечення закладаються або, навпаки, можуть виникати вразливості, що безпосередньо впливають на рівень його безпеки, тому системний підхід до управління безпекою ПЗ є необхідною умовою ефективної цифровізації, зокрема у сфері цивільної безпеки (рис.1.1). Необхідність інтеграції вимог безпеки на всіх етапах життєвого циклу програмного забезпечення підтверджується сучасними моделями безпечної розробки та міжнародними стандартами захисту застосунків [2; 5; 14].



Рисунок 1.1. Життєвий цикл програмного забезпечення

Діаграма ілюструє основні етапи життєвого циклу програмного забезпечення (SDLC): від аналізу вимог до підтримки в експлуатації. SDLC (Software Development Life Cycle) – це життєвий цикл розробки програмного забезпечення**,** тобто формалізована послідовність етапів створення, впровадження та супроводу ПЗ.

1. Планування – початковий етап життєвого циклу ПЗ, на якому визначаються цілі, обсяг функціональності, ресурси та базові вимоги до безпеки. Для систем цивільного захисту на цьому етапі також оцінюється критичність

функцій і ризики, що впливають на подальші рішення щодо захисту програмного забезпечення.

2. Аналіз вимог – формуються функціональні, нефункціональні і безпекові вимоги до ПЗ залежно від доменної області (включно з цивільною безпекою).

3. Проєктування (дизайн) – визначається архітектура системи та її компоненти з урахуванням вимог, зокрема безпеки.

4. Розробка (кодування) – реалізація функціональності у вигляді програмного коду відповідно до проєктної специфікації.

5. Тестування й інтеграція – перевірка коректності, якості і безпеки реалізації; виявлення дефектів до розгортання і впровадження програмного забезпечення в робоче середовище.

6. Супровід (support/maintenance) – експлуатація ПЗ з урахуванням подальших оновлень, виправлення дефектів, реагування на нові загрози та підтримка відповідності вимогам безпеки.

Розглянута модель життєвого циклу відповідає концепції Secure SDLC, у межах якої заходи безпеки впроваджуються на всіх етапах створення та супроводу програмного забезпечення [2; 5].

Безпека програмного забезпечення (Application Security) – це захист коду, процесів виконання і даних, що обробляються програмою.

Вона включає:

1) Аналіз вразливостей – пошук дефектів (OWASP, SAST, DAST).

2) Безпечне кодування – дотримання принципів Secure Coding.

3) Захист під час виконання – антивірусні засоби, sandbox, контроль доступу.

4) Обфускація та шифрування коду – для запобігання реверс-інжинірингу.

5) Захист даних – використання алгоритмів шифрування (AES, RSA, ChaCha20), хешування (SHA-3, Argon2), електронного підпису.

Перелічені механізми належать до базових технічних засобів забезпечення безпеки програмного забезпечення та даних [3; 8; 13].

Основні напрямки захисту даних:

- у стані зберігання (encryption-at-rest);

- під час передавання (TLS, VPN);

- у процесі обробки (secure enclaves, homomorphic encryption).

Типовими атаками на програмне забезпечення вважаються:

- SQL Injection – отримання доступу до бази даних;
- XSS – впровадження JavaScript-коду;
- CSRF – підміна дій користувача;
- Buffer Overflow – виконання довільного коду;
- RCE – повний контроль над системою.

Типи, механізми дії на наслідки атак зібрано в таблиці 1.2.

Таблиця 1.2

Приклади типових атак на програмне забезпечення

Тип атаки	**Механізм**	**Наслідки**
SQL Injection	Вставка коду SQL у запити	Витік даних
Cross-Site Scripting (XSS)	Впровадження JavaScript-коду	Викрадення cookies
CSRF	Підміна дій користувача	Несанкціоновані транзакції
Buffer Overflow	Запис за межами буфера	Виконання довільного коду
Remote Code Execution	Віддалене виконання команд	Повний контроль над системою
Hooking	Перехоплення викликів функцій або системних API	Модифікація поведінки програми, обхід захисту
Patching	Модифікація програмного коду	Відключення захисних механізмів, зміна логіки роботи

Типові атаки, наведені у таблиці, входять до переліку найбільш поширених загроз для сучасних програмних систем і вебзастосунків [6; 7].

З точки зору захисту безпосередньо програмного коду найбільш критичними для програмного забезпечення є атаки, спрямовані на аналіз, модифікацію та втручання в процес виконання програмної реалізації, зокрема статичний і динамічний аналіз, реверс-інжиніринг, ін’єкції коду, hooking та

patching, які дозволяють обходити вбудовані механізми захисту незалежно від коректності застосованих алгоритмів.

Атаки на програмні реалізації спрямовані на використання вразливостей практичного втілення програмного забезпечення незалежно від коректності застосованих алгоритмів або криптографічних примітивів. Такі атаки орієнтовані на аналіз, модифікацію або порушення процесу виконання програмного коду та можуть призводити до компрометації конфіденційності, цілісності або доступності програмних систем. Особливу небезпеку вони становлять для програмного забезпечення, що використовується в підрозділах цивільного захисту, де збої або підміна функціональності можуть мати критичні наслідки. Це статичний аналіз, динамічний аналіз коду, реверс-інжиніринг, ін’єкції, Hooking та Patching. Зазначені методи активно використовуються як під час легітимного аналізу безпеки програмного забезпечення, так і під час атак, спрямованих на дослідження або модифікацію програмних реалізацій [1; 4].

Статичний аналіз програмної реалізації полягає у дослідженні програмного коду або виконуваних файлів без їх запуску. У контексті атак він використовується для виявлення логіки роботи програми, структури коду, використаних алгоритмів, бібліотек і потенційних вразливостей. Зловмисники можуть застосовувати дизасемблювання, декомпіляцію та аналіз залежностей для пошуку слабких місць, таких як жорстко закодовані ключі, помилки обробки даних або некоректні перевірки доступу.

Динамічний аналіз передбачає дослідження поведінки програмного забезпечення під час виконання. Він дозволяє спостерігати за взаємодією програми з операційною системою, пам’яттю, файлами та мережевими ресурсами. У межах атак динамічний аналіз застосовується для обходу механізмів захисту, перехоплення викликів функцій, аналізу обміну даними та виявлення умов, за яких виникають помилки або некоректна поведінка програми.

Реверс-інжиніринг є комплексним підходом до аналізу програмних реалізацій, який поєднує методи статичного та динамічного аналізу з метою відновлення внутрішньої логіки, алгоритмів і протоколів роботи програмного

забезпечення. У результаті реверс-інжинірингу можуть бути з'ясовані механізми захисту, ліцензування або перевірки цілісності, що створює передумови для їх обходу або модифікації.

Ін'єкційні атаки полягають у введенні шкідливих або неочікуваних даних у програму з метою зміни її поведінки. У контексті програмних реалізацій це може бути ін'єкція коду, команд або даних у пам'ять процесу, що дозволяє виконувати довільні дії або отримувати підвищені привілеї. Причинами таких атак часто є відсутність належної валідації вхідних даних або помилки роботи з пам'яттю.

Hooking – це метод втручання в процес виконання програми шляхом перехоплення або підміни викликів функцій, системних API чи бібліотек. За допомогою hooking зловмисник може змінювати логіку роботи програми без безпосередньої модифікації її основного коду, що ускладнює виявлення втручання та дозволяє обходити механізми контролю.

Patching у контексті атак означає внесення змін до виконуваного коду або його сегментів з метою зміни або відключення певних функцій, зокрема перевірок ліцензії, автентифікації або цілісності. Такі модифікації можуть здійснюватися як на диску, так і безпосередньо в оперативній пам'яті під час виконання програми, що створює серйозні загрози для безпеки програмних реалізацій.

Забезпечення інформаційної безпеки програмного забезпечення передбачає обов'язкове застосування методів його аналізу на різних етапах життєвого циклу [2; 5; 14].

1.2.4. Модель інтеграції інформаційної безпеки у життєвий цикл програм

Модель інтеграції інформаційної безпеки у життєвий цикл програм передбачає вбудовування механізмів захисту на кожному етапі розроблення – від проєктування до експлуатації та оновлення – для забезпечення стійкості й надійності програмного продукту. ІБ має бути інтегрована у кожний етап SDLC:

1) визначення вимог безпеки;

2) проєктування безпечної архітектури;

3) кодування з дотриманням стандартів безпеки;

4) тестування (SAST, DAST, fuzzing);

5) впровадження і розгортання із застосуванням CI/CD контролю;

6) моніторинг, журналювання, реагування на інциденти;

7) деактивація і знищення даних.

Такий підхід відповідає принципам Secure SDLC та рекомендаціям міжнародних стандартів щодо безпечної розробки програмного забезпечення [2; 5; 14].

Програмне забезпечення, що використовується у підрозділах цивільного захисту, характеризується підвищеними вимогами до надійності, безперервності функціонування та стійкості до зовнішніх і внутрішніх загроз, оскільки його робота безпосередньо впливає на ефективність реагування на надзвичайні ситуації та безпеку населення. Такі програмні системи часто функціонують у розподілених та гетерогенних середовищах, інтегрують дані з різних джерел, взаємодіють із суміжними інформаційними ресурсами та експлуатуються в умовах обмеженого часу на прийняття рішень. У зв’язку з цим до програмного забезпечення цивільного захисту висуваються підвищені вимоги щодо забезпечення конфіденційності, цілісності та доступності інформації, захисту програмних реалізацій від реверс-інжинірингу й несанкціонованої модифікації, стійкості до відмов і атак на доступність, а також можливості оперативного виявлення та усунення вразливостей протягом усього життєвого циклу програмного забезпечення. З урахуванням SDLC (див. таблицю 1.3), інтеграція вимог безпеки має вигляд:

Таблиця 1.3

Інтеграція нормативних вимог у життєвий цикл ПЗ

Етап SDLC	**Нормативні вимоги**	**Практична реалізація**
Планування	Класифікація системи як критичної	Визначення рівня захисту (КСЗІ)
Аналіз вимог	Закон про кібербезпеку	Формування security requirements

Етап SDLC	Нормативні вимоги	Практична реалізація
Проєктування	НД ТЗІ	Threat modeling, сегментація
Розробка	Secure Coding	Контроль вхідних даних
Тестування	ISO 27001	SAST/DAST, аудит
Експлуатація	КСЗІ, SOC	Моніторинг, SIEM
Супровід	Постійне вдосконалення	Patch management

Таблиця відображає взаємозв'язок між нормативними вимогами та практичними заходами забезпечення безпеки програмного забезпечення на різних етапах життєвого циклу [9-15].

У межах життєвого циклу програмного забезпечення аналіз програм виступає невід'ємним етапом, що забезпечує відповідність реалізації вимогам безпеки. Практична реалізація зазначених вимог передбачає застосування методів аналізу програмного забезпечення, що розглядаються у наступному підрозділі [5; 14].

1.3. ЕТАПИ АНАЛІЗУ ПРОГРАМ

Аналіз програмного забезпечення є важливим етапом у забезпеченні його коректності, надійності та захищеності, особливо в контексті критично важливих систем цивільного захисту. Метою аналізу програм є виявлення особливостей функціонування програмної реалізації, потенційних вразливостей, помилок проєктування та реалізації, а також оцінка стійкості програмного забезпечення до можливих загроз [2; 5].

Процес аналізу доцільно розглядати як послідовність взаємопов'язаних етапів, що забезпечують поступове поглиблення знань про програмний об'єкт – від загального ознайомлення до детального дослідження поведінки та механізмів захисту [17; 18; 19].

Основними є вісім описаних нижче етапів аналізу програмного забезпечення.

1) Первинний аналіз (попереднє дослідження)

Передбачає швидке дослідження програмної реалізації без її виконання з метою отримання базової інформації про об’єкт. На цьому етапі визначається тип файлу, аналізуються метадані, структура виконуваного файлу, наявність підозрілих ознак (висока ентропія, обфускація, вбудовані ресурси, рядки тощо).

Результати первинного аналізу дозволяють сформувати уявлення про призначення програми, оцінити рівень потенційної загрози та визначити подальшу стратегію дослідження.

2) Підготовчий етап

Включає ознайомлення з документацією (за наявності), постановку цілей аналізу, визначення інструментів і середовища дослідження (ізольоване середовище, sandbox, віртуальна машина), а також формування плану подальших дій.

3) Статичний аналіз

Полягає у дослідженні програмного коду або виконуваних файлів без їх запуску. Дозволяє виявити структуру програми, використані алгоритми, бібліотеки, а також потенційні вразливості (жорстко закодовані ключі, помилки логіки, небезпечні виклики функцій).

Основними інструментами є дизасемблери, декомпілятори та аналізатори коду.

4) Динамічний аналіз

Передбачає дослідження поведінки програмного забезпечення під час виконання у контрольованому середовищі. Дозволяє спостерігати за взаємодією програми з операційною системою, пам’яттю, файлами та мережею, а також виявляти приховані механізми та умови активації функціоналу.

Застосовуються налагоджувачі, системи моніторингу, sandbox-середовища.

5) Експериментальний аналіз (поведінкове дослідження)

Є окремою методикою дослідження, що базується на взаємодії з програмою як із «чорним ящиком». Передбачає зміну вхідних даних, умов

виконання або середовища функціонування з метою виявлення закономірностей у поведінці програми.

На відміну від первинного аналізу, цей етап передбачає виконання програми та активне втручання в процес її функціонування.

6) Відновлення алгоритмів і логіки роботи

На основі результатів статичного та динамічного аналізу здійснюється реконструкція алгоритмів, структури та внутрішньої логіки програмної реалізації, що є ключовим завданням реверс-інжинірингу [17; 19].

7) Перевірка захищеності

Передбачає тестування механізмів безпеки, виявлення вразливостей, аналіз можливостей обходу захисту та оцінювання стійкості програмного забезпечення до атак [4; 5].

8) Модифікація та повторне тестування

Включає внесення змін (у разі дослідження власного ПЗ або в навчальних цілях), усунення вразливостей та перевірку ефективності реалізованих механізмів захисту.



Рисунок 1.2. Етапи аналізу програмного забезпечення

Схема ілюструє послідовність проведення аналізу програмної реалізації – від первинного дослідження об’єкта до перевірки захищеності та тестування модифікованої програми. Аналіз має ітеративний характер і може повторюватися на різних етапах життєвого циклу програмного забезпечення. Подібний підхід відповідає сучасним методологіям аналізу програмного забезпечення, реверс-інжинірингу та безпечної розробки програмних систем [2; 5; 17–19]. Послідовне проходження зазначених етапів дозволяє системно оцінити програмну реалізацію, виявити слабкі місця та підвищити рівень її безпеки, що є особливо важливим для систем цивільного захисту [5; 18].

1.4. ПІДХОДИ ДО ДОСЛІДЖЕННЯ АЛГОРИТМІВ

Дослідження алгоритмів функціонування програмного забезпечення є ключовим елементом аналізу програмних реалізацій, що дозволяє визначити логіку роботи програми, виявити потенційні вразливості та оцінити ефективність механізмів захисту [2; 5].

На відміну від етапів аналізу, які визначають послідовність виконання дослідження, підходи до аналізу характеризують способи, умови та рівень доступу до програмного об’єкта. Класифікація підходів дозволяє систематизувати методи аналізу та обґрунтовано обирати інструменти дослідження залежно від поставленої мети [17; 19].

Основні підходи до дослідження алгоритмів програмного забезпечення класифікують за такими ознаками:

1) рівнем доступу до програмної реалізації;

2) способом проведення аналізу;

3) характером взаємодії з програмою;

4) метою дослідження.

При застосуванні класифікації ***за рівнем доступу*** виділяють аналіз з повним доступом (White-box), аналіз без доступу до внутрішньої структури (Black-box) та частковий доступ (Gray-box).

Аналіз з повним доступом передбачає наявність повної інформації про програму, включаючи вихідний код, документацію та архітектуру. Дозволяє

здійснювати глибокий аналіз алгоритмів, перевіряти логіку реалізації та виявляти вразливості на ранніх етапах розробки.

При проведенні аналізу без доступу до внутрішньої структури програма розглядається як «чорний ящик», досліджується лише її поведінка у відповідь на вхідні дані. Використовується для виявлення зовнішніх проявів вразливостей без знання внутрішньої реалізації.

Частковий доступ поєднує властивості попередніх підходів: дослідник має обмежену інформацію про внутрішню структуру програми (наприклад, API або архітектуру), що дозволяє більш ефективно аналізувати поведінку системи.

Підходи White-box, Black-box і Gray-box широко застосовуються під час аналізу безпеки, тестування та дослідження програмних реалізацій різного рівня складності [5; 19].

При класифікації ***за способом проведення*** аналізу виділяють три типи підходів (первинний, статичний, динамічний, гібридний).

Первинний аналіз полягає у початковому дослідженні програмного забезпечення без глибокого втручання у його внутрішню структуру з метою отримання загального уявлення про будову, поведінку, механізми захисту та можливі ознаки несанкціонованого втручання.

Статичний підхід передбачає дослідження програмної реалізації без її виконання. Дозволяє аналізувати структуру коду, алгоритми, залежності та потенційні вразливості. Є ефективним для виявлення логічних помилок і небезпечних конструкцій.

Динамічний підхід полягає у дослідженні поведінки програми під час виконання. Дозволяє виявляти вразливості, що проявляються лише у процесі роботи, а також аналізувати взаємодію з операційною системою, пам’яттю та мережею.

Гібридний підхід поєднує статичний і динамічний аналіз, що дозволяє отримати більш повне уявлення про програмну реалізацію та підвищити ефективність виявлення вразливостей.

Зазначені підходи становлять основу сучасних методик аналізу програмного забезпечення, реверс-інжинірингу та дослідження шкідливого програмного забезпечення [17-19].

Для аналізу програмного забезпечення на різних етапах дослідження застосовуються спеціалізовані програмно-інструментальні засоби, класифікація яких наведена у таблиці 1.4.

Таблиця 1.4

Інструментальні засоби аналізу програм

Клас	**Інструменти**	**Призначення**
Первинний аналіз	Hex-редактори, PE-viewers, CFF Explorer	Аналіз структури файлу, заголовків, сигнатур, часових міток
Статичний аналіз	Дизасемблери, декомпілятори (IDA Free, Ghidra)	Аналіз коду без виконання
Динамічний аналіз	Дебагери (x64dbg, OllyDbg), ProcMon	Дослідження поведінки під час виконання
Гібридний аналіз	Sandbox, монітори активності, EDR	Комплексний аналіз взаємодії із системою

Таблиця відображає типові категорії інструментальних засобів, що використовуються на різних етапах аналізу програмної реалізації та дослідження її поведінки [17-19].

Класифікатор **за характером взаємодії** з програмою включає пасивний аналіз, що здійснюється без втручання у процес виконання програми, та активний аналіз. Пасивний аналіз включає первинний та статичний аналіз, під час яких не змінюється поведінка досліджуваного об’єкта, а активний аналіз передбачає взаємодію з програмою під час її виконання, зміну вхідних даних або умов середовища. До нього належать динамічний та експериментальний аналіз.

Класифікація за метою дослідження передбачає проведення аналізу безпеки, реверс-інжинірингу, тестування та аналізу шкідливого програмного забезпечення.

Аналіз безпеки спрямований на виявлення вразливостей, помилок реалізації та оцінювання стійкості програмного забезпечення до атак.

Метою етапу реверс-інжинірингу є відновлення алгоритмів, структури та логіки роботи програми без доступу до вихідного коду.

Тестування та валідація передбачає перевірку відповідності програмного забезпечення функціональним і нефункціональним вимогам.

Аналіз шкідливого програмного забезпечення спрямований на виявлення ознак шкідливої активності, механізмів приховування та способів поширення програм. Залежно від поставленої мети дослідження можуть комбінуватися різні підходи та інструменти аналізу, що дозволяє отримати максимально повне уявлення про програмну реалізацію та її безпеку.

Застосування різних підходів до дослідження алгоритмів програмного забезпечення дозволяє отримати багатовимірне уявлення про програмну реалізацію, що є необхідною умовою ефективного забезпечення її безпеки. У практиці аналізу програм, зокрема у сфері цивільного захисту, доцільним є комбінування підходів залежно від умов експлуатації, доступності інформації та рівня критичності програмного продукту.

Класифікація підходів до аналізу створює методичну основу для подальшого вивчення спеціалізованих методів дослідження, зокрема статичного та динамічного аналізу програмного забезпечення, які детально розглядаються у наступних розділах.

1.5. ПРОЦЕС СТВОРЕННЯ ЗАХИЩЕНИХ РЕАЛІЗАЦІЙ

Створення безпечних програмних реалізацій передбачає: визначення вимог до безпеки; виявлення вразливостей; розробку та впровадження механізмів протидії загрозам; перевірку ефективності захисту [2; 5; 14], (рис.1.3).



Рисунок 1.3. Сутність процесу створення захищених реалізацій

Процес забезпечення безпеки є ітеративним та включає визначення вимог, виявлення вразливостей, впровадження механізмів захисту та перевірку їх ефективності з подальшим уточненням вимог. Подібний підхід відповідає сучасним моделям життєвого циклу безпечної розробки програмного забезпечення та концепції безперервного вдосконалення безпеки [2; 5].

Сучасним напрямом у сфері захисту програм та даних є DevSecOps, що передбачає інтеграцію механізмів безпеки безпосередньо в процеси розробки та експлуатації програмних систем. DevSecOps – це безперервний цикл, у якому безпека інтегрована на всіх етапах життєвого циклу програмного забезпечення: від планування і розробки до експлуатації, моніторингу та постійного вдосконалення захисту. Такий підхід передбачає інтеграцію механізмів безпеки безпосередньо в процеси розробки та експлуатації програмних систем [20; 5].

У межах DevSecOps-підходу забезпечення захищеності програмних реалізацій розглядається як безперервний процес, інтегрований у всі етапи життєвого циклу програмного забезпечення. На початковому етапі здійснюється визначення потреби в безпеці на основі CIA-тріади (конфіденційність, цілісність, доступність), що дозволяє формалізувати вимоги до захисту відповідно до

призначення програмного продукту та моделі загроз [4; 14]. Далі виконується обробка вразливостей, яка включає їх ідентифікацію, аналіз ризиків та визначення пріоритетів усунення. На етапі проєктування безпека закладається на архітектурному рівні шляхом моделювання загроз із використанням спеціалізованих інструментів, зокрема OWASP Threat Dragon, що дозволяє виявити потенційні вектори атак ще до реалізації коду [4; 14]. Подальша розробка здійснюється з дотриманням принципів безпечного програмування, орієнтованих на мінімізацію реалізаційних помилок та зменшення поверхні атаки. Реалізація засобів захисту охоплює впровадження механізмів контролю доступу, перевірки цілісності, захисту пам’яті та протидії реверс-інжинірингу. Обов’язковим етапом є тестування безпеки, що поєднує статичний і динамічний аналіз коду, а також автоматизоване тестування веб-застосунків за допомогою інструментів типу OWASP ZAP [6; 5]. Завершальним елементом DevSecOps-циклу є підтримка безпеки в розгорнутому програмному забезпеченні, яка включає моніторинг, оновлення компонентів, реагування на нові вразливості та постійне вдосконалення захисних механізмів (рис.1.4), [20; 13].



Рисунок 1.4. DevSecOps-підхід

Діаграма ілюструє інтеграцію безпеки в усі етапи життєвого циклу програмного забезпечення: від формування вимог і проєктування до розробки, тестування, розгортання та експлуатації. Безпекові практики (threat modeling, secure coding, статичний і динамічний аналіз, моніторинг) виконуються безперервно та автоматизовано, що відповідає принципам DevSecOps.

1.6. ПРИНЦИПИ ПОБУДОВИ СИСТЕМ ЗАХИСТУ

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

Системою захисту від несанкціонованого доступу (*system of protection against unauthorized access*) називають комплекс програмних засобів, що забезпечують ускладнення або заборону нелегального розповсюдження, використання і/або зміни програмних продуктів.

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

З*ламом програми* (*breaking program, крекінг*) називають порушення функціональності об'єктів захисту програмного забезпечення (програм).

Для аналізу об'єктів захисту використовуються переважно два основні види засобів:

1) налагоджувачі, які дозволяють контролювати виконання програм і таким чином прослідкувати весь алгоритм на практиці;

2) дизасемблери, що дозволяють одержувати асемблерний лістинг програми для його подальшого статичного вивчення.

Якою б мовою не була написана програма, які б способи захисту не використовувались, все, зрештою, перетвориться на команди процесора.

Для аналізу алгоритму роботи програми порушнику в більшості випадків не важливо, за допомогою чого вона була створена.

Отримавши код програми, порушник та захисник бачать асемблерні команди, тож необхідно знати асемблер, розбиратися в роботі процесора і в архітектурі комп'ютерної системи.

Розробка захищеного програмного забезпечення ґрунтується на використанні базових принципів безпеки, які забезпечують надійність, стійкість до атак і керованість системи в процесі її експлуатації. Дотримання цих принципів дозволяє мінімізувати кількість вразливостей, зменшити поверхню атаки та забезпечити контроль доступу до критичних ресурсів [5; 13; 14]. Основні принципи, якими повинен керуватися розробник при створенні захищених реалізацій, наведено нижче (рисунок 1.5).

Простота (Simplicity). Механізми захисту повинні бути максимально простими, зрозумілими та добре перевіреними. Складні й надмірно заплутані рішення підвищують імовірність помилок у реалізації, що може призвести до появи нових вразливостей. Простота також сприяє ефективному аудиту коду, тестуванню та супроводженню програмного забезпечення. У практиці це означає

використання стандартних, перевірених бібліотек і алгоритмів замість власних складних реалізацій.



Рисунок 1.5. Основні принципи створення захищених реалізацій

Відокремленість (Isolation). Система повинна бути організована таким чином, щоб мінімізувати спільне використання ресурсів між різними користувачами або компонентами. Це дозволяє обмежити можливість несанкціонованого доступу та поширення наслідків компрометації. Відокремленість реалізується через розмежування доступу, ізоляцію процесів, використання контейнеризації та розділення середовищ виконання (production, testing, development).

Розподіл повноважень (Separation of Privilege). Доступ до критичних функцій системи повинен вимагати виконання кількох умов або використання кількох незалежних механізмів контролю. Це може включати багатофакторну автентифікацію, розподіл ролей користувачів або необхідність підтвердження дій декількома особами. Такий підхід знижує ризик несанкціонованого доступу навіть у разі компрометації одного з елементів захисту.

Відсутність доступу в звичайному режимі (Least Privilege / Default Deny). За замовчуванням доступ до ресурсів повинен бути заборонений, а механізми доступу – активуватися лише за необхідності. Користувач або процес отримує лише ті права, які є необхідними для виконання конкретного завдання, і лише на обмежений час. Це дозволяє суттєво зменшити поверхню атаки та обмежити можливості зловмисника у випадку компрометації облікового запису або системного процесу.

Незалежність захисту від секретності реалізації (Security through Design, а не через Obscurity). Ефективність механізмів захисту не повинна базуватися на приховуванні алгоритмів або внутрішньої логіки системи. Навпаки, система має залишатися захищеною навіть у разі, якщо її архітектура або код стають відомими. Цей принцип передбачає використання відкритих, перевірених криптографічних алгоритмів і стандартів. Наведені принципи є основою сучасних підходів до побудови захищених інформаційних систем і відповідають класичним принципам захисту інформації, сформульованим у фундаментальних роботах з комп'ютерної безпеки [21; 14].

Застосування наведених принципів у комплексі забезпечує формування стійкої до атак програмної системи, здатної протистояти як відомим, так і новим загрозам. Вони лежать в основі сучасних підходів до забезпечення безпеки програмного забезпечення, зокрема DevSecOps, і є обов'язковими для врахування при розробці критичних інформаційних систем.

1.6.1. Основні підходи до створення захисних механізмів

Захисні механізми програмного забезпечення формуються шляхом поєднання різнорівневих методів захисту, які можуть реалізовуватися як безпосередньо в програмному коді, так і на рівні виконуваного середовища або мережевої інфраструктури. Вибір способу створення захисту визначається критичністю програмного продукту, можливими загрозами, вимогами до продуктивності та складністю потенційного обходу захисних механізмів [5; 14]. На практиці застосовують такі основні підходи (рис.1.6).

1) *Вбудований захист*

Вбудований захист передбачає інтеграцію захисних перевірок безпосередньо у вихідний код програми. До нього належать криптографічні методи (шифрування коду, контроль цілісності за допомогою хеш-функцій, електронний підпис), програмні методи (обфускація, ліцензійні перевірки, робота з цифровими сертифікатами), а також системні перевірки (контроль прав доступу, виявлення налагоджувачів). Такий підхід забезпечує тісний зв’язок захисту з логікою програми, але ускладнює супровід і налагодження.

2) *Навісний захист*

Навісний захист реалізується шляхом додавання захисних механізмів до вже скомпільованого виконуваного файлу без зміни його вихідного коду. Зазвичай він використовується у вигляді пакувальників, захисних оболонок або зовнішніх модулів ліцензування. Навісний захист часто поєднується з криптографічним шифруванням, перевірками цифрового підпису та мережевими механізмами автентифікації, що дозволяє швидко підвищити рівень захищеності готового продукту.

3) *Шифрування виконуваного файлу*

Шифрування виконуваного файлу передбачає, що програма запускається не безпосередньо, а через спеціальний завантажувач (loader), який у процесі виконання розшифровує необхідні ділянки коду. Цей підхід широко використовується для захисту від статичного аналізу і дизасемблювання та часто поєднується з контролем цілісності, обфускацією і антидебаг-техніками. У сучасних системах такий механізм може додатково використовувати можливості ОС (наприклад, криптографічні API Windows).

4) *Комбінований захист*

Комбінований захист є найбільш ефективним підходом, оскільки поєднує кілька методів одночасно: криптографічні (шифрування, хешування, цифрові підписи), програмні (обфускація, ліцензування), системні (ACL, RBAC, антивірусний контроль) і мережеві (SSL/TLS, VPN, міжмережеві екрани). Такий підхід реалізує принцип багаторівневої оборони (defense in depth) та суттєво

підвищує вартість і складність обходу захисту для зловмисника. Концепція багаторівневого захисту є одним із базових принципів сучасної інформаційної безпеки та широко використовується під час проєктування критичних програмних систем [5; 13; 20].

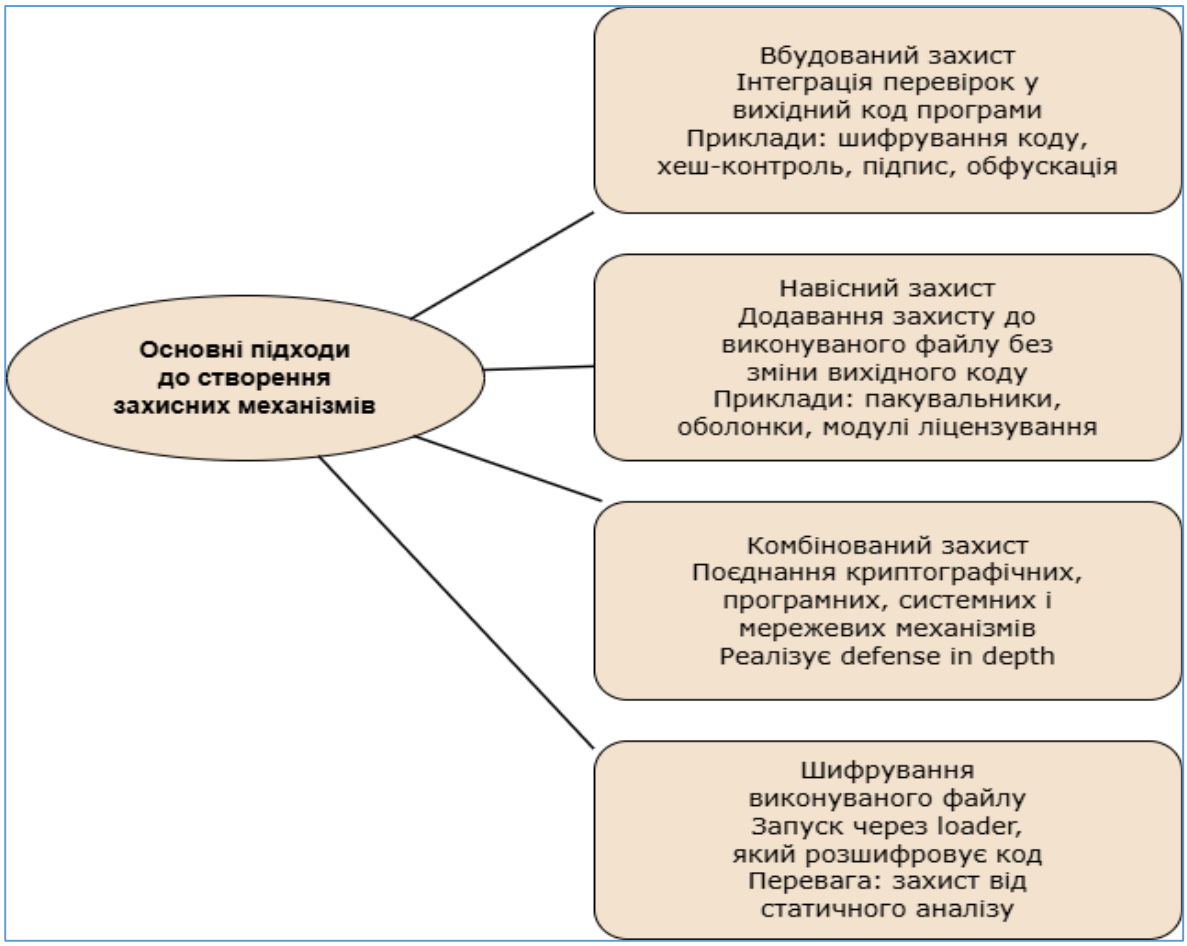


Рисунок 1.6. Підходи до створення захисних механізмів

Рисунок узагальнює основні способи реалізації захисних механізмів програмного забезпечення та демонструє можливість їх комбінованого використання для підвищення рівня захищеності [14; 5].

Вибір підходу визначається критичністю ПЗ, характером загроз, вимогами до продуктивності та складністю потенційного обходу механізмів захисту.

Наведемо програмний ілюстративний код проведення контролю цілісності файлу, що у контексті цивільного захисту може застосовуватися для перевірки незмінності програмних модулів систем оповіщення або координації рятувальних служб.

Код програмного блоку контролю цілісності файлу:

```
import hashlib
def verify_integrity(file_path, expected_hash):
  with open(file_path, "rb") as f:
    data = f.read()
  return hashlib.sha256(data).hexdigest() == expected_hash
```

Наведений фрагмент коду реалізує базовий механізм контролю цілісності файлу шляхом обчислення його криптографічного хешу та порівняння з еталонним значенням. Для цього використовується алгоритм хешування SHA-256, який формує унікальний відбиток (hash) для вмісту файлу.

На першому етапі файл відкривається у двійковому режимі, після чого зчитується його вміст. Далі за допомогою функції хешування обчислюється контрольна сума, яка порівнюється з наперед відомим (очікуваним) значенням. Якщо обчислений хеш збігається з еталонним, це означає, що файл не зазнав змін; у протилежному випадку – його цілісність порушена.

Такий підхід широко застосовується для перевірки незмінності виконуваних файлів, контролю бібліотек і залежностей, виявлення несанкціонованих модифікацій програмного коду. Контроль цілісності є одним із базових механізмів забезпечення довіри до програмного забезпечення та широко використовується в сучасних системах безпечної розробки і супроводу програмних продуктів [5; 13].

У системах цивільного захисту механізми контролю цілісності є критично важливими, оскільки дозволяють гарантувати, що програмне забезпечення, яке використовується для управління надзвичайними ситуаціями, не було змінене або скомпрометоване. Це, у свою чергу, забезпечує достовірність обробки даних і надійність прийняття рішень.

1.6.2. Криптографічний і системний захист даних

Криптографічні технології є ключовим інструментом забезпечення безпеки під час аналізу програм і даних. Вони застосовуються для шифрування передавання (TLS/SSL, VPN), зберігання (AES, Kalyna, RSA), а також для

контролю незмінності файлів і кодів за допомогою хеш-функцій (SHA-3, BLAKE2, Streebog).

У системах аналітики даних криптографія запобігає витокам під час обробки великих обсягів інформації, забезпечуючи захист персональних і конфіденційних відомостей. У поєднанні з методами цифрового підпису та сертифікації (PKI, DSS) криптографічні засоби формують довіру до результатів аналізу програм і даних, унеможливлюючи їх несанкціоновану модифікацію або підробку [13; 14].

Приклади криптографічних методів:

– симетричні – AES, GOST, Kalyna;
– асиметричні – RSA, ECC, ElGamal;
– хеш-функції – SHA-3, BLAKE2, Streebog.

Системні методи захисту призначені для контролю доступу, забезпечення цілісності, надійності та стійкості програмного середовища до несанкціонованих дій або збоїв. До системних методів відносяться [13; 5]:

– контроль доступу (RBAC, DAC, MAC);
– журналювання дій користувачів;
– антивірусний захист і цілісність файлів (MD5/SHA перевірки);
– резервне копіювання й відновлення (backup & recovery).

1.6.3. Інтеграція вимог національного законодавства та практичних підходів до захисту програмного забезпечення у підрозділах цивільного захисту

На відміну від звичайних ІТ-систем, ПЗ у підрозділах цивільного захисту має додаткові вимоги:

– жорсткі обмеження часу реагування (real-time);
– висока критичність доступності (Availability > Confidentiality у деяких сценаріях);

– робота в умовах нестабільного зв’язку, фізичних пошкоджень інфраструктури, інтеграція з різнорідними системами (датчики, GIS, мобільні пристрої).

Це означає, що класичні підходи ІБ мають адаптуватися до вимог оперативності та стійкості до відмов, а не лише до захисту даних [11; 13].

Для підвищення рівня захищеності програмного забезпечення у системах цивільного захисту доцільно впроваджувати такі заходи:

1. Організаційні: впровадження ISMS відповідно до ISO/IEC 27001; розробка політик реагування на інциденти; регулярне навчання персоналу (антифішинг, кібергігієна).

2. Технічні: використання багаторівневої автентифікації (MFA); сегментація мереж (відокремлення критичних систем); впровадження SIEM/SOC для моніторингу; використання захищених каналів (VPN, TLS).

3. Програмні: обов’язкове застосування SAST / DAST, fuzzing-тестування, контроль залежностей (supply-chain security), застосування обфускації для критичних модулів.

4. Експлуатаційні: регулярне оновлення ПЗ та ОС; резервне копіювання (backup + disaster recovery); тестування на відмовостійкість (failover).

Перелічені заходи відповідають сучасним міжнародним практикам управління інформаційною безпекою, безпечної розробки програмного забезпечення та забезпечення кіберстійкості критичних систем [5; 13; 20].

Таким чином, захист програмного забезпечення у сфері цивільного захисту є комплексною задачею, що поєднує технічні методи, стандарти та законодавчі вимоги.

Ключовим є інтегрований підхід (DevSecOps + КСЗІ + ISMS), при цьому пріоритетом виступає доступність і безперервність роботи систем, що відрізняє ці системи від класичних ІТ-рішень.

1.7. СТРУКТУРА ПРОГРАМ, ЗАХИЩЕНИХ ВІД ДОСЛІДЖЕННЯ

У п. 1.6 було розглянуто принципи та підходи до побудови систем захисту

програмного забезпечення. На їх основі формується практична структура програмних реалізацій, що забезпечують протидію несанкціонованому доступу, аналізу та модифікації.

Захист програмного забезпечення від дослідження у сучасних умовах розглядається як складова Application Security, яка включає не лише ускладнення аналізу коду, але й контроль процесу виконання, цілісності та поведінки програмної системи [14; 5].

Сучасна захищена програма має багаторівневу структуру, що включає такі основні компоненти:

- ініціалізаційний модуль (Secure Loader);
- основна функціональна частина (Core Logic);
- модуль контролю цілісності (Integrity Control);
- модуль моніторингу виконання (Runtime Protection / RASP);
- модуль журналювання та аудиту (Logging & Audit);
- модуль завершення (Secure Termination).

Подібна модульна організація захисних механізмів відповідає принципам багаторівневого захисту та безпечної архітектури програмного забезпечення [21; 14].

Функціями ініціалізаційного модуля є перевірка цілісності виконуваного файлу (hash, цифровий підпис), перевірка середовища виконання, завантаження критичних компонентів.

Прикладом реалізації етапу ініціалізації у підрозділах ДСНС є перевірка цифрового підпису виконуваного файлу під час запуску програмного комплексу координації рятувальних служб. Якщо файл змінений – запуск блокується.

Основними задачами функціональної частини є реалізація алгоритмів, обробка даних та взаємодія з користувачем і зовнішніми системами.

З точки зору захисту інформації виконується валідація вхідних даних, проводиться контроль доступу та ізоляція критичних функцій.

Приклад практичної реалізації цього етапу – у вебінтерфейсі диспетчера перевіряються всі дані, щоб запобігти SQL Injection або XSS [6].

Функціями модуля контролю цілісності (Integrity Control) є перевірка незмінності коду під час виконання; контроль підключених бібліотек; виявлення модифікацій. В якості прикладу наведемо випадок перевірки хеш-сум DLL перед завантаженням у систему моніторингу надзвичайних ситуацій.

Модуль моніторингу виконання (Runtime Protection / RASP) аналізує поведінку програми, виявляє аномалії, блокує підозрілі дії. Так, наприклад, якщо система фіксує спробу ін'єкції коду або аномальний виклик API – виконання запиту блокується. Механізми Runtime Application Self-Protection (RASP) забезпечують контроль поведінки застосунку безпосередньо під час виконання та дозволяють виявляти окремі типи атак у реальному часі [22].

Модуль журналювання та аудиту (Logging & Audit) фіксує подій безпеки, проводить аналіз інцидентів, реалізує інтеграцію з системами моніторингу. Усі дії диспетчера (вхід, зміна статусу виклику) записуються в журнал і можуть аналізуватися у SIEM [23; 13].

На етапі завершення (Secure Termination) очищується пам'ять, завершуються сесії, зберігаються логи. Й, безумовно, після завершення роботи програми видаляються ключі шифрування з оперативної пам'яті.

Багаторівнева архітектура є одним із базових принципів забезпечення безпеки програмного забезпечення, який передбачає реалізацію захисних механізмів на кількох взаємопов'язаних рівнях (рис. 1.7). Такий підхід забезпечує комплексний захист системи, оскільки заходи безпеки впроваджуються не в одному окремому компоненті, а розподіляються по всій її структурі. У сучасних програмних системах додатково застосовуються механізми контролю доступу, криптографічного захисту даних, а також засоби аудиту та моніторингу подій безпеки, зокрема з використанням систем класу SIEM [13; 5; 20].

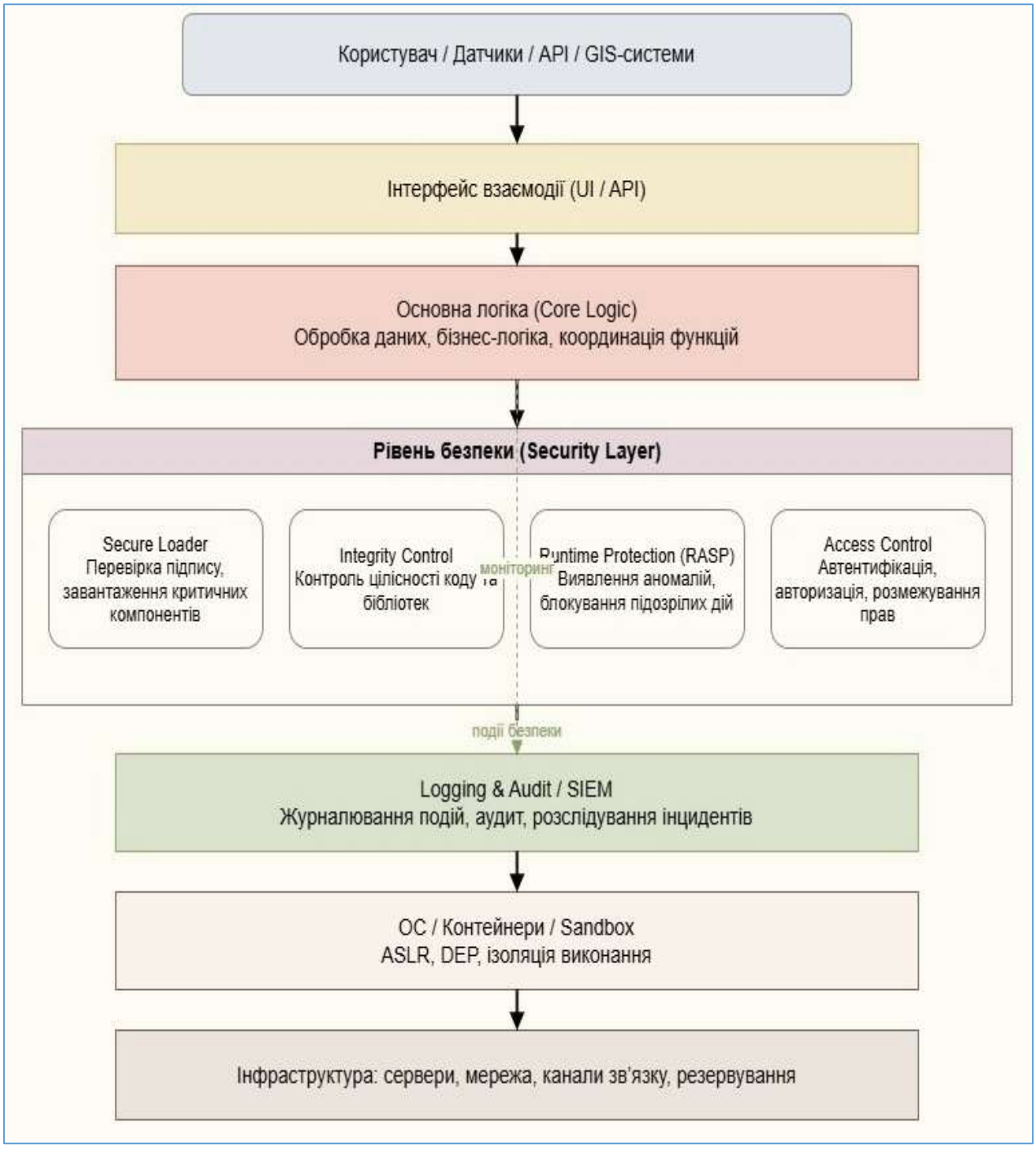


Рисунок 1.7. Багаторівнева структура захищеної програмної реалізації

У системах цивільного захисту така багаторівнева архітектура забезпечує контроль доступу до критичних функцій, захист даних під час передачі та обробки, а також можливість оперативного моніторингу і реагування на інциденти. Багаторівневий підхід до побудови захищених програмних реалізацій відповідає сучасним практикам DevSecOps та принципу Defense in Depth, відповідно до якого захисні механізми реалізуються на кількох взаємодоповнюючих рівнях [20; 21].

1.8. МЕХАНІЗМИ ЗАХИСТУ ПРОГРАМНИХ РЕАЛІЗАЦІЙ

У сучасних умовах захист програмного забезпечення реалізується на основі багаторівневого підходу, який поєднує механізми захисту коду, захист середовища виконання та ланцюга постачання програмних компонентів [5; 14; 20].

До основних методів захисту програмного коду (Code Protection), які ускладнюють аналіз і модифікацію програм належать:

- обфускація (заплутування структури коду);
- шифрування окремих фрагментів;
- цифровий підпис виконуваних файлів;
- контроль цілісності виконуваних файлів;
- моніторинг активності.

Перелічені механізми належать до базових засобів Code Protection та використовуються для підвищення стійкості програмного забезпечення до аналізу, модифікації та несанкціонованого використання [5; 14].

Захист середовища виконання – другий механізм захисту. Він реалізується за допомогою ASLR (рандомізація адрес пам’яті); DEP (заборона виконання даних); ізоляції (sandbox, контейнеризація).

Прикладом застосування цього механізму є виконання підозрілих компонентів в ізольованому середовищі для запобігання впливу на основну систему. У контексті цивільного захисту застосування таких методів дозволяє безпечно тестувати оновлення або сторонні модулі без ризику порушення роботи критичних систем [26].

Захист від дослідження та модифікації передбачає використання спеціальних механізмів, спрямованих на ускладнення аналізу програмного забезпечення та запобігання його несанкціонованим змінам. До таких механізмів належать засоби протидії налагодженню (anti-debugging), виявлення та блокування спроб модифікації коду (anti-tampering), захист від підміни бібліотек, а також протидія технікам перехоплення викликів функцій (hooking).

Наприклад, програмне забезпечення може визначати факт запуску під налагоджувачем і, у разі його виявлення, обмежувати або припиняти виконання. У системах цивільного захисту такі механізми доцільно застосовувати насамперед для критичних програмних модулів, однак їх використання має бути збалансованим і не повинно ускладнювати супровід, оновлення та аудит програмного забезпечення. Такі механізми використовуються для протидії аналізу та модифікації програмних реалізацій і є складовою сучасних підходів до Application Security [14; 5].

Захист ланцюга постачання програмного забезпечення (Supply Chain Security) спрямований на забезпечення контролю та безпеки сторонніх компонентів, що використовуються під час розробки і експлуатації програмних систем. Це включає формування переліку всіх залежностей у вигляді SBOM (Software Bill of Materials), перевірку цифрових підписів бібліотек і компонентів, а також контроль оновлень із застосуванням процедур patch management.

Наприклад, перед інтеграцією сторонніх Python-бібліотек у систему аналізу ризиків здійснюється перевірка їх походження, цілісності та наявності відомих вразливостей. У контексті цивільного захисту такі заходи дозволяють запобігти впровадженню вразливостей через зовнішні компоненти у критично важливі інформаційні системи. Контроль залежностей, використання SBOM та перевірка походження програмних компонентів є ключовими елементами сучасної концепції Supply Chain Security [24; 25].

Сучасні загрози спрямовані на експлуатацію вразливостей під час виконання програм, тому важливо застосовувати механізми динамічного захисту. Основні підходи:

– RASP (Runtime Application Self-Protection) – вбудований захист, що аналізує поведінку програми у реальному часі;

– контроль системних викликів;

– виявлення ін’єкцій (SQL, code injection);

– поведінковий аналіз.

Зазначені механізми забезпечують виявлення та блокування атак безпосередньо під час виконання програмного забезпечення [22].

Розглянемо захист програмного забезпечення під час виконання (Runtime Security) – простий ілюстративний приклад (проста перевірка вхідних даних). Приклад кодового блоку перевірки вхідних даних:

```
def validate_input(user_input):
  if not user_input.isalnum():
    raise ValueError("Invalid input detected")
  return user_input
```

Функція, яка наведена вище, реалізує базову перевірку вхідних даних користувача щоб уникнути використання небезпечних або некоректних значень. Функція validate_input приймає рядок user_input і перевіряє, чи складається він виключно з літер і цифр за допомогою методу isalnum().

Якщо вхідний рядок містить будь-які інші символи (наприклад, спеціальні знаки, пробіли або керівні символи), функція генерує виняток ValueError з повідомленням про виявлення некоректного введення. У випадку, коли перевірка проходить успішно, функція повертає вхідне значення без змін.

Такий підхід дозволяє:

– обмежити формат вхідних даних;

– зменшити ризик ін’єкційних атак (наприклад, SQL Injection або командних ін’єкцій);

– підвищити надійність обробки даних у програмі.

Слід зазначити, що ця перевірка є спрощеною і підходить для випадків, коли допустимі лише алфавітно-цифрові значення. У реальних системах, особливо в умовах складних форматів введення, доцільно застосовувати більш гнучкі механізми валідації (регулярні вирази, контекстно-залежні правила, бібліотеки перевірки даних). Перевірка вхідних даних є одним із базових механізмів протидії ін’єкційним атакам та входить до рекомендацій OWASP щодо безпечної розробки програмного забезпечення [6].

У системах цивільного захисту перевірка вхідних даних є важливою складовою безпеки, оскільки дозволяє запобігти введенню некоректної або шкідливої інформації в критичні програмні модулі, що може вплинути на прийняття рішень або функціонування систем реагування. Незважаючи на простоту, така перевірка може ефективно захистити інтерфейси диспетчерських систем від ін’єкцій та некоректних даних.

Ефективний захист програмного забезпечення повинен враховувати не лише рівень безпеки, але й вимоги до продуктивності, супроводжуваності та надійності.

Надмірне використання механізмів захисту (обфускація, шифрування, антидебаг) ускладнює внесення змін, знижує продуктивність, ускладнює аудит і тестування.

Тому при проєктуванні систем необхідно забезпечити баланс між рівнем захищеності, швидкодією, можливістю оновлення та прозорістю для адміністрування.

Ефективним механізмом, що не ускладнює систему, але значно підвищує рівень контролю, є журналювання. Далі наведено ілюстративний приклад (журналювання подій):

```
import logging
logging.basicConfig(filename="system.log", level=logging.INFO)

def log_event(event):
  logging.info(event)
```

Наведений фрагмент коду реалізує базовий механізм журналювання подій за допомогою стандартного модуля logging мови Python. На початковому етапі виконується налаштування системи журналювання за допомогою функції basicConfig, де визначається файл для збереження журналу (system.log) та рівень реєстрації подій (INFO). Це означає, що у файл будуть записуватися інформаційні повідомлення та всі події вищого рівня важливості (наприклад, попередження або помилки).

Функція log_event приймає повідомлення event і передає його до журналу через виклик logging.info(). У результаті кожна подія фіксується у файлі журналу, що дозволяє відстежувати дії користувачів, стан системи та можливі інциденти безпеки.

Журналювання є важливим елементом забезпечення безпеки програмного забезпечення, оскільки воно:

– забезпечує можливість аудиту дій користувачів і системних процесів;
– дозволяє виявляти аномалії та підозрілу активність;
– підтримує розслідування інцидентів безпеки;
– сприяє контролю відповідності політикам безпеки.

У системах цивільного захисту журналювання подій має особливе значення, оскільки дозволяє фіксувати всі критичні дії (наприклад, обробку викликів, зміну статусу подій, доступ до системи) та забезпечує можливість оперативного аналізу ситуації під час надзвичайних подій. Це сприяє підвищенню прозорості роботи систем і забезпеченню контролю за діями персоналу. Журналювання та аудит є важливими складовими моніторингу безпеки, реагування на інциденти та забезпечення підзвітності дій користувачів і системних процесів [23; 13].

Незалежно від способу реалізації, захисні механізми повинні відповідати таким вимогам:

– висока складність обходу і відсутність єдиної точки компрометації;
– неможливість простого редагування або блокування захисних фрагментів;
– мінімальний вплив на продуктивність і стабільність програми;
– стійкість до статичного й динамічного аналізу;
– сумісність із системними та мережевими засобами безпеки

Перелічені вимоги відповідають принципам побудови захищених програмних систем і забезпечення їх стійкості до сучасних кіберзагроз [5; 14].

У сучасних умовах забезпечення безпеки програмного забезпечення неможливе без використання спеціалізованих професійних інструментів, які

дозволяють автоматизувати процеси виявлення загроз, аналізу вразливостей і реагування на інциденти. Такі інструменти охоплюють різні аспекти кіберзахисту – від аналізу шкідливого програмного забезпечення та перевірки вебресурсів до дослідження мережевої інфраструктури та тестування безпеки застосунків.

Розповсюдженими професійними програмними інструментами ефективного захисту вважаються: VirusTotal, Shodan, OWASP ZAP, Polyswarm, Windows Sandbox, CheckPhish.ai, Hybrid Analysis, App.Any.Run, DNSDumpster. Таблиця 1.5 узагальнює зазначені інструменти за їх призначенням, функціональними можливостями та сферами застосування. Це дозволяє систематизувати знання про сучасні засоби кіберзахисту та обрати відповідні інструменти залежно від задачі – аналізу програмного забезпечення, тестування безпеки або моніторингу мережевих загроз.

Таблиця 1.5

Професійні програмні інструменти захисту

№	Інструмент	Призначення
1	VirusTotal	Аналіз підозрілих файлів
2	Shodan	Виявлення відкритих сервісів
3	OWASP ZAP	Тестування веббезпеки
4	Polyswarm	Перевірка шкідливого ПЗ
5	Windows Sandbox	Безпечне виконання коду
6	CheckPhish.ai	Виявлення фішингу
7	Hybrid Analysis	Автоматичний динамічний аналіз
8	App.Any.Run	Аналіз поведінки програм
9	DNSDumpster	Перевірка мережевих структур

Таблиця відображає різні категорії інструментів, що використовуються на практиці:

– інструменти аналізу файлів і шкідливого ПЗ (VirusTotal, Hybrid Analysis, Any.Run, Polyswarm);

– засоби дослідження мережевої інфраструктури та відкритих сервісів (Shodan, DNSDumpster);

– інструменти тестування безпеки вебзастосунків (OWASP ZAP);

– середовища ізольованого виконання (Windows Sandbox);

– інструменти виявлення фішингових ресурсів (CheckPhish.ai).

Застосування цих засобів дозволяє реалізувати комплексний підхід до захисту, який включає як превентивні заходи (виявлення вразливостей, аналіз залежностей), так і реактивні (аналіз інцидентів, дослідження шкідливої активності). Використання спеціалізованих інструментів аналізу, моніторингу та тестування безпеки є невід'ємною складовою сучасних процесів DevSecOps і забезпечення кіберстійкості програмних систем [20; 5].

У контексті систем цивільного захисту використання таких інструментів є особливо важливим, оскільки вони дозволяють оперативно виявляти загрози, оцінювати стан інформаційних ресурсів і забезпечувати захист критично важливих програмних систем, що використовуються для реагування на надзвичайні ситуації. Комплексне використання механізмів захисту програмного забезпечення дозволяє забезпечити необхідний рівень безпеки, надійності та безперервності функціонування інформаційних систем цивільного захисту [13; 20; 24].

ПИТАННЯ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ ЗА ТЕМОЮ 1:

1. Що таке багаторівнева архітектура безпеки програмного забезпечення та які її основні переваги?

2. Які основні принципи створення захищених реалізацій програмного забезпечення?

3. У чому полягає підхід DevSecOps та як він інтегрує безпеку у життєвий цикл програмного забезпечення?

4. Які основні підходи до створення захисних механізмів програмного забезпечення існують?

5. Що таке захист під час виконання (Runtime Security) і які механізми до нього належать?

6. У чому полягає захист від дослідження та модифікації програмного забезпечення?

7. Що таке захист ланцюга постачання (Supply Chain Security) та які його основні елементи?

8. Що таке контроль цілісності програмного забезпечення та які основні способи його реалізації?

9. Які професійні інструменти використовуються для забезпечення безпеки програмного забезпечення?

10. Які особливості забезпечення безпеки програмного забезпечення в системах цивільного захисту наведені в розділі?

ПРАКТИЧНІ ЗАВДАННЯ ДО ТЕМИ 1

ЗАВДАННЯ 1. Побудова моделі загроз для програмного забезпечення підрозділу цивільного захисту

Постановка завдання:

Оберіть один із наведених типів програмних систем, що застосовуються у підрозділах цивільного захисту:

а) система моніторингу надзвичайних ситуацій;

б) автоматизована система координації аварійно-рятувальних служб;

в) програмний комплекс аналізу ризиків і прогнозування.

Для обраного типу системи

1) Визначте активи системи (дані, програмні модулі, канали зв'язку) та складіть їх перелік з короткою характеристикою.

2) Побудуйте таблицю загроз (таблиця 1.6) за трьома класифікаційними ознаками:

- за джерелом (внутрішні / зовнішні);
- за характером дії (технічні / програмно-логічні / соціоінженерні);

– за наслідками (порушення конфіденційності / цілісності / доступності).

3) Для кожної загрози визначте відповідну вразливість (коду, конфігурації, архітектури або середовища виконання), яка робить її реалізацію можливою.

4) Сформулюйте висновок щодо найбільш критичних загроз для обраної системи.

Приклад розв'язку:

Обрана система: автоматизована система координації аварійно-рятувальних служб.

Активи системи:

– оперативна база даних викликів і ресурсів (критичні дані);

– модуль розподілу завдань між бригадами (виконуваний код);

– канал зв'язку з мобільними підрозділами через захищений VPN;

– вебінтерфейс диспетчера (клієнтський застосунок).

Таблиця 1.6

Модель загроз програмної системи

Загроза	**Джерело**	**Характер дії**	**Наслідок**	**Вразливість**
SQL-ін'єкція у формі пошуку	Зовнішнє	Програмно-логічна	Порушення конфіденційності	Відсутність валідації вхідних даних
Перехоплення VPN-трафіку	Зовнішнє	Технічна	Порушення конфіденційності	Використання застарілого протоколу (наприклад, PPTP)
Фішинговий лист диспетчеру	Зовнішнє	Соціоінженерна	Порушення конфіденційності	Відсутність MFA та контролю поштового доступу
Підміна записів про статус виклику	Внутрішнє (інсайдер)	Програмно-логічна	Порушення цілісності	Відсутність контролю цілісності БД
Впровадження коду через вразливість RCE	Зовнішнє	Технічна	Порушення цілісності	Невиправлена вразливість у серверному програмному модулі або залежностях
Підроблення команд управління	Зовнішнє	Програмно-логічна	Порушення цілісності	Відсутність двофакторної автентифікації
DoS-атака на сервер диспетчеризації	Зовнішнє	Технічна	Порушення доступності	Відсутність обмеження запитів (rate limiting)

Загроза	Джерело	Характер дії	Наслідок	Вразливість
Збій модуля через некоректне оновлення	Внутрішнє	Технічна	Порушення доступності	Відсутність резервування (single point of failure)
Блокування доступу через злом облікового запису	Зовнішнє	Соціоінженерна	Порушення доступності	Слабка парольна політика, відсутність блокування після невдалих спроб

Висновок: найбільш критичними є загрози доступності (DoS) та цілісності (RCE, підміна даних), оскільки система координації функціонує в режимі реального часу. Будь-яке переривання роботи або спотворення оперативних даних безпосередньо впливає на ефективність реагування рятувальних служб. Пріоритетними заходами є впровадження rate limiting, резервування серверних компонентів та обов'язкова двофакторна автентифікація для персоналу.

ЗАВДАННЯ 2. Аналіз підозрілого файлу за допомогою онлайн-інструментів безпеки

Постановка завдання:

Знайдіть у відкритих джерелах (наприклад, репозиторій зразків Any.Run Public Submissions або бібліотека загроз VirusTotal) хеш-суму або опис підозрілого файлу (Файл на власній системі не запускати!).

Проведіть аналіз з використанням не менше двох із наведених сервісів:

– VirusTotal (https://www.virustotal.com) – перевірка за базами антивірусних сигнатур;

– Hybrid Analysis (https://www.hybrid-analysis.com) – автоматичний динамічний аналіз;

– App.Any.Run (https://app.any.run) – інтерактивний аналіз поведінки програми.

За результатами аналізу:

1) Зафіксуйте ідентифікатор файлу (хеш), дату аналізу та назви використаних сервісів.

2) Складіть зведену таблицю: кількість виявлень, категорія загрози, виявлені MITRE ATT&CK-техніки.

3) Назвіть ознаки шкідливої активності: мережеві з'єднання, файлові операції, зміни реєстру.

4) Визначте, які механізми захисту з Теми 1, зокрема контроль цілісності, ізоляція виконання, журналювання подій безпеки могли б запобігти запуску файлу або забезпечити його своєчасне виявлення.

Приклад розв'язку:

Досліджуваний зразок: відомий дроппер сімейства Emotet. SHA-256: a4b2c1d3e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0c1d2e3f4a5b6c7d8e9f0a1b2.
(SHA-256 наведено як умовне ілюстративне значення; під час виконання роботи студент має використати реальний хеш із відкритого репозиторію загроз). Дата аналізу: 15.03.2026.

Результати аналізу:

Таблиця 1.7

Аналіз сервісів

Сервіс	**Виявлень (із загальної кількості)**	**Категорія загрози**	**MITRE ATT&CK-техніки**
VirusTotal	61 / 72	Trojan.Emotet	T1059 (Command Scripting), T1547 (Boot Autostart), T1071 (Application Layer Protocol)
Hybrid Analysis	Verdict: Malicious	Banking Trojan	T1055 (Process Injection), T1082 (System Information Discovery)
App.Any. Run	Interactive verdict: Malicious	Trojan Downloader / Dropper	T1105 (Ingress Tool Transfer), T1055 (Process Injection), T1562 (Impair Defenses)

Ознаки шкідливої активності:

– мережева активність: спроби з'єднання з C2-сервером через порт 8080 (нестандартний трафік); DNS-запити до доменів, зареєстрованих менше ніж 30 днів тому;

—файлові операції: створення копії у каталозі %APPDATA%\Local\ <випадкове ім'я>.exe; запис до тимчасових каталогів ОС;

—зміни реєстру: додавання ключа HKCU\Software\Microsoft\Windows\CurrentVersion\Run для забезпечення автозапуску;

– ін'єкція коду: впровадження шкідливого потоку у процес explorer.exe (Process Hollowing).

Висновок: файл є шкідливим (banking trojan / дроппер). Із механізмів Теми 1 запобіганню могли б сприяти:

– використання ізольованого середовища виконання (наприклад, Windows Sandbox) – унеможливлює зараження реальної системи;

– контроль цілісності виконуваних файлів (хеш-перевірка) – виявив би несанкціоновані копії;

– антивірусний захист з актуальними сигнатурами (61 з 72 рушіїв VirusTotal вже виявляють цей зразок).

ЗАВДАННЯ 3. Порівняльний аналіз підходів до захисту програмного забезпечення

Постановка завдання:

Розгляньте чотири підходи до захисту ПЗ: вбудований захист, навісний захист (зовнішний), шифрування виконуваного файлу, комбінований захист.

Для кожного підходу:

1) Стисло опишіть сутність методу.

2) Наведіть реальний приклад програмного продукту або інструменту, де цей підхід застосовується.

3) Складіть порівняльну таблицю за характеристиками:

- складність реалізації;
- стійкість до статичного аналізу;
- стійкість до динамічного аналізу;

- вплив на продуктивність;
- складність обходу.

4) Сформулюйте рекомендації щодо застосування у підрозділах цивільного захисту.

Приклад розв'язку:

Вбудований захист – захисні перевірки інтегруються безпосередньо у вихідний код програми: ліцензійні перевірки, хешування модулів, обфускація критичних функцій. Приклад: бібліотека Dotfuscator (Community Edition) для .NET – вбудовується в процес збірки і виконує обфускацію зібраної .NET-збірки перед розгортанням.

Навісний захист – захисні механізми накладаються на вже скомпільований виконуваний файл без доступу до вихідного коду. Приклад: пакувальник UPX у поєднанні із зовнішнім модулем ліцензування – обгортає EXE-файл, не змінюючи оригінальний код.

Шифрування виконуваного файлу – програма запускається через спеціальний завантажувач (loader), який розшифровує код під час виконання. Приклад: Themida – комерційний протектор, що шифрує секції коду і розшифровує їх безпосередньо перед виконанням кожного блоку.

Комбінований захист – поєднання кількох підходів одночасно (шифрування + обфускація + ліцензійні перевірки + мережева автентифікація). Приклад: StarForce – використовує фізичний аналіз носія, SDK для вбудовування перевірок і шифрування виконуваних модулів.

Таблиця 1.8

Порівняльна таблиця підходів захисту ПЗ

Характеристика	**Вбудований**	**Навісний**	**Шифрування файлу**	**Комбінований**
Складність реалізації	Висока	Низька	Середня	Дуже висока
Стійкість до статичного аналізу	Середня	Середня	Висока	Дуже висока
Стійкість до динамічного аналізу	Середня	Низька	Середня	Висока

Характеристика	Вбудований	Навісний	Шифрування файлу	Комбінований
Вплив на продуктивність	Мінімальний	Помірний	Помітний	Значний
Складність обходу для зловмисника	Середня	Низька	Середня	Дуже висока

Рекомендація: для програмного забезпечення у підрозділах цивільного захисту найбільш доцільним є комбінований підхід – він реалізує принцип «багаторівневої оборони» (defense in depth). Вбудовані перевірки цілісності захищають від модифікації коду, шифрування виконуваних модулів ускладнює статичний аналіз, а мережева автентифікація забезпечує додатковий рівень контролю. Незважаючи на вищу складність реалізації, критичність функцій систем ЦЗ виправдовує ці витрати. При цьому захисні механізми не повинні знижувати доступність системи в режимі реального часу

2. СУЧАСНИЙ СТАН ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1. ОСОБЛИВОСТІ СУЧАСНИХ СИСТЕМ ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У сучасних умовах програмне забезпечення є не лише функціональним інструментом обробки інформації, а й критично важливим елементом систем управління, моніторингу та реагування, особливо у сфері цивільного захисту. Порушення його цілісності, доступності або керованості може призвести до суттєвих наслідків, пов’язаних із зупинкою технологічних процесів, втратою даних або неможливістю своєчасного реагування на надзвичайні ситуації.

У Розділі 1 було розглянуто основні принципи безпеки програмного забезпечення, етапи аналізу програм, підходи до дослідження алгоритмів, а також процес створення захищених реалізацій. На основі цих положень формується практичне розуміння сучасних систем захисту програмного забезпечення, які забезпечують протидію несанкціонованому доступу, модифікації, копіюванню та експлуатації вразливостей.

Сучасний захист програмного забезпечення розглядається як складова Application Security і реалізується за принципами Secure by Design та DevSecOps, коли механізми безпеки інтегруються на всіх етапах життєвого циклу програмного продукту – від проєктування до супроводу та оновлення. Такий підхід дозволяє не лише реагувати на вже виявлені загрози, але й запобігати їх виникненню ще на етапі розробки [5; 14; 20].

Принцип Secure by Design передбачає, що механізми безпеки повинні закладатися ще на етапі проєктування архітектури програмного забезпечення, а не додаватися після завершення розробки або після виявлення інцидентів. Такий підхід забезпечує системну стійкість програмного продукту до загроз, зменшує кількість вразливостей і підвищує ефективність подальшого супроводу та оновлення [5; 14].

Основою захисту виступає забезпечення властивостей CIA+AA:

– конфіденційності (Confidentiality);

– цілісності (Integrity);

– доступності (Availability);

– автентичності (Authenticity);

– підзвітності (Accountability).

Для систем цивільного захисту особливе значення має саме доступність, оскільки навіть короткочасне порушення роботи програмного забезпечення може призвести до затримки реагування на надзвичайні події та збільшення масштабу наслідків.

Захист програмного забезпечення реалізується на кількох взаємопов’язаних рівнях:

– захист програмного коду (обфускація, шифрування, цифровий підпис);

– захист середовища виконання (ASLR, DEP, sandbox, контейнеризація);

– контроль цілісності та захист від модифікації (anti-tampering, anti-debugging, hooking detection);

– контроль доступу та автентифікація;

– журналювання, аудит і моніторинг подій безпеки;

– захист ланцюга постачання програмного забезпечення (Supply Chain Security, SBOM, patch management) [5; 24; 25].

Окремим напрямом є DRM (Digital Rights Management) – технології управління цифровими правами, що використовуються для контролю копіювання, використання та розповсюдження програмного продукту. У сучасних системах DRM не є основним механізмом кіберзахисту, а виконує допоміжну функцію, переважно пов’язану із захистом інтелектуальної власності та ліцензійного контролю [29].

Для критичних систем, зокрема у сфері цивільного захисту, пріоритетними є механізми забезпечення безперервності роботи, контролю цілісності, захисту конфігурацій, стійкості до зовнішніх атак і можливості оперативного відновлення після інцидентів. Саме тому сучасні системи захисту програмного забезпечення повинні поєднувати класичні механізми контролю доступу з

динамічними засобами виявлення аномалій, Runtime Security та інтеграцією з системами класу SIEM, SOAR та ITSM.

Таким чином, сучасна система захисту програмного забезпечення є не окремим захисним модулем, а комплексною багаторівневою архітектурою, що охоплює всі етапи життєвого циклу програмного продукту. Такий підхід забезпечує не лише протидію загрозам, а й підтримку надійного, безперервного та контрольованого функціонування програмних систем у критичних умовах експлуатації [13; 20].

2.1.1. Механізми перевірки цілісності програмних реалізацій

Контроль цілісності програмного забезпечення є одним із базових механізмів захисту, спрямованих на виявлення несанкціонованих змін у коді, конфігурації або виконуваних модулях програми. Порушення цілісності може свідчити про впровадження шкідливого коду, зміну логіки роботи програми, підміну бібліотек або спробу обходу захисних механізмів [5; 13].

Особливе значення контроль цілісності має для критично важливих інформаційних систем, зокрема у сфері цивільного захисту, де зміна навіть одного програмного компонента може призвести до порушення функціонування систем реагування, втрати достовірності даних або відмови в обслуговуванні.

Основними об’єктами контролю цілісності є:

– виконувані файли;

– динамічні бібліотеки (DLL, shared libraries);

– конфігураційні файли;

– системні служби;

– скрипти автоматизації;

– журнали подій і службові файли.

До основних механізмів перевірки цілісності належать:

Контроль хеш-сум.

Для кожного критичного файлу формується контрольне значення (hash), яке дозволяє перевірити його незмінність. Найчастіше використовуються

алгоритми SHA-256, SHA-3 або інші криптографічно стійкі функції хешування. Будь-яка зміна навіть одного байта призводить до зміни хеш-значення.

Цифровий підпис.

Використовується для підтвердження справжності та цілісності програмного забезпечення. Якщо виконуваний файл підписаний довіреним сертифікатом, система може перевірити, чи не був він змінений після підписання.

Перевірка цілісності під час виконання (Runtime Integrity Check).

Програма може самостійно контролювати власну незмінність під час роботи: перевіряти важливі ділянки пам’яті, критичні бібліотеки, таблиці імпортів, системні виклики або контрольні суми внутрішніх модулів [26].

Контроль конфігурацій та середовища виконання.

Окрім самого коду, важливо контролювати незмінність параметрів запуску, конфігураційних файлів, політик доступу та параметрів середовища виконання.

Інтеграція з SIEM та системами моніторингу.

Події порушення цілісності повинні реєструватися, передаватися до централізованих систем моніторингу та аналізуватися як потенційні інциденти безпеки [23].

Практична реалізація контролю цілісності передбачає використання різних методів перевірки залежно від типу програмного компонента. Найпоширеніші приклади наведено в таблиці 2.1.

Таблиця 2.1

Приклад контролю цілісності

Об’єкт перевірки	**Метод**	**Результат**
Виконуваний файл	SHA-256	Виявлення модифікації
Конфігурація	HMAC	Захист від підміни
Оновлення	Цифровий підпис	Підтвердження джерела

Найбільш критичними для систем цивільного захисту є перевірка цілісності виконуваних файлів та підтвердження достовірності оновлень,

оскільки саме ці компоненти безпосередньо впливають на безперервність функціонування програмних систем.

Прикладом практичної реалізації є перевірка цифрового підпису виконуваного файлу під час запуску програмного комплексу координації рятувальних служб. Якщо система виявляє зміну файлу або відсутність коректного підпису, запуск блокується, а подія фіксується в журналі безпеки.

У веб-системах додатково застосовується контроль цілісності бібліотек і модулів сторонніх компонентів, що особливо важливо при використанні зовнішніх залежностей. У таких випадках використовується перевірка контрольних сум, перевірка цифрових підписів пакетів та формування SBOM (Software Bill of Materials) [24; 25].

Контроль цілісності тісно пов’язаний із механізмами anti-tampering, anti-debugging та захистом від hooking і patching, оскільки всі ці методи спрямовані на виявлення або ускладнення несанкціонованої модифікації програмного забезпечення.

Таким чином, перевірка цілісності є не окремою процедурою, а постійним процесом підтримки довіри до програмної реалізації. Для систем цивільного захисту це є критично важливою умовою забезпечення безперервності роботи, достовірності обробки даних і стійкості до зовнішнього втручання [5; 13].

2.1.2. Контроль доступу та управління ідентифікацією

Ідентифікація передбачає встановлення особи користувача або пристрою, автентифікація – підтвердження цієї належності, а авторизація – визначення дозволеного рівня доступу до ресурсів системи. Сукупність цих механізмів забезпечує захист від несанкціонованого використання програмного забезпечення та зменшує ризик внутрішніх і зовнішніх загроз [28].

У сучасних системах дедалі частіше застосовуються багаторівневі схеми доступу, які враховують не лише облікові дані користувача, а й контекст його дій, характеристики пристрою, місце підключення або час виконання операцій. Наприклад, доступ до адміністративних функцій може бути дозволений лише з

визначених робочих станцій і за умови використання апаратного токена або смарт-картки [28].

У підрозділах цивільного захисту контроль доступу має особливе значення, оскільки програмні системи використовуються різними категоріями персоналу – від операторів до керівників і адміністраторів. Для таких систем доцільно застосовувати рольову модель доступу (RBAC), за якої кожен користувач отримує лише ті права, що необхідні для виконання його службових функцій. Це зменшує ризик як випадкових помилок, так і навмисних дій з боку інсайдерів [27], (Таблиця 2.2).

Таблиця 2.2

Приклад розмежування доступу у системі ЦЗ

Роль користувача	**Доступ до функцій**	**Ризики при компрометації**
Оператор	Перегляд даних, запуск процедур	Обмежені
Адміністратор	Налаштування системи	Високі
Керівник	Прийняття рішень, запуск оповіщення	Критичні

Додатковим рівнем захисту є багатофакторна автентифікація, що поєднує кілька способів підтвердження особи: пароль, апаратний носій, одноразовий код або біометричні параметри. Такий підхід значно підвищує стійкість системи до компрометації облікових записів [28].

Таким чином, контроль доступу є не лише механізмом обмеження прав користувачів, а й важливим інструментом забезпечення надійності та безперервності функціонування програмних систем. Для критичних систем цивільного захисту правильне розмежування прав доступу безпосередньо впливає на безпеку управлінських рішень і стійкість до несанкціонованого втручання.

2.1.3. Програмні механізми захисту виконуваного коду

Захист виконуваного коду є важливою складовою безпеки програмного забезпечення, оскільки саме виконувані файли найчастіше стають об’єктом

несанкціонованого дослідження, модифікації або підміни. Основною метою таких механізмів є ускладнення аналізу програмної реалізації, запобігання зміні логіки роботи програми та забезпечення контролю її цілісності під час запуску і виконання.

Відповідно до принципу Secure by Design, механізми захисту виконуваного коду необхідно закладати ще на етапі проєктування програмної системи, а не впроваджувати лише після виявлення інцидентів або спроб компрометації. Такий підхід дозволяє формувати стійку архітектуру захисту ще до введення програмного продукту в експлуатацію [5; 14].

На відміну від механізмів контролю доступу або перевірки цифрового підпису, програмні механізми захисту безпосередньо вбудовуються у структуру програмного продукту та супроводжують його протягом усього життєвого циклу виконання. Вони спрямовані не лише на запобігання несанкціонованому копіюванню, але й на протидію реверс-інжинірингу, обходу захисту та прихованій модифікації коду.

До основних програмних механізмів захисту належать [30; 14]:

Обфускація коду (Code Obfuscation)

Передбачає навмисне ускладнення структури програмного коду без зміни його функціональності. Це може включати перейменування змінних, заплутування логіки переходів, вставку зайвих інструкцій, шифрування рядків та приховування викликів функцій. Обфускація значно ускладнює аналіз коду дизасемблерами та декомпіляторами [14].

Пакування виконуваних файлів (Executable Packing)

Передбачає стиснення або шифрування виконуваного файлу з подальшим автоматичним розпакуванням під час запуску. Такий підхід ускладнює статичний аналіз та приховує внутрішню структуру програми. До типових прикладів належать UPX, Themida, AsProtect [26].

Anti-Debugging

Механізми виявлення налагоджувачів дозволяють програмі визначати факт запуску під контролем debugger (OllyDbg, x64dbg, WinDbg тощо). У разі

виявлення програма може завершувати роботу, змінювати логіку виконання або блокувати критичні функції.

Anti-Tampering

Спрямований на виявлення спроб зміни виконуваного коду, таблиць імпортів, бібліотек або конфігураційних параметрів. Такі механізми часто поєднуються з перевіркою контрольних сум та Runtime Integrity Check.

Захист від hooking та patching

Hooking використовується для перехоплення викликів функцій, а patching – для зміни виконуваного коду з метою обходу захисту. Для протидії застосовуються перевірка цілісності API-викликів, контроль адрес переходів, перевірка таблиць імпорту та контроль системних викликів [26].

Самоперевірка під час виконання (Self-Protection / Runtime Verification)

Програма може самостійно перевіряти власний стан під час роботи: контролювати пам’ять, завантажені бібліотеки, наявність підозрілих процесів або зміну конфігурації середовища [22].

У системах цивільного захисту застосування таких механізмів особливо важливе для програмних модулів, що відповідають за координацію рятувальних операцій, системи оповіщення, моніторинг надзвичайних ситуацій та обробку критичних даних. Наприклад, захист від підміни бібліотек може запобігти несанкціонованому втручанню в систему автоматизованого оповіщення, а контроль цілісності коду – блокувати запуск зміненого програмного модуля.

Надмірне використання механізмів захисту може ускладнювати супровід, оновлення та аудит програмного забезпечення, тому при проєктуванні необхідно забезпечувати баланс між рівнем захищеності, продуктивністю та можливістю подальшого адміністрування.

Таким чином, програмні механізми захисту виконуваного коду є важливою складовою багаторівневої системи безпеки програмного забезпечення. Їх ефективність значною мірою визначається саме правильним застосуванням принципу Secure by Design, коли захист є частиною архітектури програмного

продукту, а не зовнішнім доповненням після завершення розробки. Вони забезпечують стійкість програмної реалізації до аналізу, модифікації та несанкціонованого втручання, що є особливо критичним для інформаційних систем цивільного захисту [5; 14].

2.1.4. Захист програмного забезпечення під час виконання (runtime security)

Сучасні загрози для програмного забезпечення часто реалізуються не на етапі зберігання або встановлення, а безпосередньо під час виконання програми. Саме тому важливим напрямом забезпечення безпеки є захист програмного забезпечення під час роботи – Runtime Security, який передбачає постійний контроль поведінки програми, виявлення аномалій та оперативне реагування на спроби несанкціонованого втручання [22].

У межах концепції Secure by Design захист під час виконання розглядається не як додатковий інструмент реагування, а як заздалегідь передбачений елемент архітектури програмної системи. Механізми Runtime Security повинні бути інтегровані у програмний продукт ще на етапі його проєктування, що дозволяє забезпечити стійкість до атак у реальному середовищі експлуатації [5; 14].

На відміну від статичних механізмів захисту, що працюють до запуску програми (цифровий підпис, контроль хеш-сум, обфускація), Runtime Security функціонує безпосередньо у процесі виконання та дозволяє виявляти атаки, які неможливо повністю передбачити на етапі розробки.

До основних механізмів захисту під час виконання належать:

RASP (Runtime Application Self-Protection)

RASP є вбудованим механізмом захисту, який аналізує поведінку програми в реальному часі та дозволяє виявляти небезпечні дії безпосередньо в процесі виконання. Такий механізм може блокувати спроби SQL Injection, code injection, несанкціоновані API-виклики або аномальні сценарії роботи. Саме RASP є одним із найбільш показових прикладів реалізації принципу Secure by Design у Runtime Security [22].

Контроль системних викликів

Дозволяє відстежувати взаємодію програми з операційною системою, файлами, пам’яттю, мережею та службами. Аналіз системних викликів дає змогу виявляти спроби несанкціонованого доступу, ін’єкції коду або запуску шкідливих процесів [26].

Виявлення ін’єкцій

Окрему категорію загроз становлять SQL Injection, Command Injection, Code Injection та інші атаки, що використовують некоректну обробку вхідних даних. Захист реалізується через перевірку параметрів, валідацію введення, контроль форматів даних та безпечну взаємодію з базами даних [6].

Поведінковий аналіз

Система аналізує нетипову активність програми: різке збільшення кількості запитів, незвичні звернення до пам’яті, аномальні мережеві з’єднання або спроби зміни критичних файлів. Такі ознаки можуть свідчити про компрометацію програмного забезпечення.

Журналювання та аудит подій безпеки

Усі підозрілі події повинні реєструватися в журналах безпеки та передаватися до централізованих систем моніторингу (SIEM). Це дозволяє не лише оперативно реагувати на інциденти, а й проводити подальший аналіз причин порушення безпеки [23].

Простим прикладом Runtime Security є перевірка вхідних даних користувача перед їх обробкою програмою. Наприклад, система може блокувати введення спеціальних символів, які використовуються в ін’єкційних атаках, або перевіряти допустимість формату вхідних параметрів.

У системах цивільного захисту захист під час виконання має особливе значення, оскільки програмні комплекси часто працюють у режимі реального часу та безперервно взаємодіють із критичними даними. Помилка або компрометація такого програмного забезпечення може призвести до затримки реагування на надзвичайну ситуацію або порушення координації рятувальних підрозділів.

Наприклад, система диспетчерського управління повинна виявляти аномальні спроби зміни статусів викликів, несанкціоноване втручання в маршрути передачі повідомлень або спроби запуску сторонніх процесів у середовищі виконання.

Водночас механізми Runtime Security не повинні надмірно знижувати продуктивність системи, особливо у критичних програмних комплексах. Тому важливим є баланс між рівнем контролю, швидкодією та надійністю функціонування програмного забезпечення.

Таким чином, захист програмного забезпечення під час виконання є необхідною складовою сучасної системи безпеки, оскільки дозволяє виявляти та блокувати загрози безпосередньо в момент їх реалізації. Саме завдяки реалізації принципу Secure by Design ці механізми стають частиною внутрішньої логіки програмного продукту, а не зовнішнім засобом реагування. Для систем цивільного захисту це є важливою умовою забезпечення безперервності роботи, стійкості до атак та оперативного реагування на інциденти [22; 13].

2.1.5. Захист ланцюга постачання програмного забезпечення (supply chain security)

У сучасних умовах значна частина програмного забезпечення створюється з використанням сторонніх бібліотек, фреймворків, пакетів, контейнерів та зовнішніх сервісів. Це значно прискорює розробку, однак одночасно створює нові ризики безпеки, пов’язані з компрометацією компонентів, які не розробляються безпосередньо власником системи [24; 25].

Захист ланцюга постачання програмного забезпечення (Supply Chain Security) спрямований на забезпечення контролю цілісності, достовірності та безпеки всіх компонентів, що використовуються під час розробки, розгортання та експлуатації програмних систем. Метою такого захисту є запобігання впровадженню вразливостей або шкідливого коду через зовнішні залежності [25].

Принцип Secure by Design у цьому випадку означає, що безпека зовнішніх компонентів повинна враховуватися ще на етапі проєктування архітектури системи, а не лише після виявлення вразливостей у вже розгорнутому програмному забезпеченні. Контроль залежностей, перевірка джерел походження та управління оновленнями повинні бути частиною процесу розробки від самого початку [5; 25].

Особливо актуальним цей напрям є для критичних інформаційних систем, зокрема у сфері цивільного захисту, де використання ненадійного програмного компонента може вплинути на функціонування систем моніторингу, оповіщення або координації дій під час надзвичайних ситуацій.

До основних механізмів Supply Chain Security належать:

Формування SBOM (Software Bill of Materials)

SBOM є структурованим переліком усіх програмних компонентів, бібліотек, залежностей і версій, що використовуються у програмному продукті. Це дозволяє швидко визначати наявність уразливих компонентів та оцінювати вплив нових загроз [24].

Перевірка цифрових підписів і походження компонентів.

Перед використанням бібліотек, пакетів або оновлень необхідно перевіряти їх справжність, джерело походження та цілісність. Це дозволяє уникнути використання підроблених або скомпрометованих компонентів.

Patch Management

Систематичне управління оновленнями дозволяє своєчасно усувати відомі вразливості у сторонніх компонентах. Важливо не лише встановлювати оновлення, а й контролювати їх безпечність та сумісність із критичними системами.

Аналіз вразливостей залежностей

Спеціалізовані інструменти дозволяють автоматично перевіряти використовувані компоненти на наявність відомих CVE, оцінювати рівень ризику та формувати пріоритети оновлення [25].

Контроль середовища збірки та розгортання

Безпеку необхідно забезпечувати не лише на рівні бібліотек, а й у середовищі CI/CD, контейнеризації, автоматизованих pipeline та інфраструктури розгортання. Компрометація середовища збірки може призвести до поширення шкідливого коду навіть у перевірених продуктах.

Прикладом практичної реалізації є перевірка сторонніх Python-бібліотек перед інтеграцією у систему аналізу ризиків або систему підтримки прийняття рішень. Перед впровадженням перевіряють джерело походження пакета, цифровий підпис, контрольну суму, а також наявність відомих вразливостей у CVE/NVD.

У системах цивільного захисту особливо важливо контролювати програмні компоненти, що використовуються у системах автоматизованого оповіщення, диспетчерських системах, інформаційно-аналітичних платформах та засобах міжвідомчої взаємодії. Компрометація навіть одного зовнішнього модуля може створити ризик порушення роботи всієї системи.

Захист ланцюга постачання тісно пов’язаний із контролем цілісності, журналюванням, аудитом та інтеграцією із системами SIEM і SOAR, оскільки будь-які зміни в залежностях потрібно фіксувати та аналізувати як потенційні події безпеки [23].

Таким чином, Supply Chain Security є важливою складовою сучасної системи захисту програмного забезпечення, оскільки дозволяє контролювати безпеку не лише власного коду, а й усіх зовнішніх компонентів, від яких залежить функціонування програмного продукту. Реалізація цього підходу відповідно до принципу Secure by Design дозволяє запобігати ризикам ще до введення системи в експлуатацію. Для систем цивільного захисту це є критично важливим елементом забезпечення надійності, стійкості та безперервності роботи.

2.1.6. DRM у сучасних системах захисту програмного забезпечення

Digital Rights Management (DRM) – це сукупність технологій і програмно-апаратних механізмів, призначених для контролю використання, копіювання,

розповсюдження та модифікації цифрового контенту і програмного забезпечення [29]. Первинно DRM розроблявся як засіб захисту авторських прав та ліцензійного контролю, однак окремі його механізми можуть використовуватися і як допоміжний елемент загальної системи захисту програмного забезпечення.

На відміну від сучасних підходів – Application Security, Secure by Design, DevSecOps або Runtime Security, DRM не забезпечує повноцінного кіберзахисту програмних систем і не вирішує завдання протидії експлуатації вразливостей, захисту конфіденційних даних або забезпечення безперервності роботи. Його основне призначення – ускладнення несанкціонованого копіювання, модифікації та використання програмного продукту.

Принцип Secure by Design передбачає формування безпеки на етапі проєктування архітектури програмної системи, тоді як DRM здебільшого виконує допоміжну функцію контролю ліцензування та захисту інтелектуальної власності [29]. Саме тому DRM не може розглядатися як основа безпеки критичних програмних систем, а лише як додатковий інструмент у складі загальної архітектури захисту.

До типових механізмів DRM належать:

Ліцензійні ключі (Serial Keys)

Використовуються для підтвердження права користувача на встановлення та використання програмного забезпечення. Ключ може перевірятися локально або через сервер активації.

Апаратна прив’язка (Hardware Binding)

Програма прив’язується до конкретного обладнання – серійного номера пристрою, MAC-адреси, TPM-модуля або апаратного токена. Це ускладнює перенесення ліцензії на інші пристрої.

Захист носія та інсталяційних файлів

У класичних системах використовувалися спеціальні механізми захисту фізичних носіїв (CD-COPS, SafeDisc, LaserLock), а також перевірка оригінальності інсталяційного середовища.

Обмеження функціональності

Програмний продукт може надавати частковий доступ до функцій до моменту активації, перевірки ліцензії або завершення процедури реєстрації.

Контроль цілісності ліцензійних модулів

DRM-система може перевіряти незмінність модулів активації, ліцензійних бібліотек і механізмів перевірки прав доступу [29].

У сучасних критичних системах, зокрема у сфері цивільного захисту, DRM не виступає основним механізмом безпеки, однак окремі його елементи можуть бути корисними. Наприклад, апаратна прив’язка або ліцензійний контроль можуть застосовуватися для спеціалізованого програмного забезпечення управління системами оповіщення, диспетчерськими комплексами або аналітичними платформами, де важливо контролювати офіційне використання програмного продукту та захищати його від несанкціонованого копіювання.

Водночас надмірне використання DRM у критичних інформаційних системах може створювати додаткові ризики: ускладнювати оновлення, аварійне відновлення, перенесення систем між майданчиками та оперативне реагування під час надзвичайних ситуацій. Саме тому DRM доцільно розглядати як допоміжний, а не основний механізм захисту.

Основу безпеки повинні становити контроль цілісності, контроль доступу, захист під час виконання, журналювання, аудит і захист ланцюга постачання програмного забезпечення, які формуються відповідно до принципу Secure by Design, тоді як DRM виконує переважно функцію ліцензійного контролю та захисту інтелектуальної власності.

Таким чином, DRM є важливим, але допоміжним інструментом захисту програмного забезпечення. Його застосування не повинно підміняти собою системний підхід до побудови безпеки програмного продукту. У сучасних системах цивільного захисту DRM має використовуватися обґрунтовано і не суперечити головній меті – забезпеченню доступності, надійності та безперервності функціонування програмних систем.

У системах цивільного захисту контроль цілісності має критичне значення, оскільки навіть незначна зміна програмного коду може призвести до неправильних управлінських рішень або відмови системи.

2.2. СИСТЕМА ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Система захисту програмного забезпечення – це сукупність організаційних, програмних, апаратних і криптографічних заходів, спрямованих на забезпечення конфіденційності, цілісності, доступності, автентичності та підзвітності програмних систем і даних, що ними обробляються [13; 31].

На відміну від окремих механізмів безпеки, система захисту розглядається як цілісна багаторівнева архітектура, у якій кожен компонент виконує власну функцію, а їх взаємодія забезпечує стійкість до зовнішніх і внутрішніх загроз. Такий підхід відповідає принципам Application Security, Secure by Design та DevSecOps, за яких захист формується на всіх етапах життєвого циклу програмного забезпечення [5; 14; 20].

Для систем цивільного захисту особливого значення набуває не лише конфіденційність, а насамперед доступність і безперервність функціонування, оскільки від працездатності програмного забезпечення залежить своєчасність реагування на надзвичайні ситуації, координація рятувальних підрозділів і достовірність управлінських рішень.

Ефективна система захисту повинна забезпечувати:

– контроль доступу до критичних функцій;

– перевірку цілісності програмних компонентів;

– захист виконуваного коду від модифікації;

– контроль поведінки під час виконання (Runtime Security);

– захист ланцюга постачання програмного забезпечення;

– журналювання та аудит подій безпеки;

– резервування та відновлення працездатності після інцидентів.

Перелічені функції відповідають сучасним підходам до побудови комплексних систем захисту та забезпечення кіберстійкості програмних систем [31; 13].

Формування ефективної системи захисту програмного забезпечення базується на поєднанні програмних, апаратних, криптографічних та організаційних механізмів, що взаємодіють між собою та забезпечують багаторівневий захист програмної системи. Кожен із цих засобів виконує власну функцію: програмні механізми протидіють аналізу та модифікації коду, апаратні засоби підвищують довіру до середовища виконання, криптографічні методи забезпечують захист даних і автентичність компонентів, а організаційні заходи підтримують безпечну експлуатацію та контроль змін. Лише їх комплексне застосування дозволяє досягти необхідного рівня стійкості програмного забезпечення до сучасних загроз [21; 31], (рис.2.1).

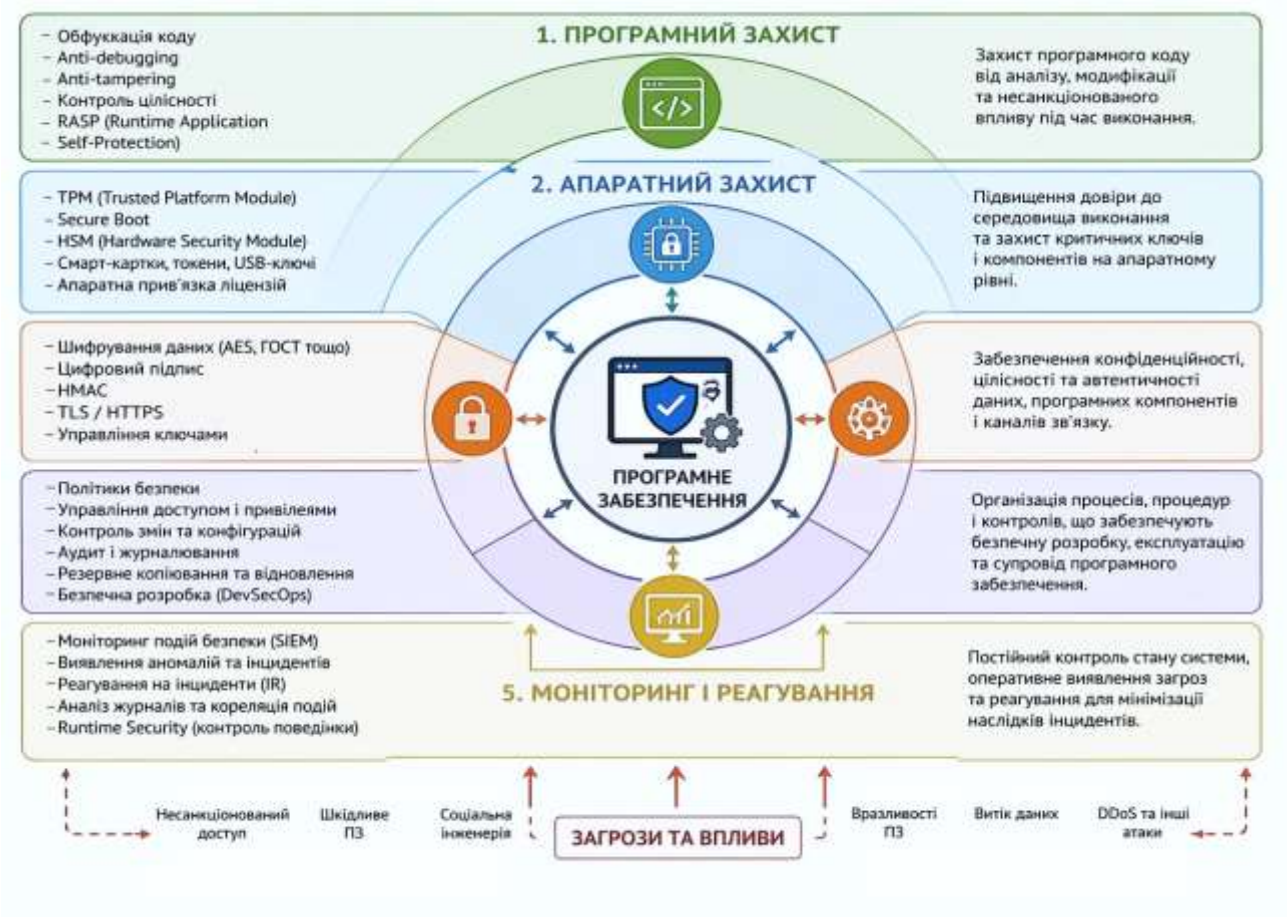


Рисунок 2.1. Багаторівнева система захисту програмного забезпечення (defense in depth)

Принцип Defense in Depth передбачає побудову кількох взаємодоповнюючих рівнів захисту, що зменшує ризик повної компрометації системи у випадку обходу окремого захисного механізму [21; 32].

Програмні засоби захисту реалізуються безпосередньо у структурі програмного продукту та забезпечують протидію несанкціонованому доступу, аналізу, модифікації та експлуатації вразливостей. До них належать механізми обфускації, пакування, anti-debugging, anti-tampering, контроль цілісності, RASP, контроль доступу, журналювання та аудит [22; 5].

Основною перевагою програмних засобів є можливість їх інтеграції у внутрішню логіку програмної системи відповідно до принципу Secure by Design. Це дозволяє не лише виявляти загрози, а й запобігти їх виникненню ще на етапі розробки [5; 20].

У системах цивільного захисту програмні засоби захисту застосовуються для забезпечення стійкості диспетчерських систем, систем оповіщення, модулів моніторингу та аналітичних платформ до несанкціонованого втручання.

Апаратні засоби захисту забезпечують безпеку програмного забезпечення завдяки використанню спеціалізованих фізичних компонентів, що ускладнюють несанкціонований доступ до критичних функцій і криптографічних ключів.

До таких засобів належать TPM (Trusted Platform Module), HSM (Hardware Security Module), Secure Boot, смарт-картки, апаратні токени, USB-ключі, засоби багатофакторної автентифікації та апаратна прив’язка ліцензій [31; 32].

TPM дозволяє безпечно зберігати криптографічні ключі та перевіряти довіру до середовища запуску, Secure Boot контролює цілісність завантаження системи, а HSM використовується для захисту критичних криптографічних операцій [31].

У підрозділах цивільного захисту такі механізми особливо важливі для систем централізованого оповіщення, захищених каналів зв’язку, серверів управління та систем доступу до критичної інфраструктури.

Криптографічний захист забезпечує конфіденційність, цілісність і автентичність даних, що використовуються програмним забезпеченням. До основних механізмів належать шифрування, цифровий підпис, HMAC, TLS, управління ключами та контроль автентичності обміну даними [13; 31].

Шифрування використовується для захисту інформації під час зберігання та передачі, цифровий підпис – для перевірки справжності програмних компонентів і оновлень, а HMAC – для контролю незмінності конфігурацій і службових повідомлень.

Для систем цивільного захисту криптографічний захист має критичне значення при передачі повідомлень оповіщення, роботі з персональними даними, міжвідомчому обміні інформацією та взаємодії з віддаленими пунктами управління.

Найвищий рівень безпеки досягається не окремими механізмами, а їх поєднанням у межах комбінованої системи захисту. Такий підхід відповідає принципу defense in depth (багаторівневий захист), за якого компрометація одного рівня не призводить до повного порушення безпеки системи [21; 32].

Комбіновані системи захисту поєднують програмні, апаратні та криптографічні механізми, а також організаційні процедури контролю, аудит, резервування та реагування на інциденти.

Наприклад, доступ до критичної функції може одночасно вимагати:

– автентифікації користувача;

– підтвердження через апаратний токен;

– перевірки цифрового підпису модуля;

– контролю цілісності середовища виконання;

– журналювання всіх дій у SIEM.

Подібна архітектура відповідає сучасним концепціям Zero Trust та багаторівневого захисту критичних інформаційних систем [32; 31].

У системах цивільного захисту саме комбінований підхід є найбільш ефективним, оскільки забезпечує відмовостійкість, безперервність роботи та можливість оперативного реагування на інциденти.

Технічні засоби захисту не можуть бути ефективними без належного організаційного управління. До таких заходів належать політики безпеки, управління оновленнями, контроль змін, аудит доступу, резервне копіювання, реагування на інциденти, навчання персоналу та управління ризиками [13; 31].

Відповідно до принципів DevSecOps, безпека повинна бути інтегрована не лише в архітектуру програмного продукту, а й у процеси його супроводу, експлуатації та адміністрування [20; 5].

Особливо важливим є контроль змін у критичних системах, де навіть незначна модифікація конфігурації може вплинути на працездатність усієї інфраструктури.

Для систем цивільного захисту організаційні заходи безпосередньо впливають на готовність до реагування, стійкість до кризових ситуацій та забезпечення безперервного функціонування програмних комплексів.

Таким чином, система захисту програмного забезпечення формується не окремими технічними рішеннями, а комплексною взаємодією програмних, апаратних, криптографічних та організаційних механізмів. Саме такий підхід дозволяє забезпечити стійкість, керованість і надійність функціонування програмних систем у критичних умовах експлуатації.

2.3. DRM ТА СУЧАСНІ МЕХАНІЗМИ КІБЕРЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У практиці захисту програмного забезпечення важливо розмежовувати поняття управління цифровими правами (DRM) та сучасних механізмів кіберзахисту [29; 14]. Обидва підходи спрямовані на підвищення рівня захищеності програмного продукту, однак мають різне призначення, сферу застосування та набір реалізованих механізмів.

DRM (Digital Rights Management) – це сукупність технологій і програмно-апаратних засобів, призначених для контролю використання, копіювання, розповсюдження та модифікації цифрового контенту і програмного

забезпечення. Основною метою DRM є захист авторських прав, забезпечення ліцензійного використання програмних продуктів та запобігання несанкціонованому копіюванню [29].

До типових механізмів DRM належать ліцензійні ключі, серверна активація, апаратна прив’язка до конкретного пристрою, перевірка цифрових підписів, контроль цілісності ліцензійних модулів, а також обмеження функціональності до моменту підтвердження права використання [29; 30].

Сучасні механізми захисту програмного забезпечення мають ширшу сферу застосування та спрямовані на забезпечення конфіденційності, цілісності, доступності, автентичності та підзвітності програмних систем і даних. До них належать контроль доступу, перевірка цілісності, Runtime Security, захист виконуваного коду, криптографічний захист, аудит, журналювання, захист ланцюга постачання та інтеграція з системами моніторингу безпеки [5; 14; 22].

У сучасних інформаційних системах DRM та механізми кіберзахисту не протиставляються, а доповнюють один одного. DRM забезпечує контроль прав використання програмного продукту, тоді як механізми кіберзахисту формують стійкість системи до зовнішніх і внутрішніх загроз [29; 5], (таблиця 2.3). Особливо це важливо для програмних комплексів, що використовуються у сфері цивільного захисту, де поряд із захистом інтелектуальної власності необхідно забезпечити безперервність роботи, достовірність даних та стійкість до атак.

Таблиця 2.3

Порівняння DRM та механізмів кіберзахисту

Критерій	**DRM**	**Сучасні механізми захисту**
Основна мета	Контроль ліцензійного використання та захист авторських прав	Захист програмної системи та даних
Об’єкт захисту	Програмний продукт, ліцензія, цифровий контент	Дані, процеси, доступ, середовище виконання
Конфіденційність	Реалізується частково	Один із базових принципів
Цілісність	Контроль окремих модулів	Повний контроль програмної системи
Доступність	Обмежено	Критично важлива

Критерій	DRM	Сучасні механізми захисту
Runtime Security	Реалізується частково	Один із ключових механізмів
Контроль доступу	Ліцензійний контроль	Повноцінне розмежування прав
Значення для систем ЦЗ	Додатковий рівень контролю	Основа забезпечення стійкості

Принцип Secure by Design дозволяє поєднати ці підходи у межах єдиної архітектури безпеки [5; 14]. Якщо DRM переважно реалізує контроль використання готового програмного продукту, то сучасний кіберзахист передбачає закладання механізмів безпеки ще на етапі проєктування програмної системи. Це дозволяє підвищити загальний рівень захищеності та зменшити ризики компрометації в процесі експлуатації.

Таблиця 2.3 демонструє, що DRM і сучасні механізми захисту вирішують різні, але взаємопов’язані завдання [29; 14]. DRM орієнтований переважно на контроль використання програмного продукту, тоді як кіберзахист забезпечує безпечне функціонування всієї програмної системи.

Найбільш поширені комерційні DRM-рішення та їх особливості наведено у таблиці 2.4, [33; 30].

Таблиця 2.4

Приклади комерційних DRM-систем та їх особливості

DRM-система	Основний принцип	Особливості застосування
SafeDisc	Перевірка фізичного носія	Використовувалася для захисту інсталяційних носіїв
StarForce	Драйверний контроль захисту	Підвищений рівень контролю запуску
Denuvo	Anti-tampering + anti-debugging	Захист від модифікації та реверс-аналізу
AsProtect	Пакування та захист PE-файлів	Використовується для ускладнення статичного аналізу
Themida	Обфускація + anti-debugging	Посилений захист виконуваного коду

У спеціалізованих системах, зокрема у сфері цивільного захисту, окремі елементи DRM можуть використовуватися для контролю офіційного використання програмного забезпечення, захисту службових модулів та обмеження несанкціонованого копіювання спеціалізованих програмних комплексів. Водночас основна увага приділяється поєднанню DRM із механізмами контролю цілісності, захисту доступу, журналювання та Runtime Security [22; 33].

Таким чином, DRM є важливим елементом загальної системи захисту програмного забезпечення, особливо в частині ліцензійного контролю та захисту інтелектуальної власності. У поєднанні з сучасними механізмами кіберзахисту він дозволяє формувати комплексну архітектуру безпеки, що забезпечує як контроль використання програмного продукту, так і стійкість програмної системи до сучасних загроз.

2.4. ОСНОВНІ АЛГОРИТМИ ТА МЕТОДИ ЗАХИСТУ

Ефективний захист програмного забезпечення базується не лише на організаційних принципах побудови системи безпеки, але й на використанні конкретних алгоритмів і технічних методів, що забезпечують протидію несанкціонованому доступу, копіюванню, модифікації та дослідженню програмної реалізації [5; 31].

У попередніх підрозділах було розглянуто загальні принципи побудови систем захисту, роль DRM, контроль цілісності, Runtime Security та багаторівневу архітектуру безпеки. Практична реалізація цих підходів здійснюється за допомогою алгоритмів і методів, які можуть застосовуватися як окремо, так і в межах комбінованих систем захисту.

Принцип Secure by Design передбачає, що більшість таких механізмів повинні враховуватися ще на етапі проєктування програмного продукту [5; 14]. Це дозволяє не лише підвищити стійкість системи до атак, а й зменшити ризики помилок реалізації та складність подальшого супроводу.

До основних алгоритмів і методів захисту програмного забезпечення належать:

Методи захисту від несанкціонованого копіювання

До цієї групи належать ліцензійні ключі, серверна активація, апаратна прив'язка (hardware binding), контроль цифрових прав (DRM), обмеження функціональності та перевірка ліцензійних модулів. Їх основною метою є забезпечення легального використання програмного продукту та захист інтелектуальної власності [29].

Методи захисту від дослідження та модифікації

Для ускладнення реверс-інжинірингу та несанкціонованої зміни коду використовуються обфускація, пакування виконуваних файлів, anti-debugging, anti-tampering, anti-hooking, захист від patching та самоперевірка під час виконання. Такі механізми ускладнюють аналіз програмної логіки та підвищують стійкість до атак [30; 14].

Методи контролю цілісності

До них належать криптографічні хеш-функції (SHA-256, SHA-3), HMAC, цифровий підпис, перевірка контрольних сум, Runtime Integrity Check та контроль цілісності конфігурацій. Їх використання дозволяє виявляти зміну програмних компонентів, бібліотек, конфігурацій і оновлень [31].

Методи забезпечення безпеки виконання

Ця група охоплює Runtime Security, RASP, контроль системних викликів, виявлення ін'єкцій, sandbox-середовища, ASLR (Address Space Layout Randomization), DEP (Data Execution Prevention), а також поведінковий аналіз під час виконання програми [22; 26].

Криптографічні методи захисту

Шифрування, цифровий підпис, TLS, VPN, захист ключів, автентифікація обміну даними та контроль справжності програмних компонентів забезпечують захист інформації під час передачі та зберігання [31].

Комбіновані алгоритми захисту

Найбільш ефективним є поєднання кількох механізмів одночасно за принципом defense in depth [21; 31]. Наприклад, запуск критичного програмного модуля може вимагати перевірки цифрового підпису, апаратної автентифікації, контролю цілісності середовища та Runtime-моніторингу.

У системах цивільного захисту такі методи використовуються для захисту програмних комплексів оповіщення, диспетчерських систем, платформ моніторингу надзвичайних ситуацій, систем міжвідомчого обміну інформацією та аналітичних центрів підтримки прийняття рішень.

Наприклад, програмне забезпечення автоматизованого оповіщення може поєднувати цифровий підпис оновлень, контроль доступу за ролями, TPM-модуль для зберігання ключів, Runtime Security для виявлення аномалій та журналювання всіх критичних подій у SIEM-системі [31; 22].

Ефективність застосування конкретного методу захисту залежить від типу програмного забезпечення, критичності системи, моделі загроз та вимог до безперервності функціонування. Порівняльна характеристика основних методів захисту, які мають застосування у системах цивільного захисту наведена в таблиці 2.5.

Таблиця 2.5

Порівняльна характеристика методів захисту

Метод захисту	**Основне призначення**	**Доцільність для ЦЗ**	**Особливості застосування**
Контроль цілісності	Захист від модифікації	Висока	Обов’язковий для критичних модулів
RASP / Runtime Security	Контроль під час виконання	Висока	Для систем реального часу
DRM	Контроль ліцензій	Середня	Для спеціалізованого ПЗ
TPM / HSM	Захист ключів	Висока	Для серверів управ–ління
Обфускація	Захист від аналізу	Середня	Для критичних служ–бових модулів

Наведені в таблиці дані показують, що для систем цивільного захисту найбільш ефективним є поєднання контролю цілісності, захисту під час виконання, апаратних засобів захисту та криптографічних механізмів [21; 31]. Використання лише одного методу не забезпечує необхідного рівня стійкості, тому практична реалізація безпеки повинна базуватися на комбінованому підході з урахуванням критичності функцій, вимог до доступності та необхідності безперервного функціонування програмних систем.

Вибір конкретних алгоритмів захисту залежить від критичності системи, вимог до доступності, типу програмного забезпечення, можливих моделей загроз та умов експлуатації. Для систем цивільного захисту особливо важливим є баланс між рівнем захищеності, швидкодією та безперервністю функціонування.

Таким чином, алгоритми та методи захисту програмного забезпечення є практичною основою реалізації багаторівневої архітектури безпеки. Їх правильне поєднання дозволяє забезпечити стійкість програмної системи до сучасних кіберзагроз, захистити критичні функції та підтримувати надійне функціонування програмних комплексів у реальних умовах експлуатації.

2.5. ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ ВИМОГИ ДО ПОБУДОВИ СИСТЕМ ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Оцінювання ефективності систем захисту програмного забезпечення є необхідним етапом їх проєктування, впровадження та подальшої експлуатації, оскільки дозволяє визначити відповідність обраних механізмів вимогам безпеки, умовам функціонування та реальним загрозам [34; 16; 13]. Особливо важливим це є для критичних інформаційних систем, де помилки у виборі засобів захисту можуть призвести не лише до витоку даних, а й до порушення безперервності роботи всієї системи.

У попередніх підрозділах було розглянуто основні механізми захисту програмного забезпечення: контроль цілісності, контроль доступу, Runtime Security, DRM, криптографічний захист, захист ланцюга постачання та

комбіновані системи безпеки. Проте ефективність їх застосування визначається не лише наявністю захисних механізмів, а й здатністю системи забезпечувати стабільну, безпечну та безперервну роботу в реальних умовах експлуатації [31; 13].

Для оцінювання використовується система критеріїв, наведена у таблиці 2.6, яка охоплює технічні, експлуатаційні та економічні аспекти функціонування систем захисту.

Таблиця 2.6

Критерії оцінювання систем захисту програмного забезпечення

Критерій	**Пояснення**
Рівень захисту	Здатність системи протидіяти несанкціонованому доступу, модифікації та аналізу
Стійкість до атак	Опірність сучасним методам реверс-інжинірингу, динамічного та статичного аналізу
Надійність	Стабільність роботи системи, відсутність збоїв і хибних спрацьовувань
Доступність (Availability)	Здатність системи функціонувати без перерв, особливо у критичних умовах
Сумісність	Відсутність конфліктів із операційною системою, апаратним забезпеченням і стороннім ПЗ
Продуктивність	Вплив механізмів захисту на швидкодію та ресурси системи
Економічна доцільність	Співвідношення витрат на впровадження та ефективність захисту
Зручність використання	Мінімальний вплив на користувача, простота експлуатації
Масштабованість і підтримка	Можливість оновлення, модифікації та інтеграції у сучасні системи

Запропоновані критерії відповідають сучасним підходам до оцінювання ризиків, ефективності засобів захисту та забезпечення безперервності функціонування інформаційних систем [34; 16; 13].

Критерій рівня захисту відображає здатність системи протидіяти несанкціонованому доступу, модифікації та дослідженню програмного забезпечення. Він безпосередньо пов’язаний із критерієм стійкості до атак, який характеризує ефективність протидії сучасним методам реверс-інжинірингу,

аналізу пам’яті, hooking, patching та спробам обходу захисних механізмів. Разом ці критерії визначають технічний рівень захищеності програмної системи [21; 31].

Надійність і доступність є ключовими характеристиками для систем цивільного захисту. Надійність визначає здатність системи працювати без збоїв, відмов і хибних спрацьовувань, тоді як доступність забезпечує її безперервне функціонування навіть у складних умовах або під час надзвичайних ситуацій. Для критичних систем саме ці критерії часто мають пріоритет над окремими механізмами захисту, оскільки від них безпосередньо залежить ефективність реагування [13; 31].

Сумісність і продуктивність характеризують вплив механізмів захисту на роботу програмного забезпечення. Наявність конфліктів із операційною системою, серверною інфраструктурою або апаратним забезпеченням може обмежити можливість використання навіть технічно ефективних засобів захисту. Водночас надмірне зниження швидкодії або збільшення навантаження на ресурси системи є критичним недоліком для програмних комплексів реального часу [31].

Економічна доцільність і зручність використання відображають практичні аспекти впровадження систем захисту. Надмірно складні або дорогі рішення можуть бути недоцільними, якщо їх ефективність не відповідає витратам. Крім того, складні процедури автентифікації або контролю доступу не повинні ускладнювати роботу користувачів, особливо в умовах оперативного реагування на надзвичайні події [34; 16].

Масштабованість і підтримка визначають можливість розвитку системи захисту в майбутньому. Захисні механізми повинні підтримувати оновлення, інтеграцію з новими компонентами, адаптацію до змін у законодавстві, появи нових вразливостей і нових моделей загроз. Це особливо важливо в умовах постійного розвитку кіберзагроз і цифрової трансформації систем цивільного захисту [5; 20].

Таким чином, оцінювання ефективності систем захисту програмного забезпечення повинно базуватися на комплексному підході, що враховує взаємозв'язок технічних, експлуатаційних та організаційних критеріїв. Лише збалансоване поєднання рівня захисту, надійності, доступності та практичної доцільності дозволяє обґрунтовано обирати оптимальні рішення для забезпечення безпеки програмних систем, зокрема у підрозділах цивільного захисту.

Формування ефективної системи захисту програмного забезпечення потребує дотримання низки вимог, які випливають із критеріїв оцінювання, наведених у таблиці 2.5. Ці вимоги визначають принципи побудови систем безпеки та забезпечують їх відповідність сучасним загрозам.

Однією з основних вимог є забезпечення достатнього рівня захисту та стійкості до атак. Це досягається шляхом використання багаторівневих механізмів, які поєднують різні методи захисту та ускладнюють їх одночасний обхід. Важливим аспектом є також інтеграція захисту у процес розробки програмного забезпечення, що відповідає принципам Secure by Design [5; 14].

Не менш важливою є вимога надійності та доступності системи. У системах цивільного захисту програмне забезпечення повинно залишатися працездатним навіть у разі часткового порушення захисту або зовнішніх впливів. Це передбачає використання резервування, відмовостійких архітектур і механізмів відновлення [13; 31].

Вимоги сумісності та продуктивності визначають необхідність мінімального впливу захисних механізмів на функціонування системи. Захист не повинен створювати конфлікти з програмним або апаратним забезпеченням, а також не повинен суттєво знижувати швидкодію, особливо у системах реального часу.

Економічна доцільність і зручність використання визначають можливість практичного впровадження системи захисту. Обрані рішення повинні бути обґрунтованими з точки зору витрат і не створювати надмірних труднощів для користувачів.

У сучасних умовах ці вимоги доповнюються необхідністю забезпечення масштабованості та можливості оновлення системи захисту. З огляду на швидкий розвиток кіберзагроз, програмне забезпечення повинно мати можливість адаптації до нових умов без суттєвих змін архітектури.

Таким чином, побудова систем захисту програмного забезпечення повинна базуватися на комплексному врахуванні технічних, експлуатаційних та економічних вимог. Такий підхід дозволяє створювати ефективні, надійні та адаптивні системи, що відповідають сучасним викликам і забезпечують безпечне функціонування програмного забезпечення у підрозділах цивільного захисту.

ПИТАННЯ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ ЗА ТЕМОЮ 2

1. У чому полягає відмінність між DRM та сучасними механізмами захисту програмного забезпечення?
2. Чому ефективний захист програмного забезпечення повинен будуватися на поєднанні кількох взаємопов’язаних механізмів, а не на одному окремому засобі?
3. Які переваги має принцип Secure by Design під час побудови систем захисту програмного забезпечення?
4. Які основні програмні, апаратні та криптографічні засоби використовуються для захисту програмного забезпечення?
5. У чому полягає захист програмного забезпечення під час виконання (Runtime Security) та які механізми до нього належать?
6. Як контроль цілісності впливає на надійність і безпеку програмного забезпечення? Наведіть приклад практичного застосування.
7. Які критерії оцінювання систем захисту є найбільш важливими для програмних систем цивільного захисту та чому?
8. Як забезпечити баланс між рівнем захисту, продуктивністю та зручністю використання програмного забезпечення?
9. Обґрунтуйте вибір методів захисту для системи реального часу з урахуванням вимог доступності, надійності та безперервності функціонування.

10. Які вимоги до побудови сучасних систем захисту програмного забезпечення є пріоритетними у підрозділах цивільного захисту та як вони впливають на вибір технологій?

ПРАКТИЧНІ ЗАВДАННЯ ДО ТЕМИ 2

ЗАВДАННЯ 1. Оцінювання комерційної системи захисту програмного забезпечення

Постановка завдання:

Ознайомтеся з підрозділами 2.1 - 2.5 та таблицею 2.6 (критерії оцінювання систем захисту програмного забезпечення) посібника.

Оберіть одну з наведених комерційних систем захисту програмного забезпечення:

а) StarForce;

б) AsProtect;

в) TAGES;

г) Dallas Lock.

Для обраної системи:

1) Коротко опишіть принцип дії та визначте її тип відповідно до опису основних методів захисту (підрозділ 2.4).

2) Заповніть оцінювальну таблицю за девятьма критеріями таблиці 2.6: визначте рівень відповідності кожного критерію (Високий / Середній / Низький) та стисло обґрунтуйте оцінку.

3) Виявіть три найбільш суттєві слабкі місця обраної системи та вкажіть відомі методи або інструменти їх експлуатації.

4) Запропонуйте конкретні заходи посилення захисту відповідно до вимог підрозділу 2.5.

Приклад розв'язку:

Обрана система: AsProtect. AsProtect – це комерційна система захисту виконуваних PE-файлів, що поєднує функції пакування, шифрування, протидії налагодженню та ліцензійного контролю.

Тип захисту: програмний механізм захисту коду (підрозділ 2.1.3).

Відповідно до класифікації основних методів захисту програмного забезпечення (підрозділ 2.4):

– за способом впровадження: зовнішній захисний механізм, що накладається на скомпільований модуль;

– за механізмом дії: комбінований – стиснення коду (компресія) + шифрування виконуваних секцій;

– за принципом функціонування: захисний пакувальник (software protector) відповідно до підрозділу 2.4).

Принцип дії: AsProtect обгортає виконуваний PE-файл власним завантажувачем, який стискає та шифрує оригінальний код і ресурси. При запуску завантажувач розпаковує та розшифровує образ у пам'яті, передає управління оригінальній точці входу. Підтримує антиналагоджувальні перевірки, генерацію серійних номерів і стиснення ресурсів.

Таблиця 2.7

Результати оцінювання

Критерій	**Рівень**	**Обґрунтування**
Захист	Середній	Ускладнює побітове копіювання та поверхневий статичний аналіз; PEiD і RDG Packer Detector виявляють захист автоматично
Стійкість до атак	Низький	Алгоритм публічно відомий; існують скрипти для OllyDbg та x64dbg, що автоматично знімають AsProtect
Надійність	Середній	Сумісний із більшістю версій Windows; спостерігаються конфлікти з антивірусним ПЗ (хибні спрацьовування)
Доступність (Availability)	Середній	Залежність від активації, ліцензійних ключів і реєстрових параметрів може ускладнювати аварійне відновлення та резервне розгортання
Сумісність	Середній	Підтримує Win32 PE-файли; стара версія не підтримує 64-бітні модулі нативно
Продуктивність	Середній	Збільшення розміру файлу на 15–30%; незначна затримка запуску (~0,3 с) через розпакування в пам'яті
Економічна доцільність (вартість)	Середній	Помірна комерційна ліцензія; для малих підрозділів економічно виправдано

Критерій	Рівень	Обґрунтування
Зручність використання	Висока	Графічний інтерфейс; пакування не потребує змін у вихідному коді
Масштабованість і підтримка	Низький	Обмежена підтримка 64-бітних систем, слабка інтеграція з CI/CD та складність модифікації захисту після пакування

Слабкі місця:

1. Легка ідентифікація – ExeinfoPE та RDG Packer Detector автоматично визначають версію AsProtect, після чого зловмисник обирає готовий спосіб розпакування.

2. Вразливість до дамп-розпакування – весь код розшифровується одночасно у пам'яті; за допомогою OllyDump або Scylla можна отримати повний незахищений образ після завантаження.

3. Зберігання параметрів захисту в реєстрі – легко видаляються або підмінюються без прав адміністратора, що знімає часові та лічильникові обмеження.

Заходи посилення (за підрозділом 2.5):

– Поєднати AsProtect із прив'язкою серійного номера до апаратного ідентифікатора (HWID): серійний номер, прив'язаний до MAC-адреси або серійного номера материнської плати, унеможливлює перенесення копії на інший пристрій (вимога «незалежність від конфігурації ПК»).

– Вбудувати антиналагоджувальні перевірки безпосередньо у вихідний код критичних функцій (IsDebuggerPresent, timing checks) ще до застосування пакувальника – це утворює двошаровий захист і підвищує стійкість.

– Застосувати SDK AsProtect для розподілу ліцензійних перевірок по всьому коду, а не лише у завантажувачі (вимога «можливість модифікації», підрозділ 2.5): навіть після зняття оболонки ліцензійні вузли залишаться у коді.

ЗАВДАННЯ 2. Порівняльний аналіз алгоритмів та методів захисту програмного забезпечення

Постановка завдання:

На основі підрозділу 2.4 посібника («Основні алгоритми та методи захисту») дослідіть два методи, що доповнюють один одного.

Оберіть одну пару методів для аналізу:

а) обфускація та мутація коду;

б) компресія та шифрування виконуваного файлу;

в) контроль цілісності та емуляція процесора.

Для обраної пари методів:

1) Опишіть принципову різницю між двома методами: мету, що саме змінюється в програмі та на якому етапі життєвого циклу застосовується кожен.

2) Наведіть по одному конкретному прикладу інструменту або бібліотеки для кожного методу та опишіть, яким чином вони перетворюють структуру або поведінку програми.

3) Складіть порівняльну таблицю за характеристиками: етап застосування у ЖЦ ПЗ, стійкість до статичного аналізу, стійкість до динамічного аналізу, вплив на продуктивність, можливість автоматизації.

4) Сформулюйте рекомендацію: у яких умовах і для яких компонентів систем ЦЗ доцільно застосовувати кожен із методів.

Приклад розв'язку:

Обрана пара: обфускація та мутація коду.

Принципова різниця:

– Обфускація – одноразове перетворення вихідного або скомпільованого коду таким чином, щоб зберегти функціональність, але максимально ускладнити розуміння алгоритму при аналізі. Застосовується на етапі підготовки дистрибутиву. Мета – протидія реверс-інжинірингу та ускладнення ручного або автоматичного аналізу коду.

– Мутація – динамічна зміна послідовності машинних команд при кожному запуску або компіляції (заміна інструкцій еквівалентними, перестановка незалежних операцій, вставка «сміттєвих» команд). Застосовується під час виконання або при кожній новій збірці. Мета – унеможливити сигнатурне порівняння: код ніколи не є однаковим у двох запусках.

Приклади інструментів:

– Обфускація: Dotfuscator Community Edition (для .NET). Вбудовується у процес збірки проєкту (MSBuild). Перейменовує класи, методи та поля на безглузді ідентифікатори (A, a0, b1), видаляє рядки налагодження, вносить фіктивні умовні гілки. Після обробки при дизасемблюванні логіка програми стає важкочитабельною навіть для досвідченого аналітика.

– Мутація: поліморфний рушій у складі Themida (опція Code Mutation). При кожній захищеній збірці критичні блоки коду отримують нову послідовність семантично еквівалентних інструкцій: замість «MOV EAX, 1» може з'явитися «XOR EAX, EAX / INC EAX», а порядок незалежних операцій довільно змінюється. Хеш-сума виконуваного образу щоразу різна, що унеможливлює сигнатурне порівняння.

Таблиця 2.8

Результати порівняння

Характеристика	**Обфускація**	**Мутація**
Етап застосування у ЖЦ	Підготовка дистрибутиву	Під час виконання або кожної збірки
Стійкість до статичного аналізу	Висока	Дуже висока (код різний у кожній версії)
Стійкість до динамічного аналізу	Середня	Середня (логіка виконується штатно)
Вплив на продуктивність	Незначний (не потребує додаткових обчислень)	Помірний (генерація варіантів потребує додаткових обчислень)
Можливість автоматизації	Висока (інтеграція в CI/CD)	Середня (потребує спеціального рушія)

Рекомендація: обфускацію доцільно застосовувати для всіх програмних компонентів систем ЦЗ як обов'язковий базовий захід при кожному релізі – вона не ускладнює архітектуру і легко автоматизується в CI/CD-конвеєрі. Мутацію варто застосовувати лише для критичних модулів (автентифікація диспетчера, шифрування оперативних даних), де пріоритетом є захист від сигнатурного

аналізу та захисту від сигнатурного аналізу та автоматизованого реверс-інжинірингу, ураховуючи вищий вплив на продуктивність і складність реалізації.

ЗАВДАННЯ 3. Проєктування системи захисту програмного забезпечення для програмного комплексу цивільного захисту

Постановка завдання:

Ознайомтеся з підрозділами 2.1-2.5 посібника.

Оберіть один із наведених програмних комплексів, що застосовуються у підрозділах цивільного захисту:

а) автоматизована система обліку матеріально-технічних ресурсів підрозділу;

б) програмний комплекс дистанційного моніторингу об'єктів підвищеної небезпеки;

в) мобільний застосунок оперативного оповіщення населення.

Для обраного комплексу:

1) Визначте тип розповсюдження та середовище розгортання (відомче, комерційне ліцензоване, відкрите тощо). Обґрунтуйте, які типи захисту з підрозділу 2.1 принципово застосовні, а які – недоцільні, та поясніть чому.

2) Класифікуйте заплановані методи захисту відповідно до підрозділу 2.4: за методом установлення, за механізмом дії та за принципом функціонування.

3) Складіть перелік конкретних алгоритмів та методів захисту з підрозділу 2.4, що будуть застосовані, та стисло обґрунтуйте вибір кожного.

4) Перевірте відповідність спроєктованої системи захисту вимогам підрозділу 2.5: заповніть таблицю відповідності, де для кожної вимоги вкажіть конкретне рішення, що її забезпечує.

Приклад розв'язку:

Обраний комплекс: програмний комплекс дистанційного моніторингу об'єктів підвищеної небезпеки.

Тип розповсюдження та середовище: внутрішнє відомче ПЗ; розгортається виключно на сертифікованих автоматизованих робочих місцях (АРМ) підрозділу

ЦЗ. Зовнішнє розповсюдження виключене; доступ – за індивідуальними обліковими даними.

Застосовність типів захисту (підрозділ 2.1):

– Захист на основі серійного номера (2.1.6) – застосовний: прив'язка ліцензії до апаратного ідентифікатора конкретного АРМ (HWID) виключає несанкціоноване перенесення.

– Secure by Design та вбудовані механізми захисту – застосовний: вбудовування перевірок цілісності та ліцензування безпосередньо у вихідний код модулів моніторингу.

– Контроль доступу та апаратна автентифікація – застосовні: автентифікація оператора через апаратний токен або смарт-картку.

– Демо-версії (2.1.1) і часові обмеження (2.1.2) – не застосовні: відомче ПЗ не потребує маркетингових обмежень.

– Захист через унікальні ідентифікатори носія (2.1.4) – не застосовний: комплекс не розповсюджується на фізичних носіях.

Класифікація методів захисту (підрозділ 2.4):

За методом установлення: комбінований – ліцензійні та цілісністні перевірки вбудовані у вихідний код (до компіляції); пакувальник накладається на готові модулі (після компіляції).

За механізмом дії: комбінований – логічні перевірки (HWID-прив'язка, перевірка токена) + криптографічні (шифрування конфігураційних файлів і ліцензійних даних за AES-256).

За принципом функціонування контроль доступу та апратна автентифікація + пакувальник (ускладнення статичного аналізу розгорнутих виконуваних модулів).

Алгоритми та методи захисту (підрозділ 2.4):

– Обфускація – застосовується до всіх модулів при збірці: ускладнює реверс-інжиніринг у разі несанкціонованого отримання виконуваних файлів.

– Компресія та шифрування – виконувані модулі упаковані пакувальником (AsProtect): перешкоджає тривіальному копіюванню та дизасемблюванню.

– Контроль цілісності – при запуску обчислюються SHA-256 хеші всіх модулів і звіряються з еталонними значеннями, збереженими у захищеному сховищі: виявляє несанкціоновану модифікацію файлів.

– Контроль доступу та апаратна автентифікація – зчитування HWID (серійний номер материнської плати, MAC-адреса мережевого адаптера) для прив'язки ліцензії до конкретного АРМ: виключає запуск на сторонньому обладнанні.

Таблиця 2.9

Таблиця відповідності вимогам

Вимога	**Реалізація у спроєктованій СЗПЗ**
Висока стійкість до злому	Комбінований захист: обфускація + пакувальник ASProtect +контроль цілісності SHA-256
Контроль середовища виконання	Прив'язка до HWID конкретного АРМ; при зміні апаратури – обов'язкова повторна реєстрація через адміністратора
Сумісність із ОС	ASProtect підтримує Win32/Win64; перевірки цілісності реалізовані через стандартний Win32 API
Низька вартість	Використання бібліотек з відкритим кодом (OpenSSL для SHA-256); ASProtect – одноразове ліцензування
Мінімальний вплив на швидкодію	Контроль цілісності виконується лише при запуску; пакувальник додає не більше 0,5 с до часу завантаження
Можливість модифікації	вбудовані механізми захисту відповідно до принципу Secure by Design
Ефективна стратегія оновлення	Еталонні хеші автоматично переобчислюються і підписуються при кожному офіційному релізі через захищений внутрішній канал розгортання

Висновок: спроєктована система захисту реалізує принцип економічної невигідності обходу захисних механізмів, за якого сукупна складність подолання апаратної прив’язки (HWID), контролю цілісності, пакування виконуваного коду та механізмів автентифікації значно перевищує потенційну вигоду від несанкціонованого використання програмного забезпечення. Такий підхід відповідає вимогам багаторівневої системи захисту та є доцільним для відомчих програмних комплексів цивільного захисту, де пріоритетними є надійність, доступність і контроль цілісності програмної реалізації.

3. ЗАГАЛЬНІ ОЗНАКИ ВРАЗЛИВОСТІ ПРОГРАМ ТА ОЗНАКИ НЕСАНКЦІОНОВАНОГО ВТРУЧАННЯ

3.1. ВРАЗЛИВОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Незважаючи на використання сучасних механізмів захисту, програмне забезпечення залишається вразливим до несанкціонованого втручання, прихованих модифікацій та впровадження шкідливого функціоналу.

Вразливість програмного забезпечення – це недолік або помилка на етапі проєктування, реалізації, конфігурації чи експлуатації, яка може бути використана для порушення конфіденційності, цілісності або доступності системи [3; 15; 36].

Вразливість слід відрізняти від загрози. Вразливість є слабким місцем системи, а загроза – потенційною можливістю використання цієї слабкості для порушення безпеки. Саме наявність вразливості створює практичні умови для реалізації загроз і перетворення потенційного ризику на реальний інцидент.

Вразливість програмного забезпечення слід розглядати як об’єктивно існуючий недолік або помилку, що виникає на будь-якому етапі життєвого циклу програмного продукту – від проєктування до експлуатації – і створює можливість для порушення властивостей безпеки системи, реалізації атак, включаючи:

– несанкціонований доступ;

– виконання довільного коду;

– впровадження програмних закладок;

– спотворення результатів обчислень.

Такими властивостями традиційно є конфіденційність, цілісність і доступність, однак у контексті систем цивільного захисту пріоритетними є насамперед цілісність і доступність інформації, оскільки спотворення або недоступність оперативних даних безпосередньо впливають на ефективність реагування на надзвичайні ситуації.

З позиції сучасних підходів до управління інформаційною безпекою, зокрема відповідно до ISO/IEC 27005, вразливість виступає зв’язувальною ланкою між загрозою та активом, визначаючи практичну можливість реалізації інциденту [15]. Саме наявність вразливості дозволяє перетворити потенційну загрозу на реальний вплив, що має вимірювані наслідки для системи. Частина таких вразливостей формується ще на етапах проєктування та розробки, що підкреслює необхідність інтеграції вимог безпеки протягом усього життєвого циклу програмного забезпечення (SDLC).

Особливістю вразливостей програмного забезпечення є їх латентний характер: за звичайних умов експлуатації вони можуть не проявлятися і не впливати на функціонування системи. Проте за спеціально сформованих умов, таких як нетипові вхідні дані, перевищення граничних значень або навмисне сформовані сценарії взаємодії, вони активуються і дозволяють змінити логіку виконання програми. Саме ця властивість робить їх зручним інструментом для прихованого впровадження шкідливого функціоналу.

У системах цивільного захисту наслідки експлуатації вразливостей набувають особливої ваги. Якщо в класичних інформаційних системах основною проблемою є витік або компрометація даних, то в системах, що забезпечують реагування на надзвичайні ситуації, критичним стає порушення доступності сервісів, затримка обробки інформації або викривлення результатів аналізу. Вразливість, яка дозволяє несанкціоновано змінити алгоритм розрахунку часу прибуття рятувальних підрозділів або пріоритетність обробки викликів, фактично трансформується у фактор ризику для життя та здоров’я людей (рис. 3.1).



Рисунок 3.1. Критичні вразливості систем цивільного захисту

Вразливості програмного забезпечення безпосередньо пов’язані з можливістю впровадження програмних закладок. Кожен тип вразливості створює відповідний технічний канал, через який може бути реалізовано той чи інший механізм втручання. Наприклад, помилки керування пам’яттю дозволяють виконувати довільний код, що відкриває шлях до встановлення серверної частини закладки, тоді як логічні помилки в алгоритмах можуть бути використані для прихованого спотворення результатів без явного втручання у код. Таким чином, вразливість, метод її експлуатації та модель програмної закладки утворюють єдиний причинно-наслідковий ланцюг, що лежить в основі більшості сучасних атак. Окремим випадком використання таких вразливостей є приховане впровадження спеціальних механізмів несанкціонованого впливу на систему, що розглядається далі у підрозділі 3.2.

3.1.1. Основні типи вразливостей та потенційні ризики безпеки

До найбільш поширених вразливостей належать [6; 35]:

1) помилки керування пам’яттю (переповнення буфера, use-after-free, double free), які можуть призводити до виконання довільного коду або аварійного завершення програми;

2) недостатня перевірка вхідних даних, що відкриває шлях до ін’єкцій (SQL, command, code injection) та підміни параметрів виконання;

3) логічні помилки в алгоритмах, коли програма коректно працює синтаксично, але допускає обхід обмежень або некоректні стани;

4) слабкі механізми автентифікації та авторизації, зокрема жорстко закодовані паролі, відсутність перевірки ролей або некоректне управління сесіями, що призводить до несанкціонованого доступ або ескалації привілеїв;

5) небезпечні виклики API та бібліотек, використання застарілих або вразливих сторонніх компонентів.

Приклад вразливості переповнення буферів:

```
# include <stdio.h>
int main ( )
{
   char String[10];
   puts ("Введіть будь-який текст:");
   gets (String);
   printf ("Ви ввели:%s", String);
}
```

Якщо користувач вводить в програму рядок довжиною дев'ять або менше символів (десятим символом є нуль, завершальний рядок), програма працює коректно.

Але якщо користувач ввів з клавіатури рядок довжиною десять символів або більше, буфер, виділений в стеці під змінну String, переповнюється, і «хвіст» введеного рядка затирає дані, які прямують у стеці безпосередньо за буфером.

Якщо подати на вхід програми спеціально підібрані дані, можна не тільки викликати в програмі помилку, але і змусити програму виконати деякі дії, спочатку не передбачені її функціональністю.

Вразливості є основним механізмом впровадження програмних закладок [37; 18]. Кожен тип вразливості пов’язаний із методом впровадження (exploit, ін’єкція, підміна бібліотек), моделлю закладки («спостерігач», «перехоплювач», «спотворення»).

Наявність вразливостей безпосередньо трансформується у ризики безпеки, які залежать від критичності програми, середовища виконання та цінності оброблюваних даних. Основні ризики включають[3; 15]:

1) порушення конфіденційності – витік персональних, комерційних або службових даних;

2) порушення цілісності – несанкціонована модифікація коду, даних або результатів обчислень;

3) порушення доступності – відмова в обслуговуванні (DoS), збої та блокування роботи системи;

4) приховане впровадження шкідливого функціоналу, зокрема програмних закладок або довільного шкідливого коду, що може тривалий час залишатися непоміченим.

Особливу небезпеку становлять вразливості, які дозволяють інжекцію або модифікацію коду без явних ознак втручання. Саме вони часто використовуються для впровадження програмних закладок, прихованих каналів зв’язку або механізмів дистанційного керування. У таких випадках ризик полягає не лише в одноразовій атаці, а у тривалій компрометації системи, що підриває довіру до програмного продукту в цілому.

У системах цивільного захисту особливо критичними є вразливості, що впливають на час реагування (RTO), призводять до втрати або затримки даних (RPO), або викликають деградацію функціональності системи. Особливо критичними є спотворення результатів (можуть призвести до неправильних рішень) та втрата доступності (затримка реагування).

Рівень ризику значно зростає за наявності таких факторів [6; 7]:

1) складна або погано документована архітектура програм;

2) відсутність контролю цілісності та аудиту змін;

3) використання застарілих бібліотек і компонентів;

4) недостатнє тестування, зокрема безпекове (security testing);

5) відсутність процесів управління вразливостями та оновленнями.

Таким чином, вразливості програмного забезпечення є ключовою причиною виникнення ризиків інформаційної безпеки. Вони можуть призводити як до явних порушень роботи програм, так і до прихованих атак із довготривалими наслідками. Ефективне зниження ризиків можливе лише за умови системного підходу, що поєднує безпечне проєктування, аналіз коду, тестування, контроль цілісності та постійний моніторинг стану програмного середовища.

3.1.2. Ознаки несанкціонованого втручання

Несанкціоноване втручання в роботу програмного забезпечення являє собою будь-яку зміну коду, логіки виконання або середовища функціонування системи, що здійснюється без дозволу власника або адміністратора [3; 31]. Таке втручання може бути як явним, так і прихованим, причому останній варіант є значно небезпечнішим, оскільки дозволяє тривалий час експлуатувати систему без виявлення (рис. 3.2).

З наукової точки зору ознаки втручання доцільно трактувати як індикатори компрометації або аномальної поведінки системи. Вони можуть проявлятися на різних рівнях – від зміни поведінки прикладного програмного забезпечення до відхилень у мережевій активності або структурі пам’яті процесу. Найбільш типовою ознакою є відхилення від штатної логіки роботи, коли програма починає виконувати дії, не передбачені її функціональним призначенням, або демонструє нестабільну поведінку без очевидних причин.

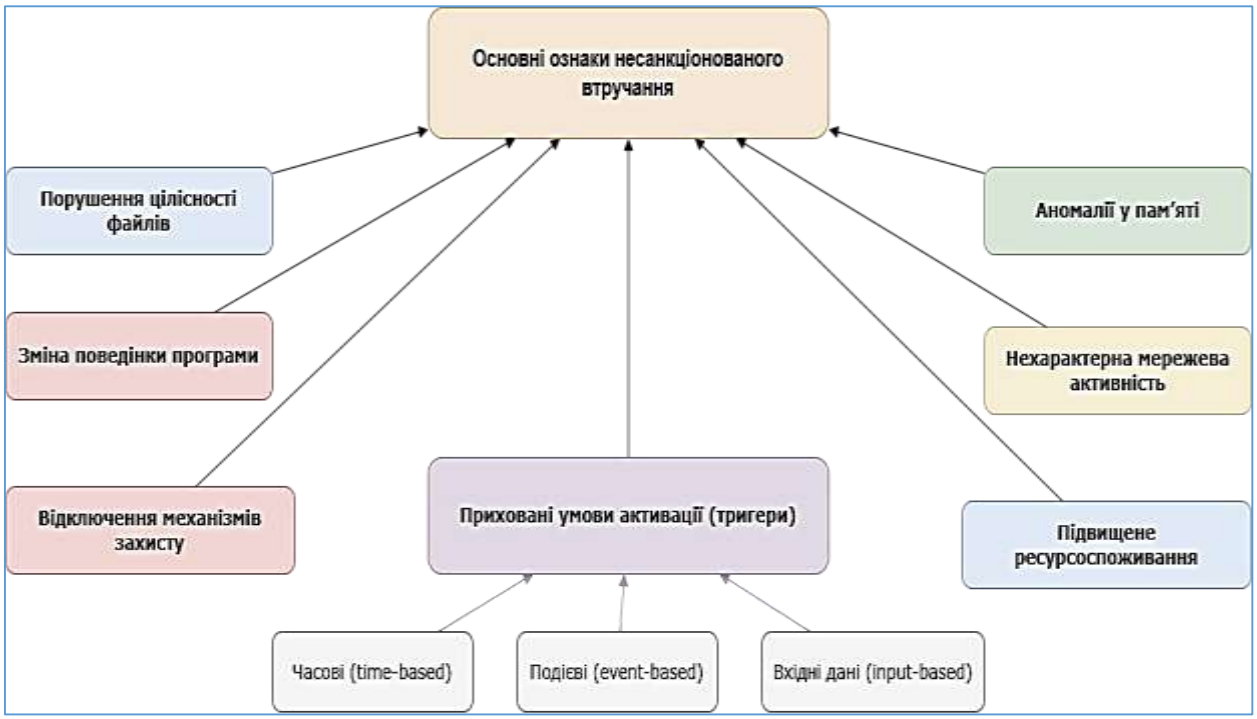


Рисунок 3.2. Основні ознаки несанкціонованого втручання

Окрему групу становлять ознаки, пов’язані з порушенням цілісності програмного коду або виконуваних файлів. Такі зміни можуть бути мінімальними з точки зору обсягу, але суттєвими за наслідками, оскільки навіть незначна модифікація умов переходу або додавання декількох інструкцій здатні повністю змінити логіку роботи системи. У сучасних атаках такі зміни часто поєднуються з використанням легітимних механізмів операційної системи, що ускладнює їх виявлення. До практичних ознак такого втручання належать зміна контрольних хеш-сум файлів, невідповідність цифрового підпису, поява нових записів автозапуску, нехарактерні мережеві з’єднання, аномалії журналів подій та зміна часових міток системних файлів [31; 23].

Важливою характеристикою несанкціонованого втручання є наявність прихованих умов активації. Такі умови можуть бути пов’язані з конкретними значеннями вхідних даних, часовими параметрами або певними подіями в системі. За відсутності цих умов програма функціонує штатно, що створює ілюзію її коректної роботи та значно ускладнює виявлення втручання під час стандартного тестування.

У контексті систем цивільного захисту особливо небезпечними є втручання, які не призводять до явного порушення працездатності системи, але змінюють її поведінку у критичних ситуаціях. Наприклад, система може коректно обробляти більшість запитів, але за певних умов ігнорувати виклики з високим пріоритетом або занижувати оцінки ризику. Такі втручання безпосередньо впливають на ефективність реагування і можуть мати серйозні наслідки, навіть якщо вони залишаються непоміченими з технічної точки зору.

Ознаки несанкціонованого втручання часто залежать від моделі функціонування шкідливого механізму: приховане спостереження, перехоплення інформаційних потоків або спотворення результатів обробки. Саме ці моделі взаємодії будуть детально розглянуті у підрозділі 3.3.

3.2. ПРОГРАМНІ ЗАКЛАДКИ

Програмна закладка є одним із найбільш складних і небезпечних видів прихованих загроз [38; 39], оскільки вона поєднує у собі властивості вразливості, механізму експлуатації та інструменту довготривалого контролю над системою. Під програмною закладкою розуміють прихований механізм несанкціонованого впливу на систему, який може використовуватися для обходу штатних засобів захисту або отримання прихованого доступу до функцій програмного забезпечення.

На відміну від звичайних вразливостей, які лише створюють потенційну можливість для атаки, програмна закладка вже є реалізованим механізмом втручання. Вона може бути впроваджена на різних етапах життєвого циклу програмного забезпечення, включаючи етап розробки, оновлення або експлуатації, і часто використовує вже наявні вразливості як канал проникнення.

У системах цивільного захисту програмні закладки становлять особливу загрозу через свою прихованість і здатність впливати на критичні функції без явних ознак порушення. Вони можуть використовуватися для дистанційного контролю, збору оперативної інформації, зміни результатів аналізу або порушення координації дій між підрозділами. Найбільш небезпечними є

закладки, які реалізують приховане спотворення даних або логіки роботи системи, оскільки їх виявлення потребує глибокого аналізу і не може бути здійснене лише за допомогою стандартних засобів захисту.

3.2.1. Визначення і класифікація програмних закладок

Програмна закладка (software backdoor) – прихований механізм у програмному забезпеченні, що дозволяє обійти штатні механізми автентифікації або змінювати поведінку системи без відома власника [38; 39]. Закладки можуть бути свідомо вбудовані розробником, впроваджені під час компрометації коду або додані шкідливим програмним забезпеченням.

На відміну від звичайних помилок або вразливостей, програмна закладка є вже реалізованим механізмом втручання, який забезпечує можливість тривалого прихованого доступу або впливу на систему. Вона може використовуватися для дистанційного керування, збору інформації, модифікації даних або порушення логіки функціонування програмного забезпечення. Не кожне шкідливе програмне забезпечення є програмною закладкою, але кожна програмна закладка містить прихований несанкціонований функціонал, спрямований на контроль або вплив на систему.

Програмні закладки класифікують за рядом ознак [38; 39], що відображають їх функціональне призначення, спосіб реалізації, здатність до розповсюдження та рівень видимості. Така класифікація дозволяє систематизувати підходи до їх аналізу та виявлення. Класифікація програмних закладок відображена на рисунку 3.3.

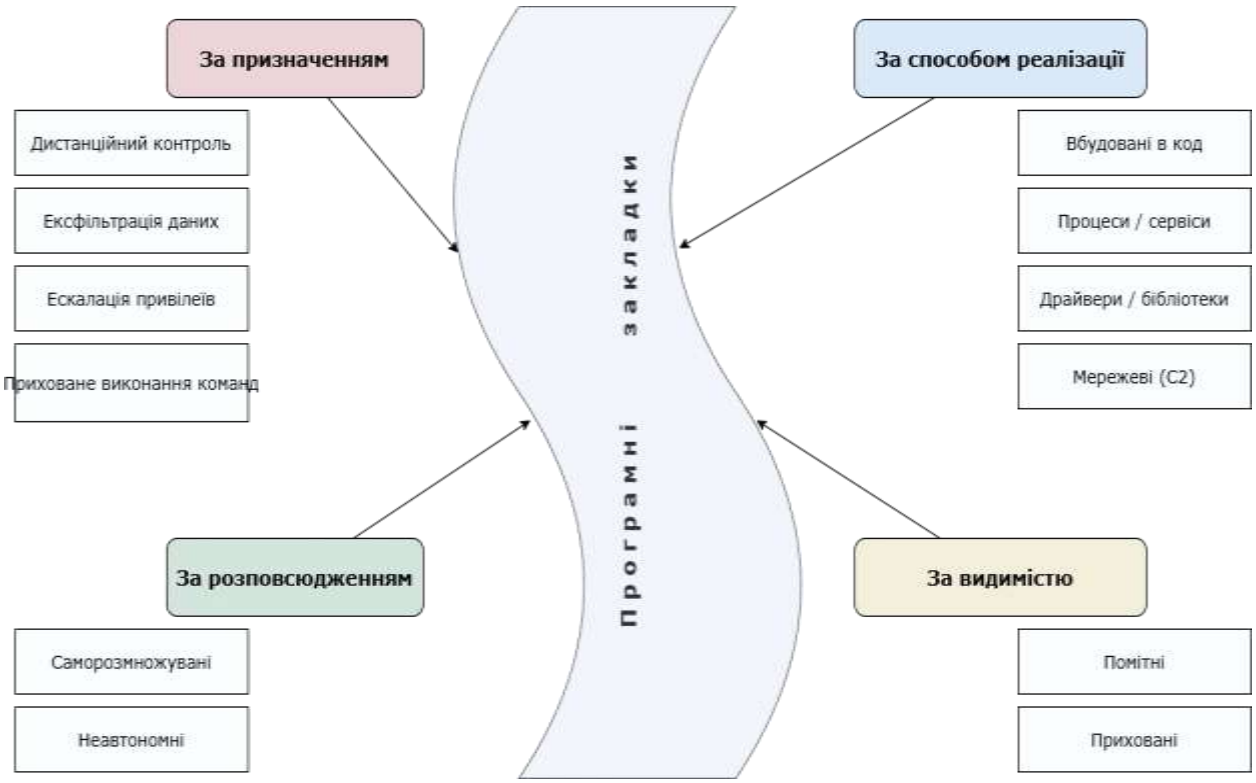


Рисунок 3.3. Класифікація програмних закладок

Програмні закладки відрізняються за функціональним призначенням, способом реалізації, рівнем інтеграції в систему та характером прояву. Їх класифікація дозволяє систематизувати підходи до аналізу та виявлення і використовується під час виконання практичних завдань цього розділу.

Залежно від призначення закладки можуть забезпечувати дистанційний контроль системи, виконання команд у прихованому режимі, ексфільтрацію даних або отримання підвищених привілеїв. Ці функції визначають характер їх впливу на інформаційні процеси та можливі наслідки для системи.

За способом реалізації закладки можуть бути інтегровані безпосередньо у програмний код або існувати як окремі компоненти, що взаємодіють із системою через процеси, служби, драйвери або мережеві механізми. Рівень інтеграції визначає як ефективність функціонування закладки, так і складність її виявлення.

Здатність до розповсюдження визначає характер їх поширення: одні закладки можуть самостійно копіюватися та поширюватися, поєднуючи властивості шкідливого програмного забезпечення, інші встановлюються цілеспрямовано і не мають механізмів саморозмноження.

Рівень видимості закладки визначає можливість її виявлення. Помітні закладки залишають артефакти у файловій системі або процесах, тоді як приховані маскуються під легітимні компоненти або використовують стандартні механізми взаємодії, що суттєво ускладнює їх ідентифікацію.

Програмна закладка завжди є прихованим або потенційно шкідливим функціоналом і не має відношення до легітимних механізмів збереження даних чи налаштувань. Її основна мета полягає у створенні каналу несанкціонованого доступу або впливу на роботу системи.

Для кращого розуміння взаємозв'язку між ознаками класифікації їх узагальнену структуру доцільно подати у вигляді таблиці (таблиця 3.1).

Наведені у таблиці 3.1 класифікаційні ознаки безпосередньо пов'язані з моделями взаємодії програмних закладок із системою (підрозділ 3.3) і використовуються для визначення типу закладки, способу її впровадження та методів виявлення у практичних завданнях.

Таблиця 3.1

Класифікація програмних закладок

Критерій класифікації	**Тип програмної закладки**	**Характеристика**
За способом реалізації	Вбудовані в код програми	Інтегровані безпосередньо в програмний код (на етапі розробки або модифікації).
	Окремі процеси / сервіси	Реалізуються як незалежні виконувані модулі, що працюють у фоні.
	Драйвери або модифікації бібліотек	Впроваджуються на рівні системних компонентів, забезпечуючи глибокий контроль.
	Мережеві компоненти (сервер/клієнт)	Функціонують через мережеву взаємодію, забезпечуючи віддалений доступ.
За призначенням	Дистанційний контроль	Забезпечує віддалене управління системою (запуск процесів, доступ до файлів, адміністрування).
	Ексфільтрація даних	Призначена для прихованого збору та передавання конфіденційної інформації.

Критерій класифікації	Тип програмної закладки	Характеристика
	Ескалація привілеїв	Дозволяє отримати розширені права доступу в системі.
	Приховане виконання команд	Виконує команди без відома користувача або адміністратора.
За здатністю до розмноження	Віруси-закладки	Мають здатність до саморозповсюдження (копіювання та зараження інших компонентів).
	Неавтономні бекдори	Не мають механізмів самопоширення, встановлюються вручну або через експлойт.
За видимістю	Помітні	Залишають сліди у системі (процеси, файли, записи реєстру).
	Невидимі	Маскуються під легітимні процеси або протоколи, складні для виявлення.

Наведені у таблиці 3.1 класифікаційні ознаки безпосередньо пов’язані з моделями взаємодії програмних закладок із системою [38].

У системах цивільного захисту найбільшу загрозу становлять закладки, що поєднують прихованість, можливість дистанційного керування та вплив на результати обробки інформації, оскільки вони здатні непомітно змінювати логіку роботи системи та впливати на процеси прийняття рішень. Найбільшими ризиками, пов’язаними з впровадженням програмних закладок, є порушення координації дій служб, витік оперативної інформації, спотворення результатів обробки даних та можливість дистанційного управління системами зловмисником.

3.2.2. Типи програмних закладок

Залежно від способу функціонування та характеру впливу програмні закладки можуть набувати різних форм, що відрізняються за функціональним призначенням, способом реалізації, рівнем інтеграції в систему та характером прояву. Така варіативність обумовлює необхідність їх класифікації, яка дозволяє

систематизувати підходи до аналізу, виявлення та протидії, а також використовується під час виконання практичних завдань цього розділу.

Залежно від призначення програмні закладки можуть бути орієнтовані на дистанційний контроль системи, приховане виконання команд, ексфільтрацію даних або отримання розширених прав доступу. Ці функції визначають основну мету втручання та характер впливу на інформаційні процеси.

За способом реалізації закладки можуть бути інтегровані безпосередньо у програмний код або функціонувати як окремі програмні компоненти. У першому випадку вони маскуються під легітимну логіку роботи програми, тоді як у другому – взаємодіють із системою через процеси, служби, драйвери або мережеві механізми. Рівень інтеграції визначає як ефективність закладки, так і складність її виявлення.

Важливою ознакою є здатність до розповсюдження. Деякі закладки можуть поєднувати ознаки шкідливого програмного забезпечення (malware) і самостійно копіюватися та поширюватися, тоді як інші встановлюються цілеспрямовано і не мають механізмів саморозмноження.

З точки зору виявлення принципове значення має рівень видимості закладки. Помітні закладки залишають артефакти у файловій системі або процесах, тоді як приховані маскуються під легітимні компоненти або використовують стандартні механізми взаємодії операційної системи, що значно ускладнює їх ідентифікацію.

Наведені класифікаційні ознаки узагальнені у таблиці 3.1 та використовуються як основа для подальшого аналізу моделей взаємодії програмних закладок із системою (підрозділ 3.3), а також для виконання практичних завдань, пов’язаних із визначенням типу закладки, способу її впровадження та методів виявлення.

У системах цивільного захисту найбільшу загрозу становлять закладки, що поєднують прихованість, можливість дистанційного керування та вплив на результати обробки інформації. Такі закладки здатні не лише забезпечувати несанкціонований доступ, але й порушувати достовірність даних і логіку

прийняття рішень, що безпосередньо впливає на ефективність реагування на надзвичайні ситуації та створює ризик прийняття неправильних управлінських рішень у критичних умовах.

3.3. МОДЕЛІ ВЗАЄМОДІЇ ЗАКЛАДКИ З СИСТЕМОЮ

Взаємодія програмної закладки з інформаційною системою може бути формалізована як сукупність процесів впливу на інформаційні потоки, об’єкти та суб’єкти системи. Для опису таких процесів доцільно використовувати суб’єктно-орієнтовану модель, у межах якої система розглядається як множина суб’єктів (процеси, користувачі, служби) та об’єктів (файли, пам’ять, мережеві ресурси), між якими здійснюються операції доступу та обробки інформації.

У межах такої моделі програмна закладка виступає як додатковий суб’єкт, що ініціює або модифікує інформаційні потоки, порушуючи встановлені правила взаємодії. Залежно від характеру впливу на систему виділяють три базові моделі функціонування програмних закладок: «спостерігач», «перехоплювач» і «спотворення» [38; 39], (рис. 3.4).

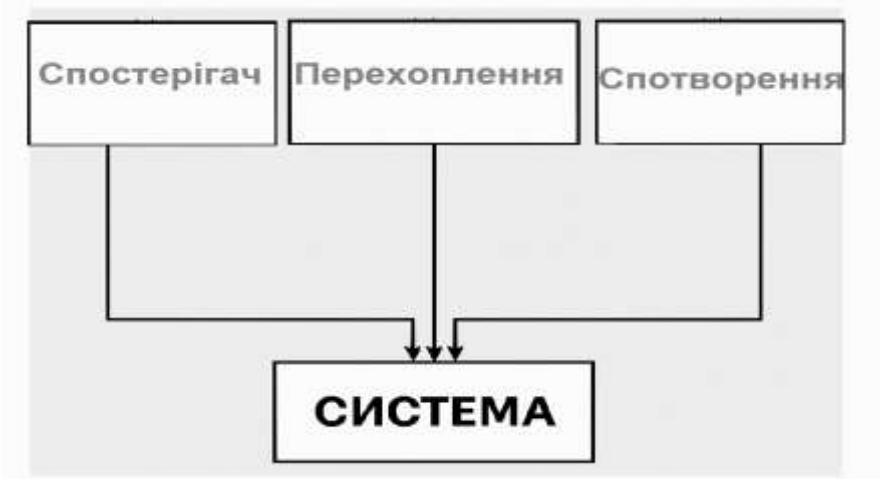


Рисунок 3.4. Моделі взаємодії програмних закладок з системою

Кожна з моделей відображає різний рівень активності закладки та ступінь впливу на функціонування системи – від пасивного збору інформації до активної зміни результатів обробки даних. Такий підхід дозволяє формалізувати аналіз загроз і безпосередньо використовується при виконанні практичних завдань цього розділу. Ознаки несанкціонованого втручання, розглянуті у підрозділі

3.1.2, безпосередньо залежать від моделі функціонування закладки: спостереження, перехоплення інформаційних потоків або спотворення результатів обробки.

3.3.1. Модель «спостерігач»

Модель «спостерігач» передбачає пасивну взаємодію програмної закладки з системою, за якої вона здійснює збір інформації без безпосереднього втручання у логіку виконання програм. У цьому випадку закладка функціонує як прихований суб'єкт, що відстежує події в системі, аналізує введення користувача або мережеву активність і передає отримані дані зовнішньому суб'єкту.

Узагальнену схему функціонування програмної закладки моделі «спостерігач» подано нижче (рис. 3.5).



Рисунок 3.5. Схема програмної закладки моделі «спостерігач»

Особливістю цієї моделі є мінімальний вплив на функціонування системи. Закладка не змінює результати обчислень і не викликає явних збоїв, що дозволяє їй тривалий час залишатися непоміченою стандартними засобами контролю. Саме ця латентність робить модель «спостерігач» однією з найбільш

небезпечних, оскільки компрометація може тривалий час залишатися непоміченою.

Типовою реалізацією цієї моделі є клієнт-серверна архітектура. У такій схемі серверна частина закладки встановлюється на цільовій системі, тоді як клієнтська частина використовується порушником для отримання інформації або керування. Взаємодія між компонентами здійснюється через мережеві протоколи, причому для маскування часто використовуються легітимні протоколи прикладного рівня (наприклад, HTTP або SMTP).

Як приклад реалізації моделі «спостерігач» можна розглянути перехоплення клавіатурного введення (keylogging), коли всі введені користувачем дані фіксуються та передаються зловмиснику [38; 18; 40]. Приклад програмного коду перехоплення введення даних:

```
while (true)
{
  char c = getch();
  log_to_file(c);
  send_to_remote(c);
}
```

На практиці програмні закладки цього типу широко використовуються для організації прихованого моніторингу системи та збору інформації. Вони можуть застосовуватися для первинного отримання доступу до системи, збору облікових даних, передачі інформації іншим компонентам шкідливого програмного забезпечення або підготовки подальших атак [40].

У сучасних операційних системах такі закладки часто реалізуються у вигляді окремих процесів або служб, що працюють у фоновому режимі та забезпечують прихований канал зв’язку із зовнішнім суб’єктом. Для забезпечення стійкості до перезавантаження системи можуть використовуватися механізми автозапуску.

Зокрема, в операційних системах сімейства Windows типовим способом реалізації автозапуску є використання ключів системного реєстру [26], таких як: HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run

HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run

Ці механізми дозволяють забезпечити запуск закладки під час старту системи та її подальше функціонування без участі користувача.

Для приховання своєї активності закладки можуть маскувати мережеву взаємодію під легітимні протоколи, що значно ускладнює їх виявлення засобами мережевого моніторингу. Додатковими ознаками такої активності можуть бути аномальні DNS-запити, нехарактерні вихідні з’єднання або періодичний обмін даними з віддаленими вузлами, що не відповідають штатній логіці роботи системи.

Програмні закладки моделі «спостерігач» можуть поєднувати у собі ознаки інших моделей, що ускладнює їх класифікацію та виявлення. Зокрема, відомі програмні засоби, які реалізують переважно функції спостереження, але водночас містять елементи активного втручання.

Як приклад можна розглянути програмну закладку Pinch, яка в основному реалізує функції збору інформації (перехоплення введення, зчитування збережених облікових даних), але також може виконувати окремі дії, характерні для моделі «перехоплювач». Це свідчить про те, що в реальних умовах програмні закладки часто мають комбінований характер і не обмежуються однією моделлю взаємодії.

У системах цивільного захисту модель «спостерігач» становить загрозу через можливість витоку оперативної інформації, включаючи дані про надзвичайні ситуації, маршрути реагування, стан інфраструктури або персональні дані. Крім того, такі закладки можуть використовуватися як підготовчий етап для більш складних атак, що реалізуються за моделями «перехоплювач» або «спотворення». Особливо небезпечним є те, що така компрометація часто не супроводжується явними ознаками відмови системи, а тому може залишатися невиявленою протягом тривалого часу.

3.3.2. Модель «перехоплювач»

Модель «перехоплювач» характеризується активним втручанням у потоки даних або виклики функцій. У цьому випадку програмна закладка інтегрується у критичні точки обробки інформації, де вона може не лише аналізувати, але й змінювати передані дані (рис.3.6).

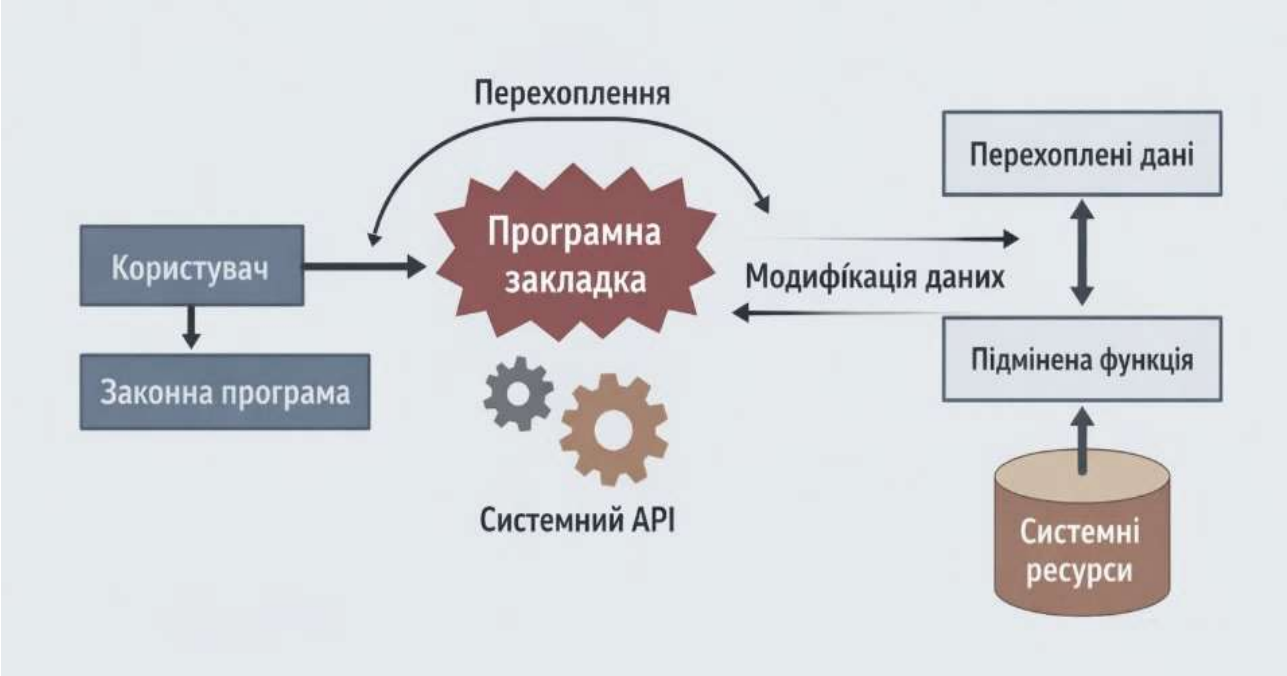


Рисунок 3.6. Схема програмної закладки моделі «перехоплювач»

На відміну від моделі «спостерігач», де закладка виконує переважно пасивний збір інформації, модель «перехоплювач» передбачає безпосередній вплив на процес обробки даних, що значно підвищує рівень ризику для системи.

Одним із типових способів реалізації моделі «перехоплювач» є перехоплення викликів системних або прикладних функцій (API hooking) [26; 19]. У цьому випадку програмна закладка змінює адресу виклику функції або підміняє її реалізацію, що дозволяє контролювати або модифікувати дані, які передаються.

Наприклад, перехоплення функції авторизації може виглядати так:

```
int auth(char* login, char* password){
      if (strcmp(login, "admin") == 0) return 1;       // обхід перевірки
      return original_auth(login, password);
}
```

У наведеному прикладі виклик функції автентифікації модифікується таким чином, що для певного користувача перевірка пароля фактично ігнорується. Це дозволяє отримати несанкціонований доступ до системи без зміни її зовнішньої поведінки.

До інших типових реалізацій цієї моделі належать перехоплення мережевого трафіку, підміна системних бібліотек (DLL hijacking), модифікація параметрів міжпроцесної взаємодії або втручання у механізми журналювання подій [26; 19]. У всіх випадках закладка розміщується у точці, через яку проходить критичний інформаційний потік.

Для систем цивільного захисту така модель є критичною, оскільки дозволяє:

– змінювати дані моніторингу;

– підміняти інформацію про стан об’єктів;

– перехоплювати та змінювати команди управління.

Це може призвести до некоректних рішень або порушення координації дій. Особливо небезпечним є те, що зовні система може продовжувати функціонувати штатно, тоді як ключові управлінські рішення вже приймаються на основі змінених або підроблених даних.

3.3.3. Модель «спотворення»

Модель «спотворення» характеризується активним і цілеспрямованим впливом програмної закладки на результати обробки інформації або логіку функціонування системи [38; 39]. На відміну від моделей «спостерігач» і «перехоплювач», у цьому випадку закладка не лише аналізує або перехоплює дані, а змінює їх або алгоритми їх обробки, що призводить до формування некоректних результатів.

Принцип функціонування програмної закладки цієї моделі полягає у її вбудуванні в критичні ділянки програмного коду – алгоритми обчислень, модулі прийняття рішень, механізми перевірки умов або формування вихідних результатів. Активація закладки відбувається за певними умовами (тригерами),

які можуть мати часовий, подієвий або логічний характер. Після активації вона змінює значення змінних, параметрів або умов переходу, що призводить до навмисного викривлення результатів обчислень або поведінки системи (рис.3.7).



Рисунок 3.7. Схема програмної закладки моделі «спотворення»

Характерною особливістю цієї моделі є відсутність явних ознак порушення: система продовжує функціонувати без аварій, однак результати її роботи стають недостовірними. Це значно ускладнює виявлення закладки стандартними засобами контролю цілісності. Саме тому модель «спотворення» вважається найбільш небезпечною для критичних систем, де помилкове рішення може мати серйозні наслідки без явного технічного збою.

Однією з типових форм реалізації є зміна логіки прийняття рішень або пріоритезації. Наприклад, штучне заниження пріоритету події за певних умов:

```
if (event.type == CRITICAL && current_date == "01.01") {
     priority = LOW; // навмисне заниження пріоритету
} else {
     priority = calculate_priority(event);
}
```

У цьому прикладі система в загальному випадку працює коректно, але за певної умови (дата) відбувається викривлення результату. Така зміна може бути практично непомітною при звичайному тестуванні.

Програмні закладки моделі «спотворення» можуть реалізовуватися у вигляді:

– прихованих умов у коді, що активуються лише за специфічних значень параметрів;

– модифікації алгоритмів (наприклад, внесення систематичної похибки в розрахунки);

– підміни результатів функцій або методів;

– маніпуляції з конфігураційними файлами або параметрами середовища;

– вибіркового викривлення даних (лише для окремих користувачів, часових періодів або типів запитів).

На відміну від явних помилок, такі зміни виглядають як допустимі варіації або особливості реалізації, що ускладнює їх виявлення.

Особливо складними для виявлення є закладки, що реалізуються із застосуванням прихованого службового коду, коли фрагменти програми формально виконують захисні або діагностичні функції, але фактично містять прихований шкідливий функціонал.

У таких випадках закладка маскується під додаткову перевірку безпеки або механізм обробки винятків, оптимізаційний або діагностичний код:

```
if (check_security_condition(user)) {
    process_request();
} else {
    //прихована модифікація результату
    process_request_with_adjustment(user);
}
```

Суть цього прикладу така: небезпечна логіка ховається не в явному шкідливому коді, а в правдоподібній альтернативній обробці. Саме тому подібні конструкції є характерними для програмних закладок моделі «спотворення»: система не ламається, не сигналізує про помилку, але починає видавати викривлені результати.

Функція check_security_condition(user) зовні виглядає як стандартна перевірка певної умови безпеки: перевірка ролі користувача, коректності токена,

цілісності сесії, або відповідності внутрішнім політикам. Якщо перевірка повертає істину, виконується штатна обробка запиту через process_request().

Проблема прихована в гілці else. Замість блокування запиту або фіксації інциденту викликається process_request_with_adjustment(user), де може бути реалізований прихований механізм спотворення.

Наприклад, ця функція може:

– занижувати пріоритет критичного повідомлення;

– змінювати значення результату розрахунку;

– приховувати частину даних;

– видавати користувачу формально коректну, але викривлену відповідь;

– пропускати заборонений запит із модифікованими параметрами;

– вести приховане журналювання або передачу даних.

Тобто код не блокує підозрілу ситуацію, а переводить її в альтернативний режим обробки, у якому і відбувається спотворення.

Такий фрагмент є ілюстрацією використання зловмисного коду, тому що шкідлива логіка:

– не виглядає як очевидна атака;

– захована в “службовій” або “захисній” конструкції;

– може бути пояснена як особливий режим роботи системи.

Аудитор може вирішити, що це “обробка нестандартної ситуації”, “режим деградації”, “резервна логіка”, або “м’яка відмова безпеки”.

Саме це і робить таку закладку особливо небезпечною.

Цей код належить саме до моделі «спотворення», а не «спостерігач» і не «перехоплювач», тому що він не просто збирає інформацію, не просто перехоплює виклик, а змінює результат обробки запиту. Тобто система формально працює, але її поведінка вже інша, ніж мала б бути.

У системі цивільного захисту аналогічна логіка може бути особливо небезпечною й приводити, наприклад, до випадків, відображених у Таблиці 3.2.

Таблиця 3.2

Наслідки реалізації коду типу «спотворення»

№	Опис	Наслідок
1	Якщо перевірка безпеки не проходить, система замість блокування може передати подію в менш пріоритетну чергу. У результаті виклик, який мав бути оброблений негайно, обробляється із затримкою	Заниження пріоритету події
2	Система оцінювання ризику може повертати “скориговане” значення небезпеки, яке буде нижчим за реальне. Це призведе до помилкового рішення	Спотворення результату аналізу
3	У модулі маршрутизації можна змінити рекомендований маршрут або склад сил реагування, не викликаючи аварії програми	Приховане викривлення координаційних даних

Такий код важко виявити, тому що:

– назви функцій можуть виглядати правдоподібно;

– синтаксично код правильний;

– він не викликає падіння програми;

– при більшості тестів може не активуватись;

– шкідлива поведінка проявляється лише в окремій гілці виконання.

Тобто стандартний контроль цілісності файлів не дозволяє виявити логічно небезпечну поведінку. Для цього необхідний аналіз алгоритмічної логіки роботи програми.

Щоб виявити подібний фрагмент, потрібно перевіряти:

– що саме робить альтернативна гілка else;

– чи відповідає її логіка задокументованим вимогам;

– чи не змінює вона критичні параметри;

– чи є підстави взагалі обробляти запит після невдалої перевірки безпеки;

– чи не маскується шкідлива дія під “корекцію”, “адаптацію”, “fallback” або “optimization”.

Таким чином, у системах цивільного захисту модель «спотворення» становить максимальну загрозу, оскільки може призводити до неправильного

визначення рівня небезпеки, некоректного розподілу ресурсів, порушення пріоритетів реагування, втрати контролю над ситуацією.

3.3.4. Приклади відомих закладок

Практичне розуміння сутності програмних закладок досягається через аналіз реальних прикладів їх використання у програмному забезпеченні, мережевих системах та криптографічних механізмах. Такі приклади демонструють різні способи реалізації закладок, рівні їх інтеграції та характер впливу на систему.

Одним із класичних прикладів програмної закладки є система віддаленого адміністрування Back Orifice, створена у 1998 році [38]. Вона використовувала клієнт-серверну архітектуру, у якій серверна частина встановлювалася на цільовій системі, а клієнтська – забезпечувала дистанційне керування.

Загальний вид інтерфейсу системи відображено на рис. 3.8.



Рисунок 3.8. Інтерфейс Back Orifice

Функціональні можливості Back Orifice включали:

- повний віддалений контроль системи;
- доступ до файлової системи;
- приховане виконання команд;

– можливість маскування присутності.

Попри заявлену мету демонстрації вразливостей, система широко використовувалася як шкідливе ПЗ. Закладка дозволяє зловмиснику здійснювати практично необмежений дистанційний контроль над операційною системою Windows (запуск програм, доступ до файлів, перехоплення введення, моніторинг активності). Особливістю Back Orifice є маскування під легітимні процеси та використання мережевих каналів для прихованої взаємодії, що ускладнює її виявлення стандартними засобами безпеки й робить її типовим прикладом закладки моделі «спостерігач» із елементами «перехоплювача».

Іншим прикладом є програма NetBus, яка також забезпечувала віддалене керування комп'ютером через мережу [38]. Вона була розроблена у 1998 році та швидко отримала поширення як інструмент несанкціонованого доступу (рис. 3.9). Функціонально вона була подібною до Back Orifice, але орієнтувалася насамперед на середовище Windows та серверні служби.

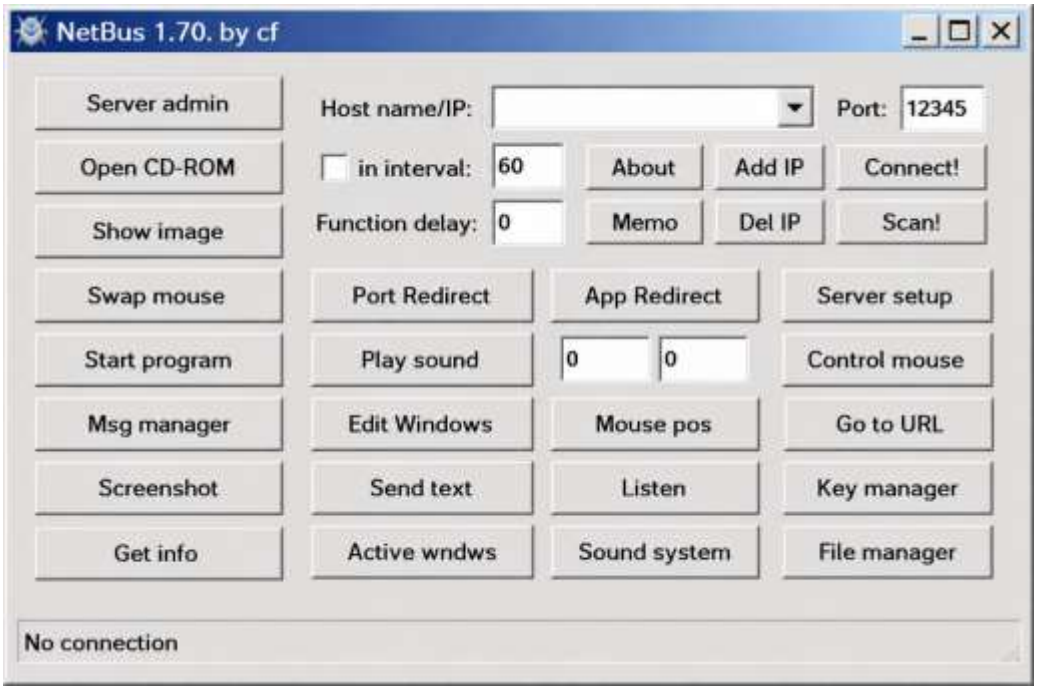


Рисунок 3.9. Інтерфейс NetBus

NetBus дозволяв керувати процесами; передавати файли; виконувати команди віддалено; здійснювати прихований моніторинг.

Цей приклад демонструє, що навіть програмні засоби, позиціоновані як адміністративні, можуть фактично виконувати функції програмних закладок.

Особливу категорію становлять закладки, вбудовані у криптографічні механізми. Найбільш відомим прикладом є генератор псевдовипадкових чисел Dual_EC_DRBG, який використовувався у стандартах NIST [5; 39].

Було встановлено, що цей алгоритм (рисунок 3.10) містив потенційний бекдор, який дозволяв передбачати згенеровані значення та, відповідно, відновлювати ключову інформацію. Це модель спотворення, а особливістю цього випадку є те, що закладка була інтегрована на рівні стандарту, непомітна для звичайного користувача, критичною для безпеки всієї системи.

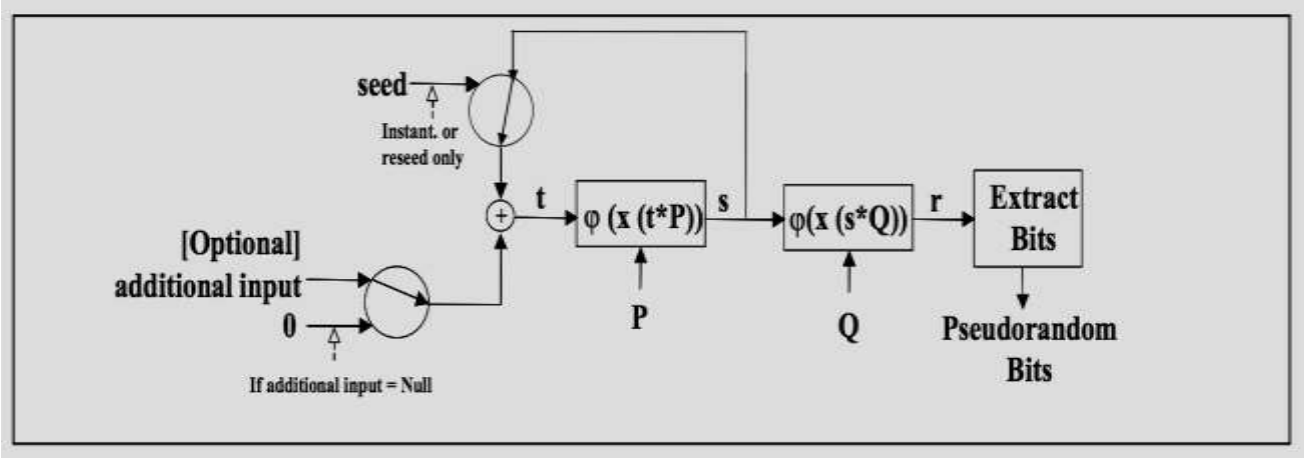


Рисунок 3.10. Схема роботи Dual_EC_DRBG

До закладок, вбудованих у криптографічні механізми можна віднести Crypto AG Backdoor, Juniper Dual EC Backdoor, Debian OpenSSL Weak Keys [39]. Ці закладки особливо небезпечні через високий рівень довіри до криптографічних компонентів. Компрометація таких механізмів призводить не лише до локального порушення безпеки, а й до системного зниження довіри до всієї інфраструктури захисту.

Програмна закладка Pinch є прикладом гібридного підходу, що поєднує функції спостереження та перехоплення. Вона орієнтована на перехоплення клавіатурного введення, зчитування збережених паролів, збір конфіденційної інформації.

Особливістю Pinch є те, що вона може не лише збирати інформацію (модель «спостерігач»), але й впливати на процеси обробки даних, що наближає її до моделі «перехоплювач». Цей приклад демонструє, що в реальних умовах

програмні закладки зазвичай не належать до однієї моделі і часто поєднують кілька механізмів впливу.

У системах цивільного захисту програмні закладки можуть мати значно більш критичні наслідки, ніж у звичайних інформаційних системах.

Розглянемо умовний приклад, коли у системі моніторингу надзвичайних ситуацій реалізовано модуль оцінки рівня небезпеки, який аналізує дані з датчиків і формує пріоритет реагування. У програмний код цього модуля вбудовується закладка, яка активується за певних умов (наприклад, для конкретного регіону або типу події) і змінює результати оцінювання.

Зовні система працює у штатному режимі, не генерує помилок, формує формально коректні звіти.

Хоча спостерігається деяка затримка в передачі сигнала, система показує, що рівень небезпеки знижується, тому ця подія класифікується як не критична.

Але фактично реалізується модель «спотворення», яка призводить до неправильного визначення пріоритетів; запізнення реагування служб; неефективного використання ресурсів; потенційного збільшення масштабів наслідків надзвичайної ситуації.

Такий приклад демонструє, що навіть незначна модифікація алгоритму може мати критичні наслідки для безпеки та життєдіяльності населення.

Для ілюстрації середовища, у якому можуть реалізовуватися програмні закладки, розглянемо типовий приклад архітектури системи критичної інфраструктури (рис.3.11).

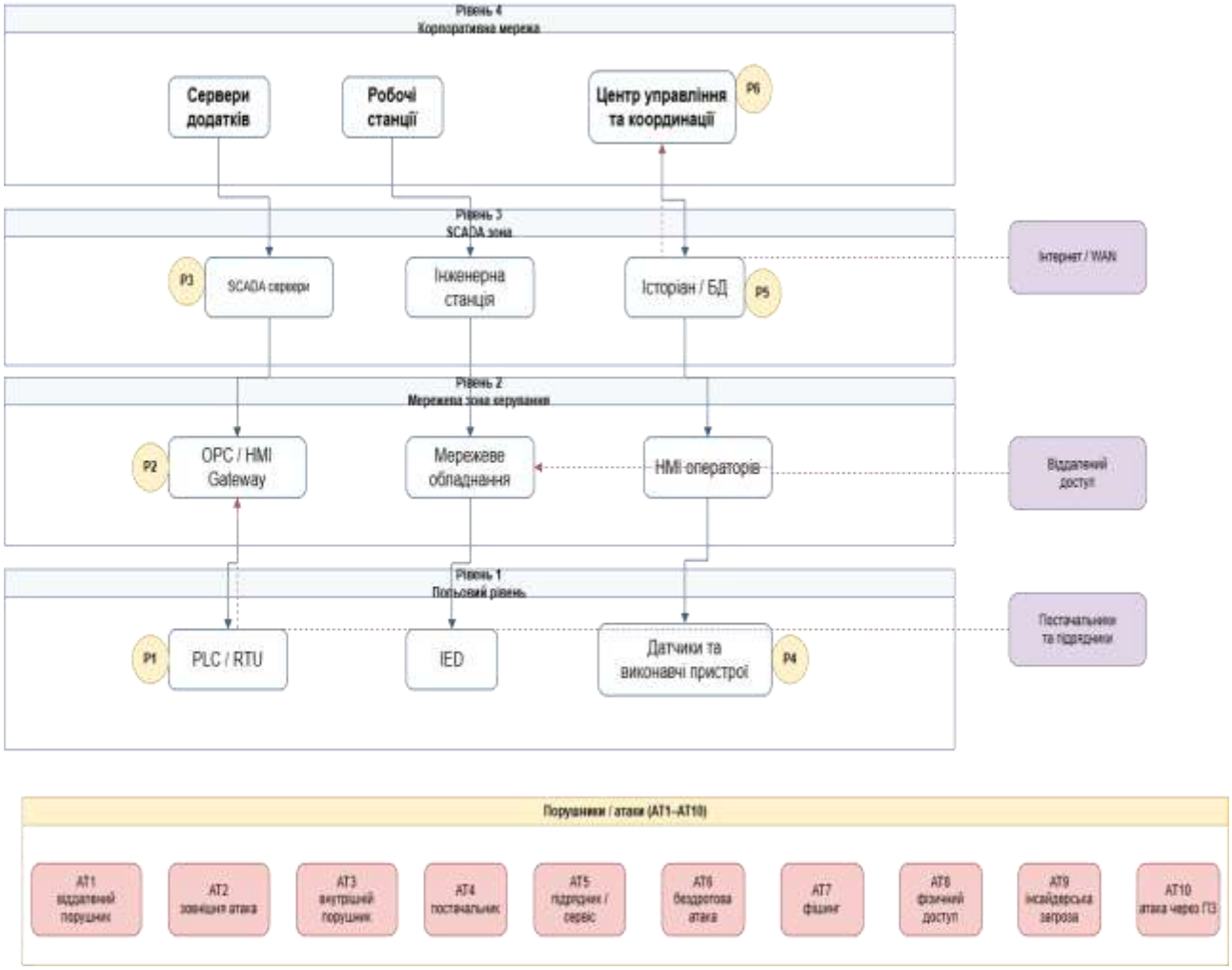


Рисунок 3.11. Узагальнена структура системи цивільного захисту та пов’язаних загроз

Наведена схема демонструє багаторівневу структуру системи (SCADA, мережеві сегменти, сервери, робочі станції), а також можливі точки впровадження програмних закладок (P1-P6) і дії порушників (AT1-AT10). У таких умовах програмні закладки можуть бути інтегровані на різних рівнях – від прикладного програмного забезпечення до мережевих компонентів.

Це дозволяє реалізувати різні моделі взаємодії закладок із системою, зокрема:

– модель «спостерігач» – для збору інформації;

– модель «перехоплювач» – для втручання у потоки даних;

– модель «спотворення» – для зміни результатів обробки інформації.

Особливо небезпечним для систем цивільного захисту є поєднання цих моделей, коли закладка спочатку виконує прихований моніторинг, далі

перехоплює критичні дані, а на завершальному етапі спотворює результати обробки або управлінські рішення.

Таким чином, наведені приклади демонструють різноманітність способів реалізації програмних закладок і підтверджують, що їх застосування можливе на всіх рівнях інформаційних систем – від прикладного програмного забезпечення до критичних інфраструктурних компонентів, що зумовлює необхідність комплексного підходу до їх виявлення та протидії, особливо у системах цивільного захисту.

3.3.5. Методи виявлення та протидії

Виявлення програмних закладок є складною задачею, що потребує поєднання різних підходів. Основу становлять аналіз поведінки системи, контроль цілісності та моніторинг інформаційних потоків.

Статичний аналіз дозволяє виявляти підозрілі фрагменти коду, тоді як динамічний аналіз – фіксувати аномальну поведінку під час виконання [17; 18; 19]. Додатково застосовуються механізми контролю цілісності, що дозволяють виявляти несанкціоновані зміни у файлах, бібліотеках або конфігураціях.

Важливим напрямом є інтеграція безпеки у процес розробки (DevSecOps), що передбачає використання методів моделювання загроз і інструментів аналізу, таких як OWASP Threat Dragon [20; 5]. Такий підхід дозволяє виявляти потенційні закладки ще на етапах проєктування та розробки, а не лише після розгортання системи.

У системах цивільного захисту особливу увагу слід приділяти забезпеченню достовірності результатів обробки даних і безперервності функціонування. Це включає перевірку результатів на еталонних даних, моніторинг критичних параметрів і використання резервних механізмів обробки інформації. Особливо важливим є контроль алгоритмічної логіки прийняття рішень, оскільки саме приховане спотворення результатів є найбільш небезпечним для критичних систем.

Для узагальнення відповідності методів виявлення програмних закладок їх функціональним моделям доцільно використати таблицю 3.3.

Таблиця 3.3

Відповідність методів виявлення програмних закладок їх моделям

Метод виявлення	**Що виявляє**	**Модель закладки (3.3)**	**Коментар**
Статичний аналіз коду	Приховані умови, підозрілі виклики, невикористаний код	С, П	Ефективний для пошуку закладок у коді до виконання
Динамічний аналіз (runtime)	Аномальна поведінка під час виконання	А, П, С	Дає змогу виявити активність, що не видно статично
Контроль цілісності (хеш, підписи)	Зміни у файлах, бібліотеках, конфігураціях	А, П, С	Не виявляє логічні зміни, лише факт модифікації
Моніторинг процесів (EDR/NGAV)	Підозрілі процеси, автозапуски, ін’єкції	А, П	Особливо ефективний для бекдорів і троянів
Мережевий моніторинг (IDS/IPS)	Аномальні з’єднання, приховані канали	А, П	Дозволяє виявити витік даних або C2-комунікацію
Sandbox-аналіз	Приховану поведінку файлів і програм	А, С, П	Дає змогу побачити поведінку в ізольованому середовищі
Аналіз алгоритмів і логіки	Викривлення обчислень, зміну результатів	С	Ключовий метод для найнебезпечнішої моделі
Тестування граничних випадків	Активацію прихованих умов (тригерів)	С	Дозволяє “спровокувати” закладку
Аудит доступу і привілеїв	Ескалацію прав, обхід контролю доступу	П	Виявляє втручання у механізми авторизації
Моніторинг автозавантаження	Закріплення закладки в системі	А	Важливий для виявлення стійких бекдорів

Примітка: застосовано позначки:

А – модель «Спостерігач»;

П – «Перехоплювач»;

С – «Спотворення».

Наведена таблиця демонструє, що жоден окремий метод не забезпечує повного виявлення всіх типів закладок, що обґрунтовує необхідність комплексного підходу.

Ефективна протидія програмним закладкам базується на комплексному підході, що поєднує технічні, організаційні та правові заходи. До технічних заходів належать контроль цілісності, сегментація мережі, використання ізольованих середовищ виконання, резервування критичних компонентів та постійний моніторинг журналів подій [31; 32]. Організаційні заходи включають контроль доступу, аудит змін, регламентоване управління оновленнями та навчання персоналу.

Отже, ефективне виявлення та протидія програмним закладкам можливі лише за умови узгодженого застосування зазначених підходів, що дозволяє підвищити рівень захищеності систем і мінімізувати наслідки потенційних атак.

ПИТАННЯ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ ЗА ТЕМОЮ 3

1. Що таке вразливість програмного забезпечення та чим вона відрізняється від загрози і програмної закладки?
2. Які основні типи вразливостей програмного забезпечення створюють умови для впровадження програмних закладок?
3. Що таке програмна закладка і чим вона відрізняється від звичайної вразливості програмного забезпечення?
4. Які основні критерії класифікації програмних закладок використовуються?
5. Які три базові моделі взаємодії програмних закладок із системою виділяють?
6. У чому полягає сутність моделі «спостерігач»?
7. Які особливості моделі «перехоплювач»?
8. Чим модель «спотворення» відрізняється від інших моделей і чому вона є найбільш небезпечною?

9. Чому контроль цілісності не завжди дозволяє виявити програмні закладки моделі «спотворення»?

10. Які методи виявлення програмних закладок є найбільш ефективними для систем цивільного захисту та чому?

ПРАКТИЧНІ ЗАВДАННЯ ДО ТЕМИ 3

ЗАВДАННЯ 1. Аналіз ознак несанкціонованого втручання в програмне забезпечення системи цивільного захисту

Постановка завдання:

На основі підрозділів 3.1.2 та 3.3 посібника розгляньте ознаки несанкціонованого втручання у контексті програмних закладок.

Оберіть один із наведених типів програмних систем цивільного захисту:

а) автоматизована система координації аварійно-рятувальних служб;

б) система моніторингу надзвичайних ситуацій;

в) програмний комплекс аналізу ризиків і прогнозування.

Для обраної системи:

1) Виберіть п'ять ознак несанкціонованого втручання (підрозділ 3.1.2) і для кожної опишіть конкретний сценарій прояву у контексті обраної системи.

2) Визначте, до якої моделі програмної закладки (підрозділ 3.3) відноситься кожна ознака: «спостерігач», «перехоплювач» або «спотворення».

3) Складіть зведену таблицю: ознака – сценарій прояву – модель закладки – основний метод виявлення (підрозділ 3.3.5).

4) Визначте дві найбільш небезпечні ознаки для обраної системи та обґрунтуйте вибір з урахуванням критичності функцій ЦЗ.

Приклад розв'язку:

Обрана система: автоматизована система координації аварійно-рятувальних служб.

Сценарії прояву ознак:

1. Аномальна мережева активність – система раз на 10 хвилин надсилає зашифровані пакети на невідому IP-адресу через нестандартний порт 53, маскуючи трафік під штатний DNS-запит.

2. Зміна поведінки за специфічних умов – при введенні дати «01.01» модуль розподілу завдань ігнорує виклики з пріоритетом «Надзвичайна ситуація», не реєструючи жодної помилки.

3. Модифікація системних бібліотек – у директорії \System32 виявлено підміну бібліотеки ws2_32.dll власною версією зі зміненими функціями send() і recv().

4. Підозрілі записи автозапуску – у реєстрі HKCU\Software\Microsoft\Windows\CurrentVersion\Run виявлено запис svchost_upd.exe, якого немає у переліку офіційних компонентів системи.

5. Некоректні результати обчислень – модуль розрахунку часу прибуття бригади систематично занижує значення на 15%, що не відповідає реальним показникам реагування.

Таблиця 3.4

Зведена таблиця

Ознака	**Сценарій прояву**	**Модель закладки**	**Метод виявлення**
Аномальна мережева активність	Передача пакетів на C2-сервер через DNS – тунелювання на порт 53	«Спостерігач»	Мережевий моніторинг, IDS/IPS; аналіз аномалій трафіку
Зміна поведінки за специфічних умов	Ігнорування НС-викликів при конкретній даті	«Спотворення»	Тестування граничних значень; порівняння з еталонною поведінкою
Модифікація системних бібліотек	Підміна ws2_32.dll для перехоплення мережевих функцій	«Перехоплювач»	Контроль цілісності файлів (SHA-256); перевірка цифрових підписів
Підозрілі записи автозапуску	svchost_upd.exe у гілці Run	переважно «Спостерігач»	Моніторинг автозапуску (реєстр, служби); EDR-системи

Ознака	Сценарій прояву	Модель закладки	Метод виявлення
Некоректні результати обчислень	Систематичне заниження часу прибуття на 15%	«Спотворення»	Порівняння результатів з еталонними даними; аналіз алгоритмів

Найбільш небезпечні ознаки:

1. Зміна поведінки за специфічних умов (модель «Спотворення») – найнебезпечніша, оскільки систематичне спотворення пріоритизації викликів безпосередньо впливає на ефективність реагування, а виявлення ускладнено тим, що програма зовні функціонує штатно. Тригер активації прихований і не виявляється стандартним тестуванням.

2. Модифікація системних бібліотек (модель «Перехоплювач») – критична через можливість прихованого перехоплення всіх мережевих операцій: передачі команд між диспетчером і бригадами, координаційних даних і паролів автентифікації, що може призвести до повної компрометації системи та втрати контролю над процесом реагування.

ЗАВДАННЯ 2. Порівняльний аналіз моделей програмних закладок

Постановка завдання:

На основі підрозділу 3.3 посібника виконайте аналіз однієї з моделей взаємодії програмних закладок із системою.

Оберіть модель для детального аналізу:

а) модель «спостерігач»;

б) модель «перехоплювач»;

в) модель «спотворення».

Для обраної моделі:

1) Опишіть формальний механізм взаємодії закладки з системою за суб'єктно-орієнтованою моделлю: визначте суб'єкти, об'єкти, тип операцій доступу та характер інформаційних потоків.

2) Наведіть конкретний реальний приклад відомої закладки, що функціонує за обраною моделлю (підрозділ “Приклади програмних закладок”), та опишіть механізм її дії і наслідки для атакованої системи.

3) Складіть порівняльну таблицю обраної моделі з однією іншою моделлю за характеристиками: тип впливу на систему, можливість виявлення стандартними засобами, наслідки для ЦЗ-системи, складність реалізації, метод виявлення.

4) Запропонуйте конкретні заходи виявлення та протидії для обраної моделі відповідно до підрозділу 3.3.5.

Приклад розв'язку:

Обрана модель: «Спотворення».

Формальний механізм:

Система описується як множина суб'єктів {S} і об'єктів {O}. Закладка «Спотворення» впроваджується як додатковий суб'єкт S_зак, що ініціює операції модифікації (W) над об'єктами системи: алгоритмами (O_alg), даними (O_data) або конфігураційними параметрами (O_conf).

На відміну від моделі «спостерігач», яка лише зчитує (R) об'єкти без їх зміни, S_зак модифікує вміст або логіку так, що результати операцій штатного суб'єкта S_prog є систематично хибними.

Активація S_зак відбувається за умовним тригером T(x), де x – значення параметра (дата, лічильник запусків, вхідний рядок), і прихована від зовнішнього спостереження.

Реальний приклад – Dual_EC_DRBG

Dual_EC_DRBG – відомий приклад програмної закладки типу «спотворення», вбудованої у криптографічний механізм генерації псевдовипадкових чисел.

Механізм дії: алгоритм використовувався у стандартах NIST як генератор випадкових чисел для криптографічних систем. Було встановлено, що за певних параметрів генератора існувала можливість передбачення згенерованих значень і, відповідно, відновлення ключової інформації.

Фактично система продовжувала працювати штатно, а генерація випадкових чисел формально відповідала очікуваному інтерфейсу, однак результати були криптографічно скомпрометованими.

Результат: користувачі довіряли механізму захисту, який фактично містив прихований канал компрометації. Це типовий приклад моделі «спотворення», де шкідливий функціонал реалізується через зміну логіки критичного алгоритму без зовнішніх ознак порушення.

Таблиця 3.5

Зведена таблиця

Характеристика	**«Спотворення»**	**«Спостерігач»**
Тип впливу на систему	Активний: змінює дані, алгоритми або логіку роботи	Пасивний: лише збирає і передає дані
Можливість виявлення стандартними засобами	Дуже низька: система функціонує зовні штатно	Низька: маскується під легітимний процес; виявляється через трафік
Наслідки для ЦЗ-системи	Хибні управлінські рішення, підрив достовірності результатів	Витік конфіденційних даних, паролів, оперативної інформації
Складність реалізації	Висока: потрібне точне знання внутрішньої логіки програми	Середня: реалізується через стандартний клієнт-серверний механізм
Метод виявлення	Порівняння результатів з еталонними даними; аналіз алгоритмів	Мережевий моніторинг, аналіз автозапуску, EDR

Заходи виявлення та протидії (підрозділ 3.3.5):

– Контроль цілісності критичних системних модулів: обчислення SHA-256 хешів усіх критичних бінарних файлів при кожному завантаженні – виявляє підміну драйверів на кшталт Турбо Криптон.

– Верифікація криптографічних операцій: незалежне тестування криптомодулів на еталонних векторах (відомих парах «відкритий текст» – «шифротекст») – виявляє підміну алгоритму без аналізу вихідного коду.

– Регулярний аудит конфігурацій і параметрів середовища: порівняння поточних конфігурацій із затвердженими базовими значеннями – виявляє маніпуляції з таблицями рішень або конфігураційними файлами.

– Тестування граничних та тригерних умов: цілеспрямовані перевірки поведінки системи за рідкісних значень параметрів (дати, лічильники, спеціальні рядки) – виявляє тригери активації, непомітні при штатній роботі.

ЗАВДАННЯ 3. Аналіз вразливостей і методів впровадження програмних закладок

Постановка завдання:

На основі підрозділу 3.1.1 посібника (типи вразливостей) та матеріалу про методи впровадження і розповсюдження закладок дослідіть взаємозв'язок між вразливостями ПЗ і шляхами впровадження прихованого шкідливого функціоналу.

Оберіть один із наведених типів програмних систем цивільного захисту:

а) програмний комплекс аналізу ризиків і прогнозування;

б) система моніторингу надзвичайних ситуацій;

в) автоматизована система координації аварійно-рятувальних служб.

Для обраної системи:

1) Визначте п'ять типів вразливостей ПЗ із підрозділу 3.1.1, що є найбільш ймовірними для обраного класу систем, та стисло обґрунтуйте, чому ці вразливості характерні саме для нього.

2) Для кожної вразливості вкажіть конкретний метод впровадження закладки, що її використовує, та модель закладки («спостерігач», «перехоплювач» або «спотворення»), яка найчастіше реалізується через цей вектор.

3) Складіть зведену таблицю: вразливість – метод впровадження – модель закладки – потенційний наслідок для системи.

4) Для кожної вразливості запропонуйте конкретний захід протидії відповідно до підрозділу 3.3.5 та обґрунтуйте його ефективність.

Таблиця 3.6

Зведена таблиця

Вразливість	Метод впровадження	Модель закладки	Потенційний наслідок
Переповнення буфера	Exploit → shellcode → встановлення серверної частини закладки	«Спостерігач»	Дистанційне керування сервером аналізу; витік прогнозних даних
Недостатня перевірка вхідних даних	SQL-ін'єкція → модифікація записів у БД	«Спотворення»	Підміна вхідних даних ризик-аналізу; хибні прогнози НС
Логічні помилки в алгоритмах|	Пряме впровадження прихованої логіки у код під час розробки або оновлення	«Спотворення»	Систематично некоректні оцінки ризику; хибні управлінські рішення
Небезпечні виклики сторонніх API	Підміна бібліотеки геоаналізу модифікованою версією	«Перехоплювач»	Перехоплення координатних даних і результатів моделювання
Слабкі механізми автентифікації	Brute-force / credential stuffing → встановлення бекдору через адмінпанель	«Спостерігач»	Несанкціонований доступ до модулів адміністрування; впровадження нових закладок

Приклад розв'язку:

Обрана система: програмний комплекс аналізу ризиків і прогнозування.

Обґрунтування вибору вразливостей: система обробляє великі масиви даних від зовнішніх джерел (датчики, звіти підрозділів), має складну архітектуру з численними модулями та сторонніми бібліотеками аналізу. Це робить її особливо вразливою до ін'єкцій, логічних помилок і компрометації сторонніх компонентів.

1. Переповнення буфера – характерне для C/C++-модулів парсингу вхідних даних із зовнішніх джерел (датчики, файли звітів без суворого формату).

2. Недостатня перевірка вхідних даних – вхідні звіти підрозділів не мають строгих форматних обмежень; SQL-ін'єкція можлива через поля вільного введення.

3. Логічні помилки в алгоритмах – модулі прогнозування мають складну умовну логіку; навмисне спотворення умов переходу важко виявити без еталонних даних.

4. Небезпечні виклики сторонніх API – система використовує бібліотеки геоаналізу без перевірки їх цілісності; підміна бібліотеки є реальним вектором атаки.

5. Слабкі механізми автентифікації – адміністративний інтерфейс доступний через веббраузер із парольним захистом без MFA.

Заходи протидії:

1. Переповнення буфера: застосування компіляторних засобів захисту (ASLR, DEP/NX, Stack Canary) і регулярний SAST-аналіз C/C++-коду – унеможливлює більшість стандартних shellcode-атак на парсери вхідних даних.

2. Недостатня перевірка вхідних даних: валідація всіх зовнішніх даних за суворою схемою (whitelist), використання параметризованих запитів до БД – усуває SQL-ін'єкцію як вектор впровадження закладки.

3. Логічні помилки в алгоритмах: тестування модулів прогнозування на еталонних наборах даних при кожному релізі та code review критичних модулів – виявляє навмисне спотворення умов переходу до публікації.

4. Небезпечні виклики сторонніх API: перевірка SHA-256 хешів усіх сторонніх бібліотек перед завантаженням (Software Composition Analysis, SCA) – виявляє підміну бібліотеки до моменту її виконання.

5. Слабкі механізми автентифікації: обов'язкова двофакторна автентифікація для адміністративного інтерфейсу, обмеження кількості спроб входу (rate limiting) і журналювання доступу – суттєво ускладнює отримання початкового доступу для встановлення закладки.

4. ПЕРВИННИЙ АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Первинний аналіз програмного забезпечення є початковим етапом дослідження програмної системи, який дозволяє сформувати загальне уявлення про її структуру, поведінку, механізми захисту та потенційні вразливості без виконання глибокого статичного або динамічного аналізу. На цьому етапі використовуються контрольовані експериментальні прийоми, що дають змогу дослідити реакцію програми на зміну вхідних даних, параметрів середовища або окремих компонентів виконуваного файлу.

Метою такого аналізу є виявлення закономірностей, залежностей і прихованих структур у програмному забезпеченні, що дозволяє:

- визначити формат і структуру виконуваних або бінарних файлів;
- розпізнати сигнатури шифрування та принципи генерації ключів;
- встановити наявність випадкових параметрів (nonce, IV, salt);
- виявити можливі слабкі місця системи захисту;
- оцінити наявність прихованих механізмів впливу або ознак програмних закладок.

Такий підхід особливо важливий у випадках, коли відсутній доступ до вихідного коду (proprietary software, malware, закриті системи керування), а дослідник може працювати лише з виконуваними файлами, результатами роботи програми та її зовнішніми проявами.

Контрольовані експериментальні прийоми є базовим інструментом бінарного аналізу у зворотному інжинірингу.

Вони дають змогу:

- досліджувати поведінку коду без вихідних текстів;
- виявляти механізми ініціалізації, генерації ключів, перевірок ліцензій;
- розуміти структуру виконуваних модулів і можливості впровадження захисних механізмів;
- ідентифікувати потенційні вразливості або приховані залежності.

На етапі первинного аналізу можуть застосовуватися й експериментальні

прийоми, але на цьому етапі вони не є самостійним методом дослідження, а виступають інструментом початкового вивчення програмної системи перед переходом до поглибленого статичного аналізу, динамічного аналізу та дослідження механізмів захисту. Саме така послідовність дозволяє забезпечити системний підхід до оцінювання безпеки програмного забезпечення.

Цей підхід зручно використовувати у випадках, коли внутрішні алгоритми недоступні, але можна спостерігати результати їхньої роботи. Дослідник змінює параметри вхідних даних (наприклад, тексти, ключі, блоки файлів) і аналізує результати (шифротексти, сигнатури, зміни в поведінці).

Ефективність такого підходу слабо залежить від програмної реалізації системи захисту (на неї, зокрема, не впливає застосування засобів захисту програмного забезпечення від вивчення) і визначається в першу чергу складністю аналізованих алгоритмів.

4.1. СУТНІСТЬ ПЕРВИННОГО АНАЛІЗУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Первинний аналіз програмного забезпечення полягає у проведенні серії контрольованих впливів на програмну систему з метою спостереження її поведінки та виявлення прихованих властивостей без необхідності негайного переходу до повного реверс-інжинірингу або глибокого статичного аналізу коду.

У загальному вигляді метод можна подати як відображення:

$$S : X \rightarrow Y,$$

де X – множина вхідних даних, Y – множина вихідних реакцій системи.

Змінюючи X, дослідник аналізує зміну Y, що дозволяє встановити:

- структуру обробки даних;
- залежності між параметрами;
- особливості реалізації алгоритмів;
- можливі приховані умови активації;
- ознаки несанкціонованого втручання або програмних закладок.

Такий підхід застосовується у випадках відсутності доступу до вихідного коду, аналізу шкідливого ПЗ, дослідження пропрієтарних систем, а також під час первинної перевірки підозрілих програмних компонентів [18; 19].

Первинний аналіз із використанням експериментальних прийомів орієнтований на поступове виявлення внутрішніх механізмів роботи програми через контрольовані зміни параметрів і спостереження реакції системи. Це дозволяє сформувати початкове уявлення про логіку функціонування програмного забезпечення без прямого втручання у його код.

У криптографічних експериментах може використовуватись додатковий випадковий параметр (nonce, IV або salt), який у цьому посібнику умовно називається маркантом – значенням, що змінюється між сеансами та впливає на результат шифрування.

У підрозділах цивільного захисту первинний аналіз програмного забезпечення застосовується для:

- аналізу підозрілих програм у ізольованих середовищах;
- перевірки цілісності програмних компонентів;
- дослідження поведінки ПЗ в умовах відмов або атак;
- тестування критичних систем у режимах деградації;
- попереднього виявлення ознак прихованого втручання або програмних закладок [12; 13].

Таким чином, первинний аналіз із використанням експериментальних прийомів є необхідним початковим етапом оцінювання безпеки програмного забезпечення, який дозволяє сформувати обґрунтовану основу для подальшого поглибленого статичного та динамічного дослідження.

4.2. ІНСТРУМЕНТАЛЬНІ ЗАСОБИ ТА ТЕХНІКИ ПЕРВИННОГО АНАЛІЗУ

Для виконання первинного аналізу програмного забезпечення використовуються спеціалізовані інструментальні засоби, що дозволяють досліджувати структуру виконуваних файлів, службові заголовки, метадані,

часові мітки та інші ознаки можливого несанкціонованого втручання без необхідності негайного переходу до глибокого реверс-інжинірингу [19; 26].

До основних засобів первинного аналізу належать Hex-редактори, PE-viewers та спеціалізовані утиліти для перегляду структури виконуваних файлів, зокрема CFF Explorer.

Hex-редактори (HxD, Hex Editor Neo, WinHex) забезпечують доступ до бінарного подання даних і дозволяють переглядати та змінювати вміст файлів побайтно. Вони використовуються для аналізу сигнатур форматів, службових заголовків, пошуку жорстко закодованих ключів, перевірки часових міток, виявлення підміни бібліотек або появи нетипових фрагментів коду. Саме можливість контрольованої побайтової модифікації робить Hex-редактори одним із базових інструментів первинного аналізу.

PE-viewers (Portable Executable Viewers) призначені для дослідження структури виконуваних файлів формату PE у середовищі Windows. Вони дозволяють переглядати DOS Header, PE Header, таблиці секцій, точки входу, імпортовані та експортовані функції, адреси завантаження та інші службові параметри. Це дає змогу швидко виявити аномалії структури файлу, ознаки пакування, обфускації або несанкціонованої модифікації.

CFF Explorer є спеціалізованим інструментом для аналізу PE-файлів, що поєднує функції PE-viewer та редактора заголовків. Він дозволяє детально досліджувати структуру виконуваного файлу, змінювати окремі поля заголовків, перевіряти цифрові підписи, імпортовані бібліотеки, ресурси та секції програми. У задачах цивільного захисту цей інструмент особливо корисний для перевірки цілісності критичного програмного забезпечення після оновлень або під час аналізу інцидентів безпеки.

Використання зазначених інструментів дозволяє сформувати початкове уявлення про програмний об’єкт, виявити непрямі ознаки компрометації та обґрунтовано перейти до поглибленого статичного або динамічного аналізу.

4.2.1. Можливості й приклади Hex-редакторів

Hex-редактор – це програмний інструмент низькорівневого аналізу, що дозволяє працювати з даними у вигляді послідовності байтів, виконувати модифікації без урахування логічної структури файлу та досліджувати внутрішню організацію програмних об’єктів [19].

У процесі первинного аналізу програмного забезпечення Hex-редактори застосовуються для виявлення сигнатур форматів, пошуку жорстко закодованих ключів, аналізу областей пам’яті, перевірки службових заголовків, часових міток та ознак прихованої модифікації виконуваних файлів. Вони можуть бути як окремими самостійними додатками, так і компонентами більш складних засобів аналізу, зокрема дизасемблерів, налагоджувачів або інтегрованих середовищ дослідження.

У задачах цивільного захисту hex-аналіз використовується для перевірки програм, що застосовуються у критичних системах, під час аналізу інцидентів безпеки, а також для контролю змін у програмному забезпеченні після оновлень або відновлення системи після збоїв.

Для подання значення байтів використовується шістнадцяткова система числення. Дані, які відображає і дозволяє редагувати Hex-редактор, можуть бути:

− образом диска;

− вмістом окремого файлу;

− вмістом оперативної пам’яті;

− вмістом віртуального адресного простору процесу;

− іншими бінарними об’єктами.

Інтерфейс HEX-редактора, як правило, розділений на кілька основних полів.

У центральній частині редактор відображає дані у вигляді матриці, де кожна клітинка відповідає одному байту, записаному у шістнадцятковій системі числення у вигляді двозначного числа. Кількість стовпців матриці зазвичай становить 16 або 8, інколи 4. Число рядків залежить від обсягу даних, які необхідно відобразити або відредагувати.

Ліворуч від матриці розташована адресна лінійка, де кожному рядку відповідає зміщення першого байта цього рядка. Крок адрес визначається кількістю колонок.

Зверху над матрицею розміщується лінійка зміщень, що показує позицію байта всередині рядка. Сума адреси рядка та зміщення колонки визначає точну адресу конкретного байта.

Праворуч від матриці часто відображається альтернативне подання тих самих даних – текстове, двійкове або символьне. Оскільки не всі байти містять текстову інформацію, у цьому полі можуть з’являтися недруковані символи або позначення у вигляді крапок ".", що зазвичай свідчить про нетекстові дані.

HEX-редактор дозволяє:

– відображати дані у вигляді шістнадцяткового коду;
– побайтно редагувати вміст будь-якого файлу незалежно від типу;
– отримувати текстове відображення вихідного або зміненого файлу;
– здійснювати пошук сигнатур, рядків або службових значень;
– виконувати операції запису, заміни та перетворення даних.

Саме можливість контрольованої побайтової зміни файлу дозволяє використовувати Hex-редактор як базовий інструмент експериментальних прийомів первинного аналізу: дослідник змінює окремі байти заголовка, службових полів або метаданих та спостерігає реакцію операційної системи чи прикладного програмного забезпечення. Це дозволяє виявити приховані залежності, перевірити стійкість механізмів захисту та зафіксувати ознаки несанкціонованого втручання.

Приклади Hex-редакторів

1**. HxD** (офіційний сайт: https://mh-nexus.de/en/hxd)

HxD Hex Editor є одним із найбільш поширених безкоштовних HEX-редакторів, що широко використовується у навчальних і практичних задачах первинного аналізу програмного забезпечення. Програма дозволяє працювати як із файлами на диску, так і з оперативною пам’яттю, підтримує великі файли, побайтове порівняння, пошук і заміну, обчислення контрольних сум і хешів, а

також експорт даних у різні формати.

Особливо важливою є можливість перевірки цілісності файлів, аналізу змін у бібліотеках, дослідження заголовків виконуваних файлів та контролю появи нетипових фрагментів коду [43, 41]. Саме тому HxD доцільно використовувати як базовий інструмент у практичних роботах з первинного аналізу ПЗ. Інтерфейс редактору HxD представлено на рисунку 4.1.

Основні можливості програми HxD:

– редактор дисків;
– редактор пам’яті;
– побайтове порівняння файлів;
– обчислення та перевірка контрольних сум і хешів;
– пошук і заміна різних форматів даних;
– робота з великими файлами;
– безпечне видалення файлів;
– експорт даних у різні формати;
– аналіз статистики файлу.

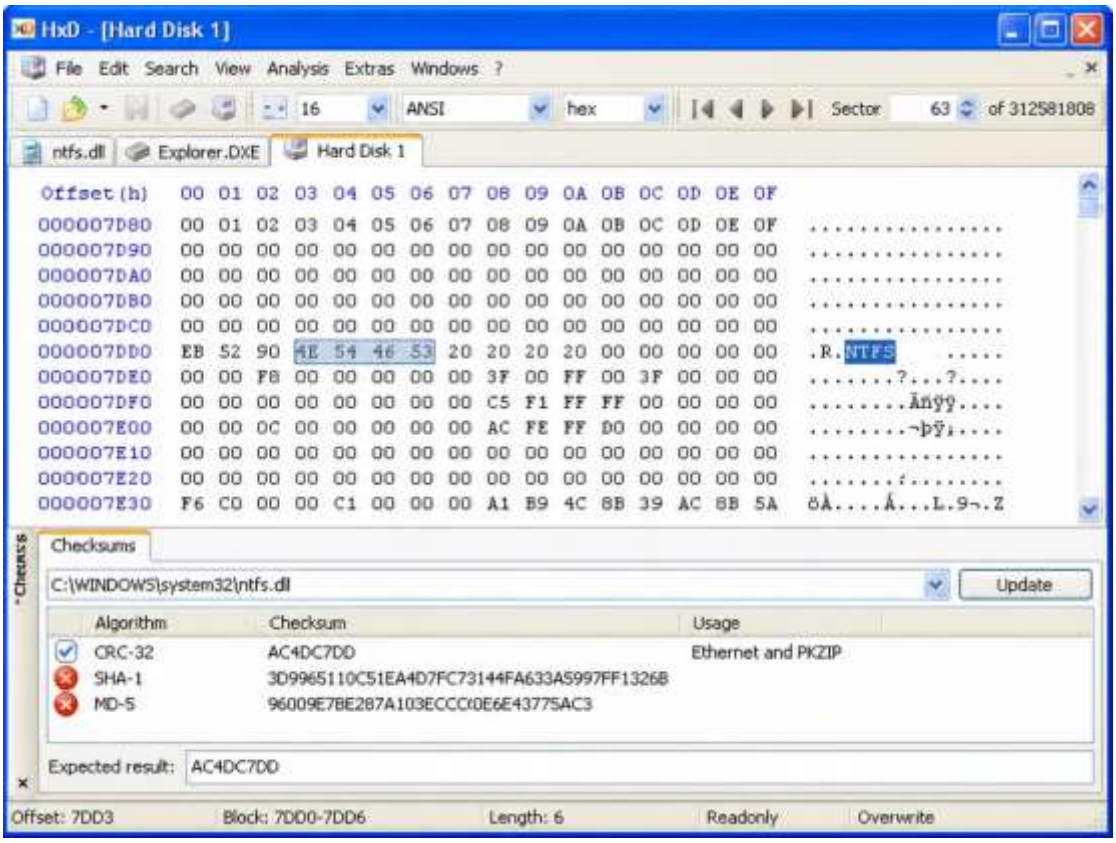


Рисунок 4.1. Інтерфейс редактора HxD

HxD є зручним інструментом для навчального та практичного первинного аналізу програмного забезпечення, оскільки поєднує простий інтерфейс із широкими можливостями низькорівневого дослідження файлів і пам’яті. На його основі часто виконують експериментальні перевірки структури заголовків, часових міток, службових полів та реакції операційної системи на контрольовану зміну окремих байтів. Це дозволяє не лише формувати практичне розуміння внутрішньої організації виконуваних файлів, а й виявляти ознаки модифікації або прихованого втручання.

2. **Hex Workshop** (офіційний сайт: https://www.hexworkshop.com/)

Hex Workshop – професійний інструмент для редагування файлів у шістнадцятковому форматі, що поєднує функції HEX-редактора та інтерпретатора даних. Програма дозволяє змінювати, копіювати, вставляти та видаляти бінарні дані, виконувати пошук і заміну, порівнювати файли, генерувати контрольні суми та працювати з різними форматами представлення даних. Важливою перевагою є зручний аналіз структури виконуваних файлів і підтримка контрольних перевірок цілісності (рис. 4.2).

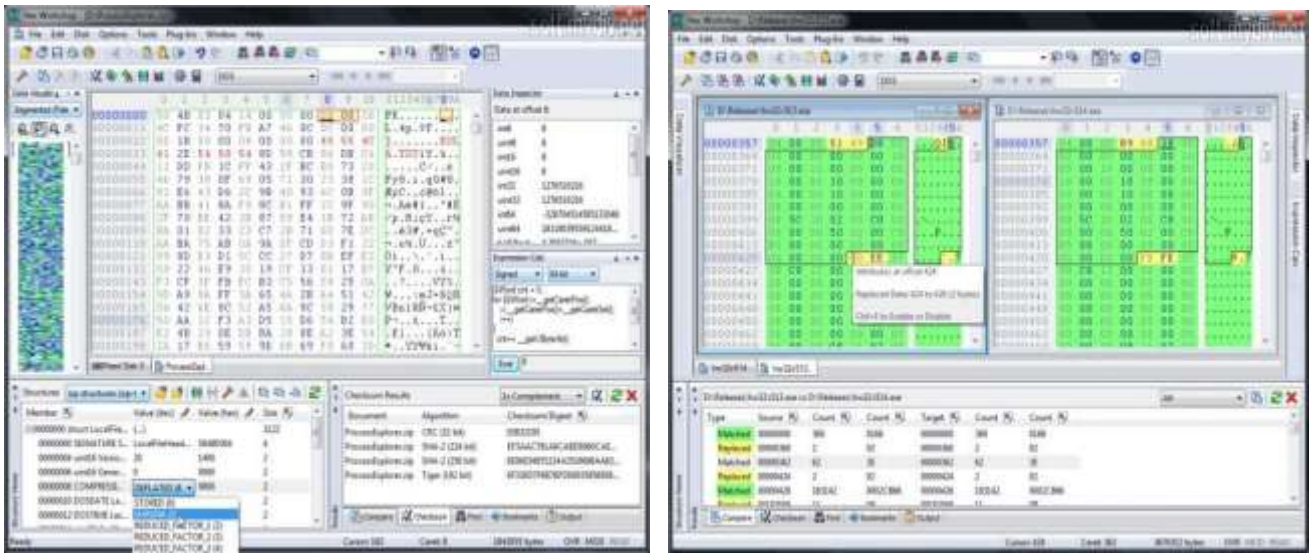


Рисунок 4.2. Інтерфейс редактора Hex Workshop

3. **Free Hex Editor Neo** (розробник HHD Software Ltd.: https://freehexeditorneo.com/)

Free Hex Editor Neo – редактор двійкових, десяткових і шістнадцяткових файлів для Windows, орієнтований на роботу з великими обсягами даних.

Програма підтримує роботу з ASCII, шістнадцятковими (HEX), десятковими (DEC), двійковими (BIN) даними, а також із 32- та 64-розрядними значеннями з плаваючою комою. Основною перевагою редактора є висока швидкодія при роботі з великими файлами та багатовіконний інтерфейс, що спрощує аналіз декількох об’єктів одночасно. Інтерфейс редактору представлено на рисунку 4.3.

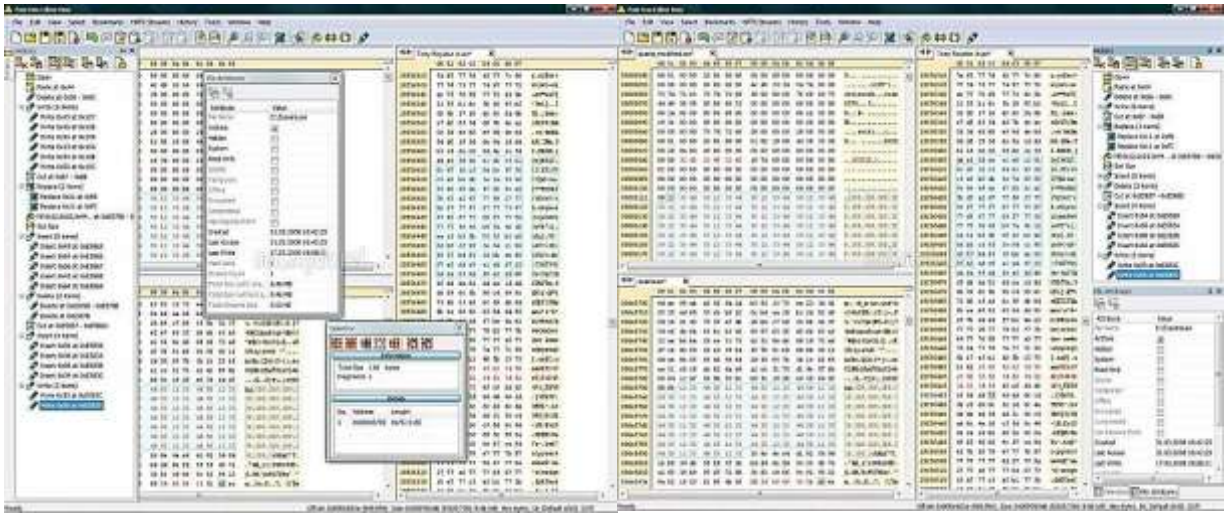


Рисунок 4.3. Інтерфейс редактора Free Hex Editor Neo

4. **HexCmp** (сторінка інструмента Fairdell Software: https://www.fairdell.com/hexcmp/)

HexCmp – HEX-редактор, основною особливістю якого є можливість швидкого порівняння вмісту бінарних файлів. Програма підтримує роботу з файлами великого розміру, синхронізоване прокручування, підсвічування відмінностей та порівняння окремих фрагментів коду. Це особливо корисно під час аналізу змін після оновлень програмного забезпечення або перевірки можливих несанкціонованих модифікацій (рис. 4.4).

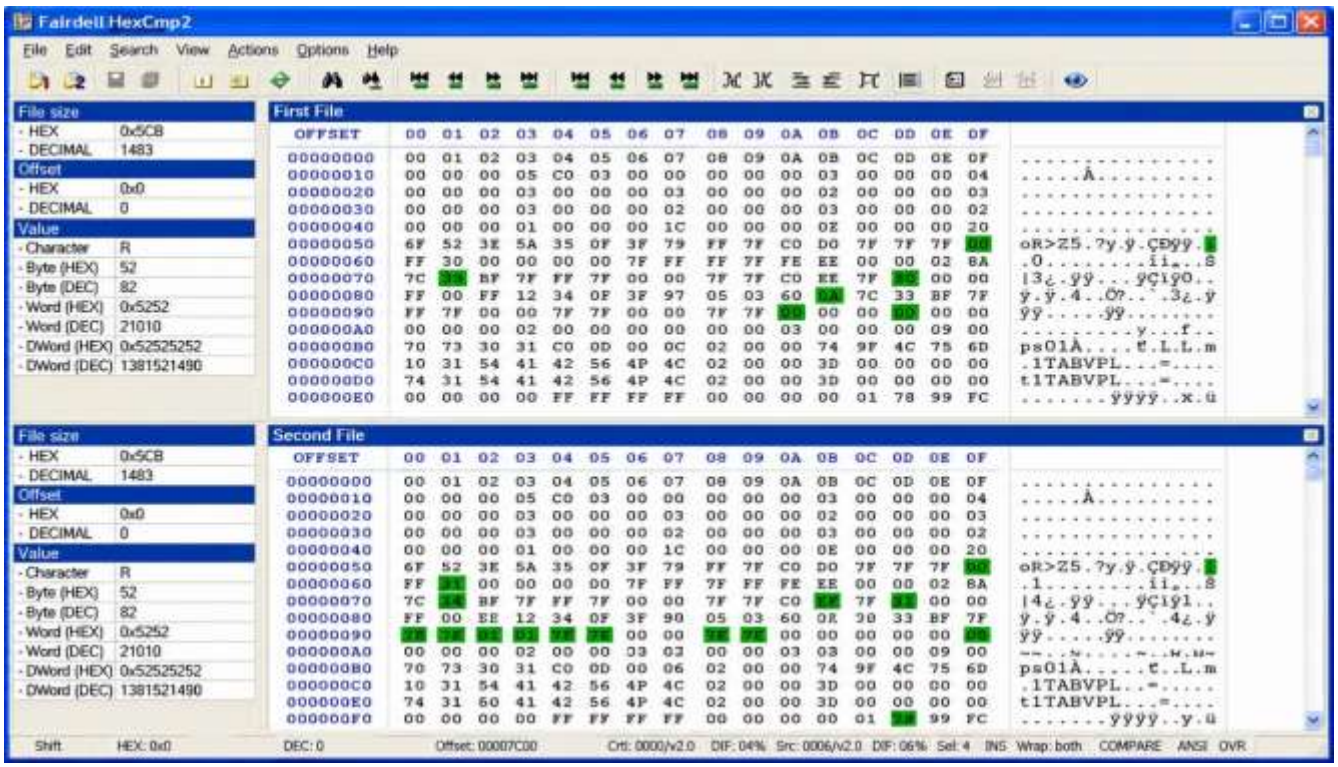


Рисунок 4.4. Інтерфейс редактора HexCmp

5. **XVI32** (офіційний сайт портативного варіанта: https://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm)

XVI32 – компактний портативний HEX-редактор, що не потребує встановлення та дозволяє працювати з файлами значного розміру. Завдяки простоті інтерфейсу він зручний для швидкого первинного аналізу структури файлів, перегляду сигнатур і базового побайтового редагування.

Інтерфейс редактору представлено на рисунку 4.5.

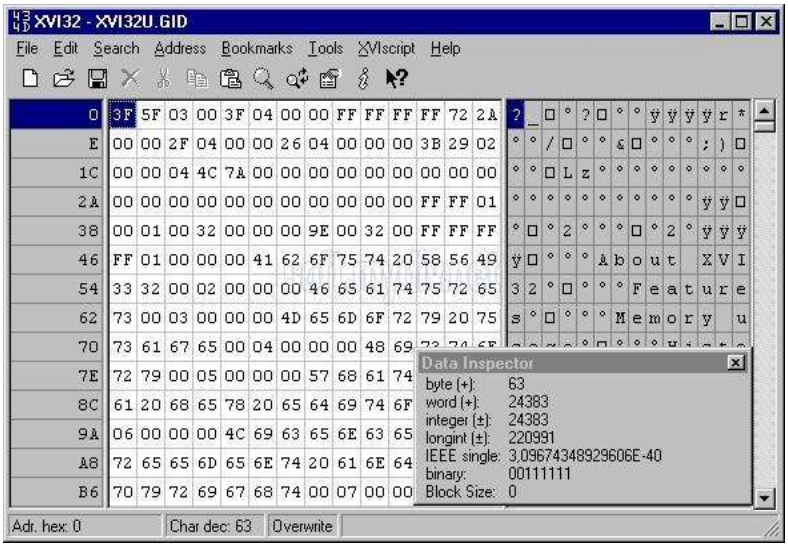


Рисунок 4.5. Інтерфейс редактора XVI32.

6. **Lister (Total Commander)** (офіційна сторінка: https://www.ghisler.com/, формум редактора https://www.ghisler.ch/board/viewtopic.php?t=51698)

Total Commander є багатофункціональним файловим менеджером, а його вбудований переглядач Lister може використовуватись як допоміжний інструмент первинного аналізу файлів. Він дозволяє швидко переглядати структуру даних, текстові та бінарні фрагменти, що зручно під час попереднього аналізу програмних компонентів і перевірки службових файлів. Сторінка форуму редактора Lister наведена на рис. 4.6

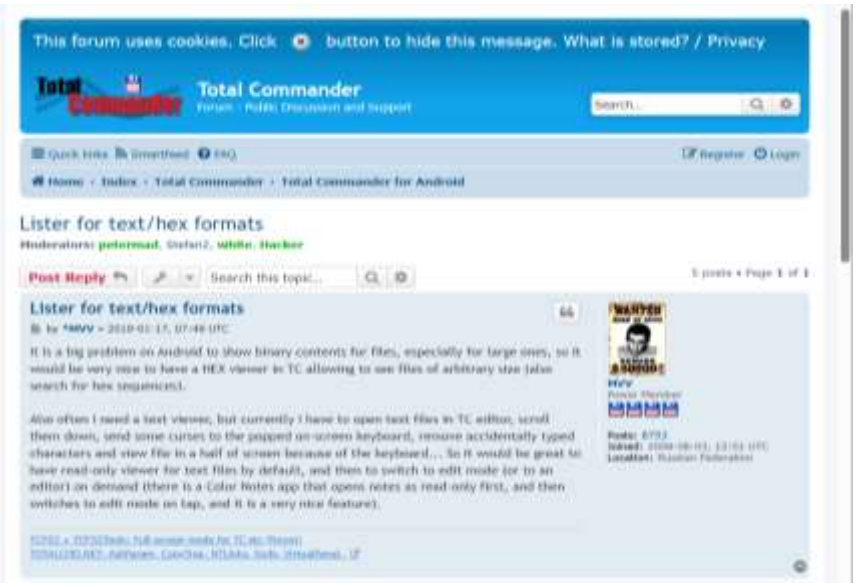


Рисунок 4.6. Сторінка форуму Lister

Отже, розглянуті HEX-редактори є ефективними інструментами первинного аналізу програмного забезпечення. Їх вибір доцільно здійснювати з урахуванням функціональних можливостей, рівня підтримки, сумісності з сучасними операційними системами та відповідності вимогам інформаційної безпеки, особливо у сфері цивільного захисту.

Такі інструменти дозволяють експериментально визначити:

– формат заголовків файлів;

– структуру секцій виконуваного коду;

– наявність марканту або випадкових параметрів у криптосистемах;

– залежність алгоритмів від системного часу.

4.2.2. Особливості й приклади PE-viewers

PE-viewers (Portable Executable Viewers) – це спеціалізовані інструменти первинного аналізу, призначені для дослідження структури виконуваних файлів формату PE (Portable Executable), що використовується в операційних системах Windows для EXE, DLL, SYS та інших виконуваних модулів.

На відміну від Hex-редакторів, які дозволяють працювати з файлом на рівні окремих байтів, PE-viewers орієнтовані на логічне представлення внутрішньої структури виконуваного файлу. Вони автоматично інтерпретують службові заголовки, таблиці секцій, імпортовані та експортовані функції, точки входу та інші елементи PE-структури, що значно спрощує первинний аналіз [42; 26].

PE-viewers використовуються для:

– перегляду DOS Header та PE Header;
– аналізу Optional Header;
– дослідження Section Headers;
– перевірки Entry Point;
– аналізу Import Table та Export Table;
– виявлення ознак пакування, обфускації або модифікації;
– перевірки часових міток і службових атрибутів файлу;
– оцінювання цифрових підписів та коректності завантаження.

Особливу увагу під час первинного аналізу приділяють таким параметрам, як адреса точки входу (Entry Point), кількість і структура секцій, права доступу до секцій (читання, запис, виконання), а також відповідність часових міток реальній історії створення або оновлення файлу. Нетипові значення цих параметрів можуть свідчити про використання пакувальників, приховану модифікацію або впровадження програмних закладок [42; 19].

У задачах цивільного захисту PE-viewers застосовуються для перевірки цілісності критичного програмного забезпечення після оновлень, аналізу підозрілих виконуваних файлів, дослідження інцидентів безпеки та контролю відповідності програм офіційним дистрибутивам. Особливо важливим є виявлення прихованих змін у системних модулях, які можуть впливати на

достовірність результатів обробки інформації або порушувати безперервність функціонування систем [12; 13].

До найбільш відомих PE-viewers належать PEview, PE-bear, LordPE, Dependency Walker, а також CFF Explorer, який поєднує функції перегляду та редагування PE-структури й розглядається окремо у наступному підрозділі.

PE-viewers дозволяють швидко сформувати загальне уявлення про виконуваний файл без переходу до повного дизасемблювання чи декомпіляції. Саме тому вони є важливим елементом первинного аналізу та забезпечують обґрунтований перехід до поглибленого статичного аналізу.

Аналіз PE-заголовків є одним із ключових етапів початкового дослідження програмного забезпечення, оскільки дозволяє виявити непрямі ознаки компрометації ще до запуску програми. Це особливо важливо для систем цивільного захисту, де своєчасне виявлення модифікованого або підозрілого програмного компонента безпосередньо впливає на надійність функціонування всієї інформаційної системи.

Приклади PE-viewers

1. **PEview** (утиліта від W.J. Radburn / офіційне поширення через дослідницькі ресурси PE-аналізу, https://download.cnet.com/pe-viewer/3000-2352_4-10966763.html#google_vignette)

PEview є одним із класичних інструментів для швидкого перегляду структури PE-файлів без редагування їх вмісту. Програма дозволяє досліджувати DOS Header, NT Headers, Optional Header, таблиці секцій, Import Table, Export Table, ресурси та службові структури виконуваного файлу. Інструмент особливо корисний для швидкого визначення точки входу, перевірки секцій і виявлення ознак пакування. Сторінка завантаження PEview наведена на рис.4.7.



Рисунок 4.7. Сторінка завантаження PEview

2. PE-bear (офіційний сайт: https://hshrzd.wordpress.com/pe-bear/ GitHub: https://github.com/hasherezade/pe-bear)

PE-bear – сучасний багатоплатформний інструмент для первинного аналізу PE-файлів, орієнтований на швидкий «перший погляд» (first view) для аналітиків шкідливого ПЗ. Підтримує PE32 та PE64, розпізнає відомі пакувальники за сигнатурами, дозволяє порівнювати два PE-файли, аналізувати секції та виконувати швидке дизасемблювання з довільного RVA або file offset. Інструмент визначається для fast and flexible first view for malware analysts (рис.4.8).

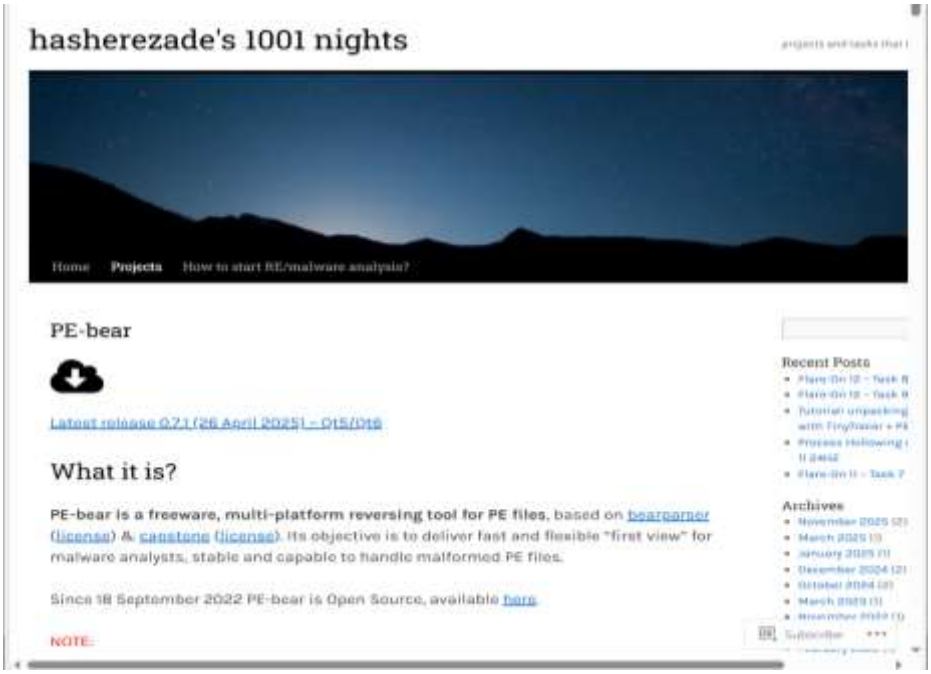


Рисунок 4.8. PE-bear на сторінці https://hshrzd.wordpress.com

3. PE Explorer (офіційний сайт: https://www.pe-explorer.com/)

PE Explorer – професійний інструмент для дослідження EXE, DLL та інших PE-файлів, який поєднує PE-viewer, редактор ресурсів, dependency scanner та дизасемблер (рис.4.9). Програма дозволяє аналізувати PE-заголовки, ресурси, API-виклики, імпортовані бібліотеки, стиснуті секції та приховані ресурси. Вона активно використовується у задачах форензики, reverse engineering та binary auditing.



Рисунок 4.9. Титульна сторінка сайту з PE Explorer

4. Dependency Walker (офіційна сторінка сайту https://www.dependencywalker.com/)

Dependency Walker використовується для перегляду імпортованих бібліотек, залежностей DLL та API-викликів програми. Він є класичним інструментом аналізу залежностей DLL Хоча він не є повноцінним PE-viewer у вузькому сенсі, цей інструмент дозволяє швидко виявляти підозрілі зовнішні залежності, приховані бібліотеки та нетипові виклики функцій, що важливо під час первинного аналізу виконуваних файлів. Сторінка завантаження містить актуальні версії x86 / x64 та опис можливостей програми (рис.4.10).

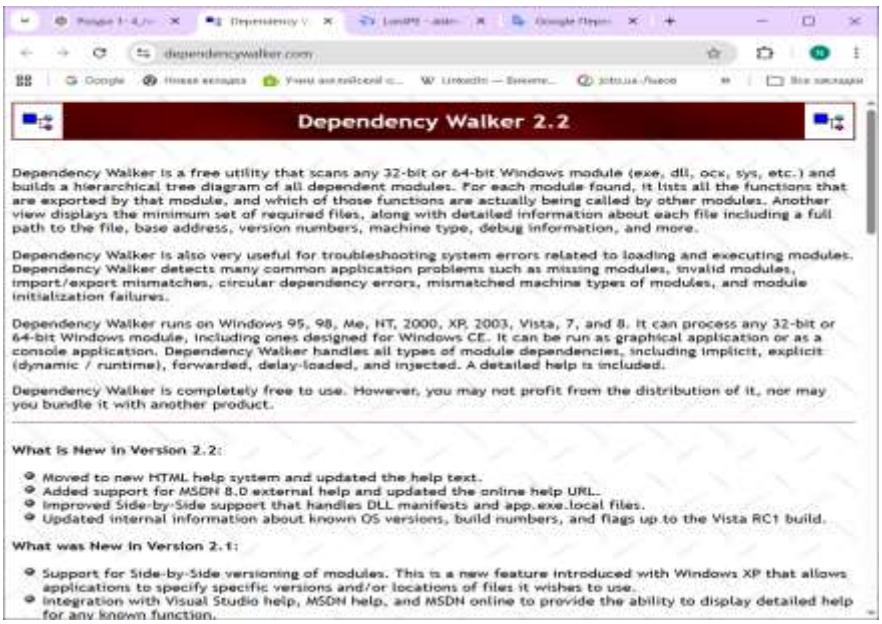


Рисунок 4.10. Сторінка опису Dependency Walker

5. LordPE (сторінка опису - https://www.aldeid.com/wiki/LordPE)

LordPE є одним із традиційних інструментів аналізу PE-структури, який широко використовувався у malware analysis та unpacking-завданнях. Він дозволяє досліджувати PE Header, таблиці імпорту, секції, а також працювати з rebuilding PE після unpacking. Попри застарілий інтерфейс, інструмент залишається корисним для демонстрації класичних підходів до аналізу виконуваних файлів.

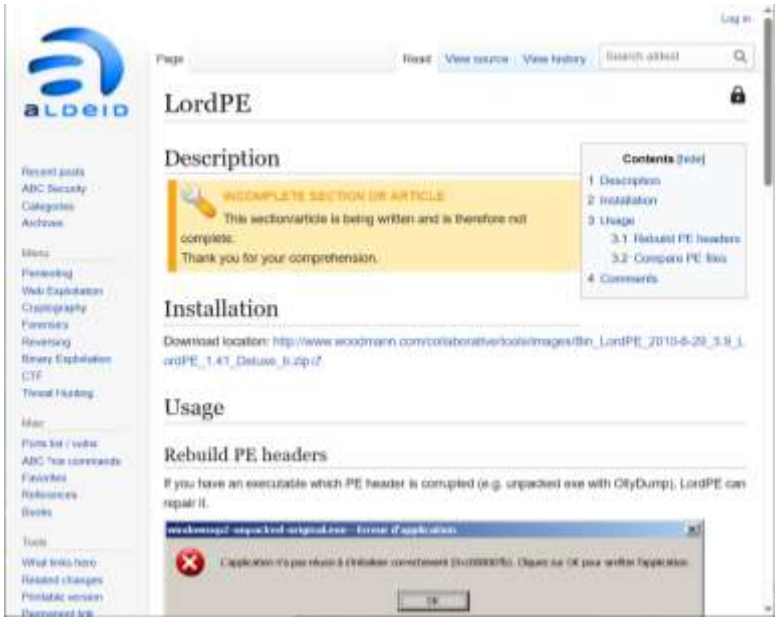


Рисунок 4.11. Сторінка опису LordPE

Отже, PE-viewers дозволяють значно швидше, ніж звичайні Hex-редактори, отримати структуроване уявлення про виконуваний файл і виявити непрямі ознаки компрометації без необхідності негайного переходу до дизасемблювання. Саме тому вони є важливою складовою первинного аналізу програмного забезпечення, особливо у системах цивільного захисту, де перевірка достовірності та цілісності програмних компонентів має критичне значення.

4.2.3. Спеціалізований інструмент CFF Explorer

CFF Explorer є спеціалізованим інструментом первинного аналізу виконуваних файлів формату PE (Portable Executable), який поєднує функції PE-viewer, редактора заголовків і засобу перевірки структури виконуваних модулів. Програма входить до складу комплексу Explorer Suite та широко використовується під час дослідження EXE, DLL, SYS та інших виконуваних файлів у середовищі Windows [4.1.3].

На відміну від звичайних PE-viewers, CFF Explorer дозволяє не лише переглядати структуру PE-файлу, але й виконувати контрольоване редагування окремих службових полів. Це робить його особливо корисним для експериментальних прийомів первинного аналізу, коли необхідно дослідити реакцію операційної системи або прикладного програмного забезпечення на зміну заголовків, секцій або службових атрибутів.

Основні можливості CFF Explorer (рис.4.12):

- перегляд DOS Header, NT Header та Optional Header;
- аналіз Section Headers;
- дослідження Import Table та Export Table;
- перевірка Entry Point;
- перегляд ресурсів (Resources);
- аналіз цифрових підписів і сертифікатів;
- перевірка Relocations, TLS, Debug Directory;
- редагування окремих полів PE-заголовків;

– -аналіз залежностей бібліотек;
– виявлення ознак пакування або прихованої модифікації [43; 42].



Рисунок 4.12. Інтерфейс CFF Explorer
(https://kb.offsec.nl/tools/windows/cffexplorer/)

Особливу практичну цінність має можливість редагування службових полів PE-файлу. Наприклад, зміна часової мітки (TimeDateStamp), точки входу (Entry Point) або параметрів секцій дозволяє дослідити, як змінюється поведінка системи під час запуску програми або її відображення у середовищі Windows. Саме такі контрольовані експерименти використовуються для виявлення непрямих ознак компрометації та аналізу механізмів захисту.

У задачах цивільного захисту CFF Explorer доцільно застосовувати для перевірки цілісності критичного програмного забезпечення після оновлень, контролю системних бібліотек, аналізу підозрілих виконуваних файлів та дослідження інцидентів інформаційної безпеки. Особливо важливим є виявлення прихованих змін у модулях, що відповідають за моніторинг надзвичайних ситуацій, маршрутизацію реагування або обробку критичних повідомлень.

Прикладом практичного використання є перевірка відповідності офіційного дистрибутива та встановленого виконуваного файлу. Якщо структура секцій, часові мітки або імпортовані бібліотеки відрізняються від еталонних значень, це може свідчити про пакування, модифікацію або впровадження програмної закладки.

CFF Explorer є одним із найбільш зручних інструментів для навчального та практичного первинного аналізу PE-файлів, оскільки дозволяє поєднати структурний перегляд виконуваного файлу з контрольованим експериментом над його службовими параметрами. Саме це забезпечує обґрунтований перехід від первинного аналізу до поглибленого статичного дослідження програмного забезпечення.

4.3. ПЕРВИННИЙ АНАЛІЗ СТРУКТУРИ PE-ФАЙЛУ

В операційних системах Windows виконувані файли мають формат Portable Executable (PE), який використовується для файлів типу *.exe, *.dll, *.sys, *.drv, .ocx та інших. На етапі первинного аналізу дослідження PE-файлу дозволяє без виконання програми визначити її базові характеристики: тип, архітектуру, структуру, точку входу, залежності від бібліотек та можливі ознаки модифікації або прихованого втручання [42; 26].

Формат PE містить:

− DOS-заголовок (сигнатура MZ – 0x5A4D);

− NT-заголовок (PE\0\0) – основна інформація про архітектуру;

− секції: .text (код), .data (дані), .bss (неініціалізовані дані), .idata (імпорт), .edata (експорт), .rsrc (ресурси), .reloc (таблиця переміщень).

Саме ці елементи є першими об’єктами перевірки під час первинного аналізу, оскільки дозволяють швидко встановити справжній тип файлу навіть у випадках підміни розширення, виявити ознаки пакування, підозрілі секції або нетипові параметри виконуваного модуля.

Цифровий підпис (сигнатура) файлу є унікальною послідовністю байтів, що записується у заголовку та дозволяє визначити реальний тип файлу. Йдеться не про електронний цифровий підпис (certificate signature), а саме про файлову сигнатуру (magic bytes), яка використовується для ідентифікації формату [42].

Аналіз PE-заголовка дає змогу:

− встановити адресу точки входу (Entry Point);

− визначити архітектуру (x86, x64);

− виявити бібліотеки, від яких залежить програма;

− перевірити таблиці імпорту та експорту;

− виявити ознаки модифікації виконуваного файлу.

Наприклад, якщо підозрілий бінарний файл має формат Portable Executable (PE), можна зробити висновок, що він призначений для операційної системи Windows. Проте покладатися лише на розширення файлу не рекомендується, оскільки зловмисники часто змінюють його зовнішній вигляд або підміняють розширення для маскування.

Тому під час первинного аналізу використовується перевірка сигнатури файлу – перших байтів заголовка, які дозволяють встановити реальний формат незалежно від назви файлу.

Виконувані файли Windows мають сигнатуру MZ або шістнадцяткове значення 4D 5A у перших двох байтах файлу. (Рис.4.13)

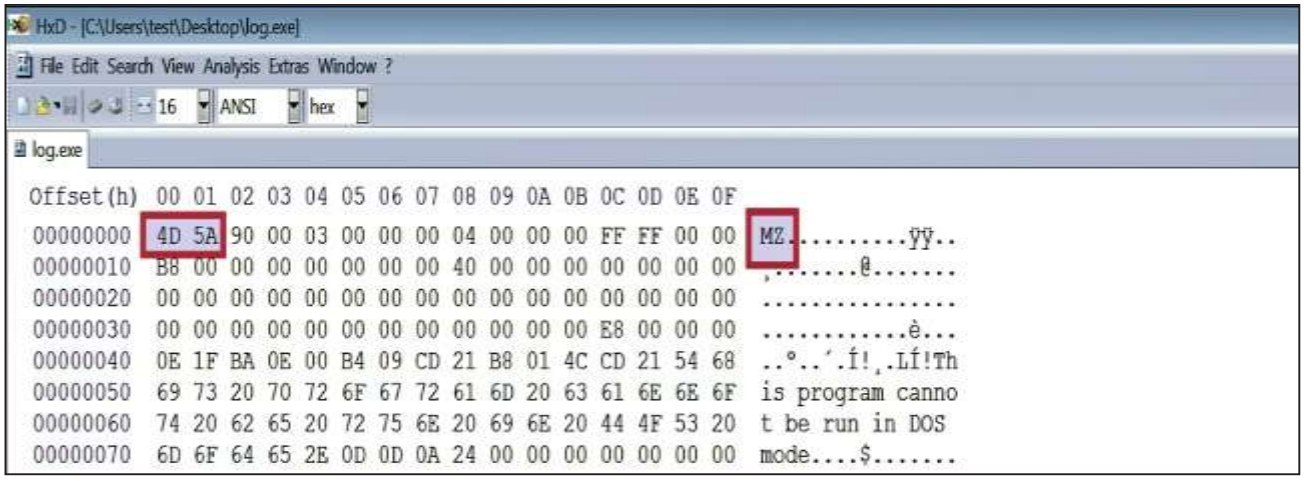


Рисунок 4.13. Цифровий підпис файлу

В системах Linux, щоб знайти сигнатуру файлу, можна використати команду xxd (ручна обробка):

```
$ xxd -g 1 log.exe | more
```

При виконанні команди буде згенеровано 16-ковий дамп файлу (рис. 4.14):

```
0000000: 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 MZ..............
0000010: b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000030: 00 00 00 00 00 00 00 00 00 00 00 00 e8 00 00 00 ................
```

Рисунок 4.14. Шістнадцятковий дамп файлу, наданий командою xxd (Linux)

Або застосувати інструментальне програмне забезпечення:

- утіліту file (для Linux);
- CFF Explorer (для Windows);
- Модуль Python - magic

Приклад для Linux (file mini):

```
$ File mini
mini: PE32 executable (GUI) Intel 80386, for MS Windows
$ File notepad.exe
notepad.exe: PE32 + executable (GUI) x86-64, for MS Windows
```

Інструментом для Windows є частина пакету Explorer (www.ntcore.com/exsuite.php) – рисунок 4.15 та 4.16.



Рисунок 4.15. Застосування ntcore

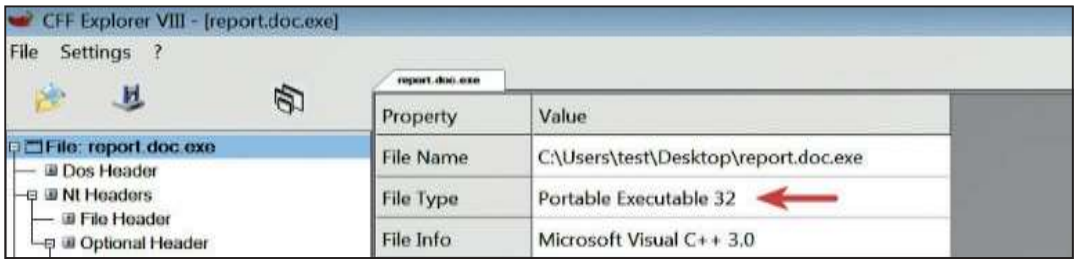


Рисунок 4.16. Виявлення сигнатури файла з пакетом Explorer

Виявлення сигнатури файла з Python-magic відбувається при застосуванні коду:

```
>>> import magic
>>> m = magic.open (magic.MAGIC_NONE)
>>> m.load ()
>>> ftype = m.file (r'log.exe ')
>>> print ftype
```

```
PE32 executable (GUI) Intel 80386, for MS Window
```

У заголовочному файлі winnt.h визначені основні сигнатури виконуваних файлів Windows [42] (табл. 4.1).

Таблиця 4.1

Сигнатури виконуваних файлів

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

Загальна структура виконуваного файлу

PE-файл представляє собою серію структур і підкомпонентів, які містять інформацію, необхідну операційній системі для завантаження програми у пам’ять.

Під час виконання двійкового файлу завантажувач операційної системи читає інформацію з PE-заголовка, після чого визначає:

− де файл має бути завантажений у пам’ять;

− з якої адреси починається виконання;

− які бібліотеки необхідні для роботи;

− які ресурси використовуються програмою.

Саме тому PE-заголовок є ключовим об’єктом первинного аналізу: його дослідження дозволяє отримати значний обсяг інформації ще до переходу до повноцінного статичного аналізу або дизасемблювання.

Процес аналізу ґрунтується на структурі PE-файлу, загальний вигляд якої наведено на рисунку 4.17.

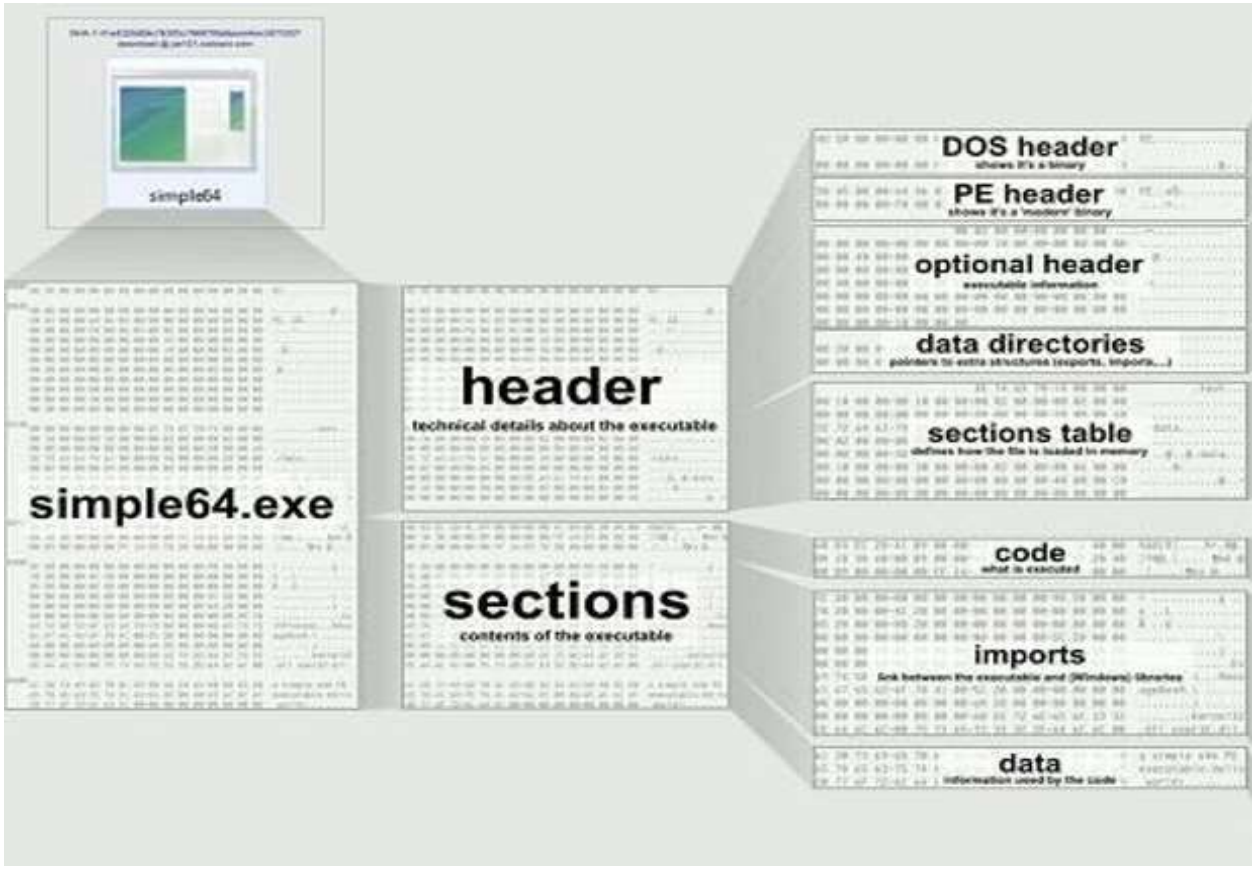


Рисунок 4.17. Структура формату Portable Executable (PE)

Перший байт образу завжди починається із сигнатури MZ, що легко перевірити за допомогою будь-якого hex-редактора, PE-viewer або спеціалізованих інструментів, таких як CFF Explorer.

Виконуваний файл складається із сигнатури, заголовків, таблиці секцій і безпосередньо самих секцій.

Найбільш застосовними в аналізі є такі секції:

.text – секція програмного коду;

.data – секція ініціалізованих даних;

.bss – секція неініціалізованих даних;

.idata – секція імпортованих функцій;

.edata – секція експортованих функцій;

.rsrc – секція ресурсів;

.reloc – секція таблиці базових поправок;

.tls – Thread Local Storage.

Аналіз структури секцій дозволяє виявити нетипові або приховані області, ознаки пакування, вставлення стороннього коду або спроби прихованої модифікації виконуваного файлу [19; 42].

Таким чином, первинний аналіз структури PE-файлу дозволяє швидко визначити базові характеристики програмного об’єкта, встановити ознаки потенційної компрометації та підготувати основу для подальшого статичного аналізу, дослідження механізмів захисту та перевірки можливих програмних закладок.

4.4. ПРИКЛАДИ ЕКСПЕРИМЕНТІВ

Первинний аналіз програмного забезпечення передбачає використання контрольованих експериментальних прийомів, які дозволяють без глибокого втручання у внутрішню логіку програми виявити її структуру, службові параметри, механізми захисту та можливі ознаки несанкціонованого втручання. На цьому етапі експеримент виступає не як самостійний метод дослідження, а як практичний інструмент початкової діагностики програмного об’єкта [18; 19].

Загальна схема проведення такого аналізу передбачає послідовне виконання таких етапів: спочатку формулюється гіпотеза щодо властивостей або поведінки досліджуваної програмної системи; далі обирається параметр або група параметрів, що підлягають контрольованій зміні; після цього здійснюється вплив на систему шляхом модифікації вхідних даних, заголовків файлів або умов виконання; отримані результати фіксуються з урахуванням змін у поведінці програми або структури даних; на завершальному етапі проводиться аналіз отриманих результатів та перевірка початкової гіпотези.

4.4.1. Аналіз формату заголовка бінарного файлу даних

Якщо програма працює з вхідними даними у вигляді бінарного файлу, формат його заголовків часто можна встановити шляхом контрольованої зміни окремих байтів на початку файлу та спостереження за реакцією системи. Наприклад, якщо після зміни певного байта архіватор відображає іншу дату

створення архіву, це означає, що змінений байт входить до складу поля дати або часу (рис.4.18, рис.4.19).

а) «Чистий експеримент»(рис.4.18)

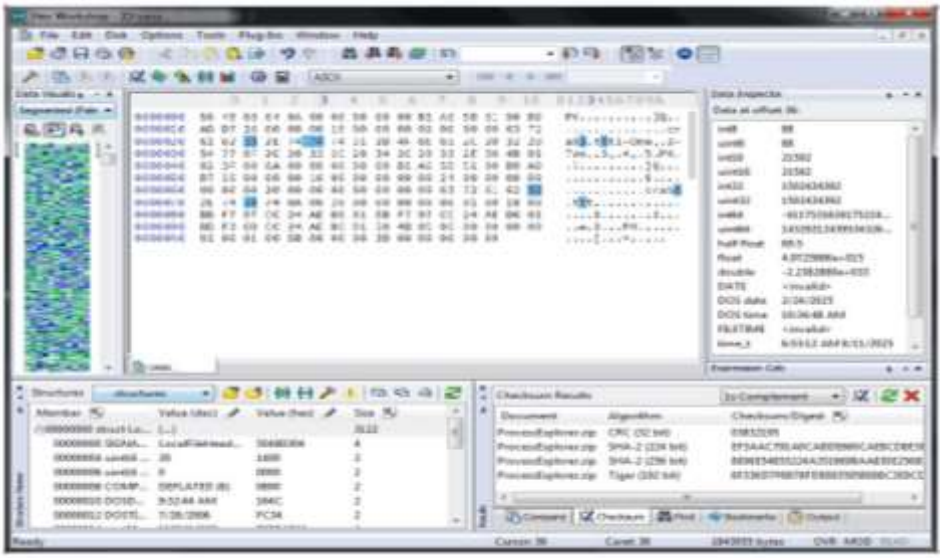


Рисунок 4.18. Результат експерименту з незміненим бінарним файлом

б) Пошук дати та часу (рис.4.19)

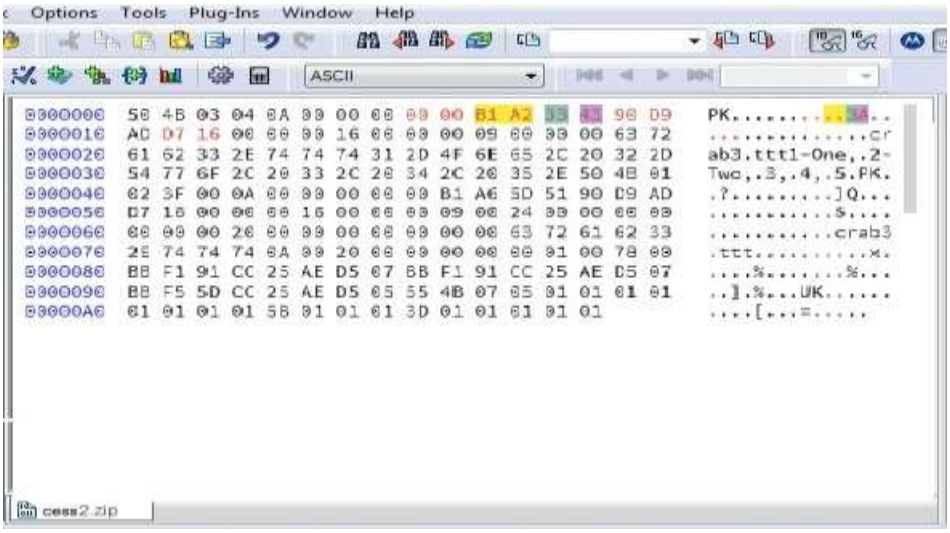


Рисунок 4.19. Виявлення значень дати та часу

Примітка: застосовано інструмент HEX WORKSHOP (http://www.hexworkshop.com/)

На рисунках 4.20, 4.21 наведено результат експерименту з модифікації заголовка файлу, поміщеного у ZIP-архів, що проявляється через зміну метаданих файлу (дати та часу й розширення) у стандартному файловому інтерфейсі операційної системи Windows і демонструє безпосередній зв’язок

між побайтовими змінами у заголовку та їх відображенням на прикладному рівні.

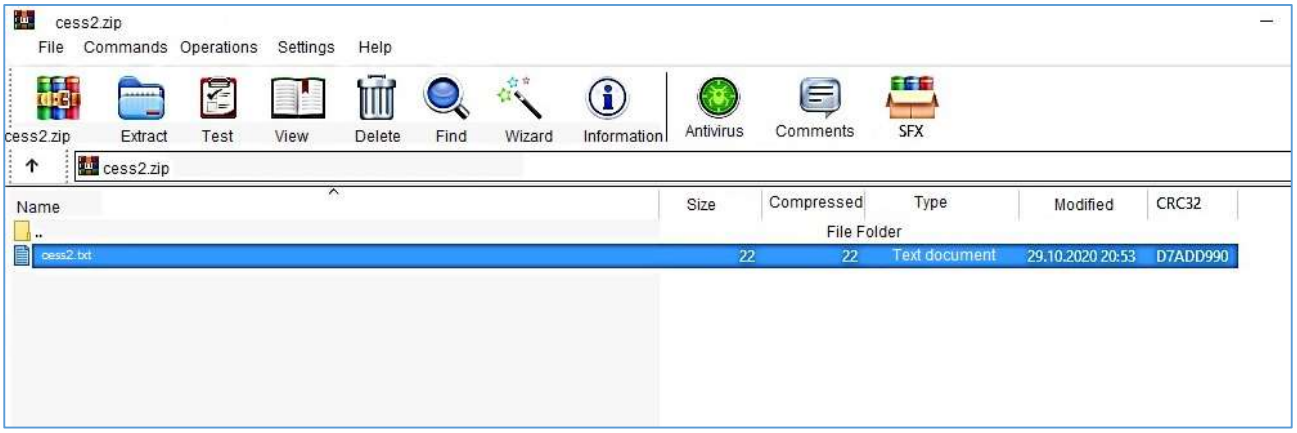


Рисунок 4.20. Початковий стан метаданих файлу в ZIP-архіві до модифікації (дата та час: 29.10.2020, 20:53, тип - .txt)

На рисунку 4.21 представлено модифікований стан бінарного файлу. Виявлені відмінності підтверджують локалізацію змінених полів і демонструють вплив точкових модифікацій на поведінку або інтерпретацію файлу системою.

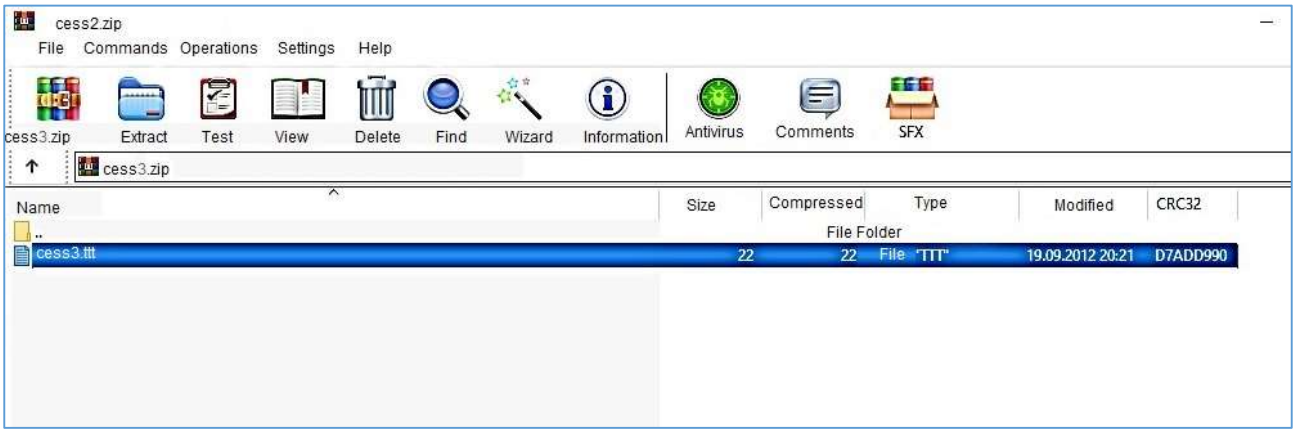


Рисунок 4.21. Змінений стан метаданих файлу після модифікації заголовка ZIP-архіву (дата та час: 19.09.2012, 20:21, тип .ttt)

Результати підтверджують, що навіть незначна зміна окремих байтів заголовка дозволяє локалізувати службові поля файлу та виявити залежність між структурою заголовка і поведінкою операційної системи.

4.4.2. Виявлення випадкового параметра у криптосистемі

Для визначення використання випадкового параметра (nonce, IV або марканту) проводиться дворазове шифрування одного й того самого відкритого тексту на одному й тому самому ключі.

Якщо в результаті двох шифрувань отримано різний шифротекст, це свідчить про використання випадкового параметра, який ускладнює виявлення закономірностей у потоці даних.

а) Перше шифрування crab2_21_04.zip (рис. 4.22)

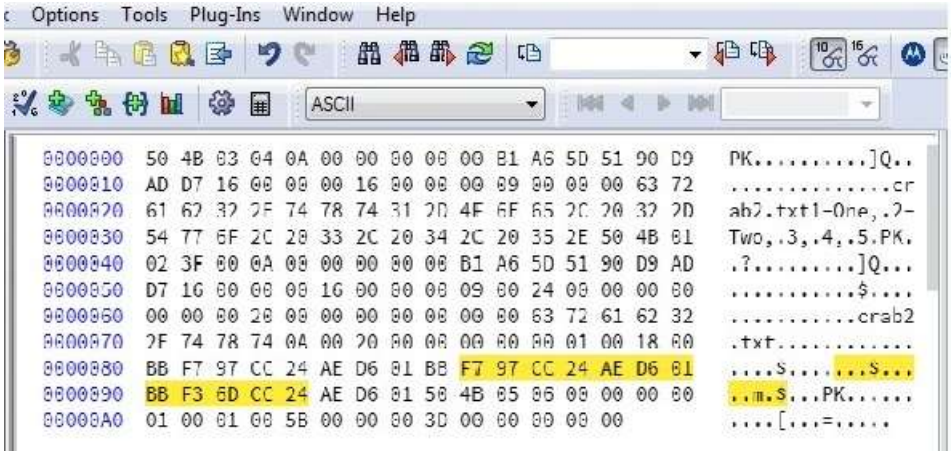


Рисунок 4.22. Виявлення випадкового параметра

б) Повторне шифрування crab2_23_35.zip (рис.4.23)

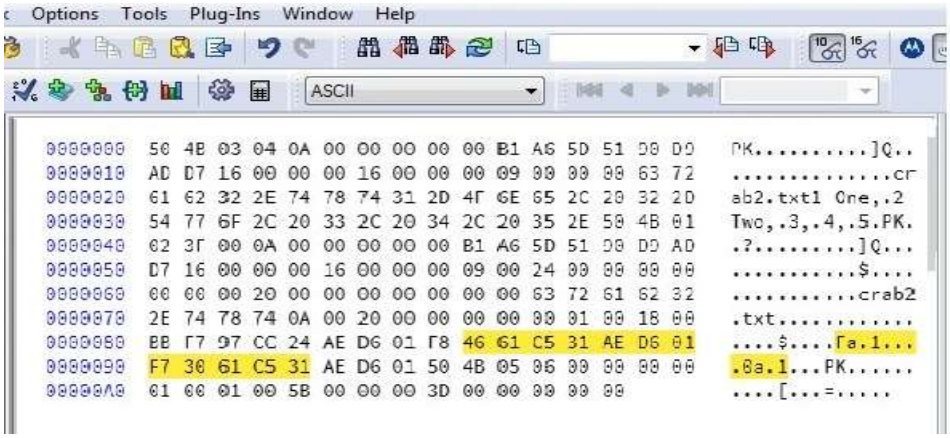


Рисунок 4.23. Повторне шифрування, використання випадкового параметра

Такий прийом дозволяє на етапі первинного аналізу оцінити наявність додаткових криптографічних механізмів захисту без необхідності повного дослідження алгоритму.

Експеримент 1. Перевірка залежності випадкового параметра від системного часу

Повторюється попередній експеримент, але в умовах контрольованої зміни системного часу або зупинки системного годинника. Якщо при повторному шифруванні отримуються однакові шифротексти, це свідчить про те, що генерація випадкового параметра не залежить від поточного часу.

Такий прийом дозволяє виявити джерело формування випадкових значень та оцінити криптографічну стійкість реалізації.

Експеримент 2. Визначення типу криптографічного перетворення

Для цього готуються два відкриті тексти, що відрізняються лише одним байтом, після чого кожен із них шифрується на одному ключі.

Якщо зміни в шифротексті:

– локалізуються в одному байті – ймовірно використовується потоковий шифр;

– поширюються на кілька байтів – імовірний блоковий шифр;

– тривають до кінця файлу – використовується режим зворотного зв’язку.

Такий підхід дозволяє на етапі первинного аналізу зробити попередній висновок щодо типу криптографічного перетворення без повного реверс-інжинірингу алгоритму.

4.4.3. Приклад проведення первинного аналізу виконуваного файлу

Розглянемо фрагмент файлу, на прикладі якого проведемо первинний аналіз виконуваного PE-файлу з використанням експериментальних прийомів.

```
#include <iostream>
#include <string>

using namespace std;

int MathPlus(int a, int b) {
     return a + b;
}
int main()
{
```

```
    string secretVar = "Secret variable that is unused";
    int varA = 3;
    int varB = 21;
    cout << "Var A = ";
    cout << varA;
    cout << "\nVar B = ";
    cout << varB;
  cout << "\n3 + 21 = ";
    cout << MathPlus(varA, varB);
    cout << "\n";
    cin.get();
}
```

Наведений фрагмент програмного коду реалізує просту консольну програму мовою C++, яка виконує арифметичну операцію додавання двох цілих чисел та виводить результат на екран.

Особливістю цього прикладу є наявність рядкової змінної secretVar, яка оголошена, але не використовується у процесі виконання програми. Це дозволяє під час первинного аналізу виконуваного файлу ідентифікувати відповідний фрагмент у бінарному представленні, оскільки рядок буде збережений у секції даних, незважаючи на відсутність його використання у логіці програми.

Після компіляції програми у формат виконуваного файлу (PE) зазначені елементи коду відображаються у різних секціях:

– інструкції функції MathPlus – у секції виконуваного коду;

– змінні та рядкові константи – у секції даних;

– текстові повідомлення – у вигляді ASCII-послідовностей.

Це створює можливість на етапі первинного аналізу встановити відповідність між логічною структурою програми та її бінарною реалізацією, відокремити код від даних і виявити потенційно чутливу інформацію, яка може залишатися у виконуваному файлі навіть без фактичного використання.

Таким чином, наведений код програми використовується як базовий приклад для демонстрації відповідності між вихідним кодом і його бінарною реалізацією, що є ключовим елементом первинного аналізу програмного забезпечення.

Після компіляції програми у формат виконуваного файлу (PE) результати дослідження підтверджуються аналізом, наведеним на рисунках 4.24-4.27:

на рис. 4.24 показано загальний формат виконуваного файлу;

на рис. 4.25 підтверджено належність файлу до середовища Windows (формат PE);

на рис. 4.26 відображено області даних програми;

на рис. 4.27 ідентифіковано рядок secretVar, що демонструє можливість виявлення навіть невикористаних змінних у бінарному представленні.

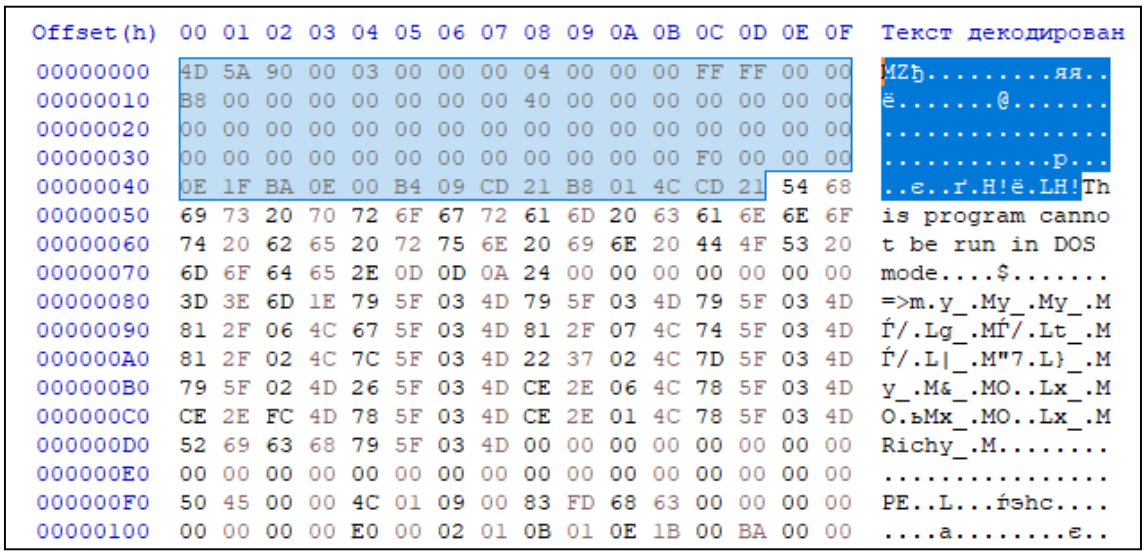


Рисунок 4.24. Формат файлу

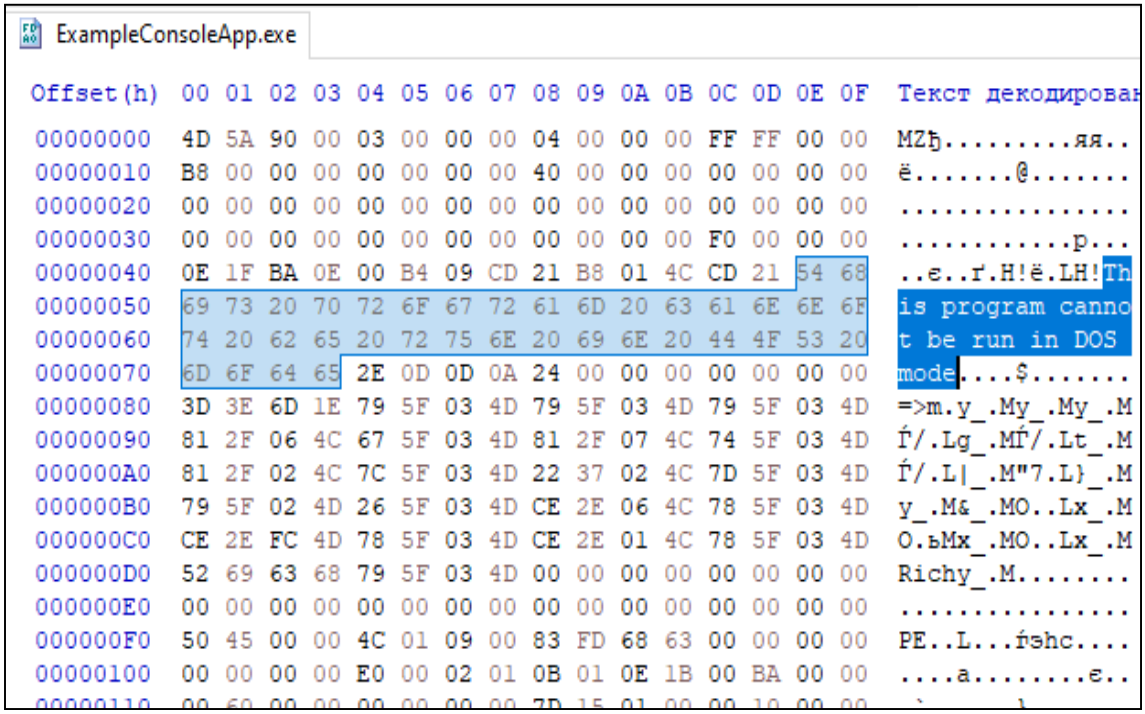


Рисунок 4.25. Підтвердження, що програма компільована для середовища «Windows»

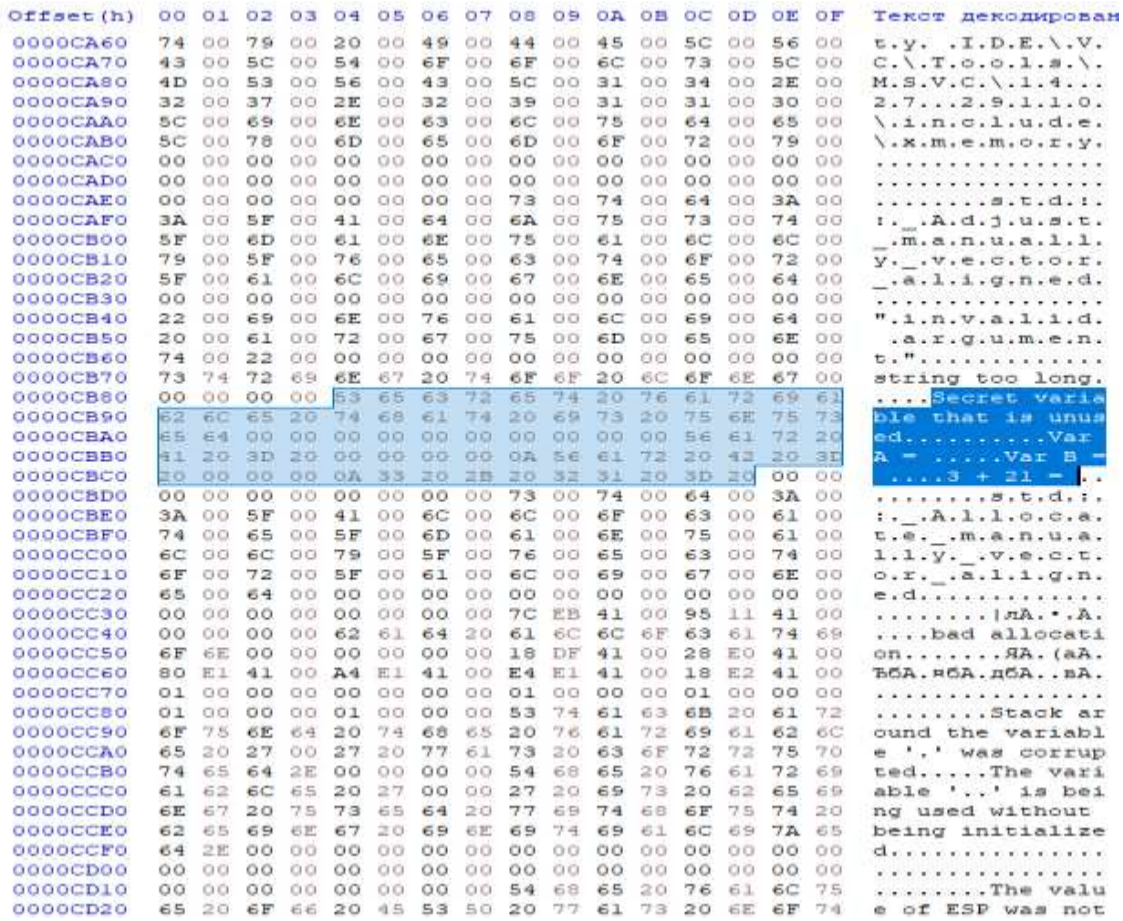


Рисунок 4.26. Області даних програми

На рисунку виділено області даних програми, що відповідають окремим сегментам виконуваного файлу або пам’яті процесу. Зокрема, показано фрагменти, які містять статичні рядки, службові повідомлення та змінні, що зберігаються у секції даних, а не в області виконуваного коду.

Такий спосіб подання наочно ілюструє можливості HEX-редакторів під час первинного аналізу виконуваних файлів і дозволяє відокремлювати код від даних, ідентифікувати потенційно чутливу інформацію та оцінювати ризики її несанкціонованого використання або модифікації.

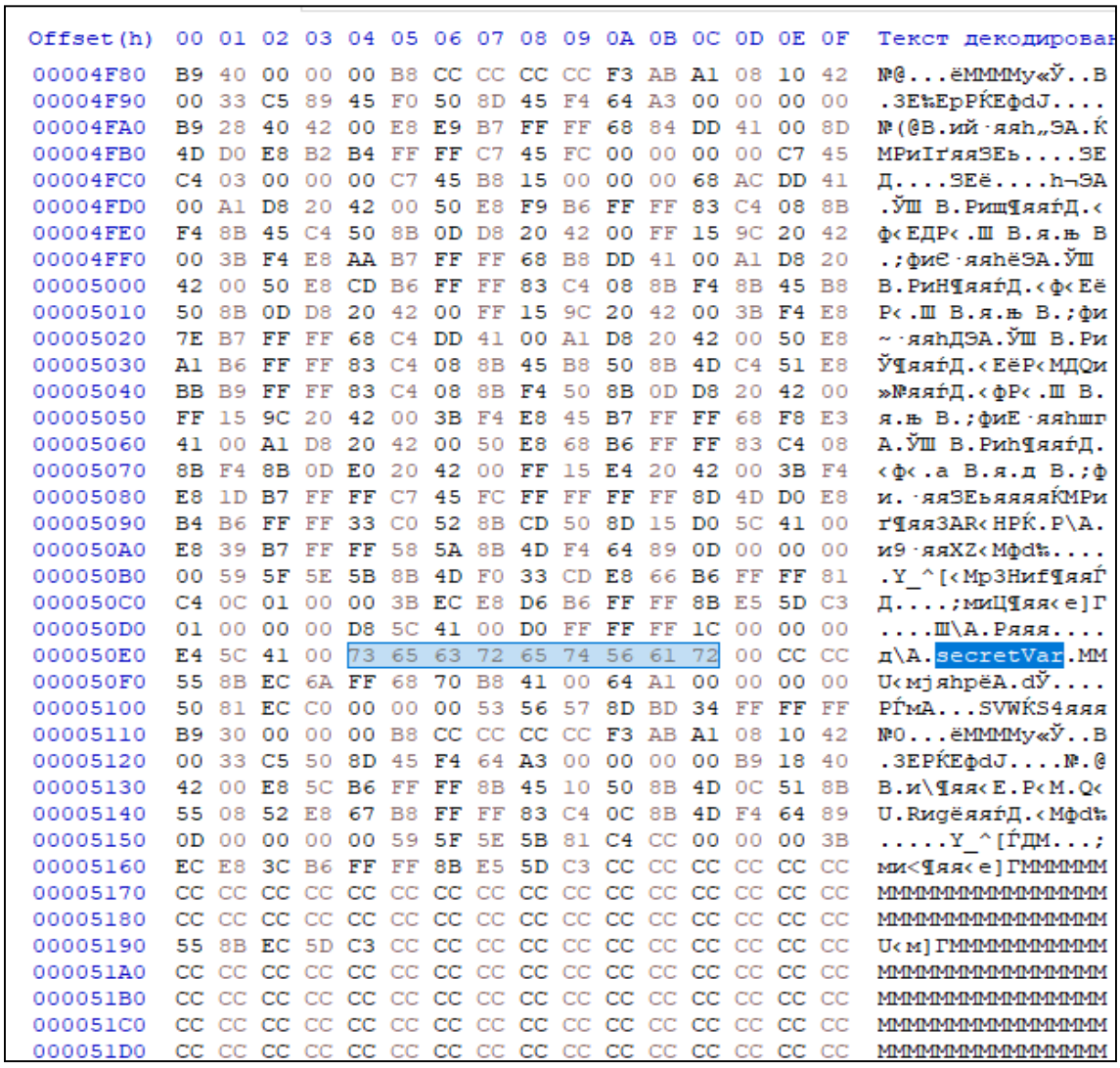


Рисунок 4.27. «Секретна» зміна, що була оголошена, але не використовувалась

Отже, зіставлення вихідного коду та результатів аналізу виконуваного файлу ілюструє можливості первинного аналізу щодо:

– встановлення відповідності між логічною структурою програми та її бінарною реалізацією;

– відокремлення коду від даних;

– виявлення потенційно чутливої інформації;

– фіксації непрямих ознак несанкціонованих модифікацій.

Це має практичне значення для задач забезпечення безпеки програмного забезпечення у системах цивільного захисту, зокрема під час перевірки критичних програмних компонентів, аналізу оновлень та виявлення прихованих змін у виконуваних файлах.

Первинний аналіз формує основу для подальшого статичного, динамічного та гібридного дослідження програмного забезпечення, дозволяючи на ранньому етапі виявити аномалії структури файлів, ознаки модифікації, порушення цілісності та непрямі індикатори несанкціонованого втручання.

У підрозділах цивільного захисту застосування первинного аналізу з використанням експериментальних прийомів дозволяє вирішувати практичні задачі забезпечення безпеки програмного забезпечення відповідно до вимог створення та супроводження комплексних систем захисту інформації (КСЗІ), а також міжнародних стандартів управління інформаційною безпекою [12; 13].

Зокрема, шляхом контрольованої перевірки заголовків файлів, порівняння контрольних сум (CRC, SHA-256), аналізу змін після оновлень або тестування реакції системи на спеціально сформовані вхідні дані можливо своєчасно виявляти несанкціоновані зміни у програмному забезпеченні, що відповідає вимогам контролю цілісності, визначеним у НД ТЗІ 2.5-004-99, ISO/IEC 27001 та ISO/IEC 27002 [13; 15].

Важливим напрямом є також перевірка коректності реалізації криптографічного захисту, зокрема виявлення використання випадкових параметрів (nonce, IV), аналіз залежності результатів шифрування від системного часу та оцінювання механізмів контролю цілісності даних. Такі перевірки підтримують процеси оцінювання ризиків відповідно до ISO/IEC 27005 та забезпечують практичне підтвердження надійності програмних систем.

Тому, первинний аналіз не є самостійним завершеним методом дослідження, а виступає необхідним початковим етапом оцінювання безпеки програмного забезпечення, який дозволяє обґрунтовано перейти до поглибленого статичного та динамічного аналізу і забезпечити надійне функціонування інформаційних систем цивільного захисту.

ПИТАННЯ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ ЗА ТЕМОЮ 4

1. У чому полягає сутність первинного аналізу програмного забезпечення з використанням експериментальних прийомів?
2. Які основні інструментальні засоби застосовуються для первинного аналізу програмного забезпечення (Hex-редактори, PE-viewers, CFF Explorer)?
3. Які основні можливості Hex-редакторів використовуються під час первинного аналізу виконуваних файлів?
4. Як визначається наявність марканту (nonce, IV) у криптосистемі?
5. Як з’ясувати залежність марканту від системного часу?
6. Як за допомогою контрольованого експерименту визначити тип криптографічного перетворення?
7. Які основні елементи містить структура PE-файлу формату Portable Executable (PE)?
8. Яку інформацію дозволяє отримати аналіз PE-заголовка на етапі первинного аналізу?
9. У чому полягає зв’язок між первинним, статичним і динамічним аналізом програмного забезпечення?
10. Чому перевірка сигнатури файлу (MZ, PE) є важливою під час первинного аналізу підозрілого програмного забезпечення?

ПРАКТИЧНІ ЗАВДАННЯ ДО ТЕМИ 4

УВАГА! Усі експерименти повинні виконуватися в ізольованому середовищі (віртуальна машина або sandbox), що є обов’язковою вимогою безпеки як у навчальних експериментах, так й у підрозділах цивільного захисту.

ЗАВДАННЯ 1. ДОСЛІДЖЕННЯ СТРУКТУРИ БІНАРНОГО ФАЙЛУ МЕТОДОМ ПЕРВИННОГО АНАЛІЗУ

Постановка завдання:

На основі підрозділів 4.1, 4.2.1, 4.2.2, 4.2.3 та 4.2.4 посібника (зокрема Експерименту 1 та Експерименту 5) проведіть дослідження внутрішньої

структури бінарного файлу за допомогою hex-редактора та засобів первинного аналізу.

Оберіть один із наведених типів файлів для дослідження:

а) ZIP-архів, що містить один або кілька файлів;

б) виконуваний PE-файл (.exe) скомпільований з відомого вихідного коду;

в) бінарний файл конфігурації або бази даних будь-якої настільної програми.

Для обраного файлу:

1) Відкрийте файл у hex-редакторі (HxD, XVI32, Hex Workshop) або CFF Explorer (для PE-файлів) та зафіксуйте початкові байти (сигнатуру), заголовки та загальну структуру файлу. Визначте, які секції, службові поля або області даних можна ідентифікувати без модифікацій.

2) Проведіть серію не менше ніж із чотирьох цілеспрямованих експериментів: для кожного змінюйте конкретні байти або діапазони байтів і фіксуйте реакцію програми, що відкриває цей файл. Для PE-файлів допускається контрольована зміна службових полів заголовка (наприклад, TimeDateStamp, Entry Point, Section Name) без запуску підозрілих зразків. Результати занесіть у таблицю.

3) На основі отриманих даних складіть схему структури файлу: позначте зміщення (offset), розмір і призначення кожного ідентифікованого поля або секції.

4) Сформулюйте висновок: які поля є критичними для коректного зчитування файлу, які – допоміжними, та які обмеження методу проявились у процесі дослідження.

Приклад розв'язку:

Обраний файл: ZIP-архів archive.zip, що містить один текстовий файл readme.txt (32 байти).

Інструмент: HxD 2.5.

Початкові спостереження: перші два байти – 50 4B (ASCII «PK») – сигнатура формату ZIP. Далі видно блоки «PK\x03\x04» (Local File Header), потім

Central Directory з блоком «PK\x01\x02», і в кінці End of Central Directory (EOCD) «PK\x05\x06».

Таблиця 4.2

Таблиця експериментів

Експеримент	**Змінені байти**	**Значення (до → після)**	**Реакція програми**
1 (сигнатура)	offset 0x00–0x01	50 4B → 00 00	7-Zip: «Неочікуваний кінець архіву»; провідник Windows: «Файл пошкоджений»
2 (дата файлу)	offset 0x0A–0x0B (Last Mod File Time)	C4 52 → C4 42	Час останньої зміни readme.txt змінився | з 10:22 на 08:22 – поле кодує час у форматі MS-DOS (5 біт год, 6 хв, 5 сек)
3 (розмір стиснених даних)	offset 0x12–0x15 (Compressed Size)	20 00 00 00 → 20 00 00 01	7-Zip витягує файл, але повідомляє | «Помилка CRC»; вміст файлу пошкоджено
4 (CRC-32)	offset 0x0E–0x11 (CRC-32 of file data)	A1 B2 C3 D4 → 00 00 | 00 00	7-Zip: «CRC не збігається» – поле захисту | цілісності функціонує як задокументовано

Таблиця 4.3

Схема структури ZIP-файлу

Зміщення	**Розмір**	**Поле**	**Статус**
0x00–0x03	4 байти	Сигнатура LFH (50 4B 03 04)	Критичне: файл не відкриється при зміні
0x04–0x05	2 байти	Версія для розпакування	Допоміжне
0x06–0x07	2 байти	Прапори	Допоміжне (шифрування тощо)
0x08–0x09	2 байти	Метод стиснення	Критичне: визначає алгоритм
0x0A–0x0B	2 байти	Час зміни файлу	Некритичне: метадані
0x0C–0x0D	2 байти	Дата зміни файлу	Некритичне: метадані
0x0E–0x11	4 байти	CRC-32	Критичне: перевірка цілісності
0x12–0x15	4 байти	Стиснений розмір	Критичне: визначає межі даних

Зміщення	Розмір	Поле	Статус
0x16–0x19	4 байти	Нестиснений розмір	Критичне: потрібний для розпакування

Висновок: критичними для коректного зчитування ZIP-файлу є сигнатура (ідентифікує формат), метод стиснення (визначає алгоритм декодування), CRC-32 і розмірні поля. Поля дати та версії – допоміжні. Виявлені обмеження первинного аналізу полягають у тому, що неможливо однозначно встановити призначення поля лише за однією модифікацією: зміна байту у складеному полі (наприклад, time/date) потребує декількох ітерацій з різними значеннями. Для великих архівів кількість необхідних експериментів суттєво зростає, а частина службових полів потребує додаткового статичного аналізу для повної інтерпретації.

ЗАВДАННЯМ 2. ВИЗНАЧЕННЯ ХАРАКТЕРИСТИК КРИПТОСИСТЕМИ МЕТОДАМИ ПЕРВИННОГО АНАЛІЗУ

Постановка завдання:

На основі підрозділу 4.1 та прикладів експериментів 2, 3 і 4 дослідіть програму шифрування, до вихідного коду якої відсутній доступ, використовуючи прийоми первинного аналізу та контрольовані експериментальні перевірки.

Для виконання завдання використовуйте будь-яку утиліту або програму, що приймає відкритий текст і ключ та повертає шифротекст (наприклад, консольний шифратор, архіватор із паролем або інструмент шифрування файлів).

1) Встановіть наявність або відсутність марканту (nonce, IV) у криптосистемі: проведіть дворазове шифрування одного й того самого відкритого тексту на одному й тому самому ключі та порівняйте отримані шифротексти.

2) Якщо маркант виявлено – визначте його можливу залежність від системного часу: виконайте обидва шифрування за максимально однакових часових умов (в межах однієї секунди або за фіксованого системного часу у віртуальному середовищі) та повторіть попередній експеримент.

3) Визначте тип криптографічного перетворення: підготуйте два відкритих тексти, що різняться лише одним байтом, зашифруйте кожен на одному ключі та порівняйте позиції відмінностей у шифротекстах.

4) Зведіть усі результати в таблицю експериментів та сформулюйте гіпотезу про клас алгоритму шифрування (потоковий / блоковий, наявність зворотного зв'язку, залежність від часу).

Приклад розв'язку:

Досліджувана програма: консольна утиліта encrypt.exe.Вхідні параметри: рядок відкритого тексту, рядок ключа. Вихідний параметр: рядок шифротексту у HEX-кодуванні.

Таблиця 4.4

Таблиця експериментів

№екс-пери-мента	**Умова**	**Вхід (текст / ключ)**	**Шифротекст (скорочено)**	**Висновок**
1а	Перше шифрування	«AAAAAAAAAAAAAAA» / «KEY1»	3F8A C201 77E4 B390 ...	—
1б	Повторне шифрування (той же вхід)	«AAAAAAAAAAAAAAA» / «KEY1»	7C2E 90FA 11D8 4401 ...	Шифротексти різні → маркант присутній
2а	Системний час заморожено (одна секунда, GetTickCount = const)	«AAAAAAAAAAAAAAA» / «KEY1»	B4F1 320C AA57 8EBB ...	—
2б	Повтор за тих самих умов	«AAAAAAAAAAAAAAA» / «KEY1»	B4F1 320C AA57 8EBB ...	Шифротексти однакові → маркант залежить від часу
3а	Тексти різняться в байті 0	«BAAAAAAAAAAAAAA» / «KEY1»	91C0 320C AA57 8EBB ...	Відмінності лише у перших 4 байтах → блоковий шифр, розмір блоку ≤ 4 байти

№екс-пери-мента	Умова	Вхід (текст / ключ)	Шифротекст (скорочено)	Висновок
3б	Тексти різняться в байті 4	«AAAABAAAAAAAAAAA» / «KEY1»	B4F1 91C0 AA57 8EBB ...	Відмінності починаються з 5-го байту → крок блоку = 4 байти підтверджено
3в	Зміна байту 0; порівняно хвіст шифротексту	«BAAAAAAAAAAAAAAA» / «KEY1»	91C0 320C AA57 8EBB ...	Хвіст шифротексту збігається → режим без зворотного зв'язку (ECB або схожий)

Гіпотеза: Утиліта encrypt.exe реалізує блоковий шифр з розміром блоку 4 байти без зворотного зв’язку (ECB-подібний режим). Маркант (nonce або IV) генерується на основі системного часу (GetTickCount або аналогічного механізму) і додається до процесу шифрування, що пояснює різницю результатів при однакових вхідних даних.

Перевірочний експеримент: подати на вхід два однакових блоки «AAAAAAAA» (8 байтів) з тим самим ключем при зафіксованому часі. Якщо два 4-байтних блоки шифротексту ідентичні – підтверджено ECB-подібний режим без зворотного зв'язку.

Вхід: «AAAAAAAA» / «KEY1» (час зафіксовано)

Очікуваний результат: B4F1 320C B4F1 320C – перші два блоки однакові.

Отримано: B4F1 320C B4F1 320C – гіпотеза підтверджена.

Прийоми первинного аналізу дозволили без доступу до вихідного коду встановити ключові характеристики криптосистеми: блоковий принцип шифрування, ECB-подібний режим роботи, розмір блоку 4 байти та залежність марканту від системного часу.

Обмеження методу полягають у тому, що для алгоритмів із більшим розміром блоку, складними режимами (CBC, CTR, GCM) або додатковими механізмами маскування кількість необхідних експериментів суттєво зростає, а

однозначна ідентифікація може потребувати додаткового статистичного або динамічного аналізу.

ЗАВДАННЯ 3. ПОРІВНЯЛЬНИЙ АНАЛІЗ HEX-РЕДАКТОРІВ ДЛЯ ЗАДАЧ ПЕРВИННОГО АНАЛІЗУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Постановка завдання:

На основі підрозділу 4.2.1 та 4.2.2 посібника ознайомтеся з характеристиками наведених hex-редакторів та оберіть три з них для детального порівняння позиції застосування у первинному аналізі програмного забезпечення.

Оберіть три hex-редактори з наведеного переліку:

а) HxD;

б) Hex Workshop;

в) Free Hex Editor Neo;

г) HexCmp;

д) XVI32;

е) Lister.

Для кожного обраного редактора:

1) Визначте тип розповсюдження (безкоштовний / комерційний / умовно-безкоштовний) та стисло опишіть ключові можливості, що відрізняють його від аналогів.

2) Оцініть придатність редактора для виконання п’яти типових задач первинного аналізу:

– перегляд і редагування бінарних файлів;

– порівняння двох файлів;

– підрахунок контрольних сум і хешів;

– редагування оперативної пам’яті процесу;

– робота з великими файлами (понад 1 ГБ).

3) Складіть порівняльну таблицю трьох редакторів за наведеними критеріями: позначте підтримку кожної можливості як «Так», «Частково» або «Ні».

4) Сформулюйте рекомендацію: який редактор найбільш доцільно використовувати для задач первинного аналізу програмного забезпечення у підрозділах цивільного захисту, та обґрунтуйте вибір з урахуванням вимог безпеки, контролю цілісності та практичних умов експлуатації.

Приклад розв'язку:

Обрані редактори: HxD, HexCmp, Hex Workshop.

HxD (розробник Maël Hörz, безкоштовний):

Відмінні особливості – підтримка редагування оперативної пам'яті та образів дисків, обчислення контрольних сум і хешів (MD5, SHA-1, SHA-256, CRC-32), порівняння файлів (побайтово), безпечне видалення файлів, розрізання та зшивання файлів. Підтримує файли великого розміру завдяки частковому завантаженню даних. Портативна версія не потребує встановлення, що особливо важливо для роботи в ізольованому середовищі.

HexCmp (розробник Fairdell Software, умовно-безкоштовний):

Відмінні особливості – спеціалізований на порівнянні бінарних файлів: швидке побайтове порівняння з підсвічуванням відмінностей, синхронне прокручування, порівняння окремих фрагментів.

Функції редагування присутні, але мають допоміжний характер. Найбільш ефективний для задач виявлення змін між двома версіями виконуваних файлів.

Hex Workshop (розробник BreakPoint Software, комерційний):

Відмінні особливості – «розумні» закладки, структурний перегляд даних (Data Inspector), арифметичні та логічні операції над блоками, генерація контрольних сум, імпорт / експорт у RTF та HTML, гнучко налаштований інтерфейс. Особливо зручний для структурного аналізу PE-файлів та дослідження заголовків виконуваних модулів.

Таблиця 4.5

Порівняльна таблиця

Критерій	**HxD**	**HexCmp**	**Hex Workshop**
Тип розповсюдження	Безкоштовний	Умовно-безк.	Комерційний
Перегляд і редагування бінарних файлів	Так	Частково	Так
Порівняння двох файлів (побайтове)	Так	Так (спеціалізовано)	Так
Підрахунок контрольних сум і хешів (MD5, SHA, CRC)	Так	Ні	Так
Редагування оперативної пам'яті процесу	Так	Ні	Ні
Робота з файлами понад 1 ГБ	Так	Так (до 4 ГБ)	Так

Рекомендація: для аналізу програмних реалізацій у підрозділах цивільного захисту найбільш доцільним є HxD з таких причин:

– безкоштовне розповсюдження виключає ліцензійні ризики при використанні в державних установах.

– підтримка редагування оперативної пам'яті процесу дозволяє виконувати Експеримент 3 (заморожування GetTickCount) та аналіз поведінки закладок у режимі реального часу без додаткових інструментів.

– вбудований підрахунок хешів (SHA-256, CRC-32) підтримує задачі контролю цілісності файлів – критичну функцію безпеки для ЦЗ-систем.

– портативна версія дозволяє використовувати редактор на ізольованих АРМ без прав адміністратора та без ризику зараження через інсталятор;

– простий інтерфейс забезпечує швидке навчання персоналу та зменшує ймовірність помилок під час первинного аналізу.

HexCmp доцільно застосовувати як допоміжний інструмент виключно для задач порівняння двох версій бінарних файлів (наприклад, оригінального та підозрілого драйвера).

5. СТАТИЧНИЙ АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Статичний аналіз є основою захисту програмних реалізацій. Він дозволяє досліджувати структуру виконуваних файлів, виявляти шкідливі фрагменти коду, приховані залежності та потенційні вразливості без реального запуску програми, що є критично важливим у випадках роботи з шкідливими або підозрілими файлами [17; 19; 44]. Метод забезпечує глибоке розуміння внутрішньої архітектури ПЗ, сприяє виявленню способів обходу захисту та використовується при аудиті, реверс-інжинірингу, розробці антивірусних рішень і систем моніторингу безпеки.

5.1. СУТНІСТЬ І ПРИНЦИПИ СТАТИЧНОГО АНАЛІЗУ

Статичний аналіз є наступним етапом дослідження програмного забезпечення після первинного аналізу та спрямований на поглиблене вивчення внутрішньої структури програмних реалізацій без їх виконання [17; 19; 44].

На відміну від первинного аналізу, який базується на спостереженні зовнішніх характеристик і контрольованих експериментальних впливах, статичний аналіз передбачає системне дослідження коду, структури виконуваних файлів і взаємозв’язків між їх компонентами.

Статичний аналіз – це метод дослідження програмного забезпечення, що полягає у вивченні вихідного, об’єктного або виконуваного коду без запуску програми з метою відновлення логіки її роботи, виявлення вразливостей, прихованих механізмів та ознак захисту.

Основна мета статичного аналізу полягає у встановленні відповідності між структурою програми та її функціональним призначенням, а також у виявленні потенційно небезпечних або модифікованих елементів.

До основних принципів статичного аналізу належать:

– цілісність дослідження – аналіз усіх структурних компонентів програми (заголовків, секцій, таблиць, коду);

– невиконуваність – відсутність запуску програмного коду, що забезпечує безпечність аналізу;

– детермінованість результатів – отримані результати не залежать від середовища виконання;

– реконструкція логіки – відновлення алгоритмів і зв’язків між компонентами програми.

Статичний аналіз є базовим інструментом реверс-інжинірингу, оскільки дозволяє дослідити структуру програми, визначити її складові, взаємодію модулів і залежності навіть за відсутності вихідного коду.

Він широко застосовується при аудиті безпеки, аналізі шкідливого програмного забезпечення, перевірці цілісності програмних компонентів, а також при дослідженні механізмів захисту та протидії несанкціонованому втручанню.

У контексті систем цивільного захисту статичний аналіз має особливе значення, оскільки дозволяє:

– перевіряти цілісність і достовірність програмних компонентів без ризику їх запуску;

– виявляти приховані зміни у критичних модулях обробки даних;

– оцінювати відповідність програмного забезпечення вимогам безпеки та нормативним документам.

Таким чином, статичний аналіз забезпечує перехід від загального уявлення про програму, отриманого на етапі первинного аналізу, до детального дослідження її внутрішньої структури, що є необхідною передумовою для подальшого динамічного аналізу та аналізу механізмів захисту.

5.2. УЗАГАЛЬНЕНА МОДЕЛЬ ПРОЦЕСУ СТАТИЧНОГО АНАЛІЗУ

Статичний аналіз програмного забезпечення виконується як послідовний ітеративний процес, що передбачає поступове уточнення уявлення про структуру та логіку роботи програми [19; 48].

На відміну від первинного аналізу, де дослідження має переважно спостережний і експериментальний характер, статичний аналіз орієнтований на

системну декомпозицію програмного об’єкта та відновлення його внутрішньої організації.

Узагальнену модель процесу статичного аналізу доцільно представити у вигляді послідовності взаємопов’язаних етапів (рис.5.1).

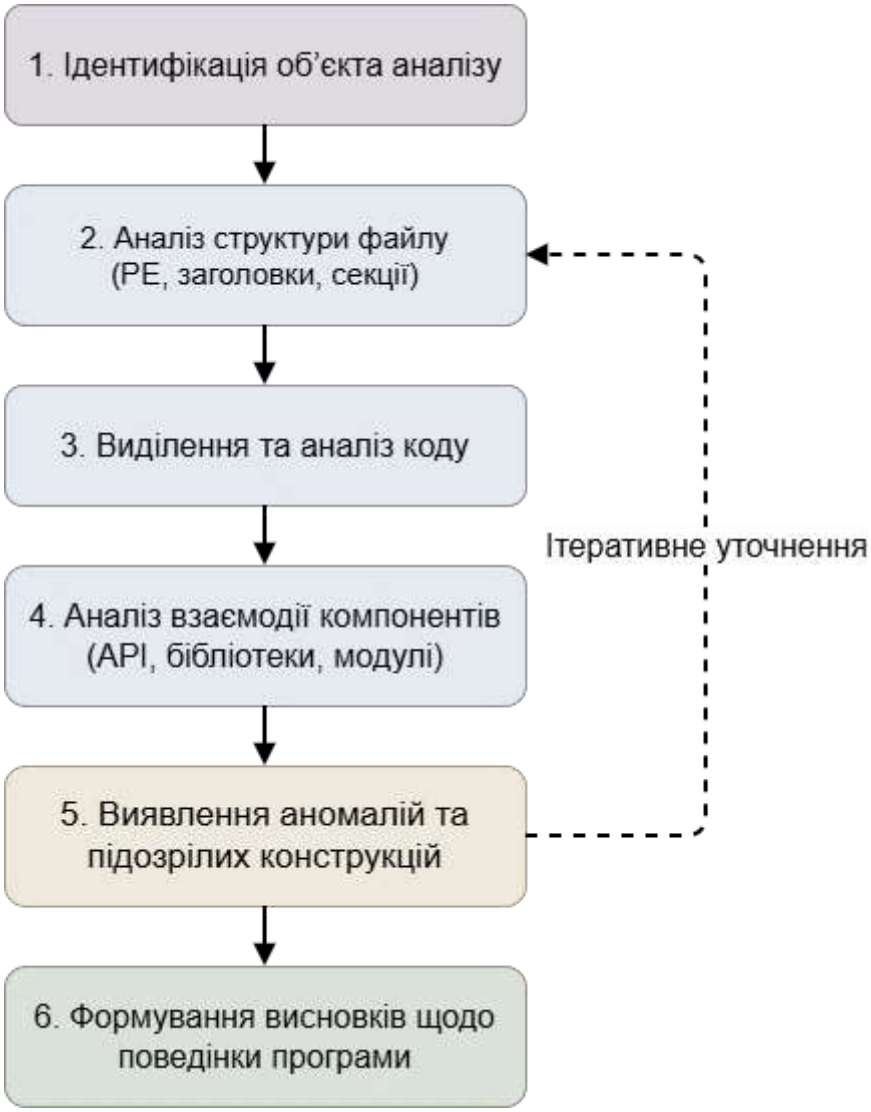


Рисунок 5.1. Типові етапи статичного методу

До основних етапів статичного аналізу належать:

1) Ідентифікація об’єкта аналізу. Виконується визначення типу програмного файлу (виконуваний модуль, бібліотека, скрипт), його формату та призначення. Уточнюються результати первинного аналізу та визначається подальша стратегія дослідження.

2) Аналіз структури виконуваного файлу. Це дослідження заголовків, секцій, службових таблиць і внутрішніх форматів даних. Етап дозволяє

встановити загальну архітектуру програми, точки входу та розміщення основних компонентів.

3) Виділення та дослідження коду програми. Виконується аналіз машинного коду або дизасембльованого представлення з метою виявлення функцій, умов переходу, циклів та викликів. Відбувається реконструкція алгоритмічної логіки та визначення ключових функціональних блоків.

4) Аналіз взаємодії компонентів. На цьому етапі встановлюються зв’язки між модулями та зовнішніми компонентами системи, досліджуються виклики функцій, імпортованих бібліотек, системних API та потоків керування.

5) Виявлення підозрілих або аномальних конструкцій. Це етап пошуку ознак вразливостей, прихованих умов, нестандартних переходів або модифікованих ділянок коду. Особлива увага приділяється нетиповим залежностям, невикористаним фрагментам коду та непрямим переходам.

6) Формування висновків щодо поведінки програми. Проводиться узагальнення результатів аналізу, побудова моделі функціонування та визначення потенційних загроз. Результати цього етапу використовуються для планування подальшого динамічного аналізу або заходів захисту.

Слід зазначити, що наведені етапи не є строго лінійними. Процес статичного аналізу має ітеративний характер, оскільки результати кожного етапу можуть вимагати повернення до попередніх кроків для уточнення або перевірки гіпотез.

У практичних задачах аналізу програмного забезпечення окремі етапи можуть виконуватися паралельно або частково перекриватися, залежно від складності об’єкта дослідження та поставлених цілей.

У системах цивільного захисту така модель процесу дозволяє:

– системно перевіряти програмні компоненти без їх запуску;

– виявляти приховані зміни або модифікації критичних модулів;

– підвищувати достовірність оцінювання безпеки програмного забезпечення.

Узагальнена модель процесу статичного аналізу забезпечує логічну та методичну основу для поглибленого дослідження програмних реалізацій і виступає ключовим етапом між первинним і динамічним аналізом.

5.3. АНАЛІЗ СТРУКТУРИ ВИКОНУВАНИХ ФАЙЛІВ (PE) ЯК СКЛАДОВА СТАТИЧНОГО АНАЛІЗУ

Однією з ключових складових статичного аналізу програмного забезпечення є дослідження структури виконуваних файлів. Для операційних систем сімейства Windows таким стандартом є формат Portable Executable (PE) [42], який використовується для представлення виконуваних файлів (.exe), динамічних бібліотек (.dll), драйверів та інших типів програмних модулів.

На відміну від первинного аналізу, де структура файлу визначається на рівні сигнатур і загальних характеристик, у межах статичного аналізу здійснюється детальне дослідження внутрішньої організації PE-файлу, його заголовків, секцій і службових таблиць.

Формат PE має ієрархічну структуру, що включає заголовки (headers), які містять службову інформацію; секції (sections), у яких розміщуються код, дані та ресурси; таблиці (directories), що визначають взаємодію з зовнішніми компонентами [42, 26].

Аналіз цієї структури дозволяє встановити не лише формальні характеристики файлу, але й отримати важливу інформацію про логіку роботи програми, її залежності та потенційні точки втручання.

Для узагальнення структури PE-файлу доцільно використати схему, що відображає взаємозв’язок його основних компонентів (рис. 5.2).

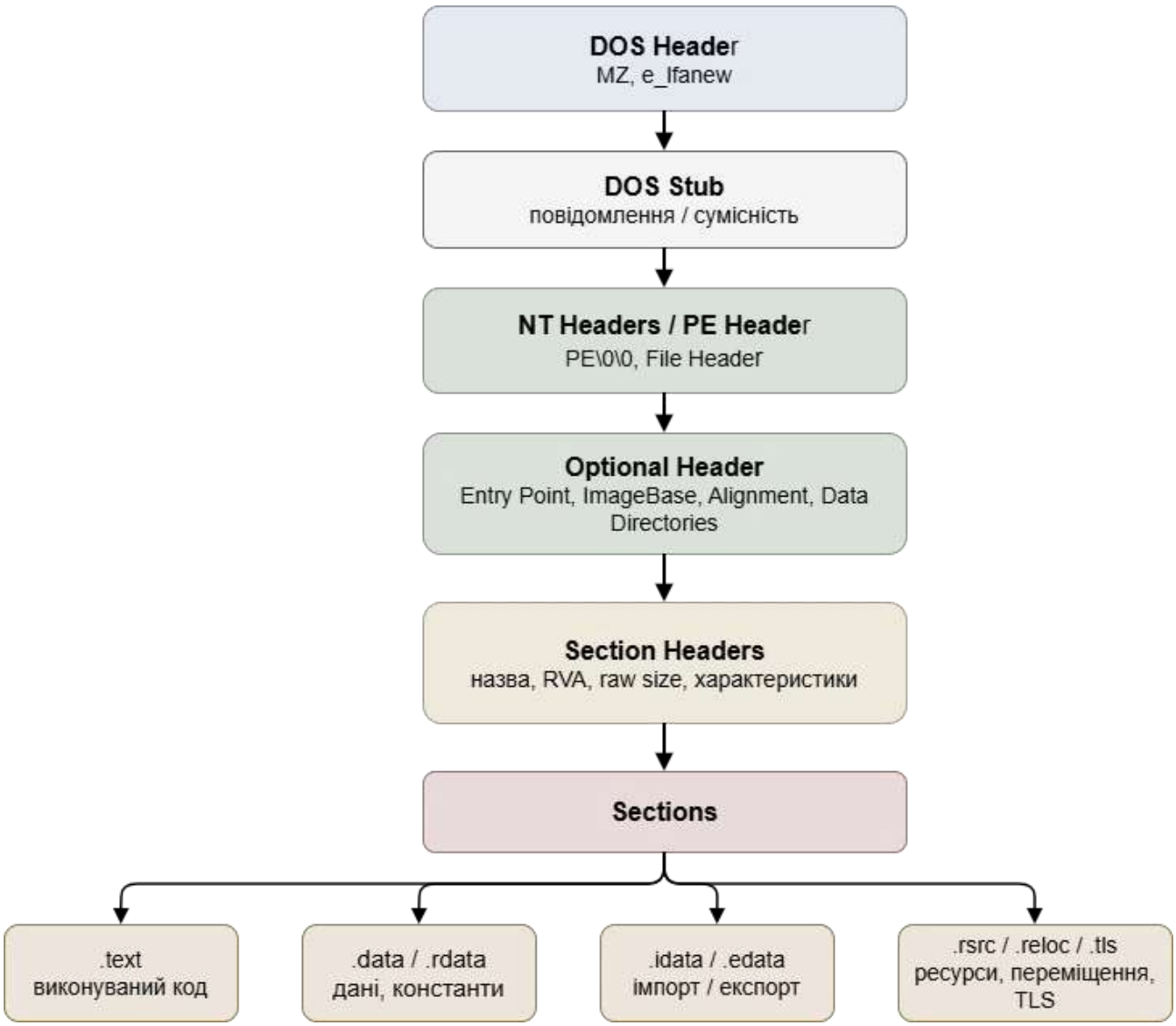


Рисунок 5.2. Узагальнена структура PE-файлу

Наведена структура відображає послідовність інтерпретації виконуваного файлу операційною системою та визначає логіку розміщення його основних компонентів. Це логіка інтерпретації PE-файлу під час первинного та статичного аналізу.

До основних елементів структури PE-файлу належать:

1) DOS-заголовок (IMAGE_DOS_HEADER), що містить службову інформацію та вказівник на PE-заголовок.

Його аналіз дозволяє визначити коректність структури файлу та виявити аномалії, пов’язані з маскуванням або модифікацією.

2) PE-заголовок (IMAGE_NT_HEADERS), який визначає тип файлу, архітектуру, характеристики виконання.

Цей блок є критичним для розуміння того, як операційна система інтерпретує файл.

3) Optional Header, який містить параметри завантаження (ImageBase, Entry Point, Alignment тощо). Особливу увагу приділяють точці входу (Entry Point), яка визначає початок виконання програми і часто використовується як точка впровадження шкідливого коду.

4) Таблиця секцій (Section Headers) Описує розміщення та характеристики секцій (.text, .data, .rdata, .rsrc тощо).

5) Аналіз секцій дозволяє виявити нетипові області пам’яті, приховані сегменти або змінені права доступу (наприклад, виконувані дані).

6) Таблиці імпорту та експорту (Import/Export Tables), що визначають взаємодію програми із зовнішніми бібліотеками. Ці таблиці є ключовими для аналізу залежностей програми та можуть містити ознаки підозрілої поведінки (наприклад, використання API для ін’єкцій або мережевої активності).

У процесі статичного аналізу дослідження структури PE-файлу виконує кілька важливих функцій:

– дозволяє визначити загальну архітектуру програми;

– забезпечує локалізацію критичних компонентів (точка входу, код, ресурси);

– дає змогу виявити ознаки модифікації або компрометації;

– формує основу для подальшого аналізу коду та поведінки.

Особливу увагу під час аналізу слід приділяти таким ознакам, як невідповідність розмірів секцій, нестандартні права доступу (наприклад, одночасно RWX), відсутність або підміна імпортів, зміщення точки входу у нетипові області й наявність додаткових або прихованих секцій.

Такі ознаки можуть свідчити про використання механізмів обфускації, впровадження програмних закладок або спроби приховування шкідливого функціоналу.

У системах цивільного захисту аналіз структури виконуваних файлів має особливе значення, оскільки дозволяє:

– контролювати цілісність критичного програмного забезпечення без його запуску;

– виявляти несанкціоновані модифікації виконуваних модулів;

– оцінювати безпечність оновлень програмних компонентів.

Аналіз структури PE-файлів виступає базовим елементом статичного аналізу, який забезпечує перехід від формального опису файлу до глибокого розуміння його внутрішньої організації та потенційних загроз.

У наступному підрозділі розглянемо структуру виконуваного файлу формату PE з позиції практичного застосування методів статичного аналізу. Метою є не тільки пояснення структури, а й демонстрація послідовності дій аналітика та інтерпретації отриманих результатів. Аналіз виконується із використанням hex-редактора та інструментів перегляду структури PE.

5.3.1. DOS Header

DOS Header є початковою структурою PE-файлу і розміщується з першого байта виконуваного модуля. Саме з нього починається інтерпретація файлу операційною системою та інструментами аналізу.

Основною ознакою DOS Header є сигнатура MZ, що відповідає шістнадцятковому значенню 4D 5A. Її наявність у перших двох байтах файлу дозволяє встановити, що об’єкт належить до виконуваних файлів Windows [42]. На рисунку 5.3 представлено початкову частину виконуваного файлу у шістнадцятковому вигляді.

Визначення сигнатури MZ є **Першим** етапом перевірки формату файлу. Наявність сигнатури MZ підтверджує, що файл є виконуваним. Відсутність або спотворення цієї сигнатури є ознакою пошкодження або модифікації.

У перших байтах ідентифікується сигнатура MZ (4D 5A), а також фрагмент DOS Stub із характерним повідомленням “This program cannot be run in DOS mode”.

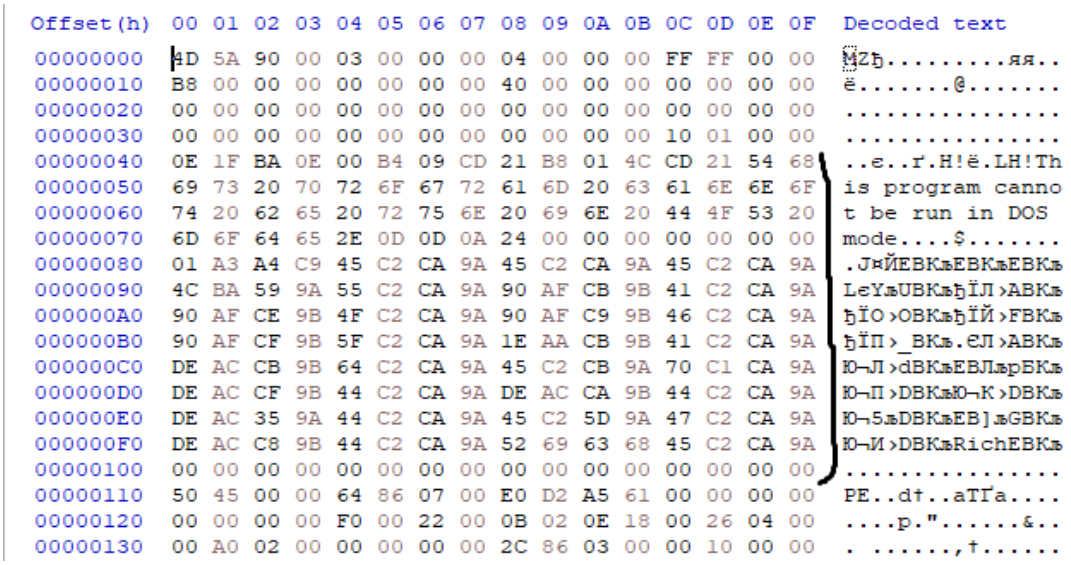


Рисунок 5.3. Фрагмент DOS Header виконуваного файлу у hex-представленні

У межах цього фрагмента можна спостерігати службові значення, що визначають структуру файлу, зокрема зміщення до основного PE-заголовка [42, 26]. Формальний опис відповідних полів наведено в таблиці 5.1.

Таблиця 5.1

Основні поля DOS Header

Поле	**Призначення**
e_magic	містить сигнатуру MZ і підтверджує тип виконуваного файлу
e_lfanew	вказує зміщення до PE-заголовка
e_csum	містить контрольну суму, якщо вона використовується
e_lfarlc	вказує зміщення таблиці переміщень
e_res, e_res2	зарезервовані поля, які зазвичай мають нульові значення

На рисунку 5.4 зображено інформацію DOS header та рядок початку блоку даних, з програми – аналізатора шістнадцяткового коду. Це структурований перелік полів із зазначенням їх зміщення та значень. На відміну від hex-подання, такий формат дозволяє безпосередньо інтерпретувати службові поля заголовка, зокрема ідентифікувати сигнатуру файлу (e_magic) та визначити зміщення до PE-заголовка (e_lfanew).

```
DOS HEADER:
------------------------------------------------------------
Line in Hex Editor              1
Offset це відступ від початку файлу.
e_magic         -       Offset - 00000000       HEX value - 0x5A4D      Size - 2
e_cblp          -       Offset - 00000002       HEX value - 0x0090      Size - 2
e_cp            -       Offset - 00000004       HEX value - 0x0003      Size - 2
e_crlc          -       Offset - 00000006       HEX value - 0x0000      Size - 2
e_cparhdr       -       Offset - 00000008       HEX value - 0x0040      Size - 2
e_minalloc      -       Offset - 0000000A       HEX value - 0x0000      Size - 2
e_maxalloc      -       Offset - 0000000C       HEX value - 0xFFFF      Size - 2
e_ss            -       Offset - 0000000E       HEX value - 0x0000      Size - 2
e_sp            -       Offset - 00000010       HEX value - 0x00B8      Size - 2
e_csum          -       Offset - 00000012       HEX value - 0x0000      Size - 2
e_ip            -       Offset - 00000014       HEX value - 0x0000      Size - 2
e_cs            -       Offset - 00000016       HEX value - 0x0000      Size - 2
e_lfarlc        -       Offset - 00000018       HEX value - 0x0040      Size - 2
e_ovno          -       Offset - 0000001A       HEX value - 0x0000      Size - 2
e_res:
e_res[0]        -       Offset - 0000001C       HEX value - 0x0000      Size - 2
e_res[1]        -       Offset - 0000001E       HEX value - 0x0000      Size - 2
e_res[2]        -       Offset - 00000020       HEX value - 0x0000      Size - 2
e_res[3]        -       Offset - 00000022       HEX value - 0x0000      Size - 2
e_oemid         -       Offset - 00000024       HEX value - 0x0000      Size - 2
e_oeminfo       -       Offset - 00000026       HEX value - 0x0000      Size - 2
e_res2:
e_res2[0]       -       Offset - 00000028       HEX value - 0x0000      Size - 2
e_res2[1]       -       Offset - 0000002A       HEX value - 0x0000      Size - 2
e_res2[2]       -       Offset - 0000002C       HEX value - 0x0000      Size - 2
e_res2[3]       -       Offset - 0000002E       HEX value - 0x0000      Size - 2
e_res2[4]       -       Offset - 00000030       HEX value - 0x0000      Size - 2
e_res2[5]       -       Offset - 00000032       HEX value - 0x0000      Size - 2
e_res2[6]       -       Offset - 00000034       HEX value - 0x0000      Size - 2
e_res2[7]       -       Offset - 00000036       HEX value - 0x0000      Size - 2
e_res2[8]       -       Offset - 00000038       HEX value - 0x0000      Size - 2
e_res2[9]       -       Offset - 0000003A       HEX value - 0x0000      Size - 2
e_lfanew        -       Offset - 0000003C       HEX value - 0x0110      Size - 4
```

Рисунок 5.4. Інформація DOS header з інтерпритацією полів

Такий спосіб подання значно спрощує аналіз структури виконуваного файлу та дозволяє швидко виявляти аномалії у значеннях полів без необхідності ручного декодування байтів.

Під час статичного аналізу DOS Header використовується не стільки для детального вивчення кожного поля, скільки для перевірки цілісності та коректності структури виконуваного файлу. Найбільше аналітичне значення мають поля e_magic та e_lfanew, які визначають формат файлу та логіку переходу до основного заголовка. Якщо значення e_lfanew є нетиповим, пошкодженим або вказує на неочікувану область файлу, це може свідчити про модифікацію заголовка, пакування або спробу приховування структури виконуваного модуля.

У DOS Header також може міститися DOS Stub – невеликий фрагмент коду, який зазвичай виводить повідомлення типу “This program cannot be run in DOS mode”. У звичайних PE-файлах цей блок виконує службову роль, однак у модифікованих або захищених файлах він може використовуватися для розміщення додаткових даних або фрагментів коду.

З позиції аналізу безпеки DOS Header виступає першим рівнем перевірки цілісності та коректності структури виконуваного файлу. Саме на цьому етапі можуть бути виявлені базові аномалії, що свідчать про втручання у файл або його нетипову побудову. До таких ознак належать:

– відсутність або спотворення сигнатури MZ;

– некоректне значення e_lfanew (вихід за межі файлу або вказівка на неструктуровану область);

– наявність нетипових або надлишкових даних у DOS Stub;

– ненульові значення у зарезервованих полях без обґрунтованої причини.

Виявлення таких аномалій на початковому етапі дозволяє своєчасно ідентифікувати потенційно модифіковані або скомпрометовані файли та визначити доцільність подальшого поглибленого аналізу.

DOS Header є першим орієнтиром під час статичного аналізу PE-файлу. Перевірка сигнатури MZ, значення e_lfanew та вмісту DOS Stub дозволяє швидко виявити ознаки пошкодження, нестандартної структури або можливої модифікації виконуваного файлу.

Аналіз DOS Header не обмежується ідентифікацією формату файлу, а виступає початковим етапом виявлення аномалій, що можуть бути пов’язані з обфускацією, пакуванням або впровадженням прихованого функціоналу.

Далі перевіряється наявність сигнатури PE\0\0. Виявлення цієї сигнатури дозволяє встановити початок основного заголовка та підтвердити коректність базової структури файлу.

5.3.2. PE Header (NT Headers)

PE Header (NT Headers) є центральним елементом структури виконуваного файлу формату PE і визначає спосіб його інтерпретації операційною системою [42,26]. Перехід до цього заголовка здійснюється за значенням поля e_lfanew з DOS Header, що задає зміщення до початку PE-структури.

PE Header починається із сигнатури PE\0\0 (шістнадцяткове значення 50 45 00 00), яка підтверджує належність файлу до формату Portable Executable (рис.5.5).

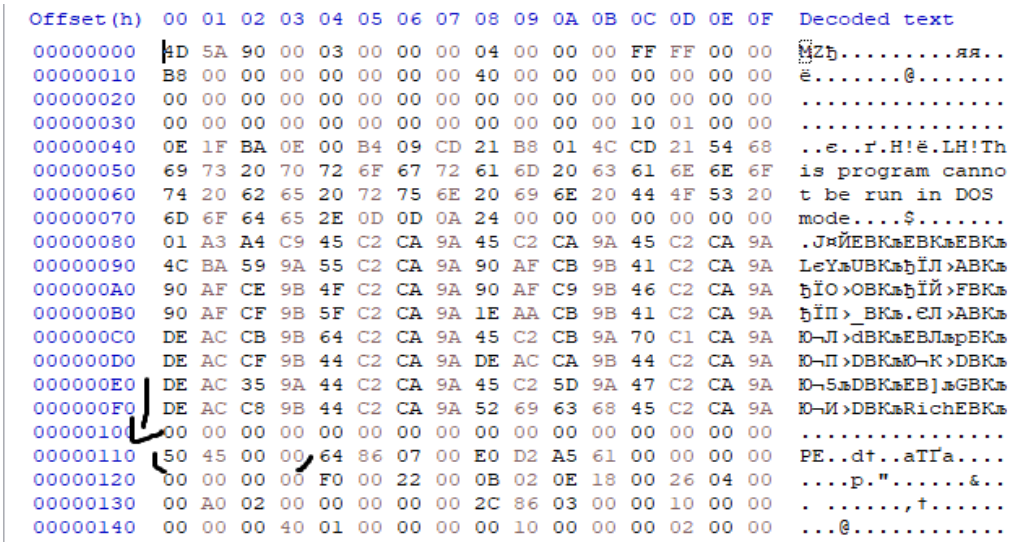


Рисунок 5.5. Фрагмент виконуваного файлу у hex-представленні з виділенням сигнатури PE Header

На рисунку представлено фрагмент виконуваного файлу у шістнадцятковому вигляді, в якому можна ідентифікувати сигнатуру PE\0\0 (50 45 00 00), що позначає початок PE Header.

Структурований вигляд сигнатури PE Header відображено на рис. 5.6.

```
NT HEADER:
-----------------------------------------------------------------
Offset це відступ від початку файлу
Line in Hex Editor      -       18
Signature       -       Offset - 00000110       HEX value - 0x00004550
Блок пам'яті :
50 45 00 00
```

Рисунок 5.6. Визначення ігнатури PE Header у структурованому вигляді

На рисунку наведено інтерпретацію сигнатури PE Header із зазначенням її зміщення у файлі. Такий вигляд дозволяє безпосередньо встановити позицію початку PE-структури без необхідності ручного аналізу байтів.

Структурно PE Header складається з двох основних частин [42]:

– File Header (IMAGE_FILE_HEADER);
– Optional Header (IMAGE_OPTIONAL_HEADER).

Аналіз параметрів заголовка є **Другим** етапом перевірки формату файлу.
На цьому етапі досліджуються ключові параметри File Header та Optional Header.

Перевіряється:

– відповідність архітектури (Machine);
– узгодженість кількості секцій;
– коректність значення точки входу (AddressOfEntryPoint);
– відповідність параметрів формату (наприклад, Magic).

Аномаліями можуть вважатися нетипові значення або суперечності між параметрами, наприклад невідповідність архітектури або зміщення точки входу у нетипову область

Перша частина – блок File Header (IMAGE_FILE_HEADER) містить загальні характеристики виконуваного файлу та параметри, необхідні для його завантаження. До ключових полів File Header належать (таблиця 5.2):

Таблиця 5.2

Ключові поля File Header

Поле	**Призначення**
Machine	визначає архітектуру процесора (наприклад, x86, x64)
NumberOfSections	кількість секцій у файлі
TimeDateStamp	час компіляції або модифікації
SizeOfOptionalHeader	розмір Optional Header
Characteristics	набір бітових ознак (тип файлу, можливість виконання, DLL тощо)

На рисунку 5.7 представлено фрагмент File Header виконуваного файлу у вигляді, наближеному до відображення в hex-редакторі, із розшифруванням основних полів та їх значень.

```
FILE HEADER:
Line in Hex Editor       -      18
Offset це відступ від початку файлу.
Machine                  -     Offset - 00000114    HEX value - 0x8664      Size - 2
NumberOfSections         -     Offset - 00000116    HEX value - 0x0007      Size - 2
TimeDateStamp            -     Offset - 00000118    HEX value - 0x61A5D2E0  Size - 4
PointerToSymbolTable     -     Offset - 0000011C    HEX value - 0x0000      Size - 4
NumberOfSymbols          -     Offset - 00000120    HEX value - 0x0000      Size - 4
SizeOfOptionalHeader     -     Offset - 00000124    HEX value - 0x00F0      Size - 2
Characteristics          -     Offset - 00000126    HEX value - 0x0022      Size - 2
```

Рисунок 5.7. File Header виконуваного файлу у структурованому вигляді)

File Header розташовується безпосередньо після сигнатури PE\0\0 та містить ключові параметри виконуваного файлу, зокрема архітектуру процесора, кількість секцій та службові характеристики. У наведеному прикладі початок File Header відповідає зміщенню 0x00000114, що узгоджується зі значенням, визначеним у DOS Header (поле e_lfanew).

Особливу увагу під час аналізу слід приділяти полю Machine, яке визначає архітектуру виконання:

0x8664 – 64-розрядна архітектура (x64);

0x014c – 32-розрядна архітектура (x86).

Також важливими є поля NumberOfSections та Characteristics, які дозволяють оцінити структуру виконуваного файлу та його призначення.

У контексті статичного аналізу File Header використовується не лише для ідентифікації параметрів файлу, але й для виявлення нетипових або суперечливих значень, що можуть свідчити про втручання у структуру виконуваного модуля.

Зокрема, аналіз поля TimeDateStamp дозволяє виявляти такі аномалії:

– нереалістичні значення (наприклад, 1970 рік або дата в майбутньому);
– однакові часові мітки для різних файлів (ознака масового пакування);
– штучно змінені значення з метою маскування походження файлу.

Поле NumberOfSections також має аналітичне значення: невідповідність між заявленою та фактичною кількістю секцій може вказувати на пошкодження структури або приховані модифікації.

Друга частина – блок Optional Header (IMAGE_OPTIONAL_HEADER) є обов’язковим для виконуваних файлів і містить ключові параметри завантаження та виконання програми. До найбільш важливих полів Optional Header належать відображені у таблиці 5.3.

Таблиця 5.3

Базові поля Optional Header

Поле	**Призначення**
AddressOfEntryPoint	точка входу в програму
ImageBase	базова адреса завантаження в пам’яті
SectionAlignment	вирівнювання секцій у пам’яті
FileAlignment	вирівнювання секцій у файлі
SizeOfImage	загальний розмір образу в пам’яті
Subsystem	тип підсистеми (GUI, Console тощо)
DataDirectory	масив вказівників на службові структури

На відміну від File Header, Optional Header безпосередньо визначає поведінку програми під час виконання, що робить його критично важливим об’єктом аналізу.

Особливу увагу слід приділяти таким полям:

– AddressOfEntryPoint – зміщення точки входу у нетипову секцію або область файлу може свідчити про пакування або впровадження шкідливого коду;

– ImageBase – нестандартні значення можуть використовуватися для ускладнення аналізу або обходу механізмів захисту;

– SectionAlignment і FileAlignment – їх некоректне співвідношення може бути ознакою модифікації або нестандартного формування файлу;

– SizeOfImage – невідповідність розміру реальному вмісту може вказувати на приховані області або помилки структури.

Поле DataDirectory є масивом структур, кожна з яких вказує на важливі службові області файлу:

– Import Table;
– Export Table;
– Resource Table;
– Base Relocation Table;
– TLS Table;
– інші.

На рисунку 5.8 наведено представлення Optional Header виконуваного файлу у вигляді, наближеному до відображення в hex-редакторі, із розшифруванням основних полів та їх значень.

```
OPTIONAL HEADER:
Line in Hex Editor        -     19
Offset це відступ від початку файлу

Magic                          -     Offset -  00000128      HEX value -  0x020B      Size -  2
MajorLinkerVersion             -     Offset -  0000012A      HEX value -  0x000E      Size -  1
MinorLinkerVersion             -     Offset -  0000012B      HEX value -  0x0018      Size -  1
SizeOfCode                     -     Offset -  0000012C      HEX value -  0x42600     Size -  4
SizeOfInitializedData          -     Offset -  00000130      HEX value -  0x2A000     Size -  4
SizeOfUninitializedData        -     Offset -  00000134      HEX value -  0x00000     Size -  4
AddressOfEntryPoint            -     Offset -  00000138      HEX value -  0x3862C     Size -  4
BaseOfCode                     -     Offset -  0000013C      HEX value -  0x1000      Size -  4
ImageBase                      -     Offset -  00000140      HEX value -  0x140000000 Size -  8
SectionAlignment               -     Offset -  00000148      HEX value -  0x1000      Size -  4
FileAlignment                  -     Offset -  0000014C      HEX value -  0x0200      Size -  4
MajorOperatingSytemVersion     -     Offset -  00000150      HEX value -  0x0006      Size -  2
MinorOperatingSystemVersion    -     Offset -  00000152      HEX value -  0x0000      Size -  2
MajorImageVersion              -     Offset -  00000154      HEX value -  0x0000      Size -  2
MinorImageVersion              -     Offset -  00000156      HEX value -  0x0000      Size -  2
MajorSubsystemVersion          -     Offset -  00000158      HEX value -  0x0006      Size -  2
MinorSubsystemVersion          -     Offset -  0000015A      HEX value -  0x0000      Size -  2
Win32VersionValue              -     Offset -  0000015C      HEX value -  0x0000      Size -  4
SizeOfImage                    -     Offset -  00000160      HEX value -  0x72000     Size -  4
SizeOfHeaders                  -     Offset -  00000164      HEX value -  0x0400      Size -  4
CheckSum                       -     Offset -  00000168      HEX value -  0x7F082     Size -  4
Subsystem                      -     Offset -  0000016C      HEX value -  0x0002      Size -  2
DllCharacteristics             -     Offset -  0000016E      HEX value -  0x8160      Size -  2
SizeOfStackReserve             -     Offset -  00000170      HEX value -  0x871B000   Size -  8
SizeOfStackCommit              -     Offset -  00000178      HEX value -  0x1000      Size -  8
SizeOfHeapReserve              -     Offset -  00000188      HEX value -  0x100000    Size -  8
SizeOfHeapCommit               -     Offset -  00000188      HEX value -  0x1000      Size -  8
LoaderFlags                    -     Offset -  00000190      HEX value -  0x0000      Size -  4
NumberOfRvaAndSizes            -     Offset -  00000194      HEX value -  0x0010      Size -  4
```

Рисунок 5.8. Optional Header виконуваного файлу у структурованому вигляді

Optional Header містить ключові параметри, що визначають поведінку програми під час виконання, зокрема точку входу (AddressOfEntryPoint), базову адресу завантаження (ImageBase), параметри вирівнювання (SectionAlignment, FileAlignment) та загальний розмір образу (SizeOfImage).

У наведеному прикладі значення поля Magic (0x20B) вказує на використання 64-розрядного формату PE (PE32+), що узгоджується з параметрами архітектури, визначеними у File Header.

Аналіз значень полів Optional Header дозволяє оцінити коректність структури виконуваного файлу та виявити можливі аномалії, пов’язані з модифікацією, пакуванням або впровадженням прихованого функціоналу.

Аналіз Data Directory дозволяє визначити, які системні функції використовує програма, які бібліотеки підключає та які механізми взаємодії з операційною системою реалізовано.

Аномалії у Data Directory можуть проявлятися у вигляді:

– відсутності очікуваних записів (наприклад, Import Table);
– вказівників на неіснуючі або перекриті області файлу;
– невідповідності розмірів структур їх фактичному вмісту.

Під час статичного аналізу можуть бути виявлені такі аномалії та підозрілої структури:

– невідповідність між NumberOfSections і реальною кількістю секцій;
– некоректні або нульові значення SizeOfOptionalHeader;
– зміщення Entry Point у секцію з нетиповими правами доступу;
– відсутність або підміна записів у DataDirectory;
– суперечливі значення вирівнювання (Alignment).

Виявлення таких ознак є підставою для подальшого поглибленого аналізу, зокрема дослідження впливу змін окремих полів заголовка на поведінку програми.

PE Header визначає логіку завантаження і виконання програми та є ключовим джерелом інформації під час статичного аналізу. Його дослідження дозволяє не лише встановити параметри виконуваного файлу, але й виявити аномалії, що можуть свідчити про модифікацію, обфускацію або впровадження прихованого функціоналу. Саме ці властивості роблять PE Header одним із основних об’єктів подальшого поглибленого аналізу та експериментального дослідження.

5.3.3. Section Headers

Після PE Header у структурі виконуваного файлу розміщується таблиця секцій (Section Headers), яка описує розташування, розмір і характеристики кожної секції [42,26]. Саме ця таблиця визначає, як логічні компоненти програми (код, дані, ресурси) відображаються у файлі та в оперативній пам’яті під час виконання.

Аналіз структури секцій є третім етапом аналізу формату файла. аналізується таблиця секцій.

Оцінюється [42]:

– наявність стандартних секцій (.text, .data, .rdata, .rsrc);

– відповідність прав доступу (відсутність одночасних прав запису і виконання);

– узгодженість розмірів та адрес.

Відхилення від типової структури (наприклад, незвичні назви секцій або некоректні атрибути доступу) можуть свідчити про обфускацію або модифікацію файлу.

Кожен запис таблиці секцій відповідає окремій секції і має фіксовану структуру (IMAGE_SECTION_HEADER). Кількість записів визначається полем NumberOfSections у File Header. Основні поля Section Header (Табл.5.4).

Таблиця 5.4

Основні поля Section Header

Поле	**Призначення**
Name	ім’я секції (наприклад, .text, .data, .rdata)
VirtualAddress (RVA)	адреса секції в пам’яті
SizeOfRawData	розмір секції у файлі
PointerToRawData	зміщення секції у файлі
Characteristics	атрибути доступу (читання, запис, виконання)

З позиції статичного аналізу таблиця секцій виконує роль карти розміщення коду та даних у файлі, що дозволяє локалізувати критичні області для подальшого дослідження. Аналіз параметрів секцій дає змогу визначити, які

частини файлу відповідають за виконуваний код, зберігання даних або взаємодію із зовнішніми компонентами.

У стандартних виконуваних файлах зазвичай присутні такі секції:

.text – містить виконуваний машинний код програми;

.data – містить ініціалізовані змінні;

.rdata – константи, рядки, таблиці;

.idata – таблиця імпорту;

.edata – таблиця експорту (для DLL);

.rsrc – ресурси (іконки, меню, діалоги);

.reloc – інформація для релокації.

Наявність, кількість і структура цих секцій дозволяють оцінити тип програми та її складність.

На рисунку 5.9 наведено представлення таблиці секцій виконуваного файлу із зазначенням їх основних параметрів. Кожна секція (зокрема .text, .rdata, .data) описується набором характеристик, що визначають її розташування у файлі та пам’яті, розмір і права доступу.

```
SECTION HEADERS:
-----------------------------------------------------------------
.text
          0x403FE                 VirtualSize
          0x1000                  VirtualAddress
          0x40400                 SizeOfRaw Data
          0x 400                  PointerToRawData
          0x0                     PointerToRelocations
          0x0                     PointerToLineNumbers
          0x0                     NumberOfRelocations
          0x0                     NumberOfLineNumbers
          0x60000020              Characteristics
.uedbg
          0x205E                  VirtualSize
          0x42000                 VirtualAddress
          0x2200                  SizeOfRaw Data
          0x 40800                PointerToRawData
          0x0                     PointerToRelocations
          0x0                     PointerToLineNumbers
          0x0                     NumberOfRelocations
          0x0                     NumberOfLineNumbers
          0x60000020              Characteristics
.rdata
          0x1F1C0                 VirtualSize
          0x45000                 VirtualAddress
          0x1F200                 SizeOfRaw Data
          0x 42A00                PointerToRawData
          0x0                     PointerToRelocations
          0x0                     PointerToLineNumbers
          0x0                     NumberOfRelocations
          0x0                     NumberOfLineNumbers
          0x40000040              Characteristics
```

Рисунок 5.9. Таблиця секцій виконуваного файлу у структурованому вигляді

Аналіз таких параметрів, як VirtualAddress, SizeOfRawData та Characteristics, дозволяє встановити відповідність структури файлу стандарту PE, а також виявити аномалії, зокрема наявність виконуваного коду у нетипових секціях або некоректні атрибути доступу.

На рисунку 5.10 представлено таблицю секцій виконуваного файлу у шістнадцятковому вигляді. У правій частині відображення можна ідентифікувати імена секцій у ASCII-поданні (зокрема .text, .rdata, .data, .rsrc), що дозволяє співставити структурні елементи PE-файлу з їх реальним розміщенням у бінарному представленні.

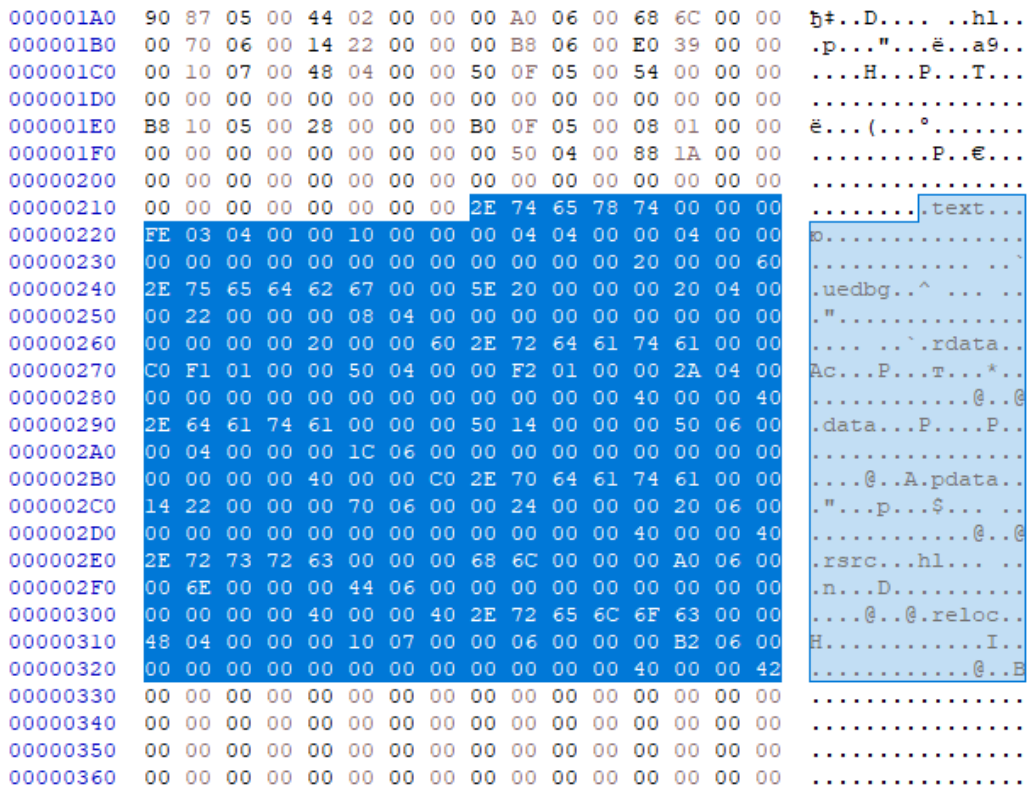


Рисунок 5.10. Таблиця секцій PE-файлу у hex-представленні

Такий підхід дає змогу перевірити коректність інтерпретації даних, а також виявити можливі аномалії, наприклад спотворені імена секцій або приховані записи, які не відображаються у структурному представленні.

Під час статичного аналізу особливу увагу приділяють:

– відповідності між SizeOfRawData і VirtualSize (значні розбіжності можуть свідчити про приховані дані або пакування);

– PointerToRawData (дозволяє точно локалізувати секцію у файлі для подальшого аналізу);

– характеристикам доступу (Characteristics) (наприклад, поєднання прав Read/Write/Execute (RWX) є нетиповим і потенційно небезпечним);

– іменам секцій (нестандартні або випадкові імена можуть вказувати на обфускацію або модифікацію).

У розглянутому прикладі структура файлу відповідає стандарту PE: сигнатури коректні, параметри заголовків узгоджені, а секції мають типовий вигляд. Явних аномалій не виявлено, але відсутність аномалій у структурі не гарантує безпечності файлу. Шкідливий функціонал може бути реалізований без порушення формату PE.

Для виявлення прихованих механізмів необхідно дослідити аномалії та вплив змін параметрів заголовка, що розглядається у наступних підрозділах.

Під час дослідження структури виконуваного файлу можуть бути виявлені такі аномалії:

– -наявність секцій із випадковими або нетиповими іменами (наприклад, .abc, .xyz1);

– невідповідність між VirtualSize і SizeOfRawData (ознака пакування або прихованих даних);

– наявність виконуваного коду поза секцією .text, або відсутність стандартних секцій (.text, .data);

– секції з одночасними правами запису та виконання (RWX);

– надмірна кількість секцій або їх незвичне розташування, перекриття адрес секцій або невідповідність їхніх розмірів.

Такі ознаки часто свідчать про використання пакувальників, обфускацію або впровадження прихованого функціоналу.

Аналіз таблиці секцій є основою для подальшого поглибленого дослідження, зокрема вивчення впливу змін параметрів секцій на поведінку виконуваного файлу, а також можливостей впровадження або приховування коду.

У статичному аналізі Section Headers виступає ключовим інструментом для визначення структури програми, локалізації виконуваного коду та вибору ділянок для подальшого дослідження (зокрема дизасемблювання). Саме на основі аналізу цієї таблиці формується уявлення про розміщення та взаємодію компонентів програми в пам’яті.

Таблиця секцій визначає логіку розподілу коду та даних у виконуваному файлі, а її дослідження дозволяє виявити аномалії, пов’язані з обфускацією, пакуванням або впровадженням прихованого функціоналу ще до переходу до аналізу інструкцій.

5.4. ПОГЛИБЛЕНИЙ АНАЛІЗ PE-ФАЙЛІВ

5.4.1. Аналіз аномалій у структурі PE-файлів

Після встановлення базової коректності структури виконуваного файлу наступним етапом поглибленого статичного аналізу є виявлення аномалій у його структурі. Аномалії визначаються як відхилення від типових або очікуваних значень параметрів PE-файлу, які можуть свідчити про модифікацію, обфускацію або впровадження прихованого функціоналу [18,19,48].

На відміну від формальної перевірки структури, аналіз аномалій передбачає інтерпретацію значень полів з урахуванням контексту виконання та характеру програмного забезпечення.

Класифікація аномалій PE-файлів

Аномалії можуть проявлятися на різних рівнях структури файлу:

1. Аномалії DOS Header

До типових відхилень належать:

– відсутність або спотворення сигнатури MZ;
– некоректне значення поля, що визначає зміщення до PE-заголовка;
– наявність нетипових або надлишкових даних у DOS Stub;
– ненульові значення у зарезервованих полях.

Такі аномалії можуть свідчити про пошкодження файлу, нестандартне формування або спроби приховування структури.

2. Аномалії PE Header

На рівні PE Header найбільш показовими є:

– нереалістичні або підозрілі значення часових міток;

– невідповідність архітектури параметрам виконання;

– некоректні значення розміру заголовків;

– зміщення точки входу у нетипову область файлу.

Зокрема, розташування точки входу поза виконуваною секцією може вказувати на пакування або впровадження додаткового коду.

3. Аномалії Optional Header

Аналіз параметрів Optional Header дозволяє виявити:

– невідповідність між параметрами вирівнювання (SectionAlignment і FileAlignment);

– некоректні значення загального розміру образу (SizeOfImage);

– відсутність або спотворення службових структур;

– нетипові значення базової адреси завантаження.

Такі ознаки можуть бути пов’язані як із технічними помилками, так і з навмисною модифікацією файлу.

4. Аномалії таблиці секцій

Найбільш інформативними є аномалії, пов’язані зі структурою секцій:

– відсутність стандартних секцій (.text, .data, .rdata);

– наявність секцій із нетиповими або випадковими іменами;

– секції з одночасними правами запису та виконання (RWX);

– невідповідність між VirtualSize і SizeOfRawData;

– перекриття секцій або вихід за межі файлу;

– наявність виконуваного коду поза секцією .text.

Такі відхилення часто використовуються для реалізації обфускації, пакування або приховування шкідливого функціоналу [33,18,19].

5. Аномалії службових структур

Аналіз службових таблиць (Data Directory) дозволяє виявити:

– відсутність імпортів у програмі, що явно використовує системні функції;

– вказівники на неіснуючі або перекриті області пам’яті;

– невідповідність розмірів структур їх фактичному вмісту.

Ці ознаки можуть свідчити про використання нестандартних механізмів взаємодії з операційною системою або спроби приховування залежностей програми.

Важливо підкреслити, що наявність окремої аномалії не завжди є ознакою шкідливого коду. Деякі відхилення можуть бути пов’язані з особливостями компіляції, оптимізації або використанням легітимних засобів захисту.

Однак сукупність декількох аномалій значно підвищує ймовірність модифікації або наявності прихованого функціоналу. Саме тому аналіз повинен виконуватися комплексно, із урахуванням взаємозв’язку між різними параметрами.

Виявлення аномалій дозволяє локалізувати підозрілі області файлу, визначити доцільність подальшого аналізу, сформувати гіпотези щодо механізмів функціонування програми й підготувати основу для експериментального дослідження.

Таким чином, аналіз аномалій є проміжним етапом між структурним дослідженням і активним втручанням у файл. Отримані результати дозволяють визначити, які саме параметри доцільно змінювати для перевірки поведінки програми.

Аналіз аномалій PE-файлу дозволяє виявити приховані ознаки модифікації, які не проявляються при формальній перевірці структури. Саме ці ознаки визначають напрям подальшого поглибленого аналізу, зокрема застосування методів контрольованої модифікації параметрів файлу.

5.4.2. Маніпулювання параметрами PE-файлу як метод аналізу

Маніпулювання параметрами PE-файлу є прийомом поглибленого статичного аналізу, що полягає у контрольованій зміні окремих службових полів виконуваного файлу з подальшою інтерпретацією результатів. Його мета полягає не у модифікації програми як такій, а у з'ясуванні ролі конкретних полів заголовка, секцій або службових таблиць у процесі завантаження та виконання файлу [19,48].

На відміну від простого перегляду структури, маніпулювання передбачає аналітичну послідовність:

ВИБІР ПАРАМЕТРА → КОНТРОЛЬОВАНА ЗМІНА → ФІКСАЦІЯ РЕЗУЛЬТАТУ → ІНТЕРПРЕТАЦІЯ ВПЛИВУ.

До параметрів, які можуть досліджуватися таким способом, належать:

– e_lfanew – зміщення до PE Header;
– AddressOfEntryPoint – точка входу програми;
– NumberOfSections – кількість секцій;
– SizeOfImage – розмір образу в пам'яті;
– SectionAlignment / FileAlignment – параметри вирівнювання;
– Characteristics секцій – атрибути читання, запису та виконання;
– PointerToRawData / SizeOfRawData – розташування та розмір секцій у файлі.

Маніпулювання такими параметрами дозволяє встановити, які поля є критичними для коректного завантаження файлу, які впливають на інтерпретацію секцій, а які можуть бути використані для приховування або впровадження додаткового коду.

Для безпечного виконання такого аналізу необхідно працювати лише з копією файлу в ізольованому середовищі. Будь-які зміни службових полів можуть призвести до пошкодження виконуваного файлу, відмови завантажувача операційної системи або некоректної роботи програми.

Для дослідження впливу модифікації окремих елементів виконуваного файлу на його структуру та поведінку доцільно використовувати контрольовані зміни, приклади яких наведено у таблиці 5.5.

Таблиця 5.5

Приклади контрольованих змін

Параметр	**Що змінюється**	**Можливий результат**	**Аналітичне значення**
e_lfanew	зміщення до PE Header	файл не розпізнається як PE	поле критичне для інтерпретації структури
AddressOf EntryPoint	точка входу	програма не запускається або переходить у нетипову область	дозволяє оцінити роль Entry Point
NumberOf Sections	кількість секцій	частина секцій не інтерпретується або файл пошкоджується	показує залежність структури від таблиці секцій
Characteris tics секції	права доступу до секції	секція може стати виконуваною або недоступною	дозволяє оцінити ризики RWX-областей
SizeOfIma ge	розмір образу	помилка завантаження або некоректна карта пам’яті	визначає межі образу в пам’яті

Маніпулювання параметрами PE-файлу має особливе значення для аналізу механізмів захисту, пакування та впровадження Х-коду [33,18]. Наприклад, зміна точки входу може імітувати ситуацію, коли керування передається не основній програмі, а додатковому захисному або прихованому модулю. Зміна характеристик секції може показати, як наявність прав виконання у секції даних створює потенційну область ризику.

Водночас такі дії не слід ототожнювати з динамічним аналізом. У цьому випадку основним об’єктом дослідження є структура файлу, а не поведінка програми під час виконання. Якщо після зміни параметра дослідник лише фіксує можливість відкриття, розпізнавання або завантаження файлу, це залишається в межах поглибленого статичного аналізу з використанням контрольованих експериментальних прийомів.

У системах цивільного захисту такий підхід може застосовуватися для перевірки стійкості критичних програмних компонентів до несанкціонованої модифікації, аналізу наслідків пошкодження службових полів, а також для оцінювання ефективності механізмів контролю цілісності.

Маніпулювання параметрами PE-файлу дозволяє перейти від пасивного перегляду структури до активної перевірки ролі окремих полів у функціонуванні виконуваного модуля. Такий підхід допомагає виявити критичні елементи структури, потенційні точки втручання та ознаки прихованої модифікації, що є важливим етапом поглибленого статичного аналізу.

5.4.3. Дослідження впливу змін заголовка на поведінку програми

Після визначення принципів маніпулювання параметрами PE-файлу доцільно перейти до дослідження впливу таких змін на поведінку виконуваного модуля. На відміну від попереднього підрозділу, де розглядалися окремі параметри ізольовано, у цьому випадку увага зосереджується на аналізі реакції системи на їх модифікацію.

Такий підхід дозволяє встановити причинно-наслідкові зв’язки між структурою файлу та процесом його завантаження і виконання.

Дослідження виконується за узагальненою схемою [19,48]:

ВИБІР ПАРАМЕТРА → ВНЕСЕННЯ КОНТРОЛЬОВАНОЇ ЗМІНИ → ЗАПУСК АБО ВІДКРИТТЯ ФАЙЛУ → ФІКСАЦІЯ РЕЗУЛЬТАТУ → ІНТЕРПРЕТАЦІЯ.

Важливою умовою є ізольоване середовище виконання (віртуальна машина або sandbox), що забезпечує безпечність експериментів та виключає вплив на робочу систему.

Нижче наведено приклади змін параметрів заголовка та їх впливу на поведінку програми (таблиця 5.6).

Таблиця 5.6

Результати дослідження впливу змін параметрів PE-файлу

Параметр	**Зміна**	**Реакція системи**	**Інтерпретація**
e_lfanew	зміщення на довільне значення	файл не розпізнається як виконуваний	порушується перехід до PE Header
AddressOfEntryPoint	вказівка на неіснуючу адресу	помилка запуску або аварійне завершення	точка входу є критичним параметром виконання
NumberOfSections	зменшення значення	частина секцій ігнорується	змінюється інтерпретація структури
Characteristics секції	додавання права виконання	секція може виконуватися	потенційна зона ризику (RWX)
SizeOfImage	зменшення значення	помилка завантаження	визначає межі образу в пам’яті

Результати експериментів показують, що різні поля заголовка мають різний рівень критичності. Частина параметрів (наприклад, e_lfanew або AddressOfEntryPoint) є визначальними для можливості виконання файлу, тоді як інші впливають на спосіб інтерпретації структури або організацію пам’яті.

Особливий інтерес становлять зміни, які не призводять до негайного порушення виконання, але змінюють логіку роботи програми. Саме такі ситуації можуть використовуватися для прихованого впровадження додаткового функціоналу або обходу механізмів захисту.

Дослідження впливу змін заголовка дозволяє:

– визначити критичні параметри структури PE-файлу;
– оцінити стійкість програм до пошкодження або модифікації;
– виявити потенційні точки втручання;
– сформувати гіпотези щодо механізмів захисту або обфускації.

У системах цивільного захисту такі експерименти можуть застосовуватися для перевірки цілісності програмних компонентів, оцінювання наслідків несанкціонованих змін та підготовки до реагування на інциденти.

Зокрема, контроль параметрів виконуваних файлів дозволяє виявляти відхилення від еталонних значень, що є важливим елементом систем моніторингу та забезпечення безперервності функціонування.

Але слід звернути увагу, що отримані результати залежать від середовища виконання, версії операційної системи та механізмів захисту. Крім того, не всі зміни призводять до однозначно інтерпретованих наслідків, що ускладнює аналіз.

Поглиблений статичний аналіз виконуваних файлів, зокрема формату PE, є важливим етапом дослідження програмного забезпечення, який забезпечує перехід від формального опису структури до розуміння принципів її функціонування. У межах цього розділу було розглянуто структуру PE-файлу, методи виявлення аномалій, підходи до контрольованої модифікації параметрів та інструменти, що підтримують такий аналіз. Особливе значення має здатність виявляти приховані зміни, які не призводять до порушення формату файлу, але можуть змінювати його функціональність.

Але, незважаючи на ефективність, поглиблений статичний аналіз має певні обмеження, а саме:

– не дозволяє безпосередньо дослідити поведінку програми під час виконання;

– залежить від якості інструментів та інтерпретації результатів;

– може бути ускладнений використанням обфускації або пакування.

Тому для отримання повної картини необхідно поєднувати його з іншими методами аналізу, зокрема динамічним.

Поглиблений статичний аналіз займає проміжне місце між первинним аналізом і динамічним дослідженням. Він уточнює результати попередніх етапів і формує основу для подальшого вивчення поведінки програмного забезпечення.

Саме на цьому етапі визначаються ключові параметри та області, які потребують більш детального дослідження, що забезпечує ефективність наступних етапів аналізу.

Поглиблений статичний аналіз PE-файлів є невід'ємною складовою дослідження безпеки програмного забезпечення, що дозволяє виявляти приховані ознаки модифікації, оцінювати цілісність і формувати обґрунтовані висновки щодо надійності програмних систем. Його застосування у підрозділах цивільного захисту підвищує рівень контролю за станом програмного забезпечення та сприяє забезпеченню стабільного функціонування критичних інформаційних систем.

Дослідження впливу змін параметрів заголовка дозволяє перейти від статичного опису структури до розуміння механізмів її інтерпретації операційною системою. Такий підхід є важливим етапом поглибленого аналізу, оскільки дозволяє виявити залежності, які не проявляються при простому перегляді структури файлу.

5.5. ІНСТРУМЕНТАЛЬНІ ЗАСОБИ СТАТИЧНОГО АНАЛІЗУ

Інструментальні засоби статичного аналізу забезпечують дослідження структури виконуваних файлів без їх запуску, дозволяючи аналізувати PE-заголовки, виявляти помилки, вразливості та потенційно небезпечні конструкції. Використання таких засобів підвищує надійність, безпечність і якість програмного забезпечення ще на етапі розроблення або первинного дослідження.

Залежно від призначення, інструменти статичного аналізу можна поділити на кілька груп.

5.5.1. Інструменти аналізу структури PE-файлів

Ця група інструментів призначена для дослідження внутрішньої структури виконуваних файлів формату PE, зокрема заголовків, секцій, таблиць імпорту та експорту [42,43].

До них належать:

– CFF Explorer (www.ntcore.com/exsuite.php) – один із найпоширеніших інструментів для аналізу та редагування PE-заголовків, дозволяє досліджувати

структуру файлу, змінювати параметри секцій та аналізувати імпорт/експорт функцій [43];

– PE Internals (www.andreybazhan.com/pe-internals.html) – інструмент для дослідження внутрішніх структур PE-файлу з акцентом на їх логічну організацію;

– PPEE (puppy) (www.mzrst.com) – компактний інструмент для аналізу заголовків і секцій, зручний для швидкого дослідження;

– PEBrowse Professional (www.smidgeonsoft.prohosting.com/pebrowse-pro-fileviewer.html) – інструмент для перегляду PE-файлів і дослідження їх структури на низькому рівні.

Ці засоби дозволяють ідентифікувати аномалії у структурі файлу, що можуть свідчити про модифікацію або наявність шкідливого коду.

5.5.2. Інструменти поглибленого аналізу PE-файлів

Для більш глибокого дослідження застосовуються інструменти, що поєднують аналіз структури з оцінкою безпеки.

– PEStudio (winitor.com) – інструмент для статичного аналізу виконуваних файлів, який дозволяє виявляти підозрілі імпорти, нестандартні секції, ознаки обфускації та інші ризики;

– PPEE – також може використовуватися для розширеного аналізу;

– Manalyzer (manalyzer.org) – онлайн-платформа для автоматизованого аналізу PE-файлів (рис.5.11).

Manalyzer.org виконує:

– аналіз заголовків і секцій;
– перевірку цифрових підписів;
– обчислення хешів (MD5, SHA-1, SHA-256);
– виявлення підозрілих конструкцій і ознак шкідливого коду;
– формування детального звіту з оцінкою ризиків.



Рисунок 5.11. Приклад звіту системи Manalyzer

Winitor.com – вебінструмент для аналізу і порівняння виконуваних файлів, який дозволяє (Рис.5.12):

– досліджувати структуру PE-файлу;

– виявляти зміни між версіями;

– аналізувати ресурси, підписи та секції;

– ідентифікувати аномалії.

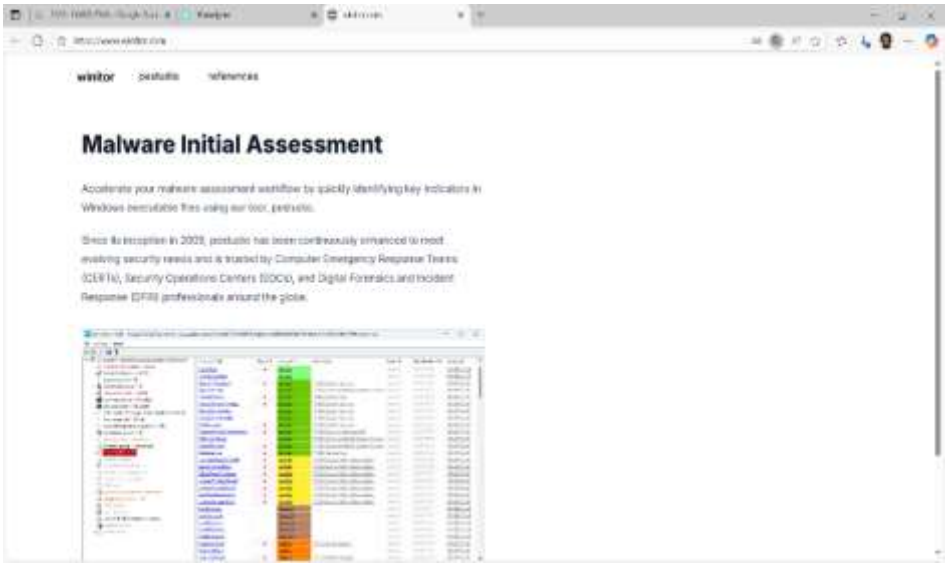


Рисунок 5.12. Аналіз PE-файлу за допомогою Winitor

5.5.3. Дизасемблери

Дизасемблери використовуються для перетворення машинного коду у зрозуміле представлення та аналізу логіки програми [44; 47; 19].

До них належать:

– IDA Pro [47];
– Ghidra [44,45];
– Radare2 / rizin;
– Gutter (GUI);
– RetDec (на базі LLVM);
– JADX;
– OllyDbg (з можливістю дизасемблювання)/

Розрізняють:

– dumb-дизасемблери – виконують пряме перетворення інструкцій;
– smart-дизасемблери – здійснюють аналіз структури програми, визначають функції, переходи та залежності.

Дизасемблери дозволяють перейти від аналізу структури до аналізу поведінки коду на статичному рівні.

5.5.4. Статичні аналізатори коду

Ці інструменти застосовуються для виявлення помилок і вразливостей у вихідному коді [5,31].

– PVS-Studio;
– Cppcheck;
– Coverity;
– Clang Static Analyzer;
– SonarQube/

Для окремих мов програмування застосовуються:

– для Python: Pylint, Pyflakes;
– для Java: FindBugs;
– для .NET: FxCop, Roslyn/

Використання статичних аналізаторів дозволяє виявити логічні помилки, небезпечні конструкції та потенційні вразливості ще до компіляції або виконання програми.

5.5.5. Сигнатурний аналіз

Сигнатурні засоби дозволяють виявляти відомі шаблони шкідливого коду. Найбільш застосовним інструментом для створення правил виявлення шкідливих програм і ознак компрометації є YARA [46,49], (рис.5.13).

YARA (Yet Another Recursive Acronym) – це інструмент зіставлення шаблонів і ідентифікації з відкритим кодом, який в основному використовується для проведення аналізу зловмисного програмного забезпечення та полювання на загрози, а також для створення користувачами спеціальних правил, які допомагають ідентифікувати будь-які потенційно шкідливі файли.



Рисунок 5.13 – Інформаційна заставка пакету

YARA не є традиційним інструментом статичного аналізу коду в тому сенсі, як це роблять, наприклад, компілятори чи інші статичні аналізатори. Однак її можна використовувати для виявлення шкідливого програмного забезпечення, що містить статичні ознаки, що робить її частиною процесу аналізу коду.

Основні особливості YARA:

1. ПРАВИЛА: YARA дозволяє користувачам визначати правила для виявлення шкідливих файлів на основі специфічних ознак (наприклад, рядків тексту, структур даних, метаданих). Кожне правило складається з умов, що вказують, які патерни потрібно шукати.

2. ВИКОРИСТАННЯ: YARA широко використовується в кібербезпеці для аналізу шкідливих програм, ідентифікації загроз і побудови індикаторів компрометації (IoC). Інструмент може працювати з файлами на диску, в пам'яті, а також інтегруватися з іншими аналітичними системами [46].

3. ГНУЧКІСТЬ: YARA підтримує розширений синтаксис, включаючи регулярні вирази, математичні операції, логічні оператори та інші конструкції, що дозволяє створювати дуже точні правила.

4. КРОСПЛАТФОРМНІСТЬ: YARA доступна на різних платформах, таких як Windows, Linux та macOS.

YARA стала важливим інструментом для фахівців з кібербезпеки, аналітиків шкідливого ПЗ та дослідників у цій галузі, оскільки вона допомагає в автоматизації процесів виявлення шкідливого програмного забезпечення та аналізу загроз й широко застосовується у кібербезпеці для ідентифікації підозрілих файлів за характерними ознаками.

5.5.6. Довідкові ресурси

Для глибшого розуміння структури PE-файлів доцільно використовувати спеціалізовані ресурси:

– детальний опис формату PE:

www.delphibasics.info/...part1

www.delphibasics.info/...part2

– специфікація формату:

www.openrce.org/.../PE%20Format.pdf

– навчальний матеріал:

https://github.com/corkami/pics/.../pe101.pdf

Ці ресурси дозволяють отримати детальне уявлення про структуру виконуваних файлів і підтримують практичне застосування методів статичного аналізу.

Інструментальні засоби статичного аналізу охоплюють широкий спектр задач – від дослідження структури виконуваних файлів до аналізу коду та виявлення вразливостей. Їх комплексне використання дозволяє отримати повне уявлення про програмну реалізацію без її запуску.

Ефективність статичного аналізу визначається правильним вибором і комбінуванням інструментів. Використання декількох засобів одночасно дозволяє підвищити точність аналізу та своєчасно виявити потенційні загрози.

5.6. ПРИКЛАД ЗАСТОСУВАННЯ СТАТИЧНОГО МЕТОДУ

Для закріплення розглянутих методів статичного аналізу доцільно розглянути практичний приклад дослідження виконуваного файлу із використанням дизасемблера.

Одним із найбільш поширених інструментів такого типу є дизасемблер IDA Pro, який широко застосовується для реверс-інжинірингу програмного забезпечення й дозволяє досліджувати машинний код і структуру програм без їх виконання [47,19]. Інструмент поєднує автоматичний аналіз із можливістю інтерактивного уточнення результатів дослідником.

IDA Pro (Interactive DisAssembler) – це інтерактивний дизасемблер, що підтримує велику кількість форматів виконуваних файлів і архітектур процесорів. Інструмент дозволяє [47]:

- виконувати дизасемблювання машинного коду;
- будувати графи викликів функцій;
- аналізувати структуру програми;
- здійснювати перейменування змінних і функцій;
- працювати з перехресними посиланнями;
- використовувати скрипти та плагіни.

На початковому етапі IDA виконує автоматичний аналіз, після чого користувач уточнює результати, додаючи семантичну інформацію (імена, коментарі, структури даних).

Процес дослідження програми у дизасемблері включає шість основних етапів.

1. Завантаження файлу (File → Open);
2. Виконання автоматичного дизасемблювання;
3. Первинний перегляд структури програми;
4. Аналіз функцій і переходів;
5. Ідентифікація ключових ділянок коду;
6. Уточнення логіки програми шляхом коментування.

Під час статичного аналізу виникає ряд характерних проблем:

1. Відновлення символічних імен (у машинному коді відсутня інформація про імена змінних і функцій, тому дизасемблер генерує умовні позначення (наприклад, var1, func2).

2. Розрізнення коду та даних (машинний код не містить явних ознак розділення інструкцій і даних, що може призводити до помилок інтерпретації.

3. Визначення меж інструкцій (у процесорах архітектури x86 інструкції мають змінну довжину, що ускладнює їх коректне розпізнавання [17,19,44].

Особливістю дослідження є використання перехресних посилань.

Перехресні посилання (cross-references) дозволяють визначити, з яких ділянок програми здійснюється звернення до певної адреси або функції. Приклади відображення точки входу програми наведено на рисунку 5.14.

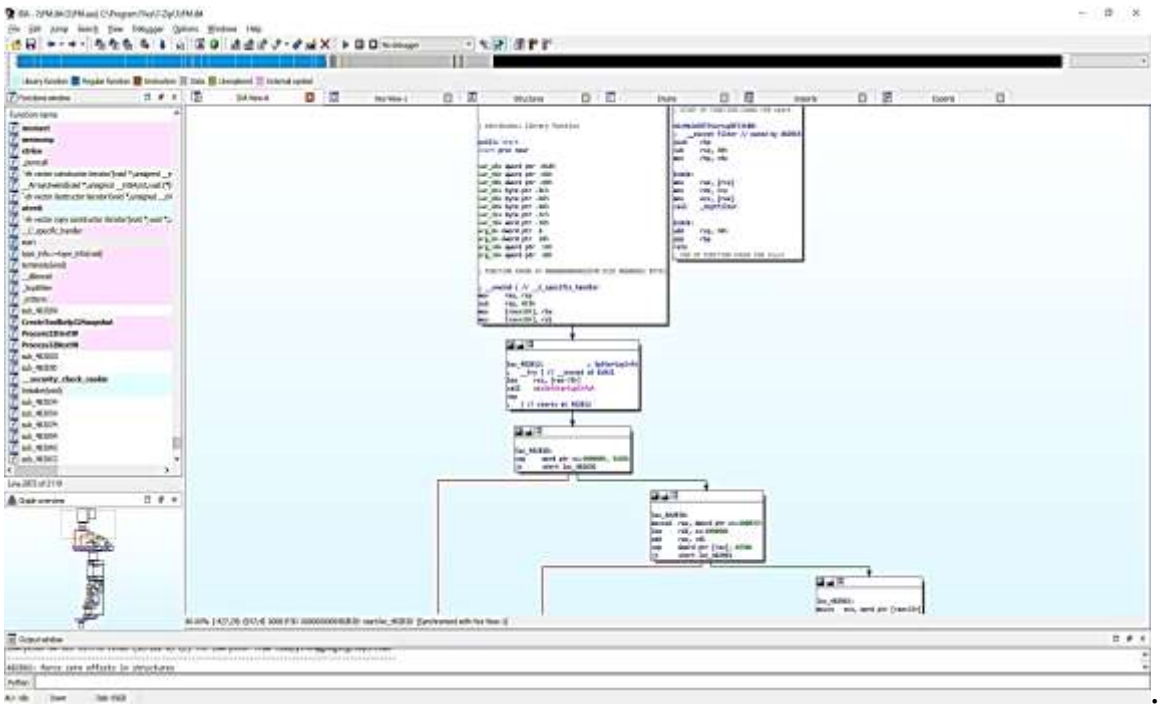


Рисунок 5.14. Функція старт

Перехресні посилання – це адреси, з яких здійснюються звернення до поточної ділянки коду або даних. Їх аналіз дозволяє встановити взаємозв’язки між функціями та визначити логіку переходів у програмі. У дизасемблері передбачено можливість переходу до відповідної адреси за перехресним посиланням, а також налаштування кількості відображуваних посилань (рис. 5.15).

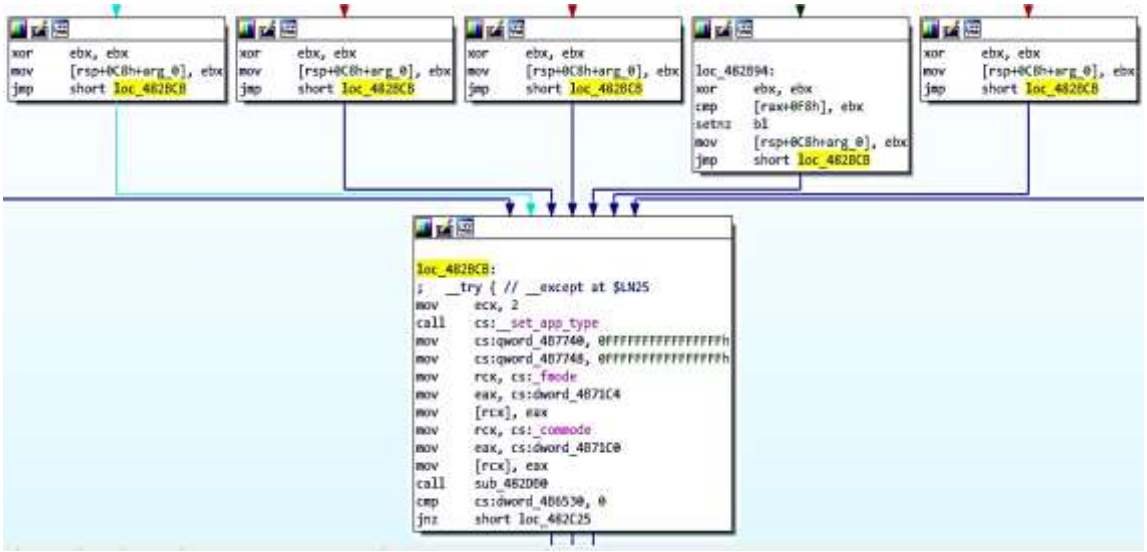


Рисунок 5.15. Перехресні посилання

Аналіз перехресних посилань є ключовим для розуміння логіки програми та взаємозв’язків між її компонентами [47, 44].

Під час дослідження особливу увагу слід приділяти сегменту даних (.data), де зберігаються:

– текстові рядки;

– константи;

– параметри функцій.

На рисунку 5.16 показано початкові значення констант, що використовуються у програмі та можуть бути ідентифіковані під час статичного аналізу.

```
char alf[28][2] = {"","a","b","c","d","e","f","g","h","i","j","k","l","m"
```

Рисунок 5.16. Початкові значення

Рисунок 5.17 ілюструє приклад розміщення констант і строкових даних у сегменті даних виконуваного файлу.

```
.data:00D44000
.data:00D44001                  align 2
.data:00D44002                  db  61h ; a
.data:00D44003                  db    0
.data:00D44004                  db  62h ; b
.data:00D44005                  db    0
.data:00D44006                  db  63h ; c
.data:00D44007                  db    0
.data:00D44008                  db  64h ; d
.data:00D44009                  db    0
.data:00D4400A                  db  65h ; e
.data:00D4400B                  db    0
.data:00D4400C                  db  66h ; f
.data:00D4400D                  db    0
.data:00D4400E                  db  67h ; g
.data:00D4400F                  db    0
.data:00D44010                  db  68h ; h
.data:00D44011                  db    0
.data:00D44012                  db  69h ; i
.data:00D44013                  db    0
.data:00D44014                  db  6Ah ; j
.data:00D44015                  db    0
.data:00D44016                  db  6Bh ; k
.data:00D44017                  db    0
.data:00D44018                  db  6Ch ; l
.data:00D44019                  db    0
```

Рисунок 5.17. Константи у сегменті даних

Аналіз таких даних дозволяє ідентифікувати призначення програми та її ключові функції, а також визначити характер оброблюваної інформації, можливі сценарії використання та взаємодію з користувачем або зовнішніми компонентами системи. Додатково дослідження констант і строкових значень сприяє виявленню прихованої логіки, діагностичних повідомлень, мережевих адрес, шляхів до файлів та інших артефактів, що можуть бути використані для подальшого аналізу або виявлення потенційно небезпечної поведінки програми [17,19].

У процесі декомпіляції можуть виникати спотворення [44,48]:

– неправильне визначення типів даних;

– зміна структури циклів;

– спрощення або трансформація логіки.

На рисунку 5.18 наведено фрагмент початкового програмного модуля до виконання декомпіляції.

```
void shfrAlf()
{
    int move;
    cout << "Vvedite sdvig";
    cin >> move;
    n = 0;
    i = 0;
    {
        while (i < k - move)
        {
            Kralf[i][2] = alf[i + move][2];
            i++;
        }
        for (int zi = i; i < k; i++)
        {
            Kralf[i][2] = alf[n][2];
            n++;
        }
    }
}
```

Рисунок 5.18. Код початкового модуля

Аналіз початкового коду дозволяє визначити структуру алгоритму, типи даних, а також логіку виконання програми у її вихідному вигляді. Це є базою для подальшого порівняння з результатами декомпіляції та оцінки точності

відновлення. Рисунок 5.19 ілюструє результат декомпіляції відповідного програмного модуля.

```
unsigned __int64 shfrAlf()
{
  signed int v0; // edx
  int v2; // [esp+DCh] [ebp-Ch]

  j_std::operator___std::char_traits_char____0(std::cout, "Vvedite sdvig");
  std::basic_istream<char,std::char_traits<char>>::operator>>(std::cin, &v2);
  n = 0;
  for ( i = 0; i < k - v2; ++i )
  {
    v0 = 2;
    Kralf[2 * i + 2] = alf[2 * (v2 + i) + 2];
  }
  while ( i < k )
  {
    v0 = i;
    Kralf[2 * i++ + 2] = alf[2 * n++ + 2];
  }
  return __PAIR__(v0, i);
}
```

Рисунок 5.19. Декомпільований модуль

Порівняння декомпільованого коду з початковим демонструє можливі спотворення, зокрема зміну типів даних, трансформацію циклічних конструкцій і втрату частини семантичної інформації [44,48]. Це підкреслює обмеження автоматизованих засобів аналізу та необхідність інтерпретації результатів дослідником.

У деяких випадках декомпіляція дає результат, близький до оригінального коду. На рисунку 5.20 наведено приклад початкової функції програмного модуля.

```
void deShfr(char sench[100], int length)
{
    for (int q = 0; q< length; q++)
    {
        for (int zn = 0; zn < 6; zn++) {
            if (sench[q] == Kralf[zn][2]) { key[q] = zn; }
        }
    }
    for (int v = 0; v < length; v++)
    {
        cout << alf[key[v]][2];
    }
    cout << "\n";
}
```

Рисунок 5.20. Початкова функція

Дослідження такої функції дозволяє визначити її призначення, вхідні параметри та логіку обробки даних, що є важливим для подальшого аналізу поведінки програми. Рисунок 5.21 демонструє результат декомпіляції відповідної функції.

```
int __cdecl deShfr(int a1, int a2)
{
  int k; // [esp+D0h] [ebp-20h]
  int j; // [esp+DCh] [ebp-14h]
  int i; // [esp+E8h] [ebp-8h]

  for ( i = 0; i < a2; ++i )
  {
    for ( j = 0; j < 6; ++j )
    {
      if ( *(char *)(i + a1) == Kralf[2 * j + 2] )
        key[i] = j;
    }
  }
  for ( k = 0; k < a2; ++k )
    j_std::operator___std::char_traits_char___(std::cout, alf[2 * key[k] + 2]);
  return j_std::operator___std::char_traits_char____0(std::cout, "\n");
}
```

Рисунок 5.21. Декомпільована функція

У даному випадку декомпіляція забезпечує відносно точне відновлення логіки функції, що свідчить про ефективність застосованих інструментів статичного аналізу. Водночас можливі незначні відхилення, зокрема у визначенні типів даних або оформленні конструкцій, потребують додаткової перевірки. Це дозволяє оцінити точність аналізу та зробити висновки щодо структури програми.

Таким чином, статичний аналіз дає змогу отримати цілісне уявлення про структуру та призначення програмного продукту, виявити ознаки втручання або аномальної поведінки, а також сформувати основу для подальшого, більш глибокого дослідження.

Статичний аналіз є необхідним етапом дослідження програмного забезпечення, що забезпечує первинне розуміння його структури та дозволяє підготувати основу для переходу до динамічного аналізу і комплексної оцінки безпеки.

ПИТАННЯ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ ЗА ТЕМОЮ 5

1. У чому полягає сутність статичного аналізу програмного забезпечення?
2. Які основні етапи виконуються під час статичного аналізу виконуваного файлу?
3. Яке призначення PE-заголовка та які його елементи є найбільш важливими для аналізу?
4. Які секції PE-файлу є найбільш важливими для дослідження безпеки програмного забезпечення?
5. Які ознаки можуть свідчити про аномалії або модифікацію PE-файлу?
6. Які інструментальні засоби використовуються для статичного аналізу програмного забезпечення?
7. Які переваги та обмеження має статичний аналіз програмного забезпечення?
8. Як статичний аналіз використовується для виявлення вразливостей і потенційно небезпечного коду?
9. Яку інформацію про програму можна отримати без її запуску за допомогою статичного аналізу?
10. Яке значення мають перехресні посилання (cross-references) під час статичного аналізу програмного забезпечення?

ПРАКТИЧНІ ЗАВДАННЯ ДО ТЕМИ 5

ЗАВДАННЯ 1. СТАТИЧНИЙ АНАЛІЗ PE-ЗАГОЛОВКУ ВИКОНУВАНОГО ФАЙЛУ

Постановка завдання

На основі підрозділів 5.3 та 5.4 посібника проведіть статичний аналіз PE-заголовка виконуваного файлу.

Оберіть будь-який виконуваний PE-файл (.exe або .dll) без використання вихідного коду:

а) системну утиліту Windows (notepad.exe, calc.exe, mspaint.exe);

б) встановлений застосунок стороннього розробника;

в) невеликий виконуваний файл, скомпільований з відомого вихідного коду (без символів налагодження).

Дослідження слід проводити лише для легітимних системних або навчальних файлів. Використання невідомих або потенційно шкідливих виконуваних файлів допускається лише у контрольованому середовищі.

Для обраного файлу:

1) За допомогою будь-якого інструменту з підрозділу 5.5 (CFF Explorer, PPEE puppy, pestudio або hex-редактора) зчитайте DOS-заголовок і заповніть таблицю значень ключових характеристик: e_magic, e_lfanew та не менше трьох інших полів (таблиця 5.3 посібника).

2) Зчитайте File Header (таблиця 5.4) і визначте: архітектуру (Machine), кількість секцій (NumberOfSections), мітку часу компіляції (TimeDateStamp) та характеристики образу (Characteristics).

3) На основі Optional Header (таблиця 5.5) визначте точку входу (AddressOfEntryPoint), бажану базову адресу (ImageBase), вирівнювання секцій (SectionAlignment і FileAlignment), а також проаналізуйте основні секції файлу з їх іменами, розмірами (VirtualSize і SizeOfRawData) та атрибутами (Characteristics).

4) Зробіть висновок: які елементи структури файлу можуть свідчити про його захищеність або потенційні аномалії (наявність цифрового підпису, підозрілі секції, нетипові поля)?

Приклад розв'язку

Досліджуваний файл: notepad.exe (Windows 11, x64). Інструмент: CFF Explorer 8.

Таблиця 5.7

DOS-заголовок (ключові поля):

Характеристика	**Зміщення**	**Значення (HEX)**	**Пояснення**
e_magic	0x0000	4D 5A	Сигнатура «MZ» – PE-файл Windows

Характеристика	Зміщення	Значення (HEX)	Пояснення
e_cblp	0x0002	90 00	144 байти використано на останній сторінці
e_cp	0x0004	03 00	3 сторінки для зберігання файлу
e_lfarlc	0x0018	40 00	Таблиця переміщень на зміщенні 0x40
e_lfanew	0x003C	F8 00 00 00	PE-заголовок знаходиться на зміщенні 0xF8

Таблиця 5.8

File Header:

Характеристика	Значення	Пояснення
Machine	0x8664	AMD64 (x86-64)
NumberOfSections	7	Файл містить 7 секцій
TimeDateStamp	0x65B3A210	Дата компіляції: 26.01.2024 (UTC)
Characteristics	0x0022	IMAGE_FILE_EXECUTABLE_IMAGE + LARGE_ADDRESS_AWARE

Таблиця 5.9

Optional Header (ключові поля):

Характеристика	Значення	Пояснення
AddressOfEntryPoint	0x000177C0	RVA точки входу програми
ImageBase	0x140000000	Бажана адреса завантаження (64-бітна)
SectionAlignment	0x1000	Вирівнювання секцій у пам'яті (4 КБ)
FileAlignment	0x200	Вирівнювання секцій у файлі (512 байт)
CheckSum	0x0003D44A	Контрольна сума образу для перевірки цілісності файлу

Таблиця 5.10

Секції файлу:

Секція	VirtualSize	SizeOfRawData	Characteristics	Призначення
.text	0x0002B8A4	0x0002BA00	0x60000020 (CODE,EXEC)	Виконуваний код
.rdata	0x00016234	0x00016400	0x40000040 (IDATA,READ)	Таблиці імпорту/експорту, константи

Секція	VirtualSize	SizeOfRawData	Characteristics	Призначення
.data	0x00001BB0	0x00000C00	0xC0000040 (DATA,R/W)	Ініціалізовані змінні
.pdata	0x000025B8	0x00002600	0x40000040	Таблиця виключень (x64)
.rsrc	0x0000B2F0	0x0000B400	0x40000040	Ресурси (діалоги, рядки)
.reloc	0x00000798	0x00000800	0x42000040	Таблиця базових поправок
.didat	0x00000120	0x00000200	0xC0000040	Дані відкладеного імпорту

Висновок: зі статичного аналізу заголовку встановлено, що notepad.exe є 64-бітним PE-файлом архітектури x86-64, скомпільованим у 2024 році. Ненульове значення поля CheckSum свідчить про наявність контрольної суми образу, яка використовується для перевірки цілісності під час завантаження. Додатковий аналіз таблиці сертифікатів (Certificate Table) показав наявність цифрового підпису Microsoft Authenticode.

Структура секцій є стандартною, без підозрілих або невідомих секцій. Для ЦЗ-систем аналогічна перевірка підтвердила б відсутність нестандартних секцій (ознака можливого Х-коду категорій В або С) та відповідність контрольної суми еталонному значенню.

ЗАВДАННЯ 2. ВИЯВЛЕННЯ АНОМАЛІЙ У СТРУКТУРІ PE-файлу

Постановка завдання

На основі підрозділів 5.4 та 5.5 посібника виконайте дослідження аномалій у структурі PE-файлу засобами статичного аналізу.

Оберіть два виконувані файли формату PE (.exe або .dll):

а) стандартний системний файл Windows або легітимний застосунок;

б) файл із ознаками модифікації, пакування або обфускації (навчальний приклад, тестовий файл або спеціально підготовлений зразок).

Дослідження слід проводити лише для легітимних або навчальних файлів. Аналіз потенційно небезпечних виконуваних об’єктів допускається виключно у контрольованому середовищі.

Для кожного файлу:

За допомогою інструментів із підрозділу 5.5 (pestudio, CFF Explorer, PEStudio, manalyzer.org або Ghidra):

1) Дослідіть структуру PE-файлу та перевірте: коректність PE-заголовків, кількість і назви секцій, вирівнювання секцій, таблиці імпорту, наявність цифрового підпису, значення entropy секцій.

2) Визначте можливі аномалії структури файлу: нестандартні або підозрілі секції, надмірно високе значення entropy, відсутність або пошкодження таблиць імпорту, нетипові характеристики секцій, невідповідність VirtualSize і SizeOfRawData, наявність overlay-даних.

3) Складіть порівняльну таблицю двох файлів за характеристиками: кількість секцій, наявність цифрового підпису, середнє значення entropy, наявність підозрілих секцій, ознаки пакування або обфускації, загальна оцінка ризику.

4) Зробіть висновок щодо того, які аномалії можуть свідчити про:

- використання пакувальників;
- модифікацію PE-файлу;
- наявність потенційно небезпечного коду;
- спроби приховування функціональності.

Приклад розв'язку

Для виконання статичного аналізу було обрано два виконувані файли формату PE. Перший файл – системний застосунок Windows calc.exe (Windows 11 x64), який використано як приклад легітимного виконуваного файлу зі стандартною структурою. Другий файл – packed_sample.exe, що являє собою навчальний зразок із попереднім пакуванням за допомогою UPX.

Дослідження виконувалося із застосуванням інструментів pestudio, CFF Explorer та онлайн-сервісу manalyzer.org, які дозволяють аналізувати структуру PE-файлу, таблиці імпорту, характеристики секцій та інші ознаки потенційної модифікації.

Аналіз файлу calc.exe

Під час аналізу файлу calc.exe встановлено, що його структура відповідає типовому системному PE-файлу Windows. DOS Header та PE-заголовок є коректними, а набір секцій (.text, .rdata, .data, .rsrc, .reloc) відповідає стандартній структурі виконуваних файлів операційної системи Windows.

Таблиця імпорту містить валідні записи системних бібліотек, а файл має цифровий підпис Microsoft Authenticode, що підтверджує його походження та цілісність.

Додатково було проаналізовано значення entropy окремих секцій. Для секції .text значення entropy становило 6.12, для .rdata – 4.91, а для .data – 3.22. Такі значення не перевищують типових меж для системних PE-файлів і не свідчать про використання пакування, шифрування або обфускації коду.

Аналіз файлу packed_sample.exe

У процесі аналізу packed_sample.exe виявлено ряд ознак, характерних для модифікованих або упакованих виконуваних файлів. Зокрема, структура містить секції UPX0 та UPX1, які є типовими для пакувальника UPX.

Крім того, у файлі спостерігається неповна таблиця імпорту, а також overlay-дані наприкінці файлу, що може свідчити про наявність додаткових даних поза межами стандартної структури PE.

Значення entropy для секцій UPX0 та UPX1 становили відповідно 7.91 та 7.84, що є суттєво вищим за типові значення для звичайних секцій виконуваних файлів. Високий рівень entropy є характерною ознакою пакування, стиснення або шифрування програмного коду.

Таким чином, сукупність виявлених ознак – нестандартні секції, високі значення entropy та overlay-дані – свідчить про використання пакувальника UPX та потребує подальшого поглибленого аналізу файлу.

Таблиця 5.11

Порівняльний аналіз файлів

Характеристика	**calc.exe**	**packed_sample.exe**
Кількість секцій	6	4
Цифровий підпис	Наявний	Відсутній
Середнє entropy	4.75	7.87

Характеристика	calc.exe	packed_sample.exe
Підозрілі секції	Ні	UPX0, UPX1
Ознаки пакування	Ні	Так
Overlay-дані	Ні	Так
Загальна оцінка ризику	Низька	Висока

Інтерпретація результатів

Для calc.exe структура PE-файлу відповідає типовому системному застосунку: секції мають стандартні назви, entropy перебуває у допустимих межах, файл містить цифровий підпис.

Для packed_sample.exe виявлено ряд аномалій: нестандартні секції, високе entropy, ознаки пакування, відсутність цифрового підпису.

Такі ознаки не є прямим підтвердженням шкідливості файлу, однак потребують додаткового аналізу та можуть свідчити про спроби приховування внутрішньої логіки програми.

Висновок: Статичний аналіз структури PE-файлу дозволяє виявляти аномалії, пов’язані з модифікацією, пакуванням або обфускацією виконуваних файлів без їх запуску. Аналіз секцій, таблиць імпорту, цифрових підписів і значень entropy є ефективним методом первинного оцінювання ризику та може використовуватися у підрозділах ЦЗ для виявлення потенційно небезпечного програмного забезпечення.

Комплексне використання декількох інструментів статичного аналізу підвищує достовірність результатів і дозволяє компенсувати обмеження окремих засобів дослідження.

ЗАВДАННЯ 3. ПОРІВНЯЛЬНИЙ АНАЛІЗ ІНСТРУМЕНТІВ СТАТИЧНОГО АНАЛІЗУ PE-ФАЙЛІВ

Постановка завдання

На основі підрозділу 5.5 посібника ознайомтеся з переліком інструментальних засобів статичного аналізу та оберіть три для детального дослідження.

Оберіть три інструменти з наведеного переліку:

а) CFF Explorer;

б) pestudio (winitor.com);

в) Ghidra;

г) YARA;

д) manalyzer.org.

Дослідження інструментів слід проводити лише для навчальних або легітимних виконуваних файлів. Аналіз потенційно небезпечних об’єктів рекомендується виконувати виключно у контрольованому середовищі.

Для кожного обраного інструменту:

1) Визначте клас інструменту (аналізатор PE-заголовків, дизасемблер/декомпілятор, сигнатурний аналізатор, онлайн-сервіс) та стисло опишіть головне призначення і ключові можливості, що відрізняють його від аналогів.

2) Оцініть придатність інструменту для виконання шести задач статичного аналізу:

– читання та інтерпретація PE-заголовку,
– аналіз таблиць імпорту та експорту,
– виявлення підозрілих секцій і сигнатур шкідливого коду,
– дизасемблювання / декомпіляція функцій,
– пошук індикаторів компрометації (IoC),
– автономна робота без доступу до мережі.

3) Складіть порівняльну таблицю трьох обраних інструментів за наведеними задачами: позначте «Так», «Частково» або «Ні».

4) Сформулюйте рекомендацію щодо комбінованого застосування обраних інструментів для аналізу підозрілого виконуваного файлу у підрозділі ЦЗ: визначте послідовність дій і роль кожного інструменту з урахуванням переваг і проблем статичного методу.

Приклад розв'язку

(Обрані інструменти: CFF Explorer, pestudio, Ghidra).

CFF Explorer

(клас: аналізатор PE-заголовків, розробник ntcore.com, безкоштовний):

Призначений виключно для розбору структури PE-файлів: графічно відображає DOS header, File header, Optional header, Data Directory, таблиці секцій, імпорту та експорту, ресурси і цифрові підписи.

Інструмент дозволяє редагувати окремі поля PE-заголовка безпосередньо в інтерфейсі та є ефективним засобом первинного аналізу структури виконуваного файлу.

Не виконує дизасемблювання та не містить сигнатурної бази шкідливого коду.

Pestudio

(клас: аналізатор PE + індикатори загроз, winitor.com, умовно-безкоштовний):

Поєднує розбір PE-структури з автоматичним виявленням підозрілих ознак: перевіряє імпортовані функції на наявність в базах шкідливого ПЗ, виявляє нетипові секції, рядки, порожні або підозрілі поля заголовку, розраховує entropy для кожної секції (висока entropy → можливе шифрування або стиснення), перевіряє хеші через VirusTotal.

Високе значення entropy може свідчити про використання пакування, шифрування або обфускації коду.

Інструмент не виконує повноцінного дизасемблювання програмного коду.

Ghidra

(клас: дизасемблер і декомпілятор, NSA / open-source, безкоштовний):

Повнофункціональне середовище зворотного інжинірингу:

– дизасемблює x86/x64/ARM та десятки інших архітектур,
– підтримує декомпіляцію до псевдокоду C,
– відображає граф потоку керування (CFG),
– відновлює сигнатури функцій стандартних бібліотек.

Ghidra підтримує велику кількість архітектур процесорів та дозволяє проводити глибокий аналіз логіки роботи програмного забезпечення.

Для роботи потребує попереднього встановлення та середовища Java Runtime Environment (JRE).

Таблиця 5.12

Порівняльна таблиця:

Задача	**CFF Explorer**	**pestudio**	**Ghidra**
Читання та інтерпретація PE-заголовку	Так	Так	Частково
Аналіз таблиць імпорту та експорту	Так	Так	Так
Виявлення підозрілих секцій і сигнатур шкідливого коду	Ні	Так	Частково
Дизасемблювання / декомпіляція функцій	Ні	Ні	Так
Пошук індикаторів компрометації (IoC)	Ні	Так	Частково
Автономна робота без мережі	Так	Так	Так

Рекомендація щодо послідовності аналізу підозрілого файлу у підрозділі ЦЗ:

Етап 1 – Первинний огляд (pestudio)

Файл відкривається у pestudio для швидкої оцінки ризику. Інструмент автоматично виявляє:

– підозрілі поля заголовка;

– нетипові секції;

– небезпечні API-функції;

– аномальні значення entropy.

Якщо entropy секції .text перевищує 7.0, файл може бути упакований або зашифрований, що ускладнює подальший статичний аналіз без попереднього розпакування.

Етап 2 – Детальний аналіз структури (CFF Explorer)

Після первинної оцінки файл відкривається у CFF Explorer для детального аналізу:

– DOS Header;

– File Header;
– Optional Header;
– Data Directory;
– таблиць імпорту та експорту.

На цьому етапі підтверджуються або спростовуються аномалії, виявлені під час первинного аналізу.

Етап 3 – Аналіз програмного коду (Ghidra)

Для дослідження логіки роботи програми файл аналізується у Ghidra.

Особлива увага приділяється функціям, пов’язаним із:

– мережевою взаємодією;
– файловою системою;
– механізмами автозавантаження;
– криптографічними операціями.

Такі функції можуть свідчити про реалізацію закладок, прихованих механізмів керування або інших потенційно небезпечних компонентів.

Запропонована послідовність аналізу відповідає типовим етапам статичного дослідження програмного забезпечення: первинна ідентифікація → аналіз структури → аналіз коду. Комплексне використання декількох інструментів дозволяє компенсувати обмеження окремих засобів і підвищити ефективність виявлення потенційних загроз без запуску підозрілого файлу, що є критично важливим для середовищ цивільного захисту.

6. ДОСЛІДЖЕННЯ ПРОГРАМНИХ ПРОДУКТІВ В ПРОЦЕСІ ВИКОНАННЯ

Статичний аналіз, якому присвячено попередній розділ, дозволяє досліджувати структуру, заголовки, секції та внутрішню організацію програмного забезпечення без його запуску. Проте значна частина функціональності сучасних програмних реалізацій проявляється лише під час виконання, коли програма взаємодіє з операційною системою, пам’яттю, файловою системою, мережею та іншими компонентами програмного середовища. У таких випадках застосовується динамічний аналіз.

Динамічний аналіз є одним із ключових методів дослідження програмного забезпечення в інформаційній та кібербезпеці. Його сутність полягає у спостереженні за поведінкою програми під час виконання з використанням спеціалізованих засобів контролю, налагодження та моніторингу. На відміну від статичного підходу, динамічний метод дозволяє аналізувати фактичну поведінку програмної реалізації, виявляти приховану логіку, механізми захисту, мережеву активність, звернення до пам’яті, файлової системи та системного реєстру [17; 19; 48].

Методи динамічного аналізу широко застосовуються під час дослідження шкідливого програмного забезпечення, аналізу вразливостей, тестування механізмів захисту та вивчення особливостей функціонування складних програмних систем. Особливого значення вони набувають у випадках, коли програма використовує механізми обфускації, пакування, динамічного завантаження бібліотек або приховування функціональності, що ускладнює застосування виключно статичного аналізу [18; 19; 33; 39].

Для проведення динамічного аналізу використовуються налагоджувачі (debugger), монітори системної активності, sandbox-середовища, аналізатори мережевого трафіку та інші інструментальні засоби, які забезпечують контроль за виконанням програми та фіксацію її дій у реальному часі. При цьому важливою умовою є використання контрольованого та ізольованого середовища дослідження, що дозволяє мінімізувати ризики негативного впливу потенційно

небезпечного програмного забезпечення на операційну систему або мережеву інфраструктуру [49–55].

Для підрозділів цивільного захисту динамічний аналіз має важливе практичне значення під час дослідження програмних компонентів автоматизованих систем управління, засобів обробки оперативної інформації, мережевих сервісів та спеціалізованого програмного забезпечення, що використовується у критичних інформаційних системах.

У цьому розділі розглянуто основні методи динамічного аналізу програмного забезпечення, інструментальні засоби їх реалізації, а також практичні аспекти дослідження поведінки програмних реалізацій у процесі виконання.

6.1. СУТНІСТЬ ДИНАМІЧНОГО АНАЛІЗУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Динамічний аналіз програмного забезпечення – це метод дослідження програмних реалізацій, який передбачає аналіз поведінки програми безпосередньо під час її виконання. На відміну від статичного аналізу, що базується на дослідженні структури файлів без запуску, динамічний метод дозволяє спостерігати фактичну взаємодію програми з операційною системою, пам’яттю, файловою системою, мережею та іншими компонентами середовища виконання [17; 19; 48].

Основною особливістю динамічного аналізу є можливість дослідження поведінки програми в реальному часі. Це дозволяє [26; 50; 51]:

– контролювати виконання машинних інструкцій;
– відстежувати виклики API-функцій;
– аналізувати створення процесів і потоків;
– виявляти звернення до системного реєстру та файлової системи;
– досліджувати мережеву активність;
– контролювати зміни пам’яті під час виконання.

Динамічний аналіз особливо ефективний у випадках, коли програмне забезпечення використовує пакування, обфускацію, шифрування або механізми приховування функціональності, що ускладнюють застосування виключно статичного аналізу [19; 33].

Для проведення динамічного аналізу використовуються налагоджувачі (debugger-и), монітори системної активності, sandbox-середовища, аналізатори мережевого трафіку, засоби трасування виконання програм [50-55].

Під час динамічного аналізу дослідник може:

– виконувати програму покроково;

– встановлювати точки зупину (breakpoints);

– контролювати вміст регістрів процесора;

– досліджувати адресний простір процесу;

– аналізувати взаємодію програми з бібліотеками та системними компонентами.

Важливою умовою безпечного проведення динамічного аналізу є використання ізольованого та контрольованого середовища виконання, зокрема віртуальних машин або sandbox-систем, що дозволяє мінімізувати ризики пошкодження операційної системи чи мережевої інфраструктури [49; 55].

Перевагою динамічного аналізу є можливість спостереження фактичної поведінки програми, включаючи приховані або умовно активовані функції. Водночас цей метод має і певні обмеження, оскільки результати аналізу можуть залежати від конкретного середовища виконання, а частина функціональності може проявлятися лише за визначених умов або після виконання певних дій користувача [17; 19; 48].

Таким чином, динамічний аналіз є важливим доповненням до статичного методу та дозволяє комплексно досліджувати поведінку програмного забезпечення у процесі його виконання.

6.2. МЕТОДИКА ДОСЛІДЖЕННЯ ПРОГРАМ ДИНАМІЧНИМ МЕТОДОМ

6.2.1. Основні принципи й поняття

Базовим інструментом динамічного методу є налагоджувач (debugger). Без розуміння його принципів – покрокового виконання, точок зупину, регістрів, стеку – неможливо провести повноцінний аналіз поведінки програм. Робота базується на апаратних механізмах процесора архітектури x86/x86-64 [17; 19; 26; 48].

- прапорець трасування (Trap Flag) – покрокове виконання машинних інструкції;
- точка зупинки (Breakpoint, int 3) – викликає штучне переривання для зупинки виконання програми;
- налагоджувальні регістри (Debug- Registers) – дозволяють фіксувати звернення до всизнаих ділянок пам’яті.

Налагоджувачі дозволяють переглядати машинний код, вміст регістрів, стек, змінювати дані в пам’яті, виконувати трасування, встановлювати точки зупину та здійснювати повторне виконання окремих фрагментів програми.

1. Прапорець трасування (Trace Flag або Trap Flag, TF).

Це спеціальний біт регістра прапорів процесора (8-й біт FLAGS/EFLAGS/RFLAGS). Коли цей біт встановлений в одиницю, процесор після виконання кожної машинної інструкції викликає переривання 1.

Прапорець трасування використовується для організації покрокового виконання програми. У цьому випадку налагоджувач перехоплює переривання 1 та після виконання кожної машинної інструкції відображає стан регістрів процесора, прапорів, сегментів пам’яті та інших параметрів середовища виконання [26; 50; 51].

2. Точка зупину (Breakpoint).

Точка зупину реалізується шляхом вставлення машинної команди виклику переривання 3 (int 3), що відповідає байту CCh.

На відміну від більшості інших переривань, команда int 3 займає лише один байт, тому може бути вставлена практично в будь-яке місце досліджуваної програми.

Налагоджувач перехоплює переривання 3 та після встановлення точки зупину передає керування досліджуваній програмі. Перед виконанням інструкції, на якій встановлено точку зупину, керування знову передається налагоджувачу. Перед продовженням виконання програми налагоджувач повинен відновити оригінальний байт машинної інструкції [19; 50; 51].

3. Налагоджувальні регістри (Debug Registers).

Починаючи з процесора Intel 80386, процесори сімейства x86 містять вісім налагоджувальних регістрів. Вони дозволяють реалізувати апаратні точки зупину.

На відміну від програмних точок зупину, апаратні точки зупину мають ширші можливості. Програма може бути зупинена не лише під час виконання певної інструкції, а й при зверненні до визначеної області пам’яті. При спрацюванні апаратної точки зупину генерується переривання 1 [26; 50; 51].

Наведені механізми характерні насамперед для архітектур x86/x86-64, однак аналогічні засоби трасування та контролю виконання реалізовані й в інших процесорних архітектурах.

Функції налагоджувача, які можуть використовуватися під час аналізу машинного коду, для якого відсутній вихідний текст:

1. покрокове трасування програми;

2. покрокове виконання програми, при якому виклик функції обробляється як одна команда;

3. покрокове виконання програми із переходом усередину обробників переривань;

4. встановлення / зняття точки зупину для визначеної інструкції;

5. встановлення / зняття апаратної точки зупину;

6. встановлення / зняття точки зупину для портів введення-виведення;

7. виконання фрагмента програми до найближчої точки зупину;

8. виконання фрагмента програми до інструкції, підсвіченої курсором, або до найближчої точки зупину;

9. перезапуск програми;

10. перегляд коду, пам’яті, регістрів, прапорів та стеку;

11. пошук рядків у пам’яті;

12. пошук блоків пам’яті, створених операційною системою;

13. зміна вмісту регістрів, прапорів, пам’яті та машинного коду [50; 51ї.

Аналіз програми динамічним методом умовно поділяється на три основні етапи (рис. 6.1):

1. вибір методу виявлення функцій програми;

2. пошук функцій програми;

3. аналіз функцій програми.



Рисунок 6.1. Основні етапи динамічного аналізу програмного забезпечення

У динамічному методі програма виконується під контролем спеціалізованих програмних засобів. Наявність таких засобів дозволяє виконувати програму в покроковому режимі, зупиняти її виконання при виникненні визначених подій та досліджувати поведінку окремих фрагментів

програмного коду [17, 19, 48]. Це суттєво спрощує виявлення й аналіз ділянок програми, що реалізують механізми захисту інформації, автентифікації, контролю доступу або прихованої функціональності.

6.2.2. І етап – Метод маяків та Step-Trace першого етапу

І етап динамічного аналізу полягає у виявленні функцій програми, що реалізують критично важливі дії або механізми захисту. Основною метою цього етапу є локалізація фрагментів програмного коду, відповідальних за:

– введення та перевірку паролів;
– роботу з файлами;
– виділення пам’яті;
– мережеву взаємодію;
– криптографічні перетворення;
– механізми автентифікації або контролю доступу [17; 19; 48].

Для реалізації першого етапу застосовуються два основні методи:

а) метод маяків;

б) метод Step-Trace першого етапу.

Метод маяків базується на встановленні точок зупину на системні виклики, API-функції або бібліотечні процедури, призначення яких легко визначити без повного знання внутрішньої логіки програми [50; 51]. Такі точки називаються «маяками», оскільки вони дозволяють локалізувати ділянки програмного коду, пов’язані з певними діями програми [19,48] (рис.6.2).

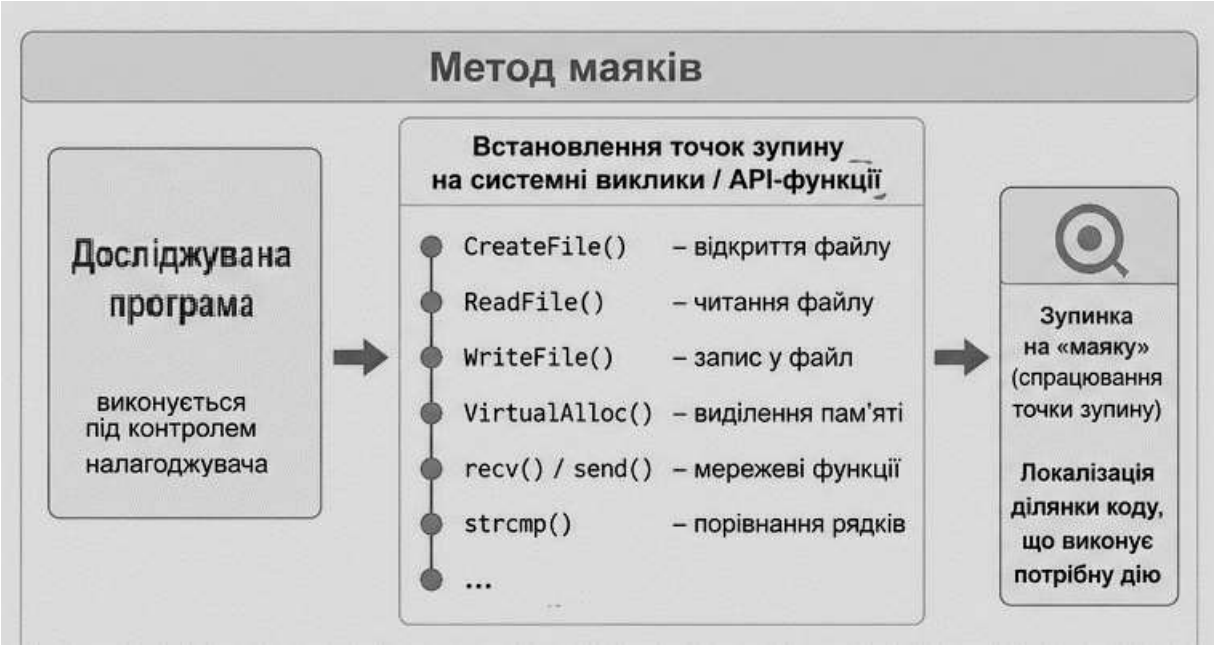


Рисунок 6.2. Метод маяків

У якості маяків можуть використовуватися:

– функції відкриття та запису файлів;
– функції роботи з пам’яттю;
– мережеві API-виклики;
– функції введення користувача;
– криптографічні бібліотеки.

Розглянемо загальну послідовність реалізації методу маяків під час динамічного аналізу програмного забезпечення. На першому етапі досліджувана програма запускається під контролем налагоджувача. Після цього налагоджувач відстежує виклики системних API-функцій або бібліотечних процедур, які виконують визначені дії, наприклад відкриття файлів, обробку мережевих даних чи перевірку введених користувачем значень (рис. 6.3).



Рисунок 6.3. Послідовність реалізації методу маяків

При зверненні програми до однієї з визначених функцій спрацьовує встановлена точка зупину («маяк»), що дозволяє тимчасово призупинити виконання програми та перейти до аналізу поточного стану процесу. У цей момент дослідник може проаналізувати:

– адресу виклику функції;
– параметри, передані у функцію;
– стек викликів;
– значення регістрів процесора;
– пов’язані ділянки програмного коду.

Отримані результати дозволяють локалізувати фрагменти програми, які реалізують потрібну функціональність, та перейти до їх подальшого детального дослідження на наступних етапах динамічного аналізу [17,19].

Наприклад, якщо досліджується механізм перевірки пароля, доцільно встановити точки зупину на функції введення даних або порівняння рядків. Це дозволяє локалізувати ділянки коду, що беруть участь у перевірці автентифікаційних даних [27,28] (рис.6.4).



Рисунок 6.4. Дослідження механізму перевірки пароля

Метод маяків є ефективним на початковому етапі дослідження, оскільки дозволяє швидко звузити область аналізу та визначити фрагменти програми, які потребують подальшого детального дослідження [19; 48].

Метод Step-Trace першого етапу

Метод Step-Trace першого етапу полягає у покроковому виконанні програми з одночасним контролем переходів між функціями та аналізом змін стану програми. Для цього використовуються режими Step Into та Step Over сучасних налагоджувачів [50; 51], (рис.6.5).



Рисунок 6.5. Метод Step-Trace першого етапу

У режимі Step Into налагоджувач переходить усередину викликаної функції та дозволяє аналізувати її виконання покроково. Режим Step Over, навпаки, виконує викликану функцію як одну інструкцію без переходу всередину її коду.

На першому етапі Step-Trace застосовується переважно для локалізації потрібних функцій програми та визначення послідовності їх виклику [17; 19; 48].

Під час покрокового трасування дослідник:

– контролює зміну регістрів процесора;
– аналізує стек викликів;
– відстежує зміни пам’яті;
– спостерігає за передаванням параметрів між функціями;
– визначає точки переходу до критичних ділянок коду.

Використання Step-Trace дозволяє досліджувати поведінку програми навіть у випадках, коли назви функцій приховані, а програмний код обфускований або частково модифікований [19; 33].

Практика показує, що найбільш ефективним є комбіноване використання методу маяків і Step-Trace. Спочатку маяки дозволяють локалізувати ділянки коду, пов’язані з потрібною функціональністю, після чого покрокове трасування дає змогу дослідити взаємодію між окремими функціями та визначити логіку роботи програми.

Таким чином, перший етап динамічного аналізу забезпечує локалізацію критичних фрагментів програмного коду та створює основу для подальшого визначення функціональності програми і дослідження реалізованих алгоритмів.

6.2.3. II етап – Визначення функціональності функцій

Після локалізації функцій програми на першому етапі динамічного аналізу виконується визначення їх функціонального призначення та дослідження логіки взаємодії між окремими компонентами програми [17; 19; 48]. Загальний процес проходження II етапу відображено на рис.6.6.



Рисунок 6.6. Процес II етапу динамічного аналізу коду

Основною метою другого етапу є встановлення того, які саме дії виконує кожна з виявлених функцій, які дані вона обробляє та яким чином взаємодіє з іншими частинами програмної реалізації.

Для реалізації другого етапу застосовуються [50; 51]:

1. апаратні точки зупину;
2. метод Step-Trace другого етапу;
3. аналіз параметрів функцій;
4. контроль змін пам’яті та регістрів процесора.

Апаратні точки зупину (рис. 6.7) використовуються для контролю доступу до визначених областей пам’яті або виконання конкретних інструкцій без модифікації машинного коду програми. На відміну від програмних точок зупину, вони не потребують вставлення інструкції int 3 та реалізуються за допомогою налагоджувальних регістрів процесора [26,50,51].

Апаратні точки зупину є особливо ефективними під час дослідження програм, які використовують механізми самоконтролю цілісності або виявлення налагоджувача, оскільки не змінюють машинний код досліджуваної програми [19,50].

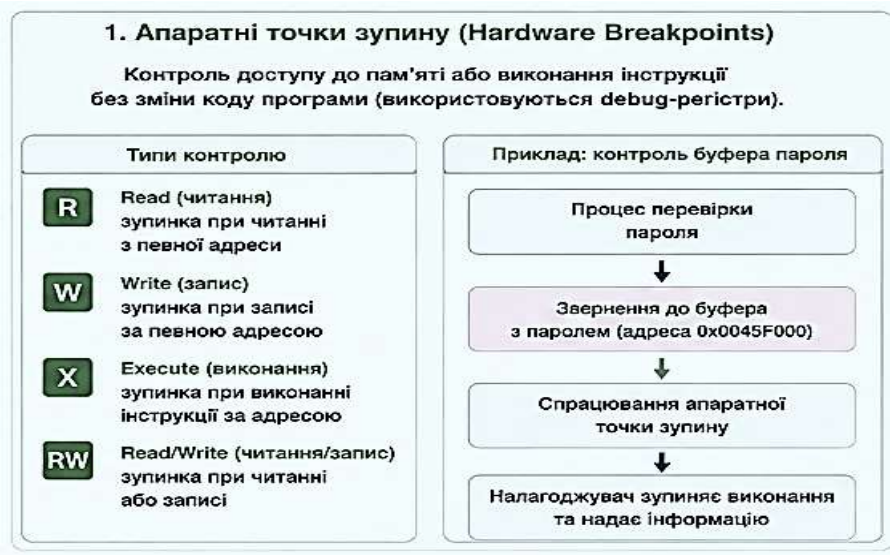


Рисунок 6.7. Апаратні точки зупину

Під час аналізу апаратні точки зупину можуть використовуватися для:

– контролю зміни критичних областей пам’яті;
– визначення моменту звернення до ключових структур даних;
– аналізу процедур шифрування та дешифрування;
– дослідження механізмів перевірки ліцензій або автентифікації.

На другому етапі метод Step-Trace використовується не для пошуку функцій, а для визначення їх функціональності та аналізу внутрішньої логіки роботи [17, 19, 48] (рис.6.8).



Рисунок 6.8. Схема методу Step-Trace II етапу

Під час покрокового виконання дослідник аналізує параметри функцій, контролює зміни регістрів, досліджує взаємодію між функціями, визначає умови переходів, аналізує цикли та алгоритмічні конструкції, контролює зміну даних у пам’яті.

Особливу увагу приділяють функціям, що здійснюють криптографічні операції, перевірку паролів, обробку мережевих даних або взаємодію із системними механізмами захисту [17,19].

Під час Step-Trace другого етапу дослідник поступово формує уявлення про:

– призначення функції;
– формат оброблюваних даних;
– логіку роботи алгоритму;
– залежності між окремими модулями програми.

Наприклад, якщо функція викликається після введення користувачем пароля та виконує серію операцій порівняння або перетворення рядків, можна зробити припущення про її участь у процедурі автентифікації.

Важливим елементом другого етапу є аналіз параметрів, що передаються між функціями. Для цього досліджуються (рис.6.9):

– стек викликів;
– регістри процесора;
– адреси буферів пам’яті;
– структури даних;
– результати виконання функцій [50,51].



Рисунок 6.9. Аналіз параметрів функцій

Аналіз параметрів дозволяє визначити типи оброблюваних даних, встановити взаємозв’язки між функціями та виявити критичні ділянки програмного коду [26, 48].

У процесі дослідження може також виконуватися контроль звернень до файлів, моніторинг мережевих з’єднань, аналіз змін системного реєстру, відстеження створення нових процесів або потоків [52, 53].

Другий етап динамічного аналізу дозволяє перейти від простого виявлення функцій до розуміння їх реального призначення та ролі у загальній логіці програми. Саме на цьому етапі визначаються:

– механізми автентифікації;
– алгоритми обробки даних;
– криптографічні процедури;
– механізми захисту;
– елементи прихованої функціональності.

Таким чином, другий етап динамічного аналізу забезпечує встановлення функціонального призначення виявлених функцій та створює основу для подальшого аналізу алгоритмів і внутрішньої логіки програмної реалізації.

6.2.4. III етап – Аналіз й дизасемблювання функцій

Третій етап динамічного аналізу полягає у детальному дослідженні внутрішньої логіки функцій, відновленні алгоритмів роботи програми та перевірці гіпотез щодо функціонального призначення окремих фрагментів коду [17, 19, 48]. На цьому етапі локалізовані та досліджені раніше функції піддаються дизасемблюванню та подальшому аналізу їх алгоритмічної структури.

Основними завданнями третього етапу є (рис. 6.10):

– отримання дизасембльованого представлення функцій;

– визначення логіки виконання алгоритмів (аналіз умов переходів та циклічних конструкцій, дослідження структур даних);

– перевірка гіпотез щодо призначення функцій;

– перевірка гіпотез щодо призначення функцій;

– відновлення взаємозв’язків між окремими компонентами програми.

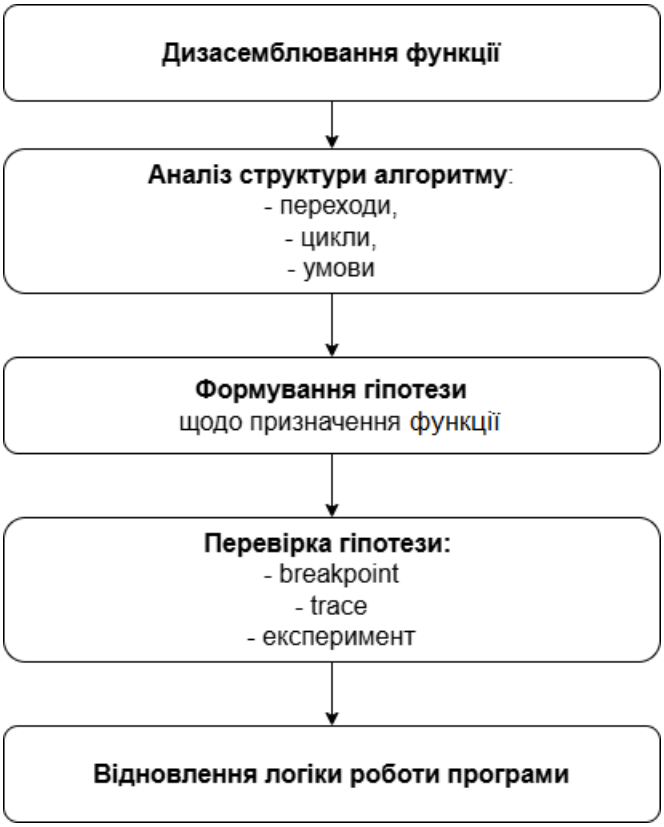


Рисунок 6.10. Етапи третього етапу динамічного аналізу коду

Дизасемблювання функцій

Після визначення функціональності окремих ділянок програми дослідник переходить до аналізу їх машинного коду за допомогою дизасемблерів або інтегрованих засобів налагодження. Для цього можуть використовуватися IDA Pro, Ghidra, Radare2, x64dbg, OllyDbg, інші засоби reverse engineering [44, 45, 47, 50].

Дизасемблювання дозволяє отримати представлення машинного коду у вигляді асемблерних інструкцій, що дає змогу досліджувати [44, 47, 48]:

– послідовність виконання команд;
– переходи між базовими блоками;
– виклики функцій;
– роботу зі стеком;
– операції з пам’яттю;
– алгоритмічні конструкції програми.

Сучасні дизасемблери підтримують не лише лінійне подання машинного коду, а й графічне представлення функцій (Graph View), що дозволяє досліджувати взаємозв’язки між базовими блоками, умовні переходи та цикли [44, 47].

Формування та перевірка гіпотези

На третьому етапі дослідник формує гіпотезу щодо функціонального призначення функції або окремого алгоритму. Така гіпотеза може базуватися на назвах API-функцій, характері оброблюваних даних, структурі переходів, послідовності викликів функцій, результатах попереднього трасування.

Наприклад, наявність циклічних операцій побітового перетворення, використання таблиць підстановок або інтенсивної роботи з буферами пам’яті може свідчити про реалізацію криптографічного алгоритму [17, 19].

Після формування припущення щодо призначення функції або окремого фрагмента програмного коду виконується перевірка цієї гіпотези за допомогою засобів динамічного аналізу. Для цього застосовуються точки зупину

(breakpoints), трасування виконання програми (trace) та експериментальні зміни вхідних даних або параметрів середовища виконання.

Встановлення точки зупину дозволяє призупинити виконання програми у визначеному місці та дослідити поточний стан процесу: значення регістрів, параметри функції, стек викликів і вміст пам’яті. Це дає можливість визначити, які дані обробляє функція та які результати формуються після її виконання.

Трасування (trace) використовується для покрокового контролю виконання інструкцій або функцій програми. Під час трасування дослідник аналізує послідовність виконання команд, переходи між функціями, зміни змінних і логіку роботи алгоритму.

Експериментальний підхід полягає у контрольованій зміні вхідних параметрів, файлів, мережевих даних або умов виконання програми з подальшим спостереженням за зміною її поведінки. Наприклад, зміна введеного пароля, структури файла або мережевого пакета дозволяє визначити, які саме функції відповідають за перевірку, обробку чи шифрування даних [17, 19, 48].

Поєднання точок зупину, трасування та експериментальних змін дозволяє підтвердити або спростувати сформовану гіпотезу щодо функціонального призначення досліджуваного фрагмента програмного коду.

Перевірка гіпотези є важливим елементом динамічного аналізу, оскільки дозволяє підтвердити або спростувати припущення щодо призначення функції та механізмів її роботи [17, 19].

Після підтвердження функціонального призначення функції виконується детальний аналіз реалізованого алгоритму. На цьому етапі досліджуються умови виконання переходів, циклічні конструкції, математичні операції, структури даних, механізми взаємодії між модулями програми, способи обробки вхідних та вихідних даних.

Особливу увагу приділяють функціям, що реалізують механізми захисту, перевірки цілісності, автентифікації, шифрування або прихованої взаємодії з операційною системою [26, 48].

У процесі аналізу також можуть використовуватися:

– перехресні посилання (cross-references);
– графи викликів функцій;
– псевдокод, сформований дизасемблером;
– контроль змін пам’яті та регістрів процесора [44, 47, 50].

Деякі сучасні засоби аналізу (наприклад, Ghidra або IDA Pro з Hex-Rays Decompiler) дозволяють автоматично формувати псевдокод, наближений до мов високого рівня, що значно спрощує аналіз складних алгоритмів [44, 47].

Процедура дизасемблювання має деякі обмеження. Ефективність дизасемблювання може суттєво знижуватися у випадках використання програмою спеціальних механізмів приховування або модифікації коду. До таких механізмів належать:

– пакування виконуваних файлів;
– обфускація програмного коду;
– самозмінюваний код;
– непрямі переходи та динамічне формування адрес переходів;
– механізми протидії аналізу (anti-debug, anti-VM);
– динамічне завантаження бібліотек та функцій [19, 33, 44].

Використання зазначених механізмів ускладнює побудову коректного графа виконання програми, автоматичне визначення меж функцій та відновлення логіки роботи алгоритмів. У таких випадках для підвищення ефективності дослідження доцільно поєднувати методи статичного та динамічного аналізу, використовуючи результати трасування, контрольованого виконання та експериментальної перевірки гіпотез.

Таким чином, третій етап динамічного аналізу забезпечує перехід від визначення функціональності окремих функцій до дослідження реалізованих алгоритмів та внутрішньої логіки програмної реалізації. Поєднання динамічного аналізу, дизасемблювання та експериментальної перевірки гіпотез дозволяє ефективно досліджувати складні програмні системи навіть за відсутності вихідного коду.

6.3. ІНСТРУМЕНТАЛЬНІ ЗАСОБИ ДИНАМІЧНОГО АНАЛІЗУ

Для проведення динамічного аналізу використовується широкий спектр інструментальних засобів, які відрізняються функціональністю, рівнем доступу до системи та сферою застосування [19, 48]. Існують спеціалізовані класи налагоджувачів і засобів моніторингу, орієнтовані на дослідження користувацьких застосунків, ядра операційної системи, драйверів, мережевої активності, використання пам’яті та інших аспектів функціонування програмного забезпечення. Класифікація налагоджувачів наведена на рис. 6.11.

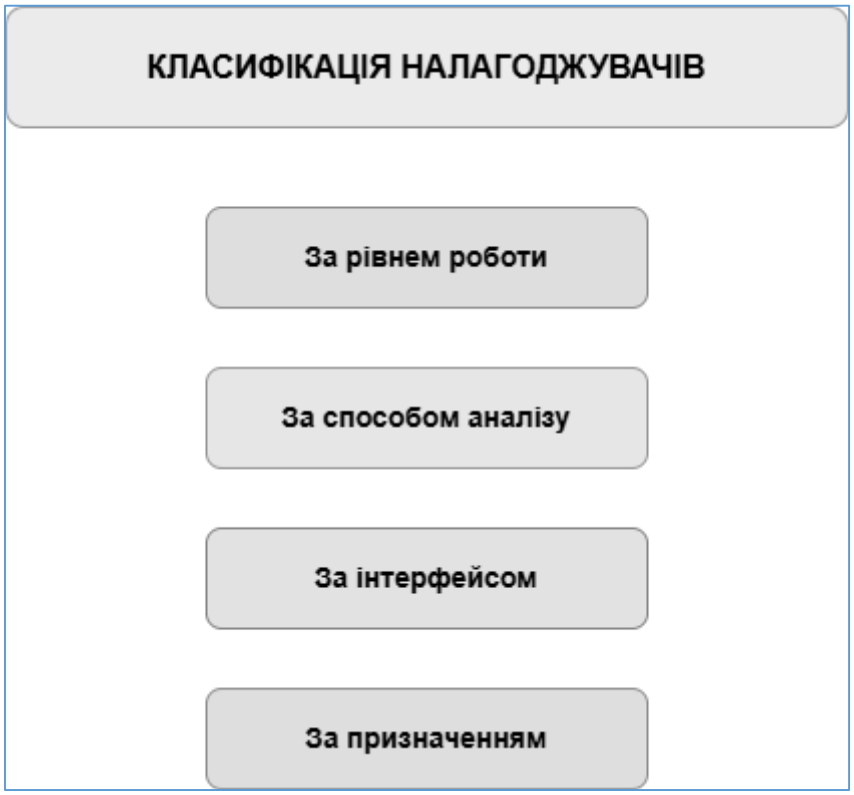


Рисунок 6.11. Класифікація налагоджувачів

Більшість сучасних середовищ розроблення містять вбудовані налагоджувачі. Наприклад, налагоджувач (debugger) у складі Microsoft Visual Studio може використовуватися не лише для тестування власних програмних реалізацій, а й для дослідження стороннього програмного забезпечення. Вибір конкретного інструмента залежить від мети аналізу, типу програмного забезпечення та рівня необхідного контролю над процесом виконання [44, 48].

Деякі засоби динамічного аналізу потребують завантаження додаткових бібліотек, інструментування програмного коду або його часткової модифікації перед запуском. Окремі сучасні засоби підтримують динамічне інструментування коду без попередньої перекомпіляції програми. Для підвищення ефективності аналізу досліджуваній програмі необхідно забезпечити достатній набір вхідних даних, що дозволяє активувати різні гілки виконання та отримати повніше покриття програмного коду.

У більшості випадків реалізація динамічного аналізу передбачає запуск налагоджувача, відкриття досліджуваної програми, встановлення точок зупину (breakpoints) та покрокове дослідження виконання окремих фрагментів коду. Під час такого аналізу можуть досліджуватися операції роботи з файлами, пам’яттю, стеком, системним реєстром, API-викликами та іншими компонентами операційної системи. Крім того, можуть аналізуватися мережеві з’єднання, міжпроцесна взаємодія, завантаження бібліотек та поведінка програми у sandbox-середовищі [50-55].

Сучасні засоби динамічного аналізу часто поєднують функції налагодження, моніторингу системної активності та поведінкового аналізу програмного забезпечення. Це дозволяє комплексно досліджувати поведінку програми, фіксувати взаємодію з операційною системою та виявляти приховані механізми функціонування або протидії аналізу.

Перед проведенням практичного динамічного аналізу доцільно ознайомитися з основними інструментальними засобами, що застосовуються для дослідження поведінки програмного забезпечення під час виконання. Такі засоби відрізняються функціональністю, рівнем доступу до системи та сферою застосування, що дозволяє використовувати їх як для налагодження програм, так і для аналізу механізмів захисту, мережевої активності та прихованої функціональності програмних реалізацій (таблиця 6.1).

Таблиця 6.1

Основні інструментальні засоби динамічного аналізу

Інструмент	Призначення	Основні можливості
x64dbg	Динамічний аналіз і reverse engineering	Покрокове трасування, breakpoints, аналіз пам’яті, API-викликів
WinDbg	Налагодження користувацького та kernel-mode коду	Аналіз дампів пам’яті, драйверів, ядра ОС
OllyDbg	Аналіз 32-бітних програм Windows	Дизасемблювання, трасування, модифікація коду в пам’яті
Process Monitor	Моніторинг системної активності	Контроль файлової системи, реєстру, процесів та потоків
Wireshark	Аналіз мережевого трафіку	Перехоплення та аналіз мережевих пакетів
API Monitor	Контроль викликів API-функцій	Аналіз параметрів функцій та взаємодії з ОС
Frida	Динамічне інструментування програм	Hooking функцій, модифікація поведінки під час виконання
Cuckoo Sandbox	Поведінковий аналіз програм	Автоматизований аналіз шкідливого ПЗ у sandbox-середовищі
Process Hacker	Аналіз процесів і пам’яті	Контроль процесів, DLL, потоків і використання пам’яті
ProcDump	Створення дампів пам’яті процесів	Аналіз аварійного завершення та стану процесів

Наведені інструменти можуть застосовуватися як окремо, так і у комплексі. Наприклад, налагоджувачі дозволяють аналізувати внутрішню логіку роботи програми, Process Monitor – контролювати взаємодію з файловою системою та реєстром, а Wireshark – досліджувати мережеву активність програмного забезпечення. Поєднання кількох засобів динамічного аналізу дозволяє отримати більш повну картину функціонування програмної реалізації [48].

x64dbg – сучасний безкоштовний налагоджувач для операційних систем Windows, орієнтований на динамічний аналіз та reverse engineering 32- і 64-бітних програм. Інструмент підтримує покрокове трасування, програмні та апаратні точки зупину, аналіз пам’яті, стеку та регістрів процесора. x64dbg широко використовується для дослідження механізмів захисту, unpacking,

аналізу шкідливого програмного забезпечення та вивчення алгоритмів роботи програм [50].

WinDbg – професійний налагоджувач компанії Microsoft, призначений для аналізу користувацького та kernel-mode коду. Інструмент дозволяє досліджувати драйвери, ядро операційної системи, аварійні дампи пам’яті та системні помилки. WinDbg активно застосовується під час аналізу критичних помилок Windows, дослідження драйверів та низькорівневого аналізу системних компонентів [51].

OllyDbg – класичний налагоджувач для 32-бітних програм Windows, орієнтований на інтерактивний аналіз машинного коду. Інструмент підтримує дизасемблювання, трасування виконання програми, модифікацію пам’яті та аналіз API-викликів. Незважаючи на обмеження щодо підтримки 64-бітних програм, OllyDbg залишається популярним засобом навчального reverse engineering.

Process Monitor (ProcMon) – засіб моніторингу системної активності із пакета Sysinternals. Інструмент дозволяє відстежувати операції файлової системи, системного реєстру, створення процесів і потоків у режимі реального часу. ProcMon широко використовується для виявлення прихованої активності програм, аналізу помилок доступу та дослідження взаємодії програмного забезпечення з операційною системою [52].

Wireshark – мережевий аналізатор, призначений для перехоплення та дослідження мережевого трафіку. Інструмент дозволяє аналізувати структуру пакетів, мережеві протоколи, підключення та обмін даними між вузлами мережі. Wireshark використовується для аналізу мережевої активності програм, виявлення підозрілих з’єднань та дослідження мережевих атак [53].

API Monitor – спеціалізований інструмент для контролю викликів API-функцій Windows. Програма дозволяє відстежувати параметри функцій, результати їх виконання та взаємодію програмного забезпечення з операційною системою. API Monitor особливо ефективний під час аналізу механізмів автентифікації, роботи з файлами, мережею та криптографічними бібліотеками.

Frida – сучасний framework для динамічного інструментування програмного забезпечення. Інструмент дозволяє перехоплювати виклики функцій, змінювати поведінку програми під час виконання та виконувати hooking системних API. Frida підтримує аналіз Windows, Linux, Android та iOS і широко використовується у mobile reverse engineering та malware analysis [54].

Cuckoo Sandbox – система автоматизованого поведінкового аналізу програмного забезпечення у sandbox-середовищі. Інструмент дозволяє запускати потенційно небезпечні програми у контрольованому середовищі та автоматично фіксувати:

– файлову активність;
– мережеві з’єднання;
– зміни реєстру;
– створення процесів;
– використання системних ресурсів.

Cuckoo Sandbox широко застосовується для аналізу шкідливого програмного забезпечення та дослідження підозрілих виконуваних файлів [55].

Process Hacker – інструмент для контролю процесів, потоків, бібліотек DLL та використання пам’яті. Засіб дозволяє досліджувати структуру процесів, виявляти приховані модулі та аналізувати взаємодію між компонентами програмного забезпечення.

ProcDump – консольний інструмент Sysinternals для створення дампів пам’яті процесів. Використовується для аналізу аварійного завершення програм, зависань та дослідження внутрішнього стану процесу у визначений момент часу [52].

Таким чином, сучасні інструментальні засоби динамічного аналізу дозволяють комплексно досліджувати поведінку програмного забезпечення під час виконання, аналізувати взаємодію програм з операційною системою та виявляти механізми захисту, приховану функціональність або потенційно небезпечну активність.

Додаткові відомості щодо безкоштовних налагоджувачів, засобів трасування та моніторингу програмної активності можуть бути отримані на спеціалізованих ресурсах, зокрема thefreecountry.com (рис. 6.12), де представлено добірки інструментів для контролю пам’яті, API-трасування та поведінкового аналізу програмного забезпечення.

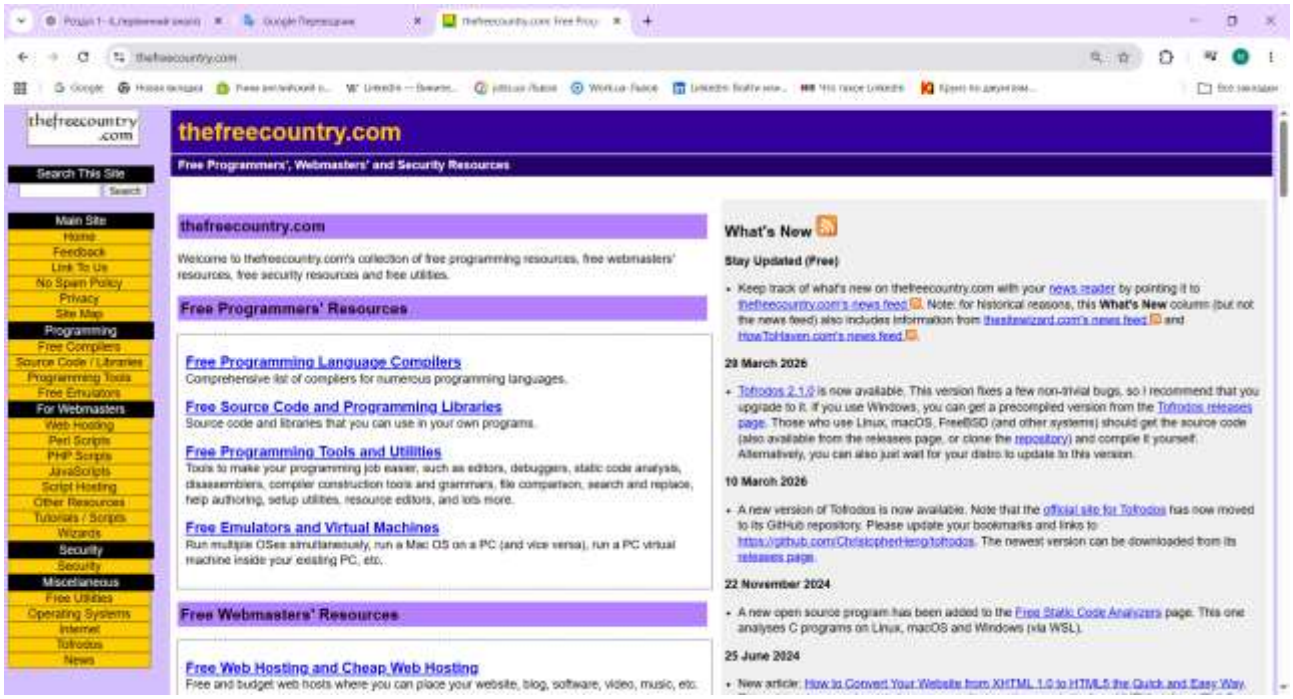


Рисунок 6.12. Приклад спеціалізованого ресурсу

6.4. ПЕРЕВАГИ ТА ОБМЕЖЕННЯ ДИНАМІЧНОГО АНАЛІЗУ

Динамічний аналіз є одним із найбільш ефективних методів дослідження програмного забезпечення, оскільки дозволяє спостерігати фактичну поведінку програми під час її виконання. На відміну від статичного аналізу, який досліджує структуру виконуваного файлу без запуску, динамічний метод забезпечує можливість контролю роботи програми у реальному середовищі виконання [19, 48].

До основних переваг динамічного аналізу належать:

– можливість спостереження реального стану пам’яті, регістрів процесора та стеку;

– дослідження поведінки програми під час виконання;

– можливість виявлення прихованої або умовно активованої функціональності;

– аналіз взаємодії програми з операційною системою;

– контроль файлової, мережевої та процесної активності;

– можливість дослідження механізмів захисту та перевірки цілісності;

– ефективне поєднання зі статичним аналізом та експериментальними методами;

– можливість аналізу поведінки шкідливого програмного забезпечення у контрольованому середовищі.

Динамічний аналіз дозволяє локалізувати функції, що реалізують механізми автентифікації, шифрування, контролю доступу або прихованої взаємодії з мережею [18, 48]. Крім того, використання налагоджувачів та засобів моніторингу дає змогу визначати послідовність викликів функцій, аналізувати структуру алгоритмів та контролювати зміну параметрів програми під час виконання.

Разом із перевагами динамічний аналіз має і певні обмеження. До них належать [49, 55]:

– залежність результатів аналізу від конкретного середовища виконання;

– необхідність використання достатнього набору вхідних даних для активування різних гілок виконання програми;

– складність аналізу програм, що використовують механізми anti-debug та anti-VM;

– можливість приховування функціональності до моменту виконання визначених умов;

– ускладнення аналізу у випадках використання пакування, обфускації або самозмінюваного коду;

– значні витрати часу на покрокове трасування складних програм;

– потреба у використанні ізольованого середовища дослідження .

Ефективність динамічного аналізу також може знижуватися у випадках використання програмами механізмів виявлення налагоджувачів, динамічного

формування коду або шифрування критичних ділянок пам’яті. У таких ситуаціях доцільно комбінувати динамічний та статичний методи аналізу [19, 44].

Некоректне використання налагоджувачів або дослідження потенційно небезпечного програмного забезпечення без належної ізоляції може призвести до нестабільної роботи операційної системи, аварійного завершення програм або пошкодження досліджуваного середовища. Тому динамічний аналіз рекомендується проводити у sandbox-системах, віртуальних машинах або спеціально підготовлених тестових середовищах [49, 55].

Таким чином, динамічний аналіз є потужним інструментом дослідження програмного забезпечення, який дозволяє виявляти поведінкові особливості програм та аналізувати їх функціонування у процесі виконання. Найбільша ефективність досягається при комплексному використанні динамічного, статичного та експериментального підходів [18, 19].

6.5. ПРАКТИЧНІ АСПЕКТИ ЗАСТОСУВАННЯ ДИНАМІЧНОГО АНАЛІЗУ

Методи динамічного аналізу широко застосовуються у сфері інформаційної та кібербезпеки для дослідження поведінки програмного забезпечення, виявлення прихованої функціональності та аналізу механізмів захисту [18, 49]. Практичне використання динамічного аналізу охоплює як дослідження легітимних програмних систем, так і аналіз потенційно небезпечного програмного забезпечення.

Одним із найбільш поширених напрямів застосування динамічного аналізу є дослідження шкідливого програмного забезпечення. У цьому випадку аналіз дозволяє виявляти мережеву активність шкідливих програм, контролювати створення та модифікацію файлів, досліджувати зміни системного реєстру, аналізувати механізми закріплення у системі, визначати способи приховування діяльності або обходу механізмів захисту [18, 38, 39].

Для таких досліджень часто використовуються sandbox-системи, які дозволяють запускати програмне забезпечення у контрольованому середовищі без ризику пошкодження основної операційної системи [49, 55].

Динамічний аналіз також активно застосовується під час дослідження механізмів автентифікації, ліцензійного контролю та захисту програмного забезпечення. Аналіз виконання програми дозволяє локалізувати функції перевірки ліцензій, механізми контролю цілісності та процедури шифрування даних [19, 48].

У процесі аналізу мережевих програмних засобів динамічний метод дозволяє досліджувати мережеві протоколи, структуру пакетів, канали обміну даними, механізми автентифікації, взаємодію із віддаленими серверами [53].

Поєднання налагоджувачів, мережевих аналізаторів та засобів моніторингу системної активності дозволяє комплексно досліджувати поведінку програмних реалізацій у реальному середовищі виконання [50, 52, 53].

Практичне значення динамічного аналізу особливо зростає у випадках використання програмами пакування, обфускації, динамічного завантаження бібліотек, самозмінюваного коду, механізмів anti-debug або anti-VM.

У таких ситуаціях саме динамічний аналіз дозволяє спостерігати реальну поведінку програми після розпакування або активації прихованої функціональності [19, 33].

Для підрозділів цивільного захисту методи динамічного аналізу можуть застосовуватися під час:

– дослідження програмних компонентів автоматизованих систем управління;

– перевірки мережевої активності спеціалізованого програмного забезпечення;

– аналізу оновлень та модулів інформаційних систем;

– виявлення потенційно небезпечних або несанкціоновано модифікованих програмних компонентів;

– тестування механізмів захисту критичних інформаційних систем.

Важливою умовою практичного застосування динамічного аналізу є дотримання вимог інформаційної безпеки та використання контрольованих середовищ виконання, що мінімізують ризики пошкодження інформаційної інфраструктури.

Таким чином, динамічний аналіз є важливим практичним інструментом дослідження сучасного програмного забезпечення, що дозволяє аналізувати поведінку програм у процесі виконання, виявляти механізми захисту та досліджувати приховану функціональність програмних систем.

6.6. ПРИКЛАД ЗАСТОСУВАННЯ ДИНАМІЧНОГО МЕТОДУ

6.6.1. Постановка задачі та програмне середовище

У якості прикладу застосування динамічного методу розглянемо дослідження програмної реалізації, що виконує шифрування та дешифрування текстових даних із використанням криптографічного алгоритму AES.

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;

namespace DynamicAnalysisExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter text:");
            string inputText = Console.ReadLine();

            Console.WriteLine("Enter key:");
            string keyText = Console.ReadLine();

            byte[] key = PrepareKey(keyText);

            string encrypted = Encrypt(key, inputText);
            Console.WriteLine("Encrypted text:");
            Console.WriteLine(encrypted);

            string decrypted = Decrypt(key, encrypted);
            Console.WriteLine("Decrypted text:");
            Console.WriteLine(decrypted);
```

```
            Console.ReadLine();
        }

        static byte[] PrepareKey(string keyText)
        {
            using (SHA256 sha256 = SHA256.Create())
            {
                return
sha256.ComputeHash(Encoding.UTF8.GetBytes(keyText));
            }
        }

        static string Encrypt(byte[] key, string plainText)
        {
            using (Aes aes = Aes.Create())
            {
                aes.Key = key;
                aes.Mode = CipherMode.ECB;
                aes.Padding = PaddingMode.PKCS7;

                ICryptoTransform encryptor = aes.CreateEncryptor();

                byte[] inputBytes = Encoding.UTF8.GetBytes(plainText);

                using (MemoryStream memoryStream = new MemoryStream())
                using      (CryptoStream      cryptoStream      =      new
CryptoStream(memoryStream, encryptor, CryptoStreamMode.Write))
                {
                    cryptoStream.Write(inputBytes, 0, inputBytes.Length);
                    cryptoStream.FlushFinalBlock();

                    return
Convert.ToBase64String(memoryStream.ToArray());
                }
            }
        }

        static string Decrypt(byte[] key, string encryptedText)
        {
            using (Aes aes = Aes.Create())
            {
                aes.Key = key;
                aes.Mode = CipherMode.ECB;
                aes.Padding = PaddingMode.PKCS7;

                ICryptoTransform decryptor = aes.CreateDecryptor();

                byte[]                  encryptedBytes                  =
Convert.FromBase64String(encryptedText);
```

```
                using (MemoryStream memoryStream = new
MemoryStream(encryptedBytes))
                using (CryptoStream cryptoStream = new
CryptoStream(memoryStream, decryptor, CryptoStreamMode.Read))
                using (StreamReader reader = new
StreamReader(cryptoStream))
                {
                    return reader.ReadToEnd();
                }
            }
        }
    }
}
```

Програма реалізована на платформі .NET Framework та використовує бібліотеку System.Security.Cryptography [56].

Метою дослідження є:

– локалізація функцій шифрування та дешифрування;

– аналіз параметрів криптографічних викликів;

– визначення способу передавання ключа шифрування;

– дослідження логіки роботи криптографічного модуля.

Для проведення динамічного аналізу використовується налагоджувач dnSpy, який дозволяє:

– виконувати дизасемблювання IL-коду;

– встановлювати точки зупину;

– здійснювати покрокове трасування;

– аналізувати стек викликів та локальні змінні;

– контролювати параметри функцій під час виконання програми.

Після завантаження досліджуваної програми у dnSpy у головному вікні відображається дизасембльований IL-код та структура модулів програми (рис. 6.13).

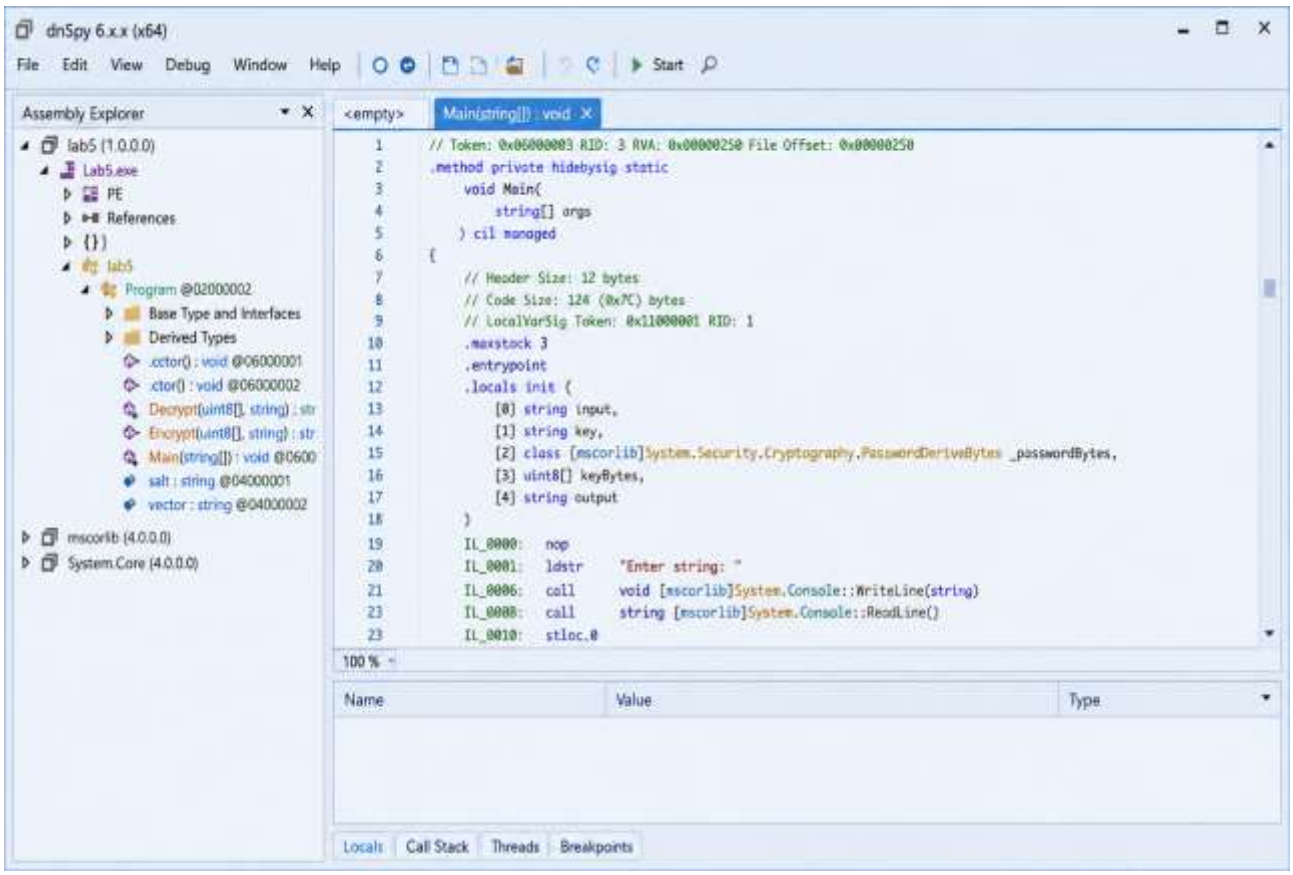


Рисунок 6.13. Головне вікно dnSpy

6.6.2. Застосування методу маяків

На першому етапі динамічного аналізу застосовується метод маяків. Його сутність полягає у встановленні точок зупину на функції або API-виклики, призначення яких можна визначити без повного аналізу програми [44, 48].

Під час дослідження було встановлено, що у програмі присутня функція Encrypt, назва якої прямо вказує на її можливе призначення. На виклик цієї функції встановлюється точка зупину (рис. 6.14).



Рисунок 6.14. Встановлення точки зупину на функцію Encrypt

Після запуску програми вводяться тестові дані:

– текст: Test string;

– ключ: 123456.

У вікні Locals налагоджувача спостерігається передавання введених значень до функції Encrypt. Покрокове трасування дозволило встановити, що ключ перетворюється у масив байтів та використовується під час виклику криптографічних процедур [56].

Подальший аналіз функції Encrypt показав використання класу System.Security.Cryptography.AesManaged із режимом ECB (рис. 6.15). Шифрування даних виконується через об’єкт CryptoStream.

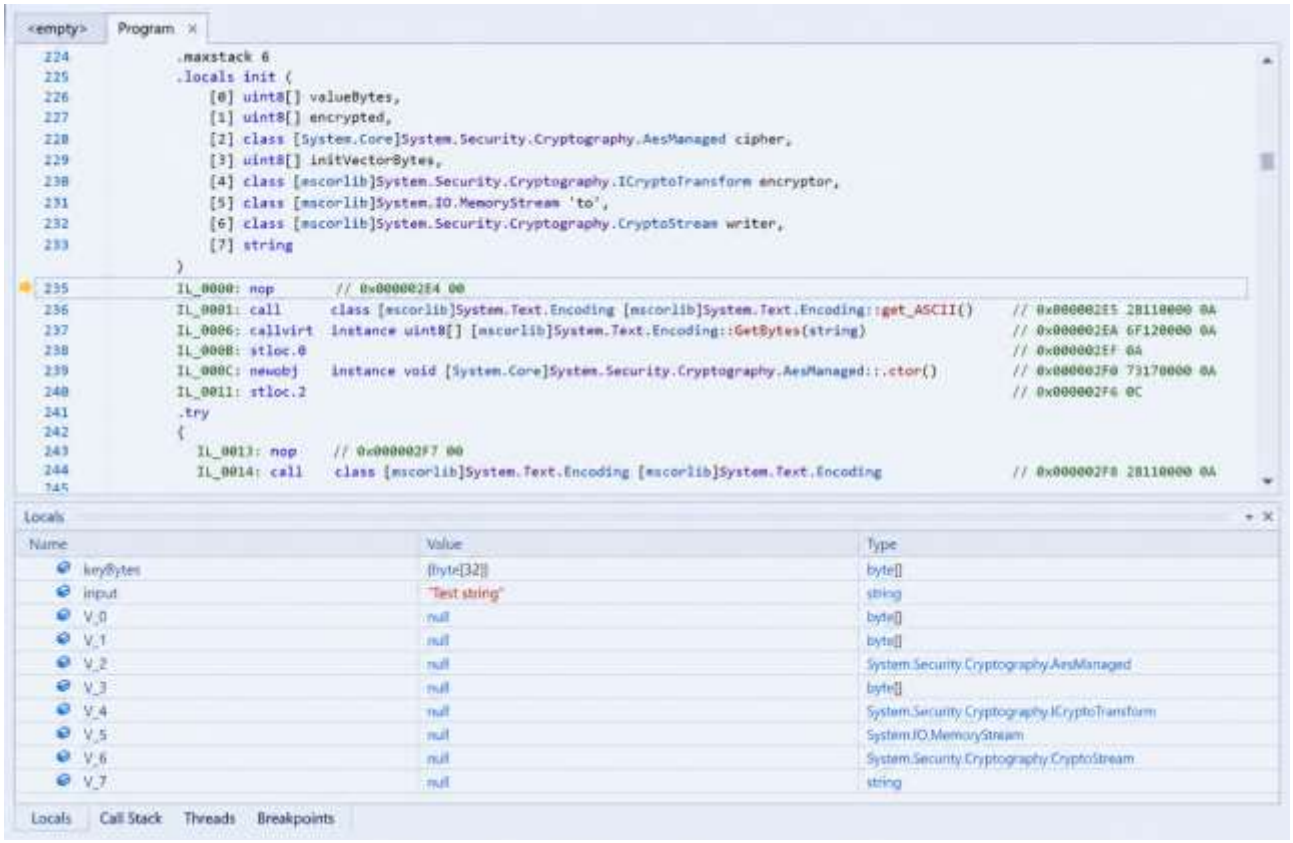


Рисунок 6.15. Аналіз функції Encrypt

Після виконання функції FlushFinalBlock (рис. 6.16) у пам’яті формується зашифрований рядок, що дозволяє локалізувати завершальний етап криптографічного перетворення.

```
279         /* 0x0000033A 06          */ IL_0056: ldloc.0
280         /* 0x0000033B 16          */ IL_0057: ldc.i4.0
281         /* 0x0000033C 06          */ IL_0058: ldloc.0
282         /* 0x0000033D 8E          */ IL_0059: ldlen
283         /* 0x0000033E 69          */ IL_005A: conv.i4
284         /* 0x0000033F 6F1D00000A  */ IL_005B: callvirt    instance void [mscorlib]System.IO.Stream::Write(uint8[],
                int32, int32)
285         /* 0x00000344 00          */ IL_0060: nop
286         /* 0x00000345 1106        */ IL_0061: ldloc.s    writer
287         /* 0x00000346 6F1E00000A  */ IL_0062: callvirt    instance void [mscorlib]System.Security.Cryptography.
                                                 CryptoStream::FlushFinalBlock()
288         /* 0x0000034C 00          */ IL_0066: nop
289         /* 0x0000034D 1105        */ IL_0069: ldloc.s    'to'
290         /* 0x0000034F 6F1F00000A  */ IL_006A: callvirt    instance uint8[] [mscorlib]System.IO.MemoryStream::ToArray()
291         /* 0x00000354 08          */ IL_006F: stloc.1
292         /* 0x00000355 00          */ IL_0071: nop
293         /* 0x00000356 DE00        */ IL_0072: leave.s    IL_0081
294     } // end .try
295     finally
296     {
297         /* 0x00000358 1106        */ IL_0074: ldloc.s    writer
298         /* 0x0000035A 2C08        */ IL_0076: brfalse.s  IL_0080
299         /* 0x0000035C 1106        */ IL_0078: ldloc.s    writer
300         /* 0x0000035E 8A          */ IL_007A: callvirt    instance void [mscorlib]System.IDisposable::Dispose()
```

Locals

Name	Value	Type
keyBytes	{byte[32]}	byte[]
input	"Test string"	string
V_0	{byte[11]}	byte[]
[0]	84	byte
[1]	101	byte
[2]	115	byte
[3]	116	byte
[4]	32	byte
[5]	115	byte

Рисунок 6.16. Результат виконання функції FlushFinalBlock

6.6.3. Застосування методу Step-Trace першого етапу

На наступному етапі використовується метод Step-Trace першого етапу. Його метою є визначення функцій, що безпосередньо реалізують необхідну функціональність [44, 48].

За допомогою режиму Step виконується покрокове проходження програми. У процесі трасування аналізуються:

- локальні змінні;
- параметри функцій;
- зміни стеку викликів;
- результати виконання окремих команд IL-коду.

У вікні Locals було встановлено, що після виклику функції Encrypt значення змінної змінюється на зашифрований рядок (рис. 6.17).

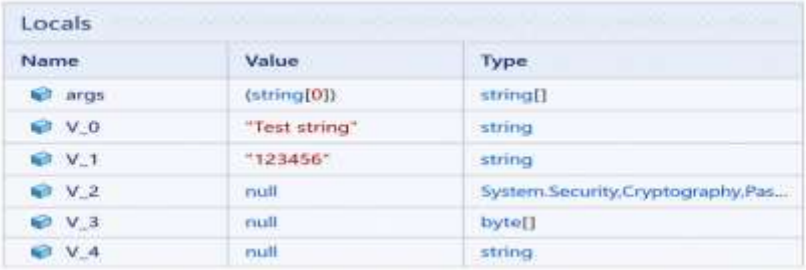


Рис. 6.17. Аналіз локальних змінних у режимі Step-Trace

Подальший аналіз дозволив встановити наявність функції Decrypt, яка використовується для дешифрування отриманого результату. Для переходу всередину функції застосовується режим Trace [56].

Під час аналізу параметрів функції Decrypt (рис. 6.18) встановлено, що до неї передаються:

– ключ шифрування;

– зашифрований текст.

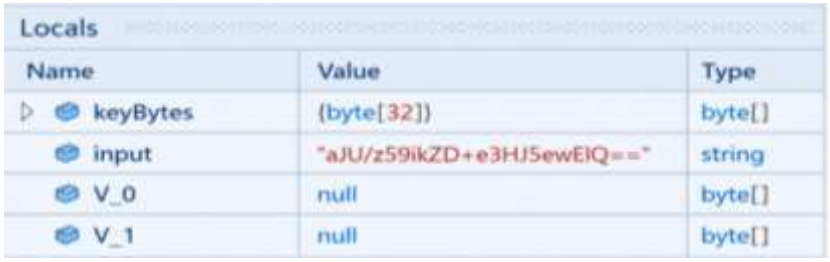


Рис. 6.18. Вхідні параметри функції Decrypt

6.6.4. Аналіз криптографічного алгоритму

Подальше покрокове трасування дозволило визначити використання функцій бібліотеки System.Security.Cryptography. Аналіз IL-коду показав, що дешифрування реалізується через виклик функції CryptoStream::Read [56].

У процесі дослідження було встановлено:

– створення потоку дешифрування;

– використання буферів пам’яті;

– перетворення масиву байтів у текстовий рядок;

– передачу даних між криптографічними функціями.

Аналіз викликів та параметрів дозволив підтвердити використання алгоритму AES у режимі ECB та відновити загальну логіку роботи криптографічного модуля (рис. 6.19).



Рис. 6.19. Використання CryptoStream під час дешифрування

Після завершення дешифрування результат перетворюється у текстовий рядок та передається до функції виведення на екран (рис. 6.20).



Рис. 6.20. Результат роботи функції дешифрування

6.6.5. Особливості застосування апаратних точок зупину

Під час дослідження встановлено, що використання апаратних точок зупину у середовищі .NET Framework має певні обмеження. Це пов’язано із особливостями роботи CLR (Common Language Runtime), динамічним розподілом пам’яті та використанням механізму автоматичного керування пам’яттю (Garbage Collector) [26].

У середовищі CLR розташування об’єктів у пам’яті може змінюватися під час виконання програми, що ускладнює стабільне використання апаратних точок зупину для контролю конкретних адрес пам’яті.

Тому для дослідження програм, реалізованих на платформі .NET, більш ефективним є використання [56]:

– покрокового трасування;
– аналізу IL-коду;
– контролю локальних змінних;
– моніторингу викликів функцій криптографічних бібліотек.

6.6.6. Висновки за результатами дослідження

Проведене дослідження показало, що застосування динамічного методу дозволяє ефективно локалізувати функції шифрування та дешифрування, визначати параметри криптографічних викликів і відновлювати логіку роботи програмного забезпечення навіть за відсутності вихідного коду.

Використання налагоджувача dnSpy, методів маяків та Step-Trace дозволило:

– виявити функції Encrypt та Decrypt;
– визначити спосіб передавання ключа шифрування;
– встановити використання алгоритму AES;
– дослідити взаємодію із криптографічними бібліотеками;
– проаналізувати процес шифрування та дешифрування даних.

Отримані результати підтверджують ефективність динамічного аналізу для дослідження сучасних програмних реалізацій, зокрема програм, що використовують криптографічні механізми захисту інформації.

ПИТАННЯ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ ЗА ТЕМОЮ 6

1. Що таке динамічний аналіз програмного забезпечення та у чому полягає його принцип дії?
2. Які основні можливості та функції налагоджувачів використовуються під час динамічного аналізу програм?
3. Назвіть основні переваги та обмеження динамічного аналізу.
4. Які інструментальні засоби застосовуються для динамічного аналізу програмного забезпечення?
5. У чому полягає сутність методу маяків?
6. У чому полягають особливості застосування методу Step-Trace на першому та другому етапах динамічного аналізу?
7. Що таке апаратна точка зупину і коли вона використовується?
8. Які завдання вирішуються на третьому етапі аналізу програми?
9. Які особливості використання динамічного аналізу у середовищі .NET Framework?
10. Які механізми можуть ускладнювати проведення динамічного аналізу програмного забезпечення?
11. Чому для проведення динамічного аналізу доцільно використовувати sandbox- або віртуалізовані середовища?

ПРАКТИЧНІ ЗАВДАННЯ ДО ТЕМИ 6

ЗАВДАННЯ 1. ТРИСТАДІЙНИЙ ДИНАМІЧНИЙ АНАЛІЗ МЕХАНИЗМУ ПЕРЕВІРКИ ЛІЦЕНЗІЇ

Постановка завдання

На основі підрозділів 6.2.2, 6.2.3 та 6.2.4 посібника проведіть тристадійний динамічний аналіз виконуваного файлу, що реалізує механізм перевірки ліцензії або автентифікації.

Оберіть один із наведених типів виконуваних модулів без доступу до вихідного коду:

а) захищена 32-бітна утиліта Windows, що перевіряє введений серійний номер перед активацією функцій;

б) програмний компонент системи цивільного захисту, що звертається до реєстру або файлу-ключа для валідації ліцензії;

в) мережевий застосунок, що надсилає запит на сервер активації та обробляє відповідь.

Для обраного виконуваного файлу:

1) Застосуйте метод маяків (підрозділ 6.2.2): визначте щонайменше чотири кандидати на роль «маяка» серед API-функцій, що викликаються модулем, та заповніть таблицю із зазначенням бібліотеки, призначення і пріоритету маяка; встановіть програмну точку зупину (int3 / байт 0xCC) на обраній функції-маяку та зафіксуйте момент її спрацювання.

2) Застосуйте Step-Trace I (підрозділ 6.2.2): після спрацювання точки зупину (software breakpoint) виконуйте код у покроковому режимі (Step/Trace); складіть фрагмент трасувального журналу (не менше восьми рядків) із зазначенням адрес, інструкції та стану регістрів; ідентифікуйте функцію, що виконує безпосередню перевірку ключа.

3) Застосуйте метод апаратної точки зупину та Step-Trace II (підрозділ 6.2.3): визначте адресу буфера, в який зчитується ліцензійний ключ або пароль; встановіть апаратну точку зупину через регістри налагодження процесора DR0–

DR3 на читання з цього буфера; складіть таблицю подій Step-Trace II із зазначенням адреси спрацювання, відповідної інструкції та вмісту регістрів.

4) Виконайте третю стадію аналізу (підрозділ 6.2.4): дизасемблюйте ідентифіковану функцію перевірки; виокремте ключову інструкцію порівняння; сформулюйте гіпотезу про алгоритм захисту та наведіть план її верифікації.

Приклад розв'язку

Досліджуваний файл: license_check.exe – гіпотетичний 32-бітний модуль реєстрації компонента системи цивільного захисту.

Інструмент: OllyDbg 1.10 (Win x86).

Сценарій запуску: програма відкриває вікно реєстрації і запитує серійний номер. При введенні «1234-5678-ABCD-EFGH» з'являється повідомлення «Ліцензійний ключ недійсний».

СТАДІЯ I – МЕТОД МАЯКІВ + STEP-TRACE I

Крок 1. Виявлення кандидатів на роль маяка

Відкрийте license_check.exe в OllyDbg, перейдіть до View → Executable modules → user32.dll → Names. Зафіксуйте усі імпортовані функції модуля та виберіть кандидати-маяки.

Таблиця 6.2

Порівняльна таблиця:

API-функція	Бібліотека	Призначення	Пріоритет маяка
GetDlgItemTextA	user32.dll	Зчитування введеного серійного номера з поля	Високий
MessageBoxA	user32.dll	Відображення повідомлення про помилку ліцензії	Середній
RegOpenKeyExA	advapi32.dll	Відкриття ключа реєстру для перевірки активації	Середній
lstrcmpA	kernel32.dll	Порівняння рядків (потенційна перевірка ключа)	Низький

Крок 2. Встановлення точки зупину на маяк з вищим пріоритетом

У вікні дизасемблера OllyDbg перейдіть до адреси точки входу GetDlgItemTextA у таблиці імпорту (IAT). Натисніть F2 – за адресою буде встановлено програмну точку зупину (перший байт інструкції замінено на 0xCC / int3).

Крок 3. Запуск та спрацювання маяка

Запустіть програму (F9), введіть серійний номер. Програма зупиняється на GetDlgItemTextA. У вікні «Stack» видно параметри виклику: дескриптор поля вводу (HWND=0x001234), ідентифікатор елемента (nIDDlgItem=0x0064), покажчик на буфер (lpString=0x00415B20), максимальна довжина (nMaxCount=0x0020).

Крок 4. Step-Trace I після маяка

Виконуйте трасування командою F8 (Step Over) / F7 (Step Into) до ідентифікації функції валідації.

Таблиця 6.3

Адреси інструкцій:

Адреса	Інструкція	Реєстри / Примітки
00401234	PUSH EBP	ESP=0018FF7C
00401235	MOV EBP,ESP	EBP=0018FF7C
00401237	SUB ESP,20h	ESP=0018FF5C (виділення кадру стеку)
0040123A	MOV EAX,[EBP+08h]	EAX=0x00415B20 (адреса буфера серійного номера)
0040123D	PUSH EAX	передача буфера наступній функції
0040123E	CALL 00402100	>>> виклик validate_license() <<<
00401243	TEST EAX,EAX	перевірка повернутого значення (0=невірна)
00401245	JNZ 00401280	перехід до виводу помилки при EAX=0

Ідентифіковано цільову функцію: validate_license() за адресою 0x00402100.

СТАДІЯ II – АПАРАТНА ТОЧКА ЗУПИНУ + STEP-TRACE II

Крок 1. Визначення адреси буфера

Зі Step-Trace I встановлено: буфер серійного номера знаходиться за адресою 0x00415B20 (поточна сесія, ASLR вимкнено для навчального прикладу).

Крок 2. Встановлення апаратної точки зупину через DR

У OllyDbg: правий клік у вікні дизасемблера → Breakpoint → Hardware, on read → введіть адресу 0x00415B20, розмір 4 байти. Це налаштовує регістр DR0=0x00415B20, відповідне поле DR7 задає умову «читання 4-байтового рядка» (RW=01, LEN=11).

Крок 3. Трасування після спрацювання апаратної точки зупину (Step-Trace II)

Таблиця 6.4

Адреси інструкцій:

Адреса	Інструкція	DR-подія	Вміст регістрів
00402115	MOV EAX,[ESI]	DR0 READ @ 0x00415B20	EAX=0x31323334 ("1234")
00402119	CMP EAX,0x4E4D4C4B	—	Порівняння з очікуваним словом
0040211C	JNZ 00402140	—	Стрибок при невідповідності
00402120	MOV EAX,[ESI+04h]	DR0 READ @ 0x00415B24	EAX=0x35363738 ("5678")
00402124	CMP EAX,0x52515049	—	Другий фрагмент ключа
00402127	JNZ 00402140	—	Перехід «невірний ключ»

Висновок стадії II: validate_license() читає серійний номер фрагментами по 4 байти та порівнює з константами, вшитими у код. Процес читання буфера відстежено через апаратну точку зупину без жодної модифікації коду модуля.

СТАДІЯ III – ДИЗАСЕМБЛЮВАННЯ ТА АНАЛІЗ ФУНКЦІЇ

Після локалізації функції validate_license() виконується її дизасемблювання та аналіз логіки перевірки серійного номера.

Крок 1. Дизасемблювання validate_license() (0x00402100)

```
; Пролог
00402100: PUSH EBP
00402101: MOV EBP,ESP
00402103: PUSH ESI
00402104: MOV ESI,[EBP+08h]      ; ESI = покажчик на серійний номер

; Перевірка першого фрагменту (байти 0-3)
00402107: MOV EAX,[ESI]          ; зчитати 4 байти
0040210A: CMP EAX,4E4D4C4Bh     ; порівняти з еталоном фрагмент 1
0040210F: JNZ 0040213Ch          ; якщо ≠ → повернути 0
```

```
; Перевірка другого фрагменту (байти 5-8, пропускаючи «-»)
00402112: MOV EAX,[ESI+05h]
00402115: CMP EAX,52515049h      ; фрагмент 2
0040211A: JNZ 0040213Ch

; Перевірка третього фрагменту (байти 10-13)
0040211C: MOV EAX,[ESI+0Ah]
0040211F: CMP EAX,46454443h      ; фрагмент 3 ("CDEF")
00402124: JNZ 0040213Ch

; Перевірка четвертого фрагменту (байти 15-18)
00402126: MOV EAX,[ESI+0Fh]
00402129: CMP EAX,4A494847h      ; фрагмент 4 ("GHIJ")
0040212E: JNZ 0040213Ch

; Всі фрагменти відповідають → ліцензія дійсна
00402130: MOV EAX,01h
00402132: JMP 0040213Eh

; Повернення «ліцензія невірна»

0040213Ch: XOR EAX,EAX
0040213Eh: POP ESI
0040213Fh: POP EBP
00402140h: RET
```

Крок 2. Гіпотеза про алгоритм захисту

Функція validate_license() виконує пряме порівняння чотирьох 4-байтових фрагментів серійного номера з константами (plaintext), жорстко закодованими в секції .text. Алгоритм не використовує хешування – захист є надзвичайно слабким (клас «механізм перевірки серійного номера» без криптографічного підсилення, підрозділ 2.3 посібника).

Крок 3. Верифікація гіпотези

Для перевірки гіпотези: запустіть license_check.exe повторно з серійним номером «NMLK-RQPI-CDEF-GHIJ» (байтові значення констант у порядку little-endian, характерному для архітектури x86). Якщо програма видає підтвердження ліцензії – гіпотезу підтверджено. Очікуваний результат: успішна активація, що свідчить про відсутність додаткового захисного рівня (хешування, salt, лічильник спроб).

Підсумковий висновок: тристадійний динамічний аналіз (підрозділ 6.2) дозволив ідентифікувати механізм захисту без доступу до вихідного коду. Метод маяків локалізував функцію перевірки; Step-Trace I та апаратна точка зупину розкрили звернення до буфера; дизасемблювання підтвердило відсутність криптографічного захисту. Для системи цивільного захисту такий рівень захисту є неприйнятним: рекомендовано застосування хешування ключа з сіллю та обмеження кількості спроб введення.

ЗАВДАННЯ 2.ПОРІВНЯЛЬНИЙ АНАЛІЗ ІНСТРУМЕНТІВ ДИНАМІЧНОГО АНАЛІЗУ

Мета: Ознайомитися з сучасними інструментальними засобами динамічного аналізу програмного забезпечення, дослідити їх функціональні можливості та обґрунтувати вибір засобів для вирішення практичних задач аналізу безпеки програмних реалізацій.

Постановка завдання

На основі підрозділів 6.3 «Інструментальні засоби динамічного аналізу» та 6.4 «Переваги та обмеження динамічного аналізу» посібника проведіть порівняльний аналіз інструментів динамічного аналізу та обґрунтуйте вибір для конкретного сценарію дослідження програмного забезпечення цивільного захисту.

Оберіть один із наведених сценаріїв аналізу:

а) 32-бітний Windows-застосунок моніторингу умовної системи цивільного захисту, що систематично аварійно завершується через пошкодження пам'яті;

б) Linux-компонент серверної частини системи цивільного захисту із підозрюваними витоками пам'яті під навантаженням;

в) багатопотоковий Windows-сервіс системи цивільного захисту, що проявляє ознаки race condition під час паралельної обробки подій.

Для обраного сценарію:

1) На основі таблиці інструментів (підрозділ 6.3 посібника) складіть порівняльну таблицю щонайменше п'яти інструментів динамічного аналізу із зазначенням: платформи підтримки, типів точок зупину, ключових можливостей та обмежень.

2) Оберіть найбільш придатний інструмент для обраного сценарію та обґрунтуйте вибір, спираючись на переваги та недоліки методу (підрозділ 6.4 посібника); поясніть, чому інші розглянуті інструменти є менш придатними.

3) Опишіть покрокову процедуру налаштування та запуску обраного інструменту для аналізу заданого компонента: вкажіть команди запуску, ключові параметри та критерії зупину аналізу; заповніть таблицю очікуваних подій або артефактів, що будуть зафіксовані інструментом.

4) Сформулюйте висновок: які категорії вразливостей може виявити обраний інструмент, а які залишаються поза його можливостями; запропонуйте доповнюючий інструмент для покриття недоліків.

Приклад розв'язку

Обраний сценарій: а) 32-бітний Windows-застосунок моніторингу системи цивільного захисту «monitor32.exe», що систематично аварійно завершується (Access Violation) при обробці XML-звітів про надзвичайні ситуації.

Крок 1. Порівняльний аналіз інструментів

Таблиця 6.5

Порівняльна характеристика інструментів динамічного аналізу

Інструмент	Платформа	Типи точок зупину	Ключові можливості	Обмеження
OllyDbg 1.10	Win x86 (32-bit)	Програмні (int3 / байт 0xCC), апаратні (DR0-DR3)	GUI, плагіни, аналіз API-викликів, трасування	Тільки 32-bit; не підтримує сучасний kernel-debug
x64dbg	Win x86/x64	Програмні (int3), апаратні (DR), умовні (conditional BP)	Підтримка 32/64-bit, скрипти, trace, плагіни	Не виконує автоматичного виявлення memory leaks
SoftICE	Win/Linux (ring0)	Ядерні (ring0), апаратні	Низькорівневе kernel-debugging та аналіз драйверів	Історично важливий debugger; не підтримується у

Інструмент	Платформа	Типи точок зупину	Ключові можливості	Обмеження
				сучасних версіях Windows
WinDbg	Win x86/x64	Програмні, апаратні, kernel-mode	Аналіз дампів пам’яті, драйверів та ядра ОС	Складніший для початкового навчання
Valgrind/Memcheck	Linux	Інструментування коду	Виявлення витоків пам'яті, вихід за межі масивів, use-after-free	Тільки Linux; значне сповільнення виконання програм (~10x)
Intel Thread Checker	Win/Linux	Аналіз доступу до потоків пам'яті	Виявлення race condition, deadlock	Орієнтований переважно на багатопотокові застосунки

Крок 2. Вибір інструменту та обґрунтування

Для сценарію а) обрано: OllyDbg 1.10.

Обґрунтування:

– monitor32.exe є 32-бітним Windows-застосунком, для якого OllyDbg є ефективним навчальним інструментом динамічного аналізу;

– аварійне завершення (Access Violation) свідчить про некоректне звернення до пам’яті, яке OllyDbg перехоплює як виняток першого шансу (First Chance Exception);

– апаратні точки зупину (метод Step-Trace II, підрозділ 6.2.3) дозволяють відстежувати записи у підозрілий буфер без модифікації коду програми;

– програмні точки зупину на функціях роботи з рядками (lstrcpyA, sprintf, strcpy) дозволяють локалізувати переповнення буфера до моменту збою.

Чому інші інструменти менш придатні:

– x64dbg підтримує як 32-, так і 64-бітні застосунки, однак у межах базового навчального сценарію OllyDbg має простіший інтерфейс для аналізу 32-бітних програм;

– Valgrind не підтримує Windows-застосунки, тому для monitor32.exe не використовується;

– Intel Inspector орієнтований на аналіз race condition та проблем синхронізації потоків;

– WinDbg є потужним засобом kernel- та crash-аналізу, однак потребує більш глибокої підготовки;

– SoftICE виключено із практичного використання через застарілість та несумісність із сучасними версіями Windows.

Крок 3. Процедура налаштування та запуску

Запуск OllyDbg і відкриття досліджуваного файлу:

File → Open → обрати monitor32.exe

Налаштування обробки винятків:

Options → Exceptions

→ поставити прапорець «Ignore all» для зняття стандартних винятків

Options → Exceptions → зняти прапорець «INT3 breaks» → увімкнути «Access Violations» (Налаштувати перехоплення винятків типу Access Violation для фіксації моменту порушення доступу до пам’яті).

Встановлення точок зупину на небезпечні функції роботи з рядками:

View → Executable modules → kernel32.dll → Names

Знайти: strcpy, lstrcpyA, sprintf → натиснути F2 на кожній для програмних BP

Запуск аналізу:

F9 (Run)

Після запуску:

- завантажити XML-файл великого розміру;
- відтворити аварійне завершення програми;
- зафіксувати момент спрацювання точки зупину або виникнення Access Violation.

Таблиця 6.6

Очікувані події під час аналізу monitor32.exe:

Подія	Адреса / Джерело	Зміст фіксації
BP на lstrcpyA	user32.dll → kernel32.dll	Параметри: src, dst, довжина рядка
Access Violation (збій)	0x00XXXXXX (у .text)	Адреса запису за межами буфера
Стан стеку в момент збою	ESP=0x0018FXXX	Смуга стеку викликів (для ретроаналізу SEH)

DR0 READ на буфері	0x00415C00 (буфер XML)	Виявлення некоректного читання XML-буфера

Крок 4. Висновок

OllyDbg дозволяє виявити такі категорії вразливостей:

– переповнення буфера (buffer overflow) – через AP-точки зупину та аналіз стеку;

– використання після звільнення (use-after-free) – через апаратний BP на звільнену область пам'яті;

– некоректні покажчики (null pointer dereference) – через перехоплення Access Violation4

- помилки роботи зі стеком та буфером пам’яті.

Поза можливостями OllyDbg залишаються:

– витоки пам'яті (memory leaks) – потребують тривалого моніторингу виділення/звільнення; рекомендований доповнюючий інструмент: IBM Rational Purify або Insure++ (підрозділ 6.5 посібника), що відстежують всі виклики malloc/free та виявляють «осиротілі» блоки без звільнення.

– довготривалий моніторинг виділення пам’яті;

– аналіз багатопотокових race condition.

– Для покриття цих обмежень доцільно використовувати:

– Dr. Memory – для аналізу витоків пам’яті;

– Intel Inspector – для аналізу race condition;

– WinDbg – для дослідження аварійних дампів пам’яті та kernel-mode компонентів.

Комплексне використання кількох інструментів динамічного аналізу дозволяє отримати більш повне уявлення про поведінку програмного забезпечення та підвищує ефективність виявлення вразливостей у системах цивільного захисту.

ЗАВДАННЯ 3. ДИНАМІЧНИЙ АНАЛІЗ ПІДОЗРІЛОГО КОМПОНЕНТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ У СЕРЕДОВИЩІ УМОВНОЇ СИСТЕМИ ЦИВІЛЬНОГО ЗАХИСТУ

Постановка завдання

На основі підрозділів 6.1, 6.2, 6.3 та 6.5 посібника проведіть динамічний аналіз підозрілого виконуваного модуля, що потрапив до програмного середовища підрозділу цивільного захисту.

Оберіть один із наведених сценаріїв:

а) підозрілий виконуваний файл (.exe), виявлений в оновленні програмного забезпечення системи цивільного захисту, що здійснює мережеві з'єднання поза штатним регламентом;

б) DLL-компонент, що зареєстрований у ключах автозавантаження реєстру та модифікує файлову систему;

в) модуль із зашифрованими або запакованими секціями (виявлено засобами статичного аналізу), поведінка якого невідома.

Для обраного підозрілого модуля:

1) Опишіть процедуру підготовки ізольованого середовища для безпечного динамічного аналізу: вкажіть тип середовища (sandbox, VM, фізичний стенд), необхідні інструменти моніторингу (мережевий, файловий, реєстровий, процесний) та заходи запобігання поширенню загрози.

2) Складіть таблицю мережевої активності модуля: для кожного зафіксованого з'єднання вкажіть часову мітку, протокол, адресу призначення, порт та стислий опис корисного навантаження; для кожного рядка дайте оцінку (штатна / підозріла / шкідлива) з коротким обґрунтуванням.

3) Складіть таблицю активності з реєстром та файловою системою: зафіксуйте операції (Read/Write/Create/Delete), відповідні шляхи або ключі реєстру, встановлені значення та оцінку кожної операції; виділіть ознаки закріплення (persistence) у системі.

4) На основі зібраних даних класифікуйте модуль за моделлю закладки (підрозділ 3.3 посібника): «спостерігач», «перехоплювач» або «спотворення»;

обґрунтуйте класифікацію, посилаючись на конкретні зафіксовані події; запропонуйте заходи нейтралізації.

Приклад розв'язку

Обраний сценарій: а) підозрілий файл «update_cz.exe» (розмір 284 КБ), виявлений у пакеті оновлення системи диспетчеризації цивільного захисту; статичний аналіз (pestudio) показав наявність імпорту мережевих функцій (WSASend, InternetOpenA) та рядків у стилі C2-клієнта.

Крок 1. Підготовка ізольованого середовища

Тип середовища: ізольована віртуальна машина (VMware / VirtualBox) з Windows 10 x64 (без прямого доступу до мережі Інтернет; взаємодія із зовнішнім середовищем здійснюється через контрольований моніторинговий проксі або емулятор мережевих служб).

Таблиця 6.7

Інструменти моніторингу:

Категорія	Інструмент	Призначення
Мережевий трафік	Wireshark + FakeNet-NG	Перехоплення та аналіз TCP/UDP/DNS-з'єднань
Файлова система	Process Monitor (ProcMon)	Фіксація CreateFile, WriteFile, DeleteFile
Реєстр	Process Monitor (ProcMon)	RegSetValue, RegCreateKey, RegDeleteKey
Процеси	Process Hacker	Аналіз процесів, завантажених DLL та пам’яті
Поведінковий аналіз	ANY.RUN (за наявності контрольованого доступу)	Автоматизований sandbox-звіт

Заходи безпеки:

— створення snapshot VM перед запуском досліджуваного модуля з можливістю швидкого відновлення середовища;

— мережевий інтерфейс VM працює у режимі Host-Only із FakeNet-NG як емулятором DNS/HTTP-служб;

— відключено спільні теки між гостьовою та хост-системою;

— зовнішні носії не підключено;

— заборонено використання робочих облікових записів та реальних службових даних у середовищі аналізу.

Крок 2. Аналіз мережевої активності

Таблиця 6.8

Мережева активність модуля

Час(с)	Протокол	Адреса призначення	Порт	Оцінка
0.4	DNS	8.8.8.8	53	Підозріла
1.2	TCP/HTTP	185.220.101.47	8080	Шкідлива
3.8	TCP/TLS	185.220.101.47	443	Шкідлива
12.1	DNS	8.8.8.8	53	Підозріла
12.4	TCP/HTTP	91.108.56.23	8080	Шкідлива

Коментар: з'єднання на порт 8080 із передаванням base64-кодованих даних відповідає типовому патерну ексфільтрації інформації через HTTP-тунель. TLS-з'єднання на порт 443 свідчить про використання зашифрованого C2-каналу. Повторні DNS-запити до зовнішніх серверів можуть вказувати на спроби динамічного отримання адрес керування або перевірки доступності командного сервера.

Крок 3. Аналіз активності реєстру та файлової системи

Таблиця 6.9

Активність файлової системи та реєстру

Операція	Шлях / Ключ реєстру	Значення / Вміст	Оцінка
RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Run\CZUpdate	"C:\ProgramData\czupd\update_cz.exe"	Закріплення (persistence)
RegSetValue	HKLM\SYSTEM\CurrentControlSet\Services\CZUpdateSvc\ImagePath	"C:\ProgramData\czupd\update_cz.exe"	Закріплення (служба)
CreateFile	C:\ProgramData\czupd\update_cz.exe	Копіює себе (самокопіювання)	Підозріла
CreateFile	C:\ProgramData\czupd\cfg.bin	Зашифрований файл конфігурації C2	Шкідлива
WriteFile	C:\Windows\Temp\sys_rep_XXXX.tmp	Буфер скріншотів екрану (PNG-сигнатура)	Шкідлива

Операція	Шлях / Ключ реєстру	Значення / Вміст	Оцінка
DeleteFile	C:\ProgramData\czupd\log_XXXX.txt	Знищення власних журналів	Шкідлива
ReadFile	C:\Users*\Documents*.docx, *.xlsx	Читання документів користувача	Шкідлива

Ознаки закріплення (persistence): два незалежні механізми – ключ Run у реєстрі поточного користувача та реєстрація системної служби (HKLM\Services) – забезпечують запуск модуля після перезавантаження незалежно від рівня привілеїв користувача.

Крок 4. Класифікація та нейтралізація загрози

Класифікація: модуль «update_cz.exe» відповідає моделі закладки «перехоплювач» (підрозділ 3.3 посібника).

Обґрунтування:

– Операції ReadFile щодо документів користувача (*.docx, *.xlsx) свідчать про збір конфіденційної інформації без її підміни;

– HTTP POST – запити на зовнішню IP-адресу (185.220.101.47) підтверджують ексфільтрацію зібраних даних;

– створення тимчасових PNG-файлів у C:\Windows\Temp\ вказує на перехоплення екранних даних оператора;

– відсутність модифікації штатних виконуваних файлів системи свідчить про те, що модуль виконує приховане перехоплення інформації без порушення роботи основного програмного забезпечення, що характерно саме для моделі «перехоплювач».

Заходи нейтралізації:

1. Відновлення VM зі snapshot до стану перед запуском (ізолювання зараженого середовища).

2. Видалення ключів автозавантаження:

HKCU\...\Run\CZUpdate та HKLM\...\Services\CZUpdateSvc.

3. Видалення файлів: C:\ProgramData\czupd*, C:\Windows\Temp\sys_rep_*.tmp.

4. Блокування IP-адрес: 185.220.101.47 та 91.108.56.23 на рівні мережевого екрану.

5. Проведення аудиту пакета оновлення та перевірка цифрового підпису.

6. Перевірка інших вузлів мережі цивільного захисту на наявність аналогічних індикаторів компрометації (IOC).

7. Повідомлення відповідальних осіб та документування інциденту відповідно до процедур реагування.

Загальний висновок:

Динамічний аналіз (підрозділ 6.5 посібника) дозволив виявити активність, яка не могла бути достовірно встановлена виключно засобами статичного аналізу: використання зашифрованих конфігурацій, мережеву ексфільтрацію даних та механізми persistence.

Комплексне поєднання sandbox-середовищ, мережевого моніторингу, аналізу файлової системи та контролю реєстру забезпечує ефективне виявлення прихованої функціональності програмних модулів і дозволяє своєчасно локалізувати загрози для інформаційних систем цивільного захисту.

7. МЕТОДИ ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ВІД ЗАГРОЗ ТА НЕСАНКЦІОНОВАНОГО ДОСЛІДЖЕННЯ

7.1. ДИЗАСЕМБЛЮВАННЯ ТА АНАЛІЗ ВИКОНУВАНОГО КОДУ

Дослідження виконуваних файлів (дизасемблювання) – один із перших кроків при реверс-інжинірингу, аналізі вразливостей і несанкціонованій модифікації програмного забезпечення. У контексті кібербезпеки та захисту програмного забезпечення методи дизасемблювання застосовуються також для дослідження механізмів функціонування програм, виявлення вразливостей та аналізу потенційно небезпечного коду. Розробники захисту прагнуть ускладнити цю задачу, зменшивши ймовірність швидкого аналізу та підвищивши трудомісткість реверс-інжинірингу й несанкціонованого втручання. Методи протидії дизасемблюванню застосовують у ліцензійних механізмах, DRM, антизламних підсистемах та для захисту інтелектуальної власності [17, 19, 29, 30, 33].

Методи боротьби з дизасемблюванням мають на меті мінімізувати можливості зловмисника отримати зрозумілу структуру коду, порушити логіку реконструкції алгоритму програми та зробити результати аналізу неоднозначними. Залежно від спрямованості дії, можна виділити кілька основних груп цілей, які наведено нижче та відображено на рисунку 7.1.

1. Протидія статичному аналізу – орієнтована на ускладнення роботи автоматичних дизасемблерів, таких як IDA Pro чи Ghidra [44, 47]. Метою є створення ситуації, коли дизасемблер формує некоректний або неповний лістинг коду. До цього належать методи приховування інструкцій, перемішування коду, використання «псевдоінструкцій» та вставка фіктивних даних.

Це ускладнює аналіз структури виконуваного файлу та дезорієнтує аналітика, який намагається побудувати граф переходів або виділити підпрограми.

2. Протидія інтерактивному дизасемблюванню – спрямована на утруднення розуміння коду людиною під час ручного дослідження.

Використовуються спеціальні комбінації інструкцій, що спричиняють некоректне відображення структури коду у відомих інструментах (наприклад, IDA або Radare2), створення великої кількості міток, псевдозмінних та «засмічення» вихідного лістингу псевдокоментарями, мітками або нечитабельними підписами.

3. Протидія налагодженню та динамічному аналізу – полягає у впровадженні anti-debugging-технік, які унеможливлюють покрокове виконання програми чи відстеження її внутрішніх змін під час роботи. Для цього використовують перевірки на наявність дебагера, зміни часових затримок, динамічне дешифрування коду або самовідновлення інструкцій у пам’яті під час виконання [57, 59].

4. Імітаційні та емуляційні механізми – включають створення віртуальних шарів виконання, що моделюють іншу архітектуру або мову команд. У цьому разі навіть після дизасемблювання дослідник стикається не з реальним машинним кодом, а з псевдоінструкціями, які необхідно додатково інтерпретувати.

5. Захист інтелектуальної власності – застосовується для охорони алгоритмів, криптографічних процедур або механізмів активації. Такі методи комбінують кілька рівнів протидії, включаючи обфускацію, пакування, цифрові підписи та перевірку цілісності виконуваного модуля [30, 33, 58].

6. Відлякування або затримка аналітика – стратегія, що базується на створенні значного обсягу «сміттєвого» або беззмістовного коду. Її мета полягає не в абсолютній недоступності програми для аналізу, а у збільшенні часу дослідження, що робить процес реверс-інжинірингу економічно недоцільним.



Рисунок 7.1. Класифікація методів боротьби з дизасемблюванням

На рисунку 7.1. узагальнено методи боротьби з дизасемблюванням програмного коду: протидію статичному та динамічному аналізу, налагодженню, емуляції, а також механізми захисту інтелектуальної власності й відлякування аналітика [33, 57, 58].

7.1.1 Методи протидії дизасемблюванню (обфускація та anti-reversing)

Дизасемблювання є ключовим етапом реверс-інжинірингу (reverse engineering), спрямованим на відновлення структури та логіки виконуваного коду [17, 19, 48]. У практиці захисту програмного забезпечення широко застосовуються методи протидії його аналізу.

Методи цієї групи спрямовані на ускладнення або унеможливлення коректного дизасемблювання виконуваного коду. Їх метою є дезорієнтація аналізатора шляхом інтерпретації даних як коду, формування хибних переходів або включення надлишкових фрагментів.

Такі методи можна умовно поділити на:

– обфускацію коду (заплутування структури та логіки програми);
– ускладнення керуючих конструкцій;

– засоби протидії автоматичним та інтерактивним дизасемблерам і налагоджувачам [33, 57, 58].

Розгляд зазначених методів у цьому підрозділі здійснюється з позицій аналізу механізмів захисту програмного забезпечення та дослідження способів протидії реверс-інжинірингу.

Ці підходи широко застосовуються як для захисту програмного забезпечення, так і для приховування шкідливої функціональності [18, 39].

Одним із ефективних методів є динамічне розшифрування коду, при якому фрагменти програми перебувають у зашифрованому вигляді та розшифровуються лише безпосередньо перед виконанням.

Після виконання такі фрагменти можуть:

– знову шифруватися;

– переміщуватися в інші області пам’яті;

– знищуватися.

У більш складних реалізаціях програма складається з кількох частин, які поетапно розшифровують наступні блоки, змінюють своє розташування в пам’яті та передають керування один одному.

Такий підхід значно ускладнює дизасемблювання, оскільки статичний аналіз не дозволяє отримати повний код програми в одному місці [33, 58].

7.1.2. Методи проти автоматичних дизасемблерів

Автоматичні дизасемблери виконують аналіз машинного коду шляхом послідовного або рекурсивного розбору інструкцій та побудови графа керування програмою. Більшість таких інструментів базується на припущенні про однозначність інтерпретації байтових послідовностей та коректність механізмів передачі керування.

Статичні дизасемблери умовно поділяють на:

– лінійні (linear sweep) – аналізують байти послідовно, інтерпретуючи їх як машинні інструкції;

– рекурсивні (recursive traversal) – будують граф переходів і аналізують лише ті ділянки коду, які досяжні через команди передачі керування [17, 19, 48].

Більшість anti-disassembly-методів спрямовані на порушення роботи одного або обох підходів.

Методи протидії автоматичним (статичним) дизасемблерам спрямовані на руйнування припущень, на яких базується лінійний і рекурсивний аналіз виконуваного коду. Перемішування інструкцій, приховування передачі керування, перекривання коду та нестандартні конструкції переходів порушують коректну побудову графа керування й унеможливлюють однозначну інтерпретацію байтових послідовностей [33, 44, 47, 58].

Додаткове маскування сегментів і використання нетипових форматів модулів призводить до хибної класифікації коду як даних або до формування неповного лістингу. Основні методи боротьби з автоматичними дизасемблерами наведено на рис. 7.2.

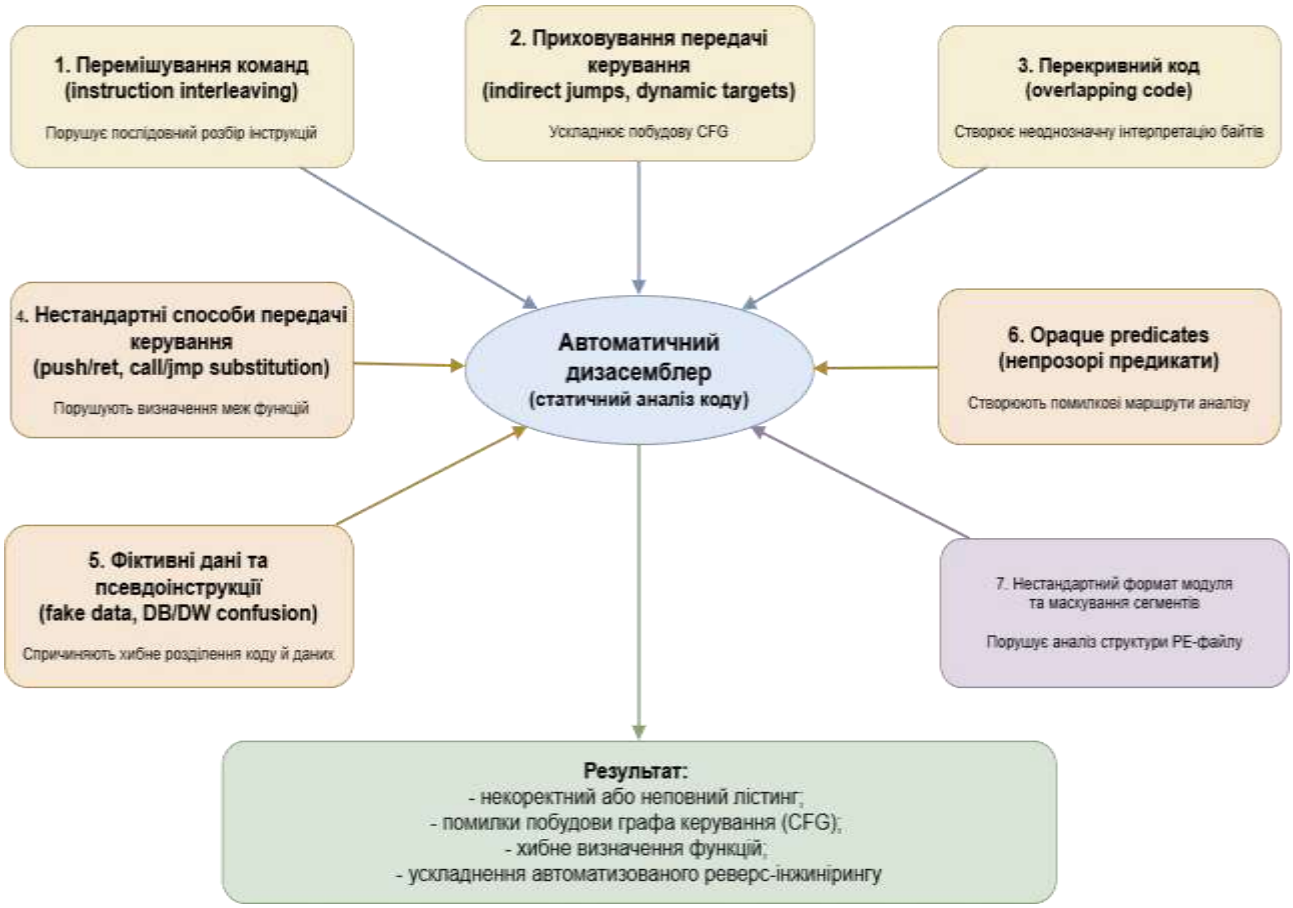


Рисунок 7.2. Методи впливу на статичний дизасемблер

До методів боротьби з автоматичними дизасемблерами відносять шість описаних нижче.

1. Перемішування команд (instruction interleaving)

Суть методу полягає у зміні природного порядку розташування інструкцій у виконуваному файлі із збереженням фактичної логіки виконання програми. Для цього програмний код розбивають на окремі блоки, між якими вставляють додаткові переходи.

У результаті, наприклад, порядок виконання програми залишається коректним, лінійний лістинг дизасемблера набуває хаотичного вигляду, ускладнюється побудова графа керування (Control Flow Graph, CFG), погіршується наочність коду для аналітика.

Такі методи особливо ефективні проти linear sweep-дизасемблерів [17, 33, 58].

2. Приховування команд передачі керування

Одним із найпоширеніших способів протидії статичному аналізу є приховування реальних маршрутів виконання програми. Для цього використовують непрямі переходи (indirect jumps), динамічно обчислювані адреси, модифікацію адрес переходів під час виконання, передачу керування через стек або таблиці переходів.

У таких випадках дизасемблер не може однозначно визначити наступну інструкцію, межі функцій або структуру викликів. Це призводить до порушення побудови графа керування та формування неповного або помилкового лістингу [17, 19, 48].

3. Використання перекривного коду (overlapping code)

Перекривний код виникає тоді, коли одна й та сама послідовність байтів може інтерпретуватися по-різному залежно від точки входу. У процесорах архітектури x86 це можливо через змінну довжину машинних інструкцій.

У результаті різні точки входу формують різні набори інструкцій, дизасемблер втрачає однозначність аналізу, частина коду може бути

інтерпретована некоректно, автоматичне відновлення структури програми значно ускладнюється.

Такі методи часто застосовуються разом із динамічними переходами та обфускацією [17, 33, 58].

4. Нестандартні способи передачі керування

Замість типових команд jmp, call або ret можуть використовуватися нестандартні конструкції передачі керування. У звичайній програмі передача керування виконується командами jmp – безумовний перехід, call – виклик функції, ret – повернення з функції. Дизасемблери добре розпізнають такі інструкції та будують граф керування (CFG), межі функцій, послідовність виконання. Але в anti-reversing використовують нестандартні конструкції, які приховують справжній маршрут виконання програми.

Наведемо декілька прикладів.

Приклад 1

```
push addr
ret;
```

Ідея полягає в тому, що адреса переходу записується у стек; команда ret бере адресу зі стеку та передає туди керування.

Звичайний перехід має вигляд:

```
jmp addr
```

Дизасемблер бачить перехід. Ціль переходу —addr (умовна адреса). Але у наведеному прикладі використовується прихований варіант:

```
push addr - адреса addr кладеться у стек;
ret - процесор бере адресу зі стеку й переходить на addr.
```

Це знижує прозорість коду, хоча для процесора результат однаковий. Для дизасемблера це виглядає як повернення з функції, а не як звичайний jmp. Через це порушується CFG, неправильно визначаються межі функцій, а recursive traversal може не проаналізувати частину коду.

Приклад 2. Передача адрес через стек.

Підхід є загальним. Адреси функцій, переходів, обробників передаються через стек, пам’ять, таблиці.

Проблема для дизасемблера полягає в тому, що статичний аналізатор не знає значення стеку під час виконання й не може точно визначити ціль переходу. Тому CFG стає неповним, а частина коду не аналізується.

Приклад 3. Самозмінні переходи (self-modifying jumps).

Програма змінює байти інструкцій під час виконання. Наприклад, адреса jmp, адреса call, або opcode переходу, можуть динамічно переписуватися у пам’яті.

Наслідки: На диску маємо

```
jmp A,
```

а у пам’яті після модифікації:

```
jmp B
```

Проблема полягає в тому що статичний дизасемблер аналізує лише файл на диску, але реальне виконання відбувається вже зі зміненим кодом. Через це статичний аналіз стає неповним, CFG не відповідає реальному виконанню, частина логіки приховується.

Подібні конструкції:

– ускладнюють розпізнавання викликів функцій;
– порушують логіку recursive traversal;
– призводять до помилок у визначенні меж процедур.

Для людини така логіка також стає менш наочною [17, 19, 33].

5. Заплутування через фіктивні дані та псевдоінструкції

У виконуваний файл можуть навмисно вставлятися фальшиві інструкції, надлишкові префікси, псевдокоманди, ділянки випадкових даних, великі масиви байтів, схожі на машинний код.

У результаті дизасемблер помилково визначає дані як код, друкує директиви DB, DW, DD, зміщує межі інструкцій, формує некоректний лістинг.

Подібні методи суттєво ускладнюють автоматичний аналіз структури виконуваного файлу [33, 44, 47].

6. Opaque predicates

Opaque predicates – це умовні конструкції, результат яких наперед відомий розробнику, але складний для статичного аналізу.

Наприклад, умова завжди істинна, умова завжди хибна, результат залежить від складних математичних виразів.

Для дизасемблера такі конструкції виглядають як повноцінні розгалуження програми, через що штучно збільшується граф керування, виникають помилкові маршрути аналізу, зростає складність автоматизованого відновлення логіки програми.

Opaque predicates широко використовуються в сучасних засобах обфускації [33, 58].

7. Нестандартний формат модуля та маскування сегментів

Ще одним методом протидії автоматичному дизасемблюванню є використання нетипових структур виконуваного файлу та маскування окремих сегментів програми. У цьому випадку модифікуються заголовки модуля, таблиці секцій, атрибути сегментів або внутрішня структура виконуваного файлу.

Зокрема, можуть застосовуватися нестандартні таблиці секцій, перекриття сегментів, приховування коду під виглядом даних, а також зміна атрибутів областей пам’яті. У результаті дизасемблер може некоректно визначати межі програмного коду, помилково інтерпретувати код як дані або не виділяти окремі функції автоматично.

Подібні модифікації також здатні порушувати коректний аналіз PE-структури виконуваного файлу та спричиняти помилки під час завантаження модуля у середовище дизасемблювання.

У результаті ускладнюється побудова графа керування програмою, погіршується якість автоматичного аналізу та знижується ефективність відновлення реальної структури виконуваного коду.

7.1.3. Методи проти інтерактивних дизасемблерів

(методи, що ускладнюють ручний аналіз у середовищах IDA Pro, Ghidra та ін.)

Методи протидії інтерактивним дизасемблерам спрямовані на навмисне ускладнення процесу ручного аналізу коду в середовищах на кшталт IDA Pro або Ghidra [44, 47].

На відміну від методів, орієнтованих переважно на автоматичні дизасемблери, ці підходи спрямовані безпосередньо на ускладнення роботи аналітика під час інтерактивного дослідження програмного коду.

Вони порушують коректне лінійне відображення інструкцій, змушують інструменти неправильно інтерпретувати окремі області як дані, а також перевантажують робочий простір аналітика штучними мітками, псевдоструктурами та оманливими сигнатурами. До таких методів належать маркувальні та заплутуючі байти («undefined region forcing»), динамічний лінеаризаційний опір, інтерактивні пастки (anti-IDA tricks) [33, 44, 47], (рис.7.3).

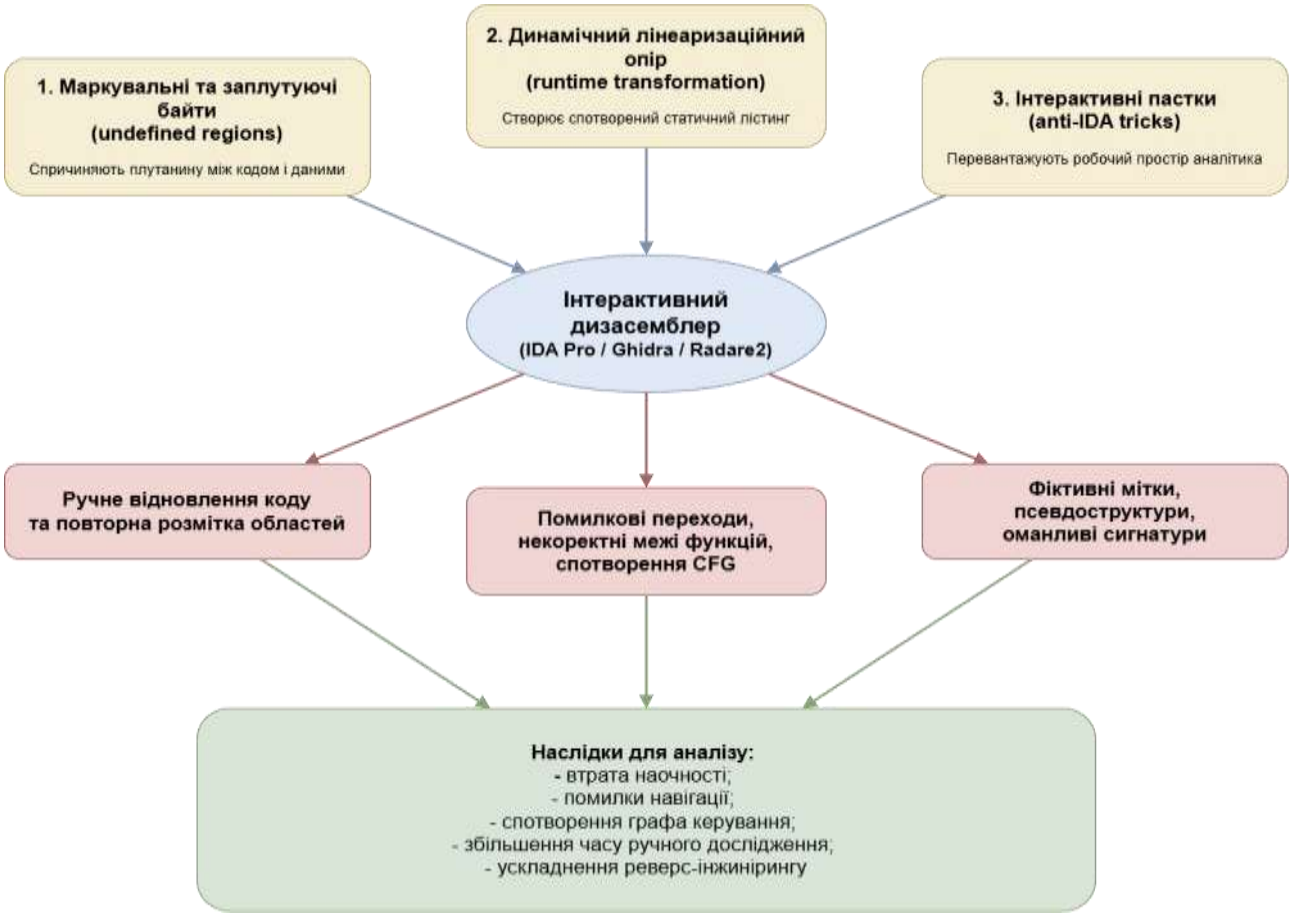


Рисунок 7.3. Методи протидії інтерактивним дизасемблерам

1. Маркувальні та заплутуючі байти («undefined region forcing»)

Суть методу полягає у вставці спеціально підібраних байтових послідовностей, які змушують дизасемблер інтерпретувати частину коду як область даних або невизначену ділянку пам’яті.

У результаті частина інструкцій не відображається коректно, дизасемблер формує фрагментарний лістинг, аналітик змушений вручну змінювати типи областей, а трудомісткість ручного аналізу суттєво збільшується [33, 44, 47].

Подібні методи особливо ефективні проти інтерактивних середовищ, де дослідник активно працює з розміткою коду та навігацією між функціями.

2. Динамічний лінеаризаційний опір

Даний підхід базується на поєднанні перемішування інструкцій та динамічної деобфускації коду. На диску програма зберігається у «забрудненому» або штучно модифікованому вигляді, тоді як під час виконання окремі фрагменти самостійно трансформуються у коректний машинний код.

У таких умовах інтерактивний дизасемблер, який працює лише зі статичним образом файлу, отримує неповний або спотворений лістинг, що включає помилкові переходи, некоректні межі функцій, надлишкові псевдоінструкції.

У результаті реальна логіка виконання програми приховується до моменту безпосереднього виконання коду [33, 57, 58].

3. Інтерактивні пастки (anti-IDA tricks)

Методи цієї групи спрямовані на створення конструкцій, які інтерактивні дизасемблери можуть інтерпретувати некоректно або неоднозначно.

До таких методів належать:

– вставка оманливих сигнатур;
– формування великої кількості псевдоміток;
– створення фіктивних структур;
– використання нестандартних таблиць імпорту;
– навмисне перевантаження робочого простору аналітика.

У результаті навігація між функціями ускладнюється, зменшується наочність графа керування, а аналітик витрачає більше часу на ручне відновлення структури програми.

Подібні методи часто комбінуються з обфускацією та anti-debugging-техніками [33, 57-59].

Таким чином, методи протидії інтерактивним дизасемблерам спрямовані на ускладнення зручності та наочності ручного аналізу виконуваного коду. Вони не завжди унеможливлюють дизасемблювання повністю, однак суттєво збільшують трудомісткість реверс-інжинірингу та ускладнюють відновлення реальної логіки функціонування програми.

7.2. X-КОД ТА МЕХАНІЗМИ ЙОГО ВИКОРИСТАННЯ

7.2.1 X-код та вимоги до нього

У процесі модифікації та захисту виконуваних файлів важливу роль відіграє впроваджуваний код, який додається до програмної реалізації з метою зміни або розширення її функціональності. Такий фрагмент коду умовно називається X-кодом [17, 19, 30].

X-код може використовуватися для:

– реалізації механізмів захисту програмного забезпечення;
– контролю цілісності та перевірки середовища виконання;
– впровадження додаткової логіки обробки;
– модифікації процесу передавання керування у програмі.

На відміну від звичайних програмних модулів, X-код функціонує у межах уже існуючої програмної реалізації, тому повинен враховувати особливості структури виконуваного файлу та механізми взаємодії з основною програмою [17, 42].

Для забезпечення коректного функціонування X-коду та мінімізації впливу на основну програму він повинен відповідати таким вимогам (рис.7.4):



Рисунок 7.4. X-код та вимоги до нього

1. Врахування особливостей середовища виконання

X-код повинен бути узгоджений із середовищем, у яке він впроваджується, зокрема враховувати архітектуру, формат виконуваного файлу та механізми завантаження й адресації пам’яті [17, 26, 42].

2. Переміщуваність (позиційна незалежність)

X-код має зберігати працездатність незалежно від базової адреси завантаження. Це досягається використанням відносної адресації та уникненням жорстко закодованих адрес [17, 19].

Позиційна незалежність є особливо важливою при модифікації виконуваних файлів, у яких адреси розміщення коду можуть змінюватися під час завантаження або переміщення модулів у пам’яті.

3. Коректність взаємодії із середовищем

X-код не повинен неконтрольовано змінювати середовище виконання, зокрема вміст пам’яті або службові структури, якщо це не передбачено його функціональністю.

Крім того, після завершення роботи X-код повинен забезпечувати коректне

відновлення регістрів, стеку та інших змінених компонентів середовища виконання [17, 26].

4. Компактність

Обсяг Х-коду повинен бути мінімальним, оскільки простір для його розміщення у виконуваному файлі зазвичай обмежений.

У практиці часто використовується розділення Х-коду на компактний завантажувач та основну функціональну частину [19, 33].

Завантажувач може розміщуватися у заголовку файлу або в інших доступних областях, тоді як основна частина – у менш помітних ділянках (наприклад, у хвості секцій або зовнішніх структурах).

Подібний підхід дозволяє мінімізувати помітність модифікацій та спростити впровадження основного функціонального модуля.

5. Мінімальний вплив на виконання

Х-код не повинен суттєво впливати на час виконання програми [33]. Значні затримки можуть свідчити про наявність стороннього коду та знижують ефективність його приховування.

Надмірне збільшення часу виконання також може призводити до нестабільної роботи програмного забезпечення або спрощувати виявлення модифікації засобами моніторингу.

6. Реєнтрантабельність

Х-код повинен забезпечувати реєнтрантабельність, тобто здатність коректно виконуватися при повторному вході до нього до завершення попереднього виконання. Це особливо важливо в умовах багатопотокового виконання або при багаторазовому виклику змінених ділянок програми.

Для цього Х-код не повинен залежати від неконтрольованого глобального стану, повинен коректно працювати з регістрами та стеком [17, 26], а також забезпечувати відновлення зміненого середовища після завершення роботи. Порушення цієї вимоги може призвести до нестабільної роботи програми або її аварійного завершення.

Таким чином, X-код є спеціалізованим інструментом модифікації програмної реалізації, який може використовуватися як для захисту програмного забезпечення, так і для впровадження додаткового функціоналу.

Ефективність застосування X-коду визначається не лише його функціональними можливостями, а й здатністю коректно взаємодіяти з основною програмою без порушення її стабільності та логіки роботи [17, 30].

X-код забезпечує можливість гнучкого впливу на структуру та поведінку виконуваного файлу, однак його застосування потребує дотримання строгих вимог, що гарантують коректність роботи та мінімальну помітність змін.

7.2.2. Механізми функціонування X-коду

Функціонування X-коду визначається сукупністю механізмів, що забезпечують його впровадження у виконуваний файл, перехоплення керування та взаємодію із системним середовищем [17, 19]. На відміну від звичайних програмних модулів, X-код працює в умовах обмежених ресурсів і невизначеного середовища, що обумовлює специфіку його організації.



Рисунок 7.5. Основні етапи функціонування X-коду

Процес роботи Х-коду можна подати як послідовність п’яти взаємопов’язаних етапів

1. Розміщення у структурі виконуваного файлу

На першому етапі Х-код інтегрується у файл-носій одним із способів [19, 42]:

– затирання – розміщення поверх існуючого коду (що може призвести до втрати частини функціональності вихідної програми);

– інтеграція – впровадження у вільні області структури (заголовок, хвости секцій);

– дописування – додавання коду на початок, у середину або кінець файлу із збереженням початкового вмісту;

– зовнішнє розміщення – винесення частини коду за межі основного файлу з організацією його підвантаження.

Вибір способу залежить від доступного простору, вимог до прихованості та особливостей модифікації виконуваного файлу.

2. Перехоплення керування

Після впровадження Х-код повинен отримати керування під час виконання програми. Це може реалізовуватися такими способами [17, 19, 42]:

– зміна точки входу (Entry Point) на адресу Х-коду;

– вставка команди переходу (JMP/CALL) у початкову область виконання;

– модифікація довільних інструкцій з подальшим поверненням керування;

– підміна записів таблиці імпорту для перехоплення викликів функцій.

Механізм перехоплення визначає баланс між надійністю виконання Х-коду та рівнем його прихованості.

На практиці найбільш поширеними є модифікація точки входу та вставка переходу до впровадженого коду.

3. Визначення необхідних ресурсів

Для коректної роботи Х-код повинен отримати доступ до системних

функцій, зокрема API операційної системи.

Основні підходи:

– використання вже наявних імпортів програми;

– динамічне отримання адрес функцій через LoadLibrary() і GetProcAddress();

– пошук необхідних функцій у пам’яті;

– модифікація таблиці імпорту.

У більш складних реалізаціях застосовуються методи обходу стандартних механізмів імпорту, що ускладнює аналіз і підвищує рівень прихованості.

4. Виконання основної функціональності

Після отримання керування та доступу до необхідних ресурсів X-код виконує свої функції, які можуть включати:

– перевірку умов виконання (середовище, цілісність);

– модифікацію поведінки програми;

– взаємодію з зовнішніми компонентами;

– передачу керування основній програмі.

На цьому етапі критично важливо забезпечити мінімальний вплив на стабільність роботи основної програми.

Крім того, X-код повинен коректно взаємодіяти із пам’яттю, регістрами процесора та стеком, не порушуючи логіку роботи основної програми [17, 26].

5. Завершення роботи та відновлення виконання

Після виконання своїх дій X-код, як правило [19, 33]:

– відновлює змінені фрагменти коду;

– повертає керування оригінальній програмі;

– маскує сліди своєї присутності.

Це забезпечує прозорість виконання та ускладнює виявлення факту модифікації.

Розглянемо спрощений підхід до отримання адрес функцій операційної системи під час виконання.

Один із найбільш поширених механізмів полягає у використанні функцій

динамічного завантаження бібліотек:

```
hKernel32 = LoadLibrary("kernel32.dll");
pFunc = GetProcAddress(hKernel32, "CreateFileA");
```

У цьому випадку [26]:

– LoadLibrary забезпечує завантаження необхідної бібліотеки;

– GetProcAddress повертає адресу функції за її іменем.

Отримана адреса може використовуватися для виклику функції під час виконання програми.

У більш складних реалізаціях можливе визначення адрес функцій без використання таблиці імпорту, наприклад, шляхом аналізу структур процесу в пам’яті, через пошук сигнатур завантажених модулів, або за допомогою обходу системних таблиць [17, 19, 26]. Такі підходи ускладнюють аналіз програми, оскільки приховують явні залежності від системних функцій.

Функціонування Х-коду базується на поєднанні механізмів впровадження, перехоплення керування та взаємодії з системним середовищем. Ефективність його роботи визначається здатністю адаптуватися до структури виконуваного файлу та залишатися непомітним для аналізу.

Механізми функціонування Х-коду забезпечують можливість гнучкого втручання у процес виконання програмного забезпечення. Їх розуміння є необхідним для аналізу сучасних методів захисту, обфускації та протидії зворотному інжинірингу.

7.2.3. Методи впровадження Х-коду

Впровадження Х-коду у виконуваний файл є ключовим етапом його використання, що визначає подальшу можливість перехоплення керування та рівень прихованості модифікації [17, 19, 33]. Вибір методу впровадження залежить від структури файлу, наявності вільного простору та вимог до збереження функціональності вихідної програми. Основні методи впровадження Х-коду наведено на рис. 7.6.

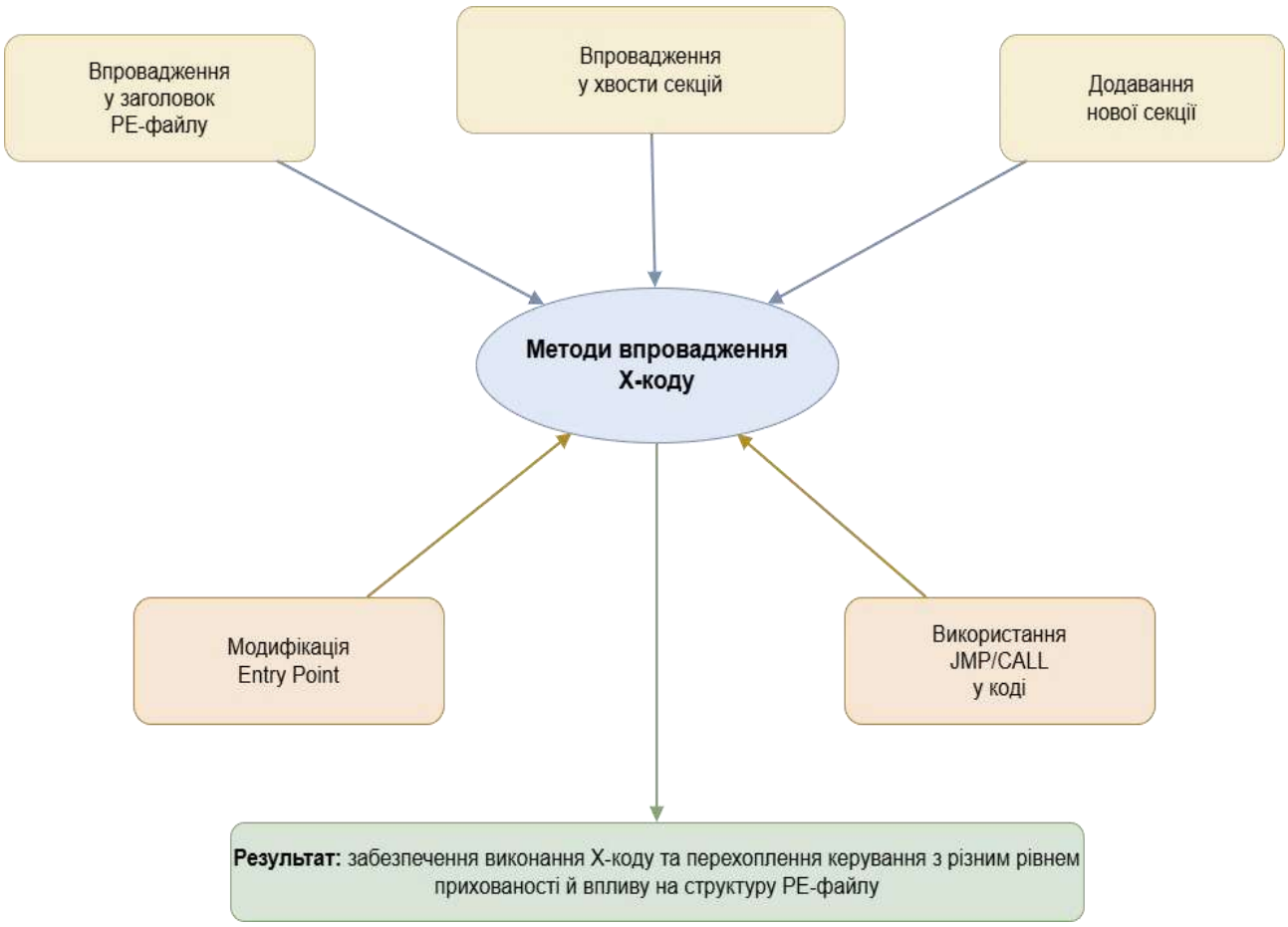


Рисунок 7.6. Основні методи впровадження Х-коду

Методи впровадження Х-коду можна умовно поділити на п’ять перелічених нижче груп.

1. Впровадження у заголовок виконуваного файлу

Цей підхід базується на використанні вільного простору між заголовком PE-файлу та першою секцією [42].

На рисунку 7.7 показано використання вільної області після заголовка для розміщення Х-коду. Такий підхід дозволяє інтегрувати додатковий функціонал без зміни розташування секцій.

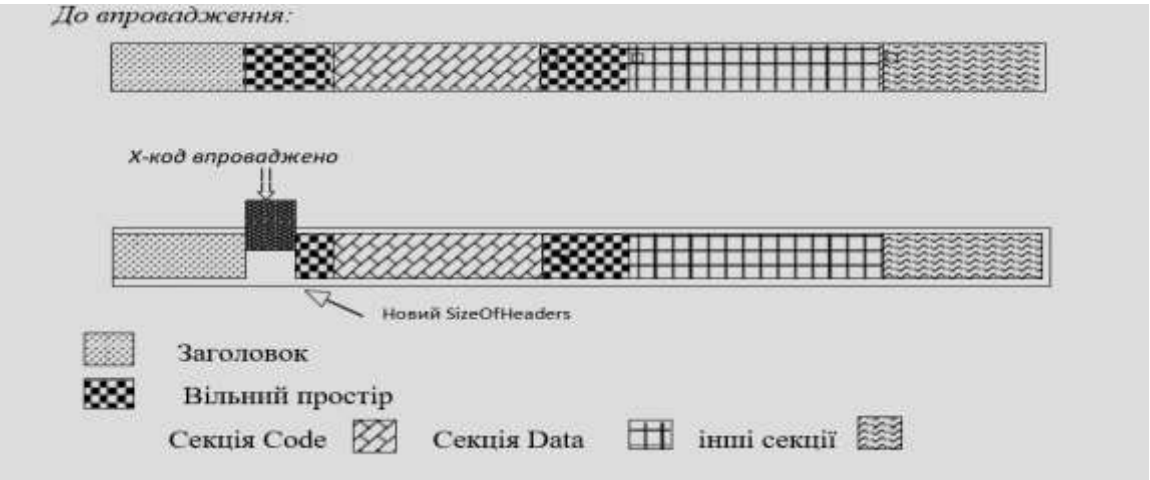


Рисунок 7.7. Схема впровадження X-коду у заголовок PE-файлу

Особливості:

– не потребує зміни структури секцій;

– забезпечує швидкий доступ до X-коду під час запуску;

– обмежений обсягом доступного простору.

Подібний підхід часто застосовується за наявності невикористаних областей у структурі заголовка виконуваного файлу.

2. Впровадження у хвостову частину секцій

Метод полягає у використанні невикористаного простору в межах існуючих секцій (так званих “хвостів”) [19, 42].

На рисунку 7.8 показано розміщення X-коду у невикористаній області секції, що дозволяє уникнути створення нових структурних елементів.

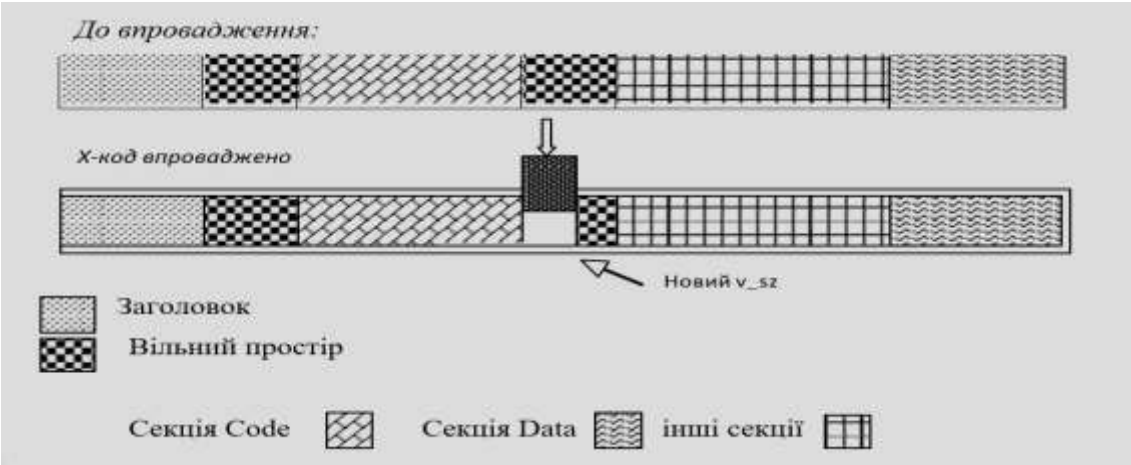


Рисунок 7.8. Схема впровадження X-коду у хвостову частину секції

Особливості:

– дозволяє зберегти зовнішню структуру файлу;

– менш помітний при поверхневому аналізі;

– потребує узгодження параметрів секції (розмірів і атрибутів).

Цей метод вважається одним із найбільш прихованих способів інтеграції Х-коду у PE-файл.

3. Впровадження шляхом додавання нової секції

У цьому випадку створюється нова секція, у якій розміщується Х-код [19, 42].

Особливості:

– забезпечує достатній обсяг для коду;

– спрощує організацію логіки;

– змінює структуру файлу, що може бути виявлено при аналізі.

Цей метод є одним із найбільш очевидних, але водночас і найбільш зручних з точки зору реалізації.

Додавання нової секції часто супроводжується модифікацією таблиці секцій та зміною параметрів PE-заголовка.

4. Впровадження з модифікацією точки входу

Метод передбачає зміну поля AddressOfEntryPoint таким чином, щоб керування передавалося Х-коду [42].

Особливості:

– гарантує виконання Х-коду при запуску;

– потребує подальшого повернення керування основній програмі;

– може бути виявлений при аналізі заголовка.

Після завершення роботи Х-код зазвичай передає керування оригінальній точці входу програми.

5. Впровадження з використанням переходів у коді

У цьому випадку в існуючий код програми вставляються інструкції переходу (JMP, CALL), що передають керування Х-коду [17, 19].

Особливості:

– дозволяє локалізувати вплив на окремі ділянки програми;
– складніше для реалізації;
– підвищує рівень прихованості.

Подібні методи часто застосовуються для перехоплення окремих функцій або модифікації логіки виконання конкретних фрагментів програми.

Методи впровадження Х-коду відрізняються за рівнем прихованості, складністю реалізації та впливом на структуру виконуваного файлу [19, 33]. Найбільш ефективними є підходи, що дозволяють зберегти початкову структуру файлу та мінімізувати видимі зміни (табл. 7.1).

Таблиця 7.1

Порівняльна характеристика методів

Метод	Прихованість	Складність	Вплив на структуру
Заголовок	середня	низька	мінімальний
Хвости секцій	висока	середня	мінімальний
Нова секція	низька	низька	значний
Зміна Entry Point	середня	низька	мінімальний
Переходи у коді	висока	висока	локальний

Для більш системного аналізу механізмів впровадження доцільно застосовувати їх класифікацію за характером впливу на структуру виконуваного файлу. Механізми впровадження Х-коду можуть бути класифіковані за характером впливу на структуру виконуваного файлу (рис.7.9).

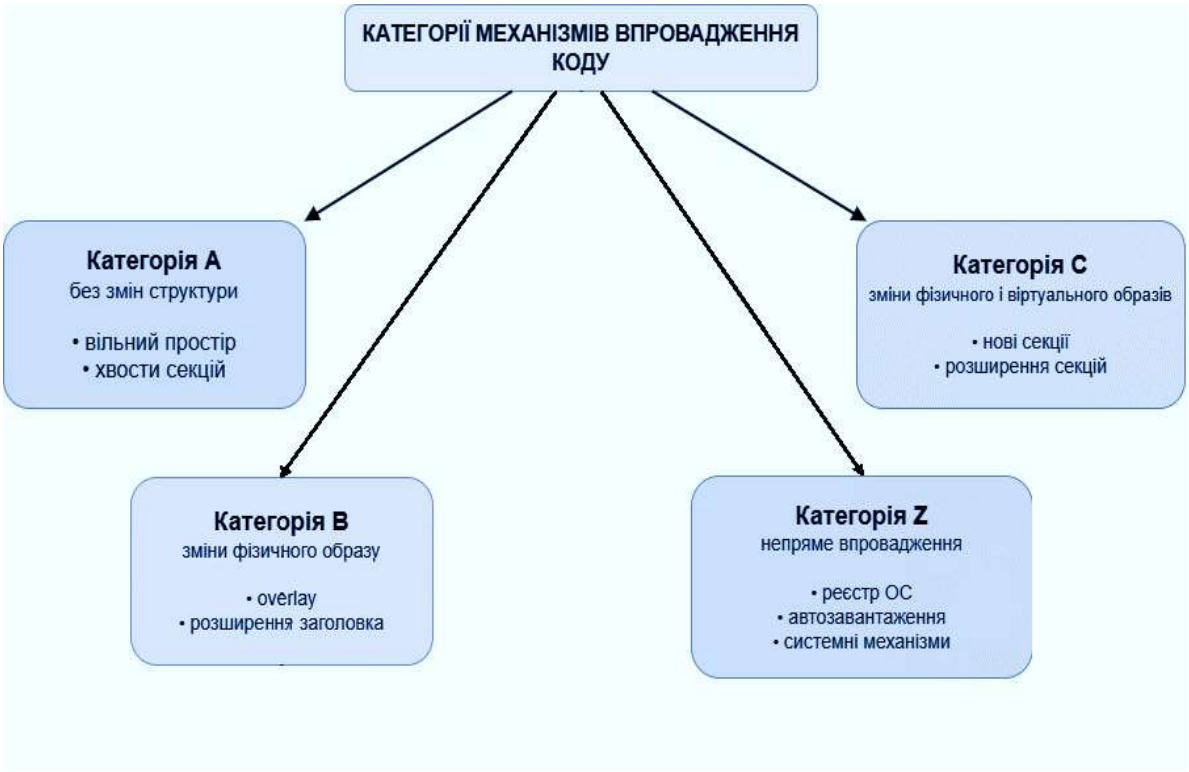


Рисунок 7.9. Класифікація механізмів впровадження коду

– Категорія A – впровадження без зміни структури (використання вільного простору, хвостів секцій);

– Категорія B – зміни лише фізичного образу (розширення заголовка, використання overlay);

– Категорія C – зміни фізичного і віртуального образу (додавання або розширення секцій);

– Категорія Z – непряме впровадження (використання механізмів ОС, наприклад автозавантаження).

Застосування статичного аналізу дозволяє визначити категорію впровадження за характером змін структури PE-файлу, що є важливим для виявлення модифікацій та оцінки їх впливу на систему.

Вибір методу впровадження X-коду визначає баланс між надійністю виконання, складністю реалізації та рівнем прихованості модифікації. Розуміння цих методів є важливим для аналізу сучасних механізмів захисту та протидії зворотному інжинірингу.

7.2.4. Запобіжні заходи та обмеження

Застосування Х-коду у виконуваних файлах пов’язане не лише з можливістю модифікації або захисту програмного забезпечення, але й із низкою обмежень і ризиків, які необхідно враховувати під час його розроблення та аналізу [30, 33].

Під час впровадження Х-коду важливо забезпечити баланс між функціональністю, прихованістю модифікації та стабільністю роботи програмного забезпечення.

При впровадженні Х-коду доцільно дотримуватися таких принципів (рис.7.10):



Рисунок 7.10. Принципи безпечного впровадження Х-коду

1. Контроль середовища виконання

Усі експерименти з модифікацією виконуваних файлів повинні проводитися виключно в ізольованому середовищі (віртуальні машини, sandbox). Це дозволяє уникнути небажаного впливу на робочу систему та забезпечити контрольованість результатів [18, 49].

Особливо важливим є використання ізольованих середовищ під час дослідження механізмів перехоплення керування та модифікації системних структур.

2. Збереження цілісності програми

Впровадження X-коду не повинно призводити до порушення основної функціональності програми, якщо це не є метою дослідження [30, 33]. Необхідно забезпечити коректне повернення керування та відновлення змінених ділянок.

Порушення цілісності виконуваного файлу або логіки виконання може призвести до нестабільної роботи програми чи її аварійного завершення.

3. Мінімізація змін структури

Рекомендується використовувати методи впровадження, що не змінюють базову структуру PE-файлу (наприклад, використання вільного простору або хвостів секцій) [19, 42]. Це знижує ймовірність виявлення модифікації та зменшує ризик порушення роботи програми.

Мінімізація структурних змін також ускладнює виявлення модифікацій засобами статичного аналізу.

4. Узгодженість із параметрами файлу

Будь-які зміни повинні відповідати параметрам заголовка та секцій (вирівнювання, розміри, адреси). Некоректні значення можуть призвести до помилок завантаження або аварійного завершення роботи.

Особливу увагу слід приділяти параметрам вирівнювання секцій (FileAlignment, SectionAlignment) та коректності адресних таблиць [42].

5. Контроль використання системних ресурсів

X-код повинен коректно працювати з системними функціями та не порушувати стан середовища виконання. Особливу увагу слід приділяти отриманню адрес API-функцій та роботі з пам’яттю [26].

Крім того, необхідно забезпечити коректну роботу зі стеком, регістрами процесора та динамічно виділеною пам’яттю.

Незважаючи на широкі можливості, використання X-коду має ряд суттєвих обмежень (рис. 7.11):



Рисунок 7.11. Обмеження використання Х-коду

1. Залежність від середовища виконання

Адреси системних функцій та структура пам’яті можуть змінюватися залежно від версії операційної системи, що ускладнює забезпечення універсальності Х-коду [26]. Додаткові труднощі виникають при використанні різних версій системних бібліотек або механізмів захисту ОС.

2. Обмеженість доступного простору

У багатьох випадках обсяг вільного простору для розміщення Х-коду є обмеженим, що вимагає оптимізації його розміру та структури. Це особливо характерно для методів, що використовують заголовок PE-файлу або хвости секцій [19, 42].

3. Складність реалізації

Розроблення коректного та стабільного Х-коду потребує глибокого розуміння внутрішньої структури виконуваних файлів, механізмів завантаження та взаємодії з операційною системою [17, 19]. Крім того, необхідно враховувати особливості архітектури процесора та механізми адресації пам’яті.

4. Обмежена надійність окремих методів

Деякі підходи (зокрема використання низькорівневих або недокументованих API) можуть бути нестабільними та залежними від

конкретної реалізації операційної системи [26]. Подібні механізми можуть працювати некоректно після оновлення ОС або зміни внутрішньої структури системних компонентів.

5. Виявлення модифікацій

Зміни структури виконуваного файлу можуть бути виявлені засобами аналізу, що знижує ефективність приховування X-коду [18, 33]. Особливо це стосується методів, які явно змінюють таблицю імпорту або додають нові секції. Ознаками модифікації можуть бути зміни структури секцій, порушення параметрів заголовка або поява нетипових областей коду.

Розуміння обмежень застосування X-коду є важливим не лише для його розроблення, але й для аналізу програмного забезпечення. Виявлення ознак некоректної реалізації або нестабільності може свідчити про наявність стороннього втручання або спроби модифікації.

Застосування X-коду є складним і чутливим процесом, що вимагає дотримання строгих вимог до сумісності, коректності та безпечності. Недотримання цих вимог може призвести до порушення функціонування програми або спростити виявлення модифікації.

Запобіжні заходи та обмеження визначають практичні межі застосування X-коду та дозволяють оцінити його ефективність як інструменту модифікації та захисту програмного забезпечення. Їх врахування є необхідною умовою як для розроблення, так і для аналізу сучасних програмних реалізацій.

7.3. ДИНАМІЧНІ ПРИЙОМИ ПРОТИДІЇ АНАЛІЗУ КОДУ

На відміну від методів, орієнтованих переважно на статичний аналіз виконуваних файлів, динамічні прийоми протидії аналізу проявляються безпосередньо під час виконання програми [19, 33, 58]. Їх основною метою є приховування реальної логіки роботи програмного забезпечення, ускладнення налагодження, а також протидія дослідженню програм у середовищах динамічного аналізу.

Особливістю таких методів є те, що значна частина функціональності програми формується лише під час виконання, відновлюється у пам’яті динамічно, залежить від параметрів середовища виконання й може змінюватися залежно від дій аналітика або засобів дослідження [19, 57, 58].

У результаті статичний аналіз виконуваного файлу часто не дозволяє отримати повне уявлення про структуру та поведінку програми.

До основних динамічних методів протидії аналізу належать runtime unpacking, самозмінний код, anti-debugging-техніки, перевірка середовища виконання, а також емуляційні та віртуалізаційні механізми [33, 57-59]. Узагальнену характеристику цих підходів наведено у табл. 7.2.

Таблиця 7.2

Характеристика динамічних методів протидії аналізу

Метод	Основний принцип	Вплив на аналіз
Runtime unpacking	Динамічне розпакування коду у пам’яті	Ускладнення статичного аналізу
Self-modifying code	Модифікація інструкцій під час виконання	Приховування реального потоку виконання
Anti-debugging	Виявлення налагоджувача	Ускладнення покрокового аналізу
Перевірка середовища	Виявлення VM/sandbox	Протидія автоматизованому дослідженню
Віртуалізація коду	Використання проміжної віртуальної машини	Приховування реального алгоритму

Наведені у табл. 7.2 методи відрізняються механізмами реалізації та рівнем впливу на процес аналізу, однак усі вони спрямовані на ускладнення дослідження реальної логіки функціонування програмного забезпечення та підвищення стійкості до реверс-інжинірингу [33, 58].

Взаємозв’язок основних динамічних методів протидії аналізу програмного коду та їх вплив на процес дослідження виконуваних файлів узагальнено на рис. 7.12.

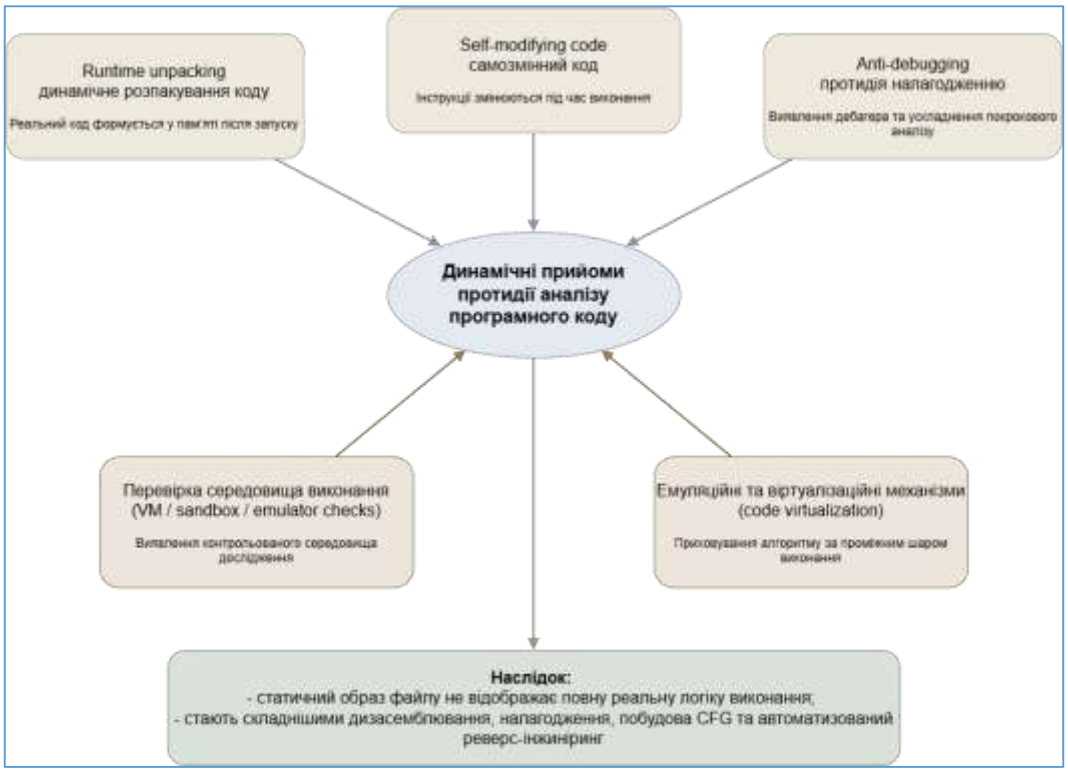


Рисунок 7.12. Основні динамічні прийоми протидії аналізу програмного коду

Як показано на рис. 7.12, динамічні методи протидії аналізу орієнтовані переважно на приховування реального потоку виконання програми та формування змінного представлення коду під час роботи програмного забезпечення [57-59]. Це суттєво ускладнює дизасемблювання, налагодження та автоматизований аналіз виконуваних модулів.

7.3.1. Динамічне саморозгортання та розшифрування коду

Одним із найпоширеніших динамічних методів є зберігання виконуваного коду у зашифрованому або модифікованому вигляді з подальшим його відновленням безпосередньо під час виконання програми [19, 33, 58].

Як показано на рис. 7.12, runtime unpacking та самозмінний код належать до динамічних методів приховування реального потоку виконання програми, при яких фактичний машинний код формується лише під час роботи програмного забезпечення [58].

У такому випадку на диску код міститься у прихованому вигляді, тоді як у пам’яті формується лише тимчасова робоча копія [19, 58]. Після виконання

окремі фрагменти можуть повторно шифруватися, змінювати своє розташування або видалятися.

Подібний підхід ускладнює статичне дизасемблювання, аналіз сигнатур, автоматичне виділення функцій, побудову графа керування програмою [17, 19, 58].

Одним із варіантів такого підходу є runtime unpacking – динамічне розпакування виконуваного коду у пам’яті. На диску програма зберігається у стисненому або обфускованому вигляді, а реальний код формується лише після запуску [19, 58].

У складніших реалізаціях застосовується самозмінний код (self-modifying code), при якому програма модифікує власні інструкції під час виконання, переписує адреси переходів або динамічно генерує нові фрагменти машинного коду [19, 33, 58]. У результаті реальний потік виконання може суттєво відрізнятися від статичного образу файлу.

Такі методи значно ускладнюють аналіз у дизасемблерах, автоматичну деобфускацію, дослідження захищеного або потенційно небезпечного програмного забезпечення.

7.3.2. Anti-debugging-техніки та перевірка середовища виконання

Методи anti-debugging спрямовані на виявлення налагоджувачів та ускладнення покрокового аналізу програми [57, 59].

На рис. 7.12 anti-debugging-техніки та перевірка середовища виконання представлені як механізми протидії контрольованому дослідженню програмного забезпечення у налагоджувальних і віртуалізованих середовищах.

Для цього використовуються перевірки системних прапорців налагодження, аналіз часових характеристик виконання, контроль цілісності окремих областей пам’яті, механізми виявлення точок зупину (breakpoints), а також перевірка виняткових ситуацій.

Якщо програма визначає наявність налагоджувача, вона може завершити роботу або приховати частину функціональності, змінити алгоритм виконання й сформувати помилкові результати аналізу.

Окрему групу становлять методи перевірки середовища виконання, які спрямовані на виявлення віртуальних машин, sandbox-середовищ, емуляторів, та інших контрольованих середовищ дослідження.

Метою таких перевірок є унеможливлення автоматизованого аналізу програми у спеціалізованому середовищі.

У сучасних системах захисту anti-debugging часто комбінується з обфускацією, динамічним дешифруванням, перевіркою цілісності й прихованими механізмами передачі керування [33, 57, 59].

7.3.3. Емуляційні та віртуалізаційні механізми

Одним із найбільш складних підходів до протидії аналізу є використання проміжного шару виконання між процесором та реальним програмним кодом [33, 58].

Як видно з рис. 7.12, емуляційні та віртуалізаційні механізми формують окрему групу динамічних методів, у межах яких реальна логіка програми приховується за проміжним шаром виконання або внутрішньою системою команд.

У таких системах реальний алгоритм представлений у вигляді спеціального байткоду, а виконання здійснюється внутрішньою віртуальною машиною або програмним інтерпретатором. У результаті навіть після дизасемблювання дослідник отримує не реальну логіку програми, а лише код внутрішнього інтерпретатора або віртуальної машини.

Основними перевагами таких підходів є:

- ускладнення реверс-інжинірингу;
- приховування алгоритмів;
- підвищення стійкості до автоматизованого аналізу;
- можливість динамічної зміни внутрішньої системи команд.

Разом із тим використання віртуалізаційних механізмів може призводити до зниження продуктивності, збільшення обсягу коду [58], ускладнення супроводу програмного забезпечення.

Подібні підходи широко застосовуються у DRM-системах, у засобах захисту програмного забезпечення, у комерційних системах обфускації та у сучасному шкідливому програмному забезпеченні.

Таким чином, динамічні методи протидії аналізу спрямовані на приховування реальної логіки функціонування програми безпосередньо під час її виконання. На відміну від статичних методів обфускації, вони формують змінне та залежне від середовища представлення програмного коду, що суттєво ускладнює дизасемблювання, налагодження та автоматизований реверс-інжиніринг.

Комплексне поєднання динамічних і статичних механізмів захисту дозволяє значно підвищити стійкість програмного забезпечення до дослідження, модифікації та несанкціонованого аналізу.

7.4. ОБФУСКАЦІЯ ПРОГРАМНОГО КОДУ

Обфускація програмного коду (code obfuscation) є одним із найбільш поширених підходів до ускладнення аналізу програмного забезпечення та протидії реверс-інжинірингу. Її основною метою є перетворення структури програми у форму, складну для розуміння людиною або автоматизованими засобами аналізу, при збереженні функціональної еквівалентності програмного забезпечення [33, 58].

На відміну від динамічних механізмів протидії аналізу, розглянутих у підрозділі 7.3, обфускація переважно орієнтована на зміну статичного представлення програмного коду та структури виконуваного файлу. Її основною метою є приховування логіки виконання програми, внутрішніх алгоритмів, а також структури потоків керування та обробки даних без активного блокування аналізу під час виконання програмного забезпечення.

Обфускація широко застосовується у системах захисту програмного забезпечення, у DRM-механізмах, комерційних засобах захисту інтелектуальної власності, у системах ліцензування, у мобільних та веб-застосунках [29, 33].

Разом із тим подібні методи можуть використовуватися і у шкідливому програмному забезпеченні для приховування реальної функціональності та ускладнення аналізу [18, 58].

Обфускація не забезпечує абсолютного захисту програмного забезпечення, однак суттєво підвищує трудомісткість дослідження, дизасемблювання та деобфускації коду.

7.4.1. Основні поняття та класифікація методів обфускації

Методи обфускації спрямовані на перетворення програмного коду у форму, складну для статичного аналізу, побудови графа керування, автоматичного виділення функцій, що ускладнює розуміння логіки виконання програми [33, 58].

Основні методи обфускації програмного коду наведено на рис. 7.13.

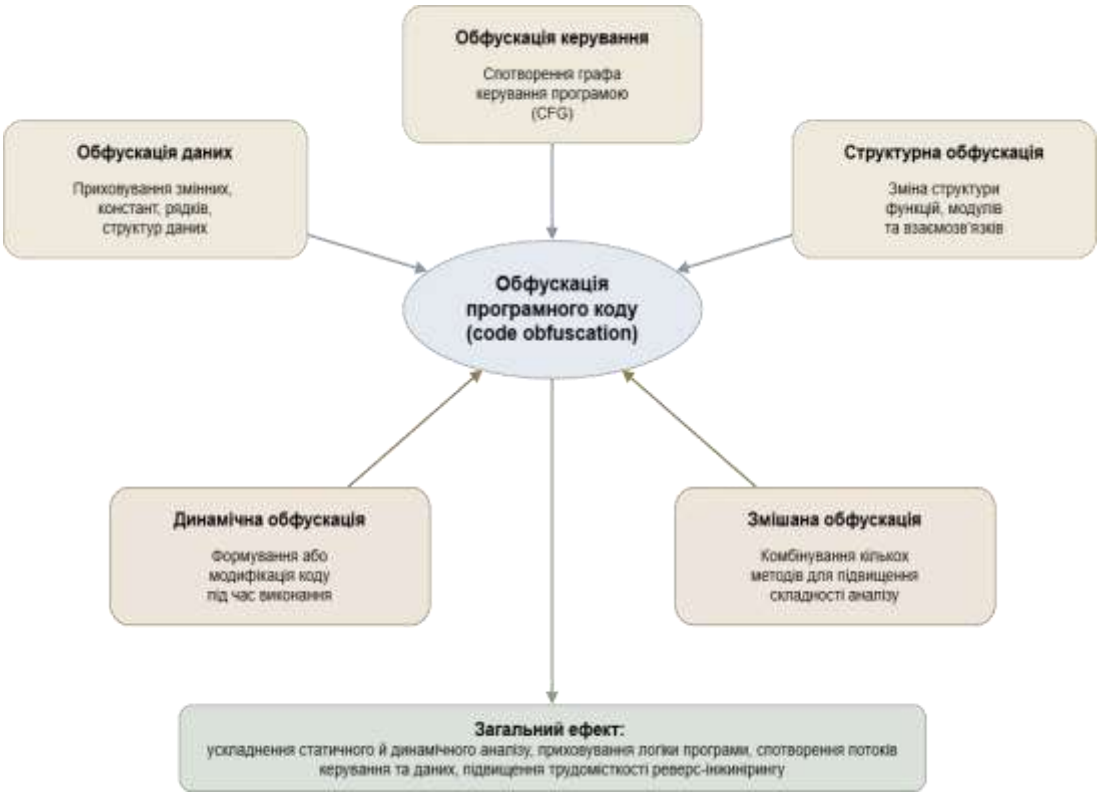


Рисунок 7.13. Класифікація методів обфускації програмного коду

Узагальнену характеристику основних груп методів обфускації наведено у табл. 7.3.

Таблиця 7.3

Основні групи методів обфускації програмного коду

Група методів	Основний принцип	Вплив на аналіз
Обфускація даних	Приховування структури та представлення даних	Ускладнення аналізу змінних і констант
Обфускація керування	Спотворення графа керування програмою	Ускладнення побудови CFG
Структурна обфускація	Зміна структури функцій та модулів	Приховування архітектури програми
Динамічна обфускація	Формування коду під час виконання	Ускладнення статичного аналізу
Змішана обфускація	Комбінування кількох підходів	Підвищення загальної складності аналізу

Як показано у табл. 7.3, різні групи методів обфускації впливають на окремі аспекти аналізу програмного забезпечення. На практиці найбільш ефективними є комбіновані підходи, у межах яких одночасно застосовуються методи приховування даних, спотворення графа керування та динамічного формування коду.

7.4.2. Обфускація даних

Обфускація даних спрямована на приховування внутрішнього представлення змінних, констант, рядків та інших структур даних, що використовуються програмою [33, 58].

Основними завданнями такого підходу є ускладнення аналізу внутрішньої логіки, приховування критичних параметрів, протидія сигнатурному аналізу, ускладнення пошуку ключових структур даних.

Обфускація даних передбачає трансформацію структур і типів даних із метою приховання логіки їх використання та ускладнення аналізу внутрішнього стану програми [33]. Наприклад, 16-розрядна двійкова послідовність

0000000000001100, може трактуватися як ціле число 12, символьний код або інша структура даних залежно від контексту використання.

До основних груп методів обфускації даних належать:

1. обфускація зберігання (storage obfuscation);
2. обфускація з'єднання (aggregation obfuscation);
3. методи розділення, об'єднання
4. методи кодування рядків, приховування констант;
5. методи реструктурування масивів;
6. модифікація області видимості змінних;

Розглянемо виділені групи методів детальніше.

1. При *обфускації зберігання* змінюється спосіб представлення даних, наприклад, рядки можуть формуватися динамічно, зберігатися у зашифрованому вигляді або відновлюватися через виклики функцій.

2. Обфускація з'єднання базується на об'єднанні незалежних змінних або розділенні пов'язаних структур даних. Наприклад заміна локальних змінних глобальними або спільними структурами даних. Допустимо, є дві різні функції:

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

Якщо ці функції не виконуються одночасно, декілька локальних змінних можуть бути замінені однією глобальною змінною,

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

Таке представлення ускладнює аналіз потоків даних та міжпроцедурних залежностей [33].

Переупорядкування змінює порядок елементів масивів або полів структур, тоді як модифікація області видимості ускладнює міжпроцедурний аналіз змінних.

3. Для ускладнення аналізу можуть застосовуватися *методи розділення, об'єднання та кодування* змінних (variable splitting).

Метод розділення. У цьому випадку значення однієї змінної представляється через декілька допоміжних змінних або арифметичних

перетворень [33, 58].

Приклад 1: Змінні фіксованого діапазону можуть бути розділені на дві й більше змінних. Для цього змінну V, що має тип x розділяють на k змінних $v_1,...,v_k$ типу y тобто $V=v_1,...,v_k$. Потім створюється набір функцій що дозволяють витягати змінну типу x зі змінних типу y і записувати змінну типу x у змінні типу y.

Приклад 2: Зміна представлення (кодування). Наприклад, цілочисельну змінну i у нижче представленому фрагменті коду (мова C):

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

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

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

Метод об’єднання змінних – це об’єднання декількох змінних в одну структуру або змінну більшого розміру [33]. Наприклад, дві 16-бітні змінні можуть бути представлені однією 32-бітною змінною, а доступ до початкових значень здійснюється через спеціальні арифметичні перетворення.

Наприклад, дві 16-бітні цілочисельні змінні X та Y можуть бути об’єднані в одну 32-бітну змінну Z за формулою:

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

У такому випадку початкові значення X та Y зберігаються у складі однієї змінної, а доступ до них здійснюється через арифметичні перетворення. Наприклад, фрагмент коду:

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

X += 5;
```

після обфускації може бути представлений у вигляді:

```
int Z = 262156;
```

```
Z += 5;
```

Подібний підхід ускладнює аналіз залежностей між змінними та приховує їх реальну структуру у програмі.

4. Кодування рядків та приховування констант. До поширених методів обфускації даних належать кодування рядків, приховування констант, динамічне формування даних, приховування таблиць переходів, а також використання надлишкових змінних і допоміжних структур даних.

Одним з підходів є приховування рядкових констант (string obfuscation) [33,58]. Це можна зробити, записуючи кожен символ рядка, у форматі ASCII-коду, наприклад символ "A" можна записати як шістнадцяткове число "0x41", але такий метод занадто простий. Ефективнішим є метод, коли текстові значення зберігаються у зашифрованому вигляді, формуються динамічно, розбиваються на фрагменти, відновлюються лише під час виконання. Наприклад, рядок може генеруватися спеціальною функцією на основі переданих параметрів, а числові константи – представлятися у вигляді арифметичних виразів. Так, число 1 можна представити виразом

(a+1-b),

де a = b.

Таке перетворення ускладнює сигнатурний аналіз та пошук критичних фрагментів коду.

5. Реструктурування масивів. Для ускладнення аналізу можуть застосовуватися методи реструктурування масивів [33], які базуються на розділенні одного масиву на декілька підмасивів, об’єднанні кількох масивів або зміні розмірності структур даних. Наприклад, елементи масиву можуть розподілятися між окремими структурами залежно від парності індексу або представлятися у вигляді багатовимірного масиву. Початковий фрагмент коду:

```
intA[] = {1, 2, 3, 4, 5, 0} ;
i = 3 ;
A[i] = ... ;
```

після реструктурування масиву може бути представлений у вигляді:

```
int A1[] = {2, 4, 0};
int A2[] = {1, 3, 5};
```

```
if ((i % 2) == 0)
    A1[i / 2] = ...;
else
    A2[i / 2] = ...;
```

Подібне перетворення знижує прозорість коду й структури даних та логіки доступу до елементів масиву.

6. Для ускладнення міжпроцедурного аналізу можуть застосовуватися *методи модифікації області дії змінних*. При локалізації змінних створюються окремі набори змінних для різних базових блоків, тоді як розширення області дії передбачає перенесення локальних змінних на рівень функції або глобальної області видимості. Подібні перетворення ускладнюють аналіз потоків даних та взаємозв'язків між окремими фрагментами програми [33].

В якості ще одного прикладу розглянемо один із найперших підходів до обфускації є символьна (лексична) обфускація, яка базується на зміні імен ідентифікаторів та текстового представлення програмного коду [33, 58].

Основною метою такого підходу є ускладнення аналізу структури програми, приховування призначення функцій і змінних, зменшення читабельності декомпільованого коду й ускладнення автоматичного аналізу.

До основних методів символьної обфускації належать:

– перейменування методів і змінних у випадкові набори символів;
– скорочення імен ідентифікаторів;
– використання Unicode-символів;
– застосування нетипових або зарезервованих ключових слів;
– зміна імен на такі, що спотворюють логіку сприйняття коду.

Приклади символьної обфускації наведено у табл. 7.4.

Таблиця 7.4

Приклади символьної обфускації

До обфускації	Після обфускації
GetPassword()	KJHS92DSLKaf()
GetConnectionString()	_0()
SecurityInformation	Car
GetInformation()	Wash()

Як показано у табл. 7.4, після обфускації імена функцій, класів та змінних втрачають логічний зміст, що ускладнює розуміння структури програми під час декомпіляції або дизасемблювання.

Окремі обфускатори використовують Unicode-символи або нестандартні кодування для ускладнення роботи окремих декомпіляторів та інструментів аналізу. Також можуть застосовуватися ідентифікатори, які збігаються із зарезервованими словами мов програмування або мають оманливий зміст.

Наприклад, фрагмент програмного коду (мова Perl) до лексичної обфускації може мати вигляд:

```
sub is_tainted {
    my $arg = shift;
    return length($arg) != 0;
}
```

Після обфускації й видалення форматування коду той самий фрагмент набуває вигляду:

```
sub z109276e1f2{my $z4fe8df46b1 = shift(@_);return length($z4fe8df46b1)! = 0;}
```

У результаті функціональність програми залишається незмінною, однак читабельність та зрозумілість коду суттєво погіршуються.

Перевагами символьної обфускації є відносна простота реалізації, незначний вплив на продуктивність, можливість автоматизованого застосування.

Разом із тим сучасні декомпілятори та системи аналізу часто автоматично замінюють подібні ідентифікатори на умовні позначення типу method_1, func_2 або var_3, що частково знижує ефективність такого підходу. Тому на практиці символьна обфускація найчастіше використовується у поєднанні з іншими методами приховування структури та логіки програмного коду.

Таким чином, методи обфускації даних спрямовані на приховування внутрішнього представлення змінних, структур і констант, що суттєво ускладнює аналіз потоків даних, реконструкцію логіки роботи програми та автоматизовану деобфускацію. На практиці найбільш ефективними є комбіновані підходи, які поєднують структурні, арифметичні та символьні методи приховування даних.

7.4.3. Обфускація та маскування графа керування програмою

Найбільш складними для реалізації є методи обфускації графа потоку керування, які базуються на створенні нетривіальних схем переходів між базовими блоками програми [33, 58]. Основною метою таких підходів є ускладнення відновлення логіки виконання та побудови коректного графа керування програмою.

Обфускація керування (control-flow obfuscation) спрямована на ускладнення аналізу графа керування програмою (CFG). Основною метою таких методів є приховування реальної послідовності виконання, ускладнення виділення функцій, формування хибних переходів, збільшення складності автоматичного аналізу.

До поширених методів обфускації керування належать:

– сплощення потоку керування (flattening);
– використання непрозорих предикатів (opaque predicates);
– вставка фіктивних переходів;
– непрямі переходи;
– перемішування базових блоків;
– зчеплення дуг (arc hooking);
– клонування базових блоків (basic block cloning).

Взаємозв’язок основних методів обфускації керування наведено на рис. 7.14.

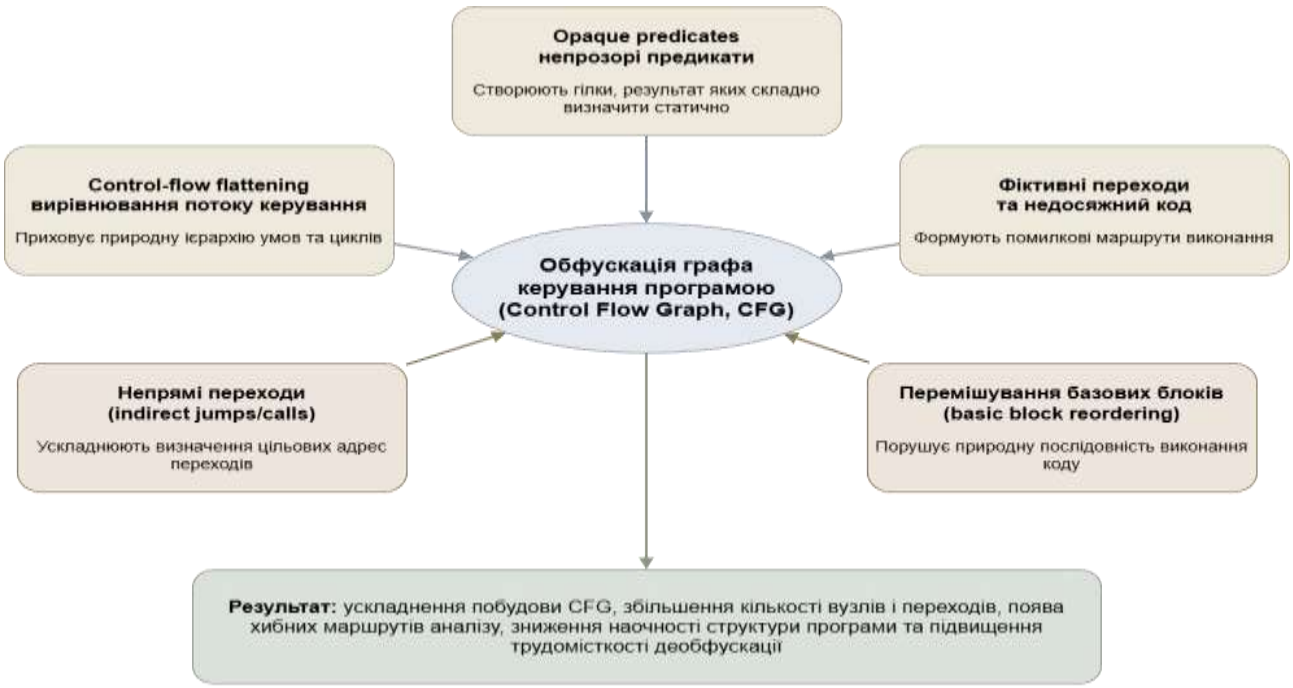


Рисунок 7.14. Методи обфускації графа керування програмою

Одним із найбільш відомих методів є сплощення потоку керування (control-flow flattening), при якому структура програми перетворюється у централізований механізм диспетчеризації переходів. У такій схемі природна ієрархія умовних конструкцій приховується, а виконання окремих блоків контролюється спеціальною змінною стану.

Іншим важливим методом є використання непрозорих предикатів – opaque predicates.

Предикатом у програмуванні називають логічний вираз або функцію, результатом виконання якої є булеве значення (`true` або `false`).

У системах обфускації особливу роль відіграють *непрозорі предикати* (`opaque predicates`) – умови, результат яких наперед відомий розробнику, але складний або практично неможливий для однозначного визначення засобами статичного аналізу.

Приклад використання opaque predicates:

```
if ((x * x) >= 0)
{
    execute_real_branch();
}
else
```

```
{
    fake_branch();
}
```

У наведеному прикладі умова (x * x) >= 0 для цілих чисел завжди є істинною, однак для аналізатора така конструкція виглядає як повноцінне розгалуження виконання [33].

Непрозорі предикати можуть бути трьох видів:

P(f) – предикат, що завжди має значення "false";

P(t) – предикат, що завжди має значення "true";

P(t,f) – предикат, що може набувати різних значень, однак під час обфускації його поточний результат відомий розробнику. На рис. 7.15 наведено варіанти використання (opaque predicates) у графі керування програмою.

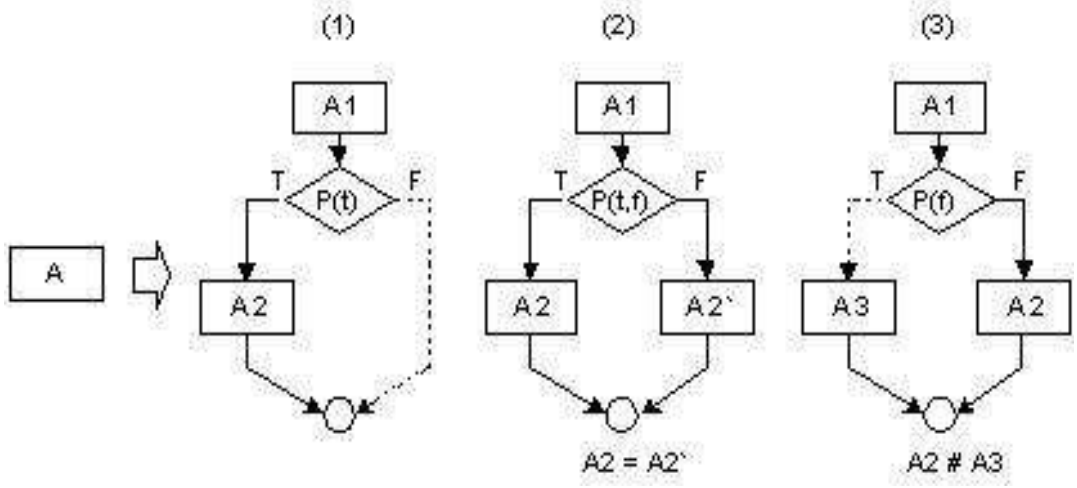


Рисунок 7.15. Варіанти побудови opaque predicates

У схемі (1) рисунку 7.15 показано базове розгалуження з непрозорим предикатом, у схемі (2) – формування еквівалентних гілок виконання (`A2 = A2′`), а у схемі (3) – створення альтернативної або фіктивної гілки керування (`A2 ≠ A3`). Для аналізатора подібні конструкції виглядають як повноцінні гілки виконання, що призводить до штучного збільшення CFG, появи помилкових маршрутів та ускладнення деобфускації.

До більш складних методів належить зчеплення дуг (arc hooking), при якому між базовими блоками додаються проміжні керуючі фрагменти коду з використанням непрозорих предикатів або допоміжних змінних стану. Такі

підходи створюють додаткові шляхи виконання та ускладнюють побудову коректного графа керування програми.

Приклад arc hooking наведено на рис. 7.16.

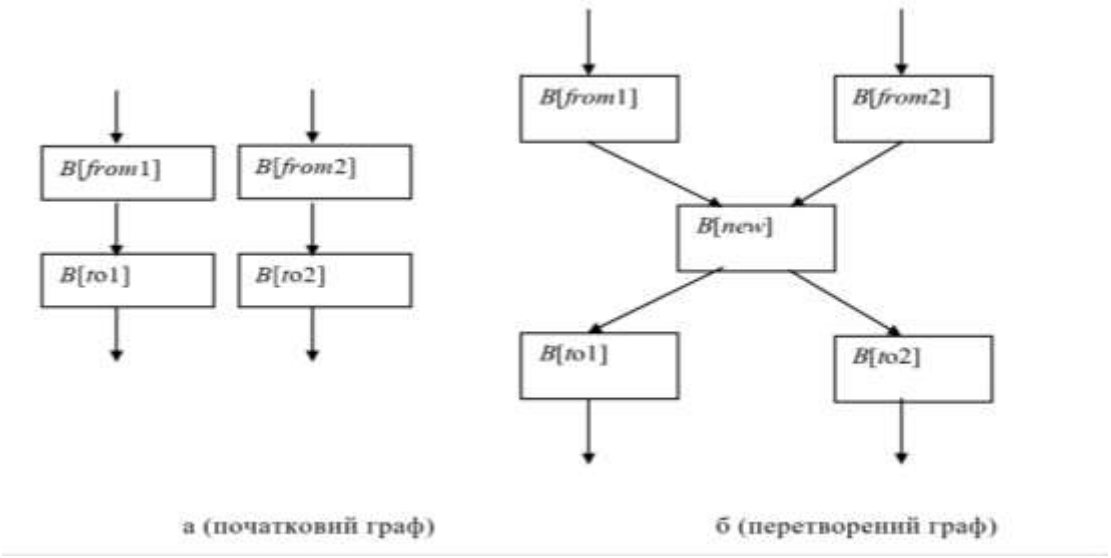


Рисунок 7.16. Приклад зчеплення дуг графа керування (arc hooking)

У такій схемі між базовими блоками вставляються додаткові керуючі фрагменти коду B[new] або проміжні переходи, що формують нові маршрути виконання. Подібні конструкції збільшують складність CFG, ускладнюють автоматичне виділення реальних шляхів виконання та підвищують трудомісткість деобфускації програмного коду.

Для додаткового ускладнення аналізу можуть використовуватися клонування базових блоків та формування множинних еквівалентних шляхів виконання, що порушує однозначність побудови CFG і ускладнює реконструкцію логіки програми.

У методах basic block cloning окремі фрагменти коду дублюються, після чого виконання може передаватися до одного з декількох еквівалентних блоків. При цьому різні шляхи виконання приводять до однакового результату, але значно ускладнюють статичний аналіз та автоматичне відновлення структури програми.

Приклад клонування базових блоків наведено на рис. 7.17.

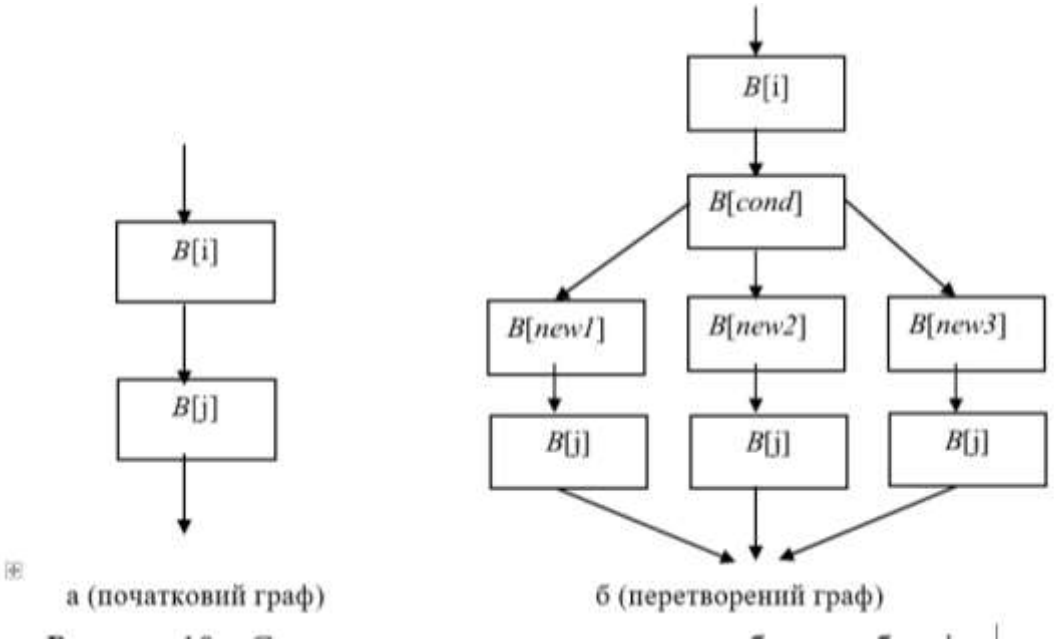


Рисунок 7.17. Клонування базових блоків (basic block cloning)

Незважаючи на однаковий результат виконання, наявність множинних шляхів ускладнює побудову коректного CFG, порушує однозначність аналізу та підвищує складність автоматичного відновлення логіки програми.

Додатково можуть застосовуватися вставка мертвого коду, ускладнення арифметичних виразів, нестандартне представлення структур даних, а також динамічне формування окремих фрагментів коду під час виконання програми.

До методів маскування графа керування також належать перетворення циклів (loop transformations), спрямовані на зміну структури виконання без порушення функціональної коректності програми. Подібні підходи можуть включати розгортання циклів (loop unrolling), розкладання циклів (loop decomposition) та зміну структури індексного простору. Такі трансформації ускладнюють відновлення початкової логіки алгоритмів і побудову коректного графа керування.

Приклад розгортання циклу (loop unrolling):

```
for (i = 0; i < n; i++)
{
    process(i);
}
```

після трансформації цикл може набувати вигляду:

```
for (i = 0; i < n; i += 2)
```

```
{
   process(i);
   process(i + 1);
}
```

У наведеному прикладі кількість переходів у циклі зменшується через обробку декількох ітерацій за один прохід, що змінює структуру графа керування та ускладнює аналіз потоків виконання.

На рис. 7.18 наведено приклад перетворення циклів у процесі маскування графа керування програми.

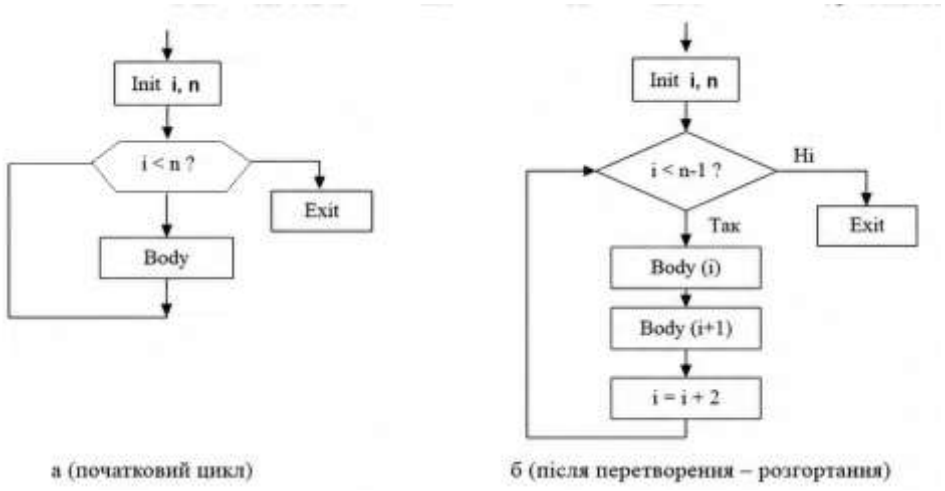


Рисунок 7.18. Приклад перетворення циклів (loop transformations)

У складних програмних системах можуть використовуватися методи перемішування логіки виконання (interleaving), при яких фрагменти декількох функцій або циклів об’єднуються у спільну керуючу структуру з використанням змінної стану. Подібні підходи суттєво ускладнюють аналіз потоків керування та контекстів виклику функцій.

Спрощений приклад interleaving:

```
switch(state)
{
   case 0:
      funcA();
      state = 1;
      break;

   case 1:
      funcB();
      state = 2;
```

```
        break;
}
```

У такій схемі логіка декількох функцій виконується в межах єдиної керуючої структури, що ускладнює відновлення початкових потоків виконання та автоматичний аналіз взаємозв’язків між функціями.

На рис. 7.19 наведено приклад interleaving у графі керування програмою.

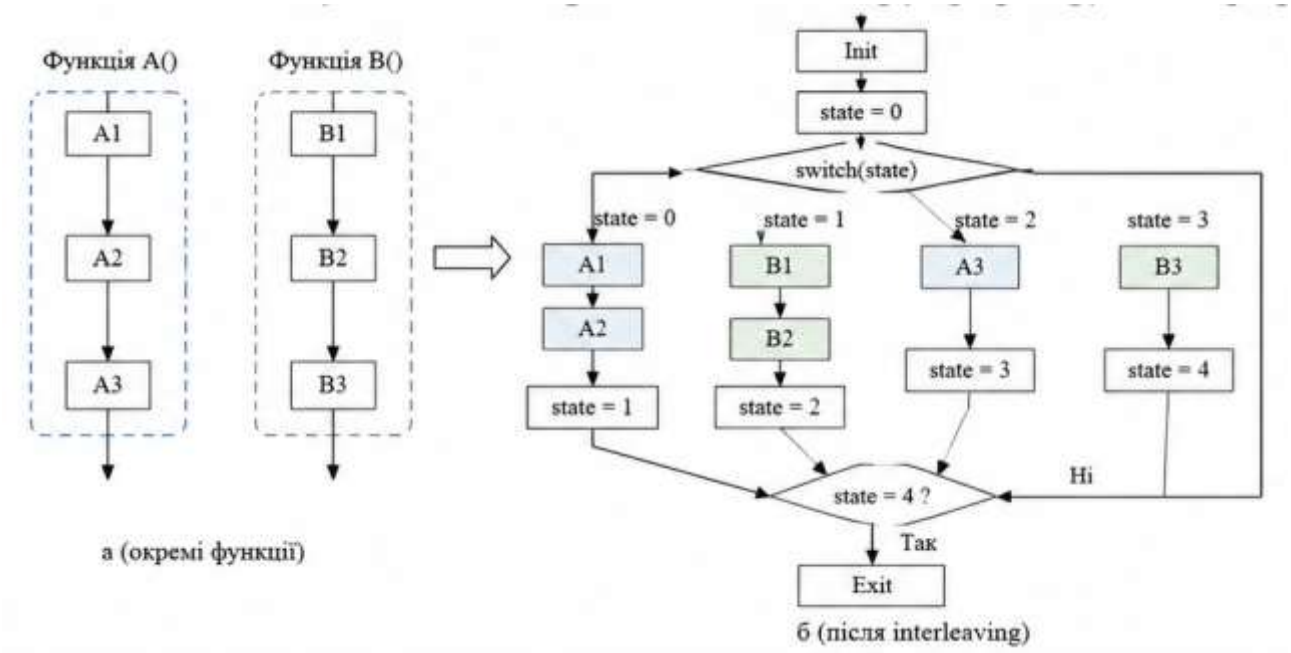


Рисунок 7.19. Приклад interleaving у графі керування програмою

У початковому варіанті функції `A()` та `B()` виконуються незалежно одна від одної, однак після трансформації їх логіка переплітається в межах єдиної керуючої структури з використанням змінної стану (`state`). Подібний підхід ускладнює аналіз потоків виконання, приховує реальні межі функцій та підвищує складність автоматичного відновлення логіки програми під час статичного й динамічного аналізу.

У сукупності такі підходи призводять до спотворення структури графа керування програми, ускладнюють автоматичне відновлення логіки виконання та підвищують стійкість програмного забезпечення до реверс-інжинірингу.

7.4.4. Структурна та модульна обфускація

Структурна обфускація спрямована на приховування архітектури програмного забезпечення та взаємозв’язків між окремими компонентами [33, 58].

Для цього можуть застосовуватися дроблення функцій, об’єднання незалежних модулів, приховування точок входу, зміна структури викликів, штучне ускладнення залежностей між компонентами. Також можуть застосовуватися методи видалення або додавання абстракцій коду, при яких окремі функції замінюються безпосередньо їх тілом або, навпаки, розбиваються на декілька менших взаємопов’язаних функцій. Подібні перетворення ускладнюють реконструкцію архітектури програмного забезпечення та аналіз міжфункціональних залежностей.

У результаті ускладнюється аналіз архітектури програми, приховуються критичні модулі, порушується логіка навігації між функціями.

Одним із поширених підходів є дроблення функцій (function splitting), при якому одна логічна функція розділяється на декілька взаємопов’язаних фрагментів із додатковими переходами між ними. Це ускладнює реконструкцію алгоритму та побудову цілісної структури програми.

У об’єктно-орієнтованих системах можуть застосовуватися методи зміни ієрархії успадкування, що базуються на введенні додаткових проміжних класів, штучному ускладненні залежностей між компонентами та реструктуризації механізмів наслідування. Подібні підходи ускладнюють аналіз архітектури програмного забезпечення та взаємозв’язків між класами.

Також можуть застосовуватися методи приховування точок входу та модифікації механізмів виклику функцій, у межах яких змінюється структура міжмодульної взаємодії та ускладнюється навігація між компонентами програми.

Подібні методи особливо ефективні у великих програмних системах та багатомодульних застосунках.

7.4.5. Динамічна обфускація

Динамічна обфускація базується на формуванні або модифікації коду безпосередньо під час виконання програми [33, 58].

На відміну від динамічних методів протидії аналізу, розглянутих у підрозділі 7.3, динамічна обфускація використовується переважно для приховування реальної структури програмного коду та формування змінного представлення інструкцій під час виконання програми.

На відміну від статичних методів, у цьому випадку частина інструкцій генерується динамічно, й код може змінюватися у пам’яті. Окремі фрагменти дешифруються лише перед виконанням.

До таких методів належать [19, 33, 58]:

– runtime unpacking;
– динамічне дешифрування;
– self-modifying code;
– генерація інструкцій у пам’яті.

У подібних системах реальний програмний код може зберігатися у зашифрованому, стисненому або модифікованому вигляді та відновлюватися лише під час виконання програми. У результаті статичний образ виконуваного файлу не містить повної інформації про фактичну логіку роботи програмного забезпечення.

Основною метою динамічної обфускації є ускладнення статичного дизасемблювання, сигнатурного аналізу й автоматичної деобфускації.

При цьому подібні методи часто поєднуються з anti-debugging, перевіркою цілісності, прихованими переходами та механізмами контролю середовища виконання [57, 59].

Динамічна обфускація є одним із найбільш складних методів приховування програмного коду, оскільки поєднує механізми модифікації інструкцій, runtime-генерації коду та протидії аналізу середовища виконання. На практиці подібні підходи широко застосовуються у DRM-системах, засобах

захисту програмного забезпечення та сучасному шкідливому програмному забезпеченні [29, 33, 58].

Реалізація складних методів обфускації потребує врахування особливостей архітектури процесора, довжин інструкцій, механізмів адресації та умовних переходів. Помилки при трансформації коду можуть призводити до зміщення адрес, порушення логіки виконання або нестабільної роботи програмного забезпечення.

7.4.6. Оцінювання ефективності обфускації

Ефективність обфускації визначається не лише складністю перетворення коду, а й впливом таких методів на стійкість до аналізу, продуктивність програми, розмір виконуваного файлу, складність супроводу програмного забезпечення [33, 58].

Основні критерії оцінювання ефективності обфускації наведено у табл. 7.5.

Таблиця 7.5

Критерії оцінювання ефективності обфускації

Критерій	Характеристика
Стійкість до аналізу	Рівень ускладнення реверс-інжинірингу
Складність CFG	Рівень ускладнення графа керування
Вплив на продуктивність	Зміна швидкодії програми
Збільшення розміру коду	Ріст обсягу виконуваного файлу
Складність деобфускації	Трудомісткість відновлення логіки програми

Як показано у табл. 7.5, ефективність обфускації оцінюється не лише рівнем приховування програмного коду, але й впливом застосованих методів на характеристики функціонування програмного забезпечення.

Одним із ключових критеріїв є стійкість до аналізу, яка визначає рівень ускладнення реверс-інжинірингу, автоматизованого дизасемблювання та деобфускації програмного коду. Важливу роль також відіграє складність графа керування (CFG), оскільки більшість сучасних методів обфускації спрямовані саме на спотворення структури переходів між базовими блоками програми.

Разом із тим застосування обфускації може негативно впливати на продуктивність програмного забезпечення, збільшувати розмір виконуваного файлу та ускладнювати супровід програмної системи.

Надмірна обфускація може призводити до зниження продуктивності, ускладнення супроводу, збільшення кількості помилок та погіршення сумісності програмного забезпечення.

Особливо це характерно для динамічної обфускації та методів, що базуються на runtime-генерації або модифікації коду під час виконання програми.

Тому на практиці обфускація застосовується як компроміс між рівнем захисту та ефективністю функціонування програмної системи.

Таким чином, обфускація програмного коду є важливим компонентом сучасних систем захисту програмного забезпечення. Її основне призначення полягає у приховуванні структури, логіки та потоків виконання програми з метою ускладнення аналізу та реверс-інжинірингу.

На відміну від механізмів runtime-захисту, обфускація переважно не блокує аналіз безпосередньо, а підвищує його трудомісткість шляхом трансформації програмного коду та спотворення його внутрішнього представлення.

Найбільша ефективність досягається при комплексному поєднанні методів обфускації даних, графа керування, структурної та динамічної обфускації з іншими механізмами захисту програмного забезпечення, зокрема anti-debugging, перевіркою цілісності та механізмами контролю середовища виконання.

7.4.7. Інструменти та приклади обфускаторів

Інструменти (обфускатори) автоматизують перетворення програмного коду: перейменування символів, шифрування рядків, зміну структури даних, ускладнення графа керування, вставку anti-debugging- та anti-emulation-механізмів, а також віртуалізацію коду [33, 58]. Окрему групу становлять packers – засоби пакування виконуваних файлів, які стискають або шифрують секції

бінарного коду. У таких системах розпакування відбувається лише під час виконання програми, що суттєво ускладнює статичний аналіз виконуваного файлу [19, 33]. Обфускатори відрізняються за рівнем застосування (вихідний код, проміжний байткод, виконуваний файл), набором підтримуваних трансформацій та типом ліцензування (open-source або комерційні рішення).

До поширених інструментів-обфускаторів належать:

− Онлайн-обфускатори для JavaScript: JS-Obfuscator, WTOOLS;

− Інструменти для C/C++: Tigress, Picheta;

− Засоби для мобільних та нативних застосунків: Obfuscator.app;

− Обфускатори для .NET та Java, що підтримують перетворення IL/CIL-коду.

Сучасні обфускатори можуть працювати як на рівні вихідного коду, так і на рівні проміжного представлення або виконуваного файлу, що дозволяє комбінувати різні методи приховування логіки програми.

Приклади онлайн-обфускаторів наведено на рис. 7.20 [58].

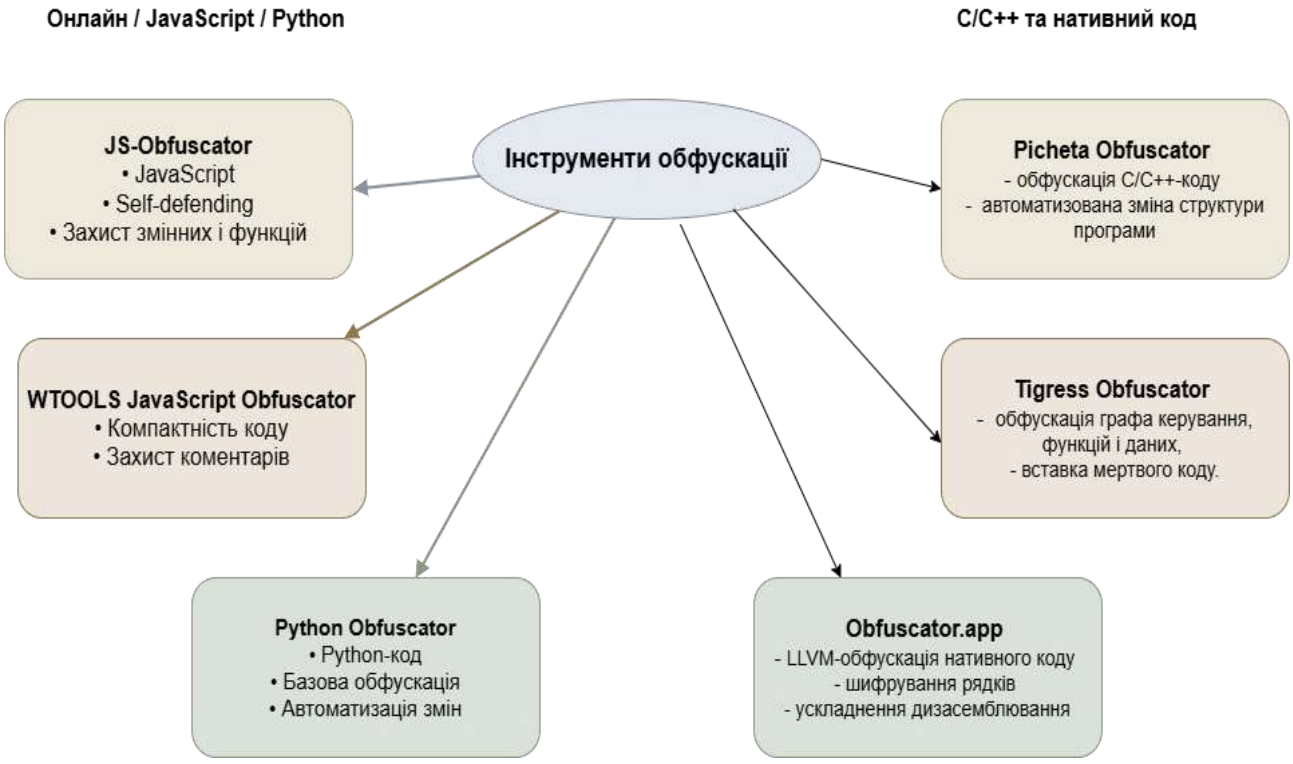


Рисунок 7.20. Приклади іструментів обфускації програмного коду

Онлайн-обфускатори для JavaScript та Python

– JS-Obfuscator – інструмент для обфускації JavaScript-коду, який підтримує декілька рівнів захисту, приховування змінних і функцій, захист від форматування та механізми self-defending.

– WTOOLS JavaScript Obfuscator – онлайн-інструмент для обфускації JavaScript-коду з можливістю налаштування параметрів захисту змінних, коментарів та компактності коду.

– Python Obfuscator – інструмент для обфускації Python-коду, що дозволяє автоматизувати базові методи приховування структури програмного коду.

– Інструменти обфускації для C/C++ та нативного коду

– Picheta Obfuscator – онлайн-інструмент для обфускації C/C++-коду на рівні вихідного тексту програми. Дозволяє автоматизовано змінювати структуру коду без втрати функціональності.

– Tigress Obfuscator – один із найбільш відомих дослідницьких обфускаторів для C/C++, який підтримує методи обфускації графа керування, даних та функцій. Інструмент широко використовується у наукових дослідженнях та експериментах із протидії реверс-інжинірингу.

– Obfuscator.app – засіб обфускації на основі технології LLVM (Low Level Virtual Machine), призначений для захисту мобільних і нативних застосунків на рівні виконуваного коду.

При виборі інструмента обфускації доцільно враховувати можливість тонкого налаштування трансформацій, підтримку CI/CD-процесів, сумісність із середовищем розроблення та можливість збереження відладної інформації для розробників.

До основних переваг обфускаторів належать:

– автоматизація процесу захисту програмного коду;

– швидке підвищення складності реверс-інжинірингу;

– можливість інтеграції у pipeline розроблення;

– підтримка багаторівневих механізмів захисту.

Разом із тим обфускатори не забезпечують абсолютного захисту програмного забезпечення. Досвідчений аналітик або спеціалізовані засоби деобфускації можуть частково відновити структуру програми. Крім того, використання складних методів обфускації може негативно впливати на продуктивність, тестування та супровід програмного забезпечення.

На практиці обфускатори та пакувальники часто використовуються у поєднанні з криптографічними механізмами, контролем цілісності, апаратними dongle або мережевою активацією, формуючи багаторівневу систему захисту програмного забезпечення.

Таким чином, інструменти обфускації є важливим практичним засобом підвищення стійкості програмного забезпечення до аналізу, однак найбільша ефективність досягається лише при їх комплексному використанні разом з іншими механізмами захисту програмного коду.

7.5. КОМПЛЕКСНІ МЕТОДИ МАСКУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Маскування програми – це сукупність трансформацій програмного коду та структури виконуваного файлу, спрямованих на ускладнення аналізу, модифікації та зворотного інжинірингу при збереженні функціональної еквівалентності програмного забезпечення [33, 58].

На відміну від окремих методів обфускації, розглянутих у підрозділі 7.4, комплексне маскування передбачає комбінування декількох трансформацій, які одночасно впливають на граф керування, структуру функцій, організацію даних та порядок виконання програми.

Загальна ідея маскування полягає в істотному підвищенні складності аналізу CFG, структури викликів функцій та взаємозв’язків між компонентами програми.

Основні етапи процесу маскування наведено на рис. 7.21.

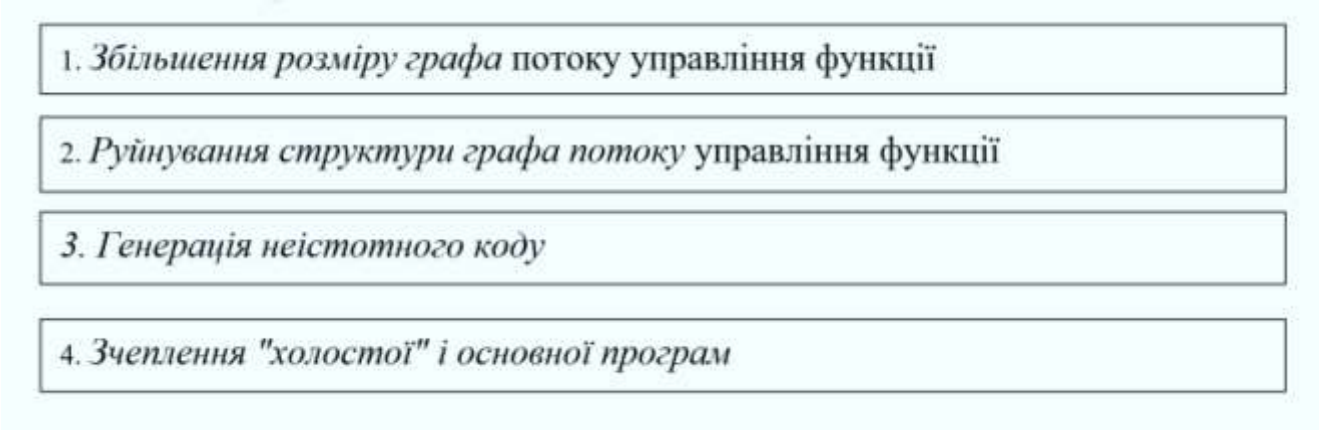


Рисунок 7.21. Етапи процесу маскування

Виділяють чотири основні етапи маскування:

– збільшення розміру графа потоку керування;
– руйнування структури CFG;
– генерація неістотного коду;
– зчеплення функціональної та «холостої» частин програми.

На етапі збільшення розміру графа потоку керування можуть застосовуватися перетворення циклів, клонування блоків і додавання нових маршрутів виконання.

На етапі руйнування структури CFG до графа потоку керування додаються нові дуги, виконуються розбиття базових блоків та створення проміжних порожніх блоків.

Етап генерації неістотного коду передбачає вставку інструкцій, які не змінюють функціональний результат роботи програми, але ускладнюють аналіз структури коду.

На завершальному етапі здійснюється зчеплення функціональних і допоміжних фрагментів коду з використанням непрозорих предикатів, непрямих переходів та інших механізмів ускладнення аналізу.

7.5.1 Маніпулювання функціями

Маніпулювання функціями є одним із поширених методів комплексного маскування програмного забезпечення. Такі трансформації спрямовані на зміну структури викликів, модульної організації та взаємозв'язків між функціями без зміни функціональної поведінки програми.

На відміну від методів обфускації графа керування, розглянутих у підрозділі 7.4.3, у даному випадку основна увага приділяється трансформації структури функцій і графа викликів (call graph).

Основні методи маніпулювання функціями наведено у табл. 7.6 [33].

Таблиця 7.6

Методи маніпулювання функціями

Метод	Сутність	Вплив на аналіз
Inlining	Підстановка тіла функції у місце виклику	Приховування структури викликів
Outlining	Виділення частини коду в окремі функції	Ускладнення реконструкції логіки
Eliminating library calls	Заміна стандартних бібліотечних викликів	Приховування семантики функцій
Interleaving	Переплетення декількох функцій	Ускладнення аналізу потоків керування
Function cloning	Створення декількох версій функцій	Ускладнення аналізу контекстів виклику

1. Відкрита вставка (inlining)

Метод полягає у підстановці тіла функції безпосередньо у місце її виклику. У результаті зменшується кількість явних викликів функцій і приховується структура взаємодії між модулями.

2. Винесення операторів (outlining)

У цьому випадку окремі фрагменти коду виділяються в нові функції. Такий підхід створює додаткові рівні викликів та змінює очікувану структуру програми.

3. Усунення бібліотечних викликів (eliminating library calls)

Метод передбачає заміну стандартних бібліотечних функцій власними реалізаціями. Це ускладнює семантичний аналіз програми, оскільки відомі бібліотечні сигнатури перестають однозначно ідентифікувати функціональність коду.

4. Переплетення функцій (interleaving)

Метод базується на об’єднанні декількох функцій у межах єдиної керуючої структури з використанням змінної стану (state).

Спрощений приклад interleaving:

```
switch(state)
{
    case 0:
        funcA();
        state = 1;
        break;

    case 1:
        funcB();
        state = 2;
        break;
}
```

У такій схемі логіка декількох функцій виконується в межах єдиної керуючої структури, що ускладнює аналіз контекстів виклику та відновлення початкової структури програми.

5. Клонування функцій (function cloning)

Метод передбачає створення декількох функціонально еквівалентних версій однієї функції із застосуванням різних маскуючих трансформацій. У результаті різні виклики можуть використовувати різні клони, що ускладнює аналіз контекстів використання функцій.

Перевагою методів маніпулювання функціями є значне ускладнення реконструкції модульної структури програми, однак їх застосування може призводити до збільшення розміру бінарного файлу та ускладнення процесу налагодження.

7.5.2. Недосяжний, мертвий та надлишковий код

О дним із поширених механізмів маскування програмного забезпечення є генерація неістотного коду, який ускладнює статичний і динамічний аналіз програми.

Основні типи такого коду наведено у табл. 7.7 [33].

Таблиця 7.7

Типи неістотного коду

Тип коду	Особливість
Недосяжний (unreachable code)	Ніколи не виконується
Мертвий (dead code)	Виконується, але не впливає на результат
Надлишковий (redundant code)	Може бути спрощений або видалений

Недосяжний код являє собою фрагменти програми, які ніколи не виконуються під час роботи системи. Подібні конструкції можуть містити довільні обчислення або псевдологіку, що ускладнює аналіз структури програми.

Мертвий код виконується під час роботи програми, однак його виконання не впливає на результат роботи системи. При цьому такий код не повинен змінювати глобальний стан програми або викликати побічні ефекти.

Надлишковий код містить операції, результати яких дублюють вже виконані обчислення або можуть бути замінені еквівалентними спрощеними виразами.



Рисунок 7.22. Етапи генерації неістотного коду

На рис. 7.22 наведено основні етапи генерації неістотного коду у процесі маскування програмного забезпечення.

На першому етапі формується набір типів і шаблонів допоміжних конструкцій, які можуть використовуватися для створення додаткових фрагментів коду.

Далі генеруються допоміжні змінні, тимчасові обчислення та проміжні структури даних.

На завершальному етапі неістотні інструкції інтегруються у базові блоки програми або між функціональними фрагментами коду.

Подібні трансформації не змінюють функціональну поведінку програми, однак збільшують складність статичного та динамічного аналізу, ускладнюють побудову графа керування та автоматичне відновлення логіки виконання. У сучасних системах захисту генерація неістотного коду часто поєднується з непрозорими предикатами, перемішуванням базових блоків та динамічним формуванням фрагментів коду під час виконання програми.

7.5.3. Практичні рекомендації щодо застосування методів маскування

Ефективність маскування програмного забезпечення значною мірою залежить від правильного комбінування трансформацій, рівня їх інтеграції у процес розроблення та збереження балансу між захистом і продуктивністю.

Основні практичні рекомендації наведено у табл. 7.8 [33, 58].

Таблиця 7.8

Практичні рекомендації щодо маскування програмного забезпечення

Напрям	Рекомендації
Вибір цілей	Маскувати насамперед критичні модулі (криптографія, ліцензії, механізми активації)
Комбінування методів	Поєднувати обфускацію, маніпулювання функціями, мертвий код та непрозорі предикати
Автоматизація	Інтегрувати засоби обфускації у CI/CD
Тестування	Виконувати автоматичні тести до і після трансформацій
Контроль стабільності	Оцінювати вплив на продуктивність і сумісність
Безпека	Зберігати mapping-файли та вихідні версії програм

Для різних рівнів програмного коду доцільно використовувати різні підходи маскування.

Для мов високого рівня (C/C++, Java, Python) ефективними є лексичні та структурні трансформації: перейменування, перестановка функцій, реструктурування класів і масивів.

Для машинного коду або байткоду більш ефективними є перетворення графа керування, клонування блоків, непрямі переходи та часткове дешифрування коду під час виконання.

Оптимальний рівень захисту досягається при комбінуванні декількох підходів, коли окремі методи компенсують слабкі сторони один одного.

Для автоматизації процесів маскування можуть застосовуватися:

для C/C++ – Tigress, Obfuscator-LLVM;

для Java – ProGuard, Zelix KlassMaster;

для .NET – ConfuserEx, Dotfuscator;

для скриптових мов – JS-Obfuscator, PyArmor.

Після застосування маскуючих трансформацій доцільно виконувати автоматичне тестування функціональності та оцінювання впливу на продуктивність системи [5, 20].

Надмірне маскування може призводити до збільшення розміру виконуваного файлу, ускладнення супроводу та погіршення сумісності програмного забезпечення. Тому ефективність таких методів визначається не лише складністю трансформацій, а й правильністю їх інтеграції у загальну систему захисту програмного забезпечення.

7.6. RUNTIME-МЕХАНІЗМИ ПРОТИДІЇ НАЛАГОДЖЕННЮ ТА АНАЛІЗУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Якщо методи обфускації та маскування, розглянуті у підрозділах 7.4-7.5, переважно спрямовані на ускладнення статичного аналізу програмного забезпечення, то runtime-механізми захисту орієнтовані на протидію дослідженню програми безпосередньо під час її виконання. Одним із найбільш поширених напрямів такого захисту є anti-debugging – виявлення та ускладнення роботи налагоджувачів [19, 57].

Налагоджувачі є важливими інструментами розроблення програмного забезпечення, однак ті самі механізми можуть використовуватися для аналізу внутрішньої логіки програм, обходу ліцензійних перевірок, дослідження криптографічних механізмів та модифікації поведінки програмного забезпечення [19, 26].

Під час запуску програми під керуванням налагоджувача зловмисник може:

– отримати доступ до змінних, ключів і конфіденційних даних у пам’яті;
– змінювати регістри, стек та параметри виконання;
– вимірювати часові характеристики виконання;
– встановлювати точки зупину (breakpoints);
– модифікувати потоки виконання;
– обходити механізми ліцензування або перевірки цілісності.

Подібні механізми часто інтегруються з контролем цілісності, обфускацією, динамічним дешифруванням коду та системами захисту від модифікації програмного забезпечення [33, 58, 59].

7.6.1. Основні механізми роботи visual studio debugger

Налагоджувач (debugger) – це програмний засіб, що забезпечує контроль виконання програми, аналіз її стану та покрокове дослідження програмного коду [19, 50, 51].

Одним із найбільш поширених середовищ налагодження є Visual Studio Debugger (рис. 7.23).

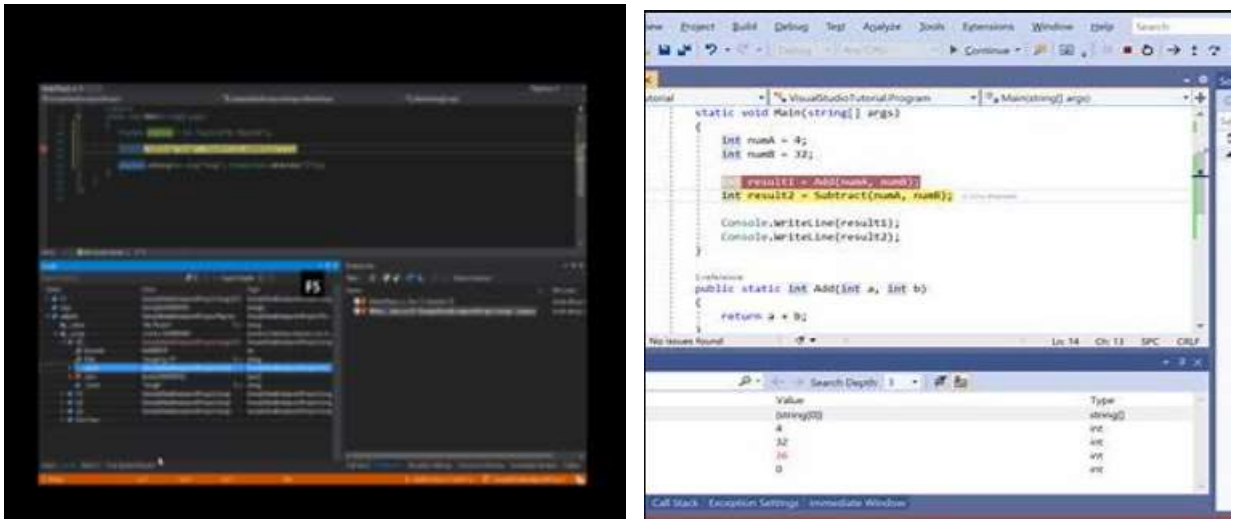


Рисунок 7.23. Інтерфейс налагоджувача Visual Studio

Налагоджувачі дозволяють:

- запускати програму у режимі налагодження (debug mode);
- виконувати покрокове трасування;
- переглядати регістри, стек та області пам’яті;
- змінювати значення змінних під час виконання;
- встановлювати програмні та апаратні точки зупину;
- аналізувати стек викликів і винятки.

З точки зору інформаційної безпеки такі можливості можуть використовуватися для аналізу захисних механізмів, вилучення критичних даних або модифікації логіки виконання програми [19].

– Сучасні налагоджувачі використовують низку системних механізмів операційної системи [26, 51]:

– WinAPI-функції DebugActiveProcess(), WaitForDebugEvent(), ContinueDebugEvent();

– прапорець Trap Flag для покрокового виконання;
– інструкцію INT 3 для створення програмних точок зупину;
– структуру PEB (Process Environment Block);
– debug-регістри DR0–DR7 для апаратних breakpoints.

Незважаючи на різні інтерфейси, більшість сучасних налагоджувачів (Visual Studio, x64dbg, WinDbg, OllyDbg) використовують однакові низькорівневі механізми взаємодії з операційною системою.

На рис. 7.23 наведено загальну схему взаємодії налагоджувача з програмою [26, 57].

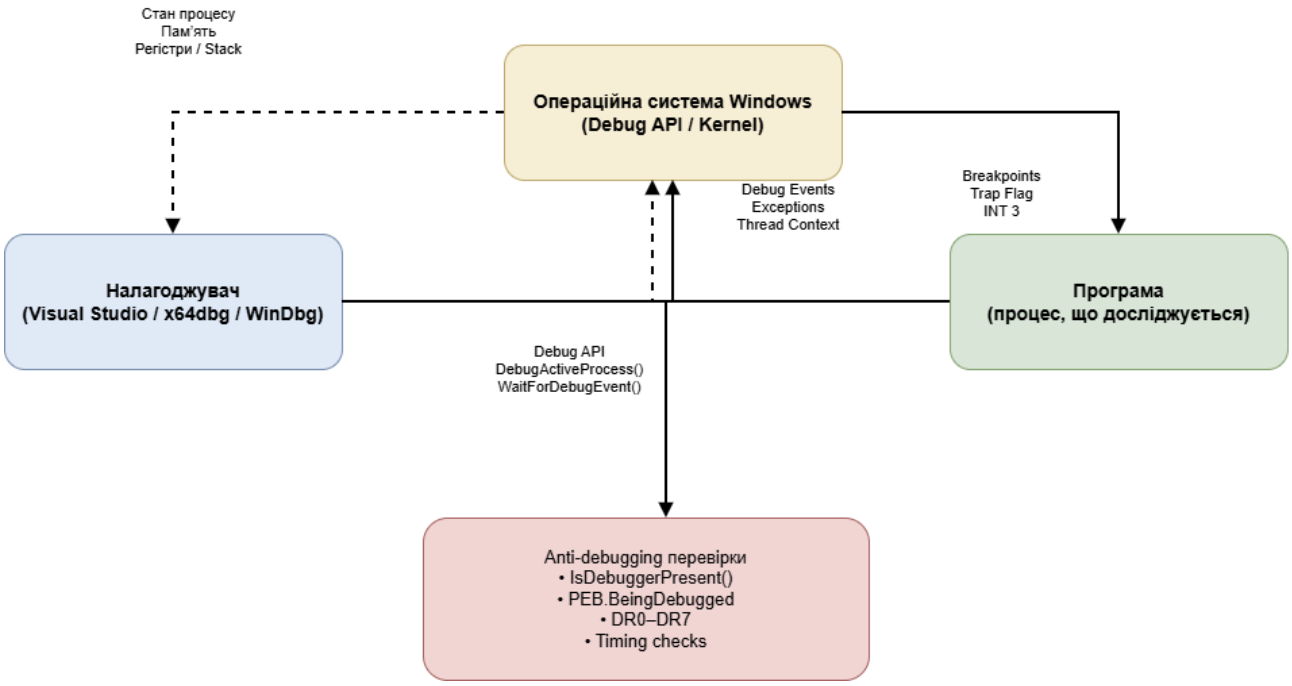


Рисунок 7.23. Загальна схема взаємодії налагоджувача з програмою

На рис. 7.23 наведено загальну схему взаємодії налагоджувача з програмою під час runtime-аналізу. Налагоджувач взаємодіє з операційною системою через механізми Debug API, використовуючи точки зупину (`INT 3`), прапорець `Trap Flag` та інші системні засоби керування виконанням процесу. Операційна система передає налагоджувачу інформацію про стан процесу, регістри, стек, винятки та події виконання. У відповідь програма може виконувати anti-debugging-перевірки, зокрема аналіз структури `PEB`, перевірку debug-регістрів `DR0-DR7` та таймінгові перевірки. Подібна взаємодія лежить в

основі сучасних механізмів runtime-захисту та протидії несанкціонованому налагодженню програмного забезпечення.

7.6.2. Методи виявлення налагоджувача

Для виявлення факту запуску програми під контролем налагоджувача застосовуються anti-debugging-механізми, що аналізують стан процесу, службові структури ОС та часові характеристики виконання. Основні методи виявлення наведено у табл. 7.9 [19,26,57].

Таблиця 7.9

Методи виявлення налагоджувача

Метод	Принцип роботи
IsDebuggerPresent()	Перевірка ознаки налагодження процесу
CheckRemoteDebuggerPresent()	Перевірка зовнішнього налагоджувача
Аналіз PEB	Перевірка прапорця BeingDebugged
Таймінгові перевірки	Аналіз затримок виконання
Перевірка DR0–DR7	Виявлення апаратних breakpoints
NtQueryInformationProcess()	Аналіз параметрів процесу

Найпростішим способом є використання WinAPI-функції IsDebuggerPresent() [26,51]:

```
if (IsDebuggerPresent())
    ExitProcess(0);
```

Для виявлення зовнішнього налагоджувача може використовуватися:

```
BOOL dbg;

CheckRemoteDebuggerPresent(
    GetCurrentProcess(),
    &dbg
);

if (dbg)
    ExitProcess(0);
```

Наведений фрагмент демонструє один із поширених методів anti-debugging, що базується на використанні функції `CheckRemoteDebuggerPresent()`. Дана функція WinAPI дозволяє перевірити, чи виконується поточний процес під контролем зовнішнього налагоджувача.

У наведеному прикладі:

– GetCurrentProcess() повертає дескриптор поточного процесу;

– змінна dbg використовується для збереження результату перевірки;

– CheckRemoteDebuggerPresent() записує у dbg значення TRUE, якщо процес налагоджується;

– у випадку виявлення налагоджувача викликається ExitProcess(0), що призводить до завершення роботи програми.

Подібні перевірки застосовуються для виявлення запуску програми під керуванням Visual Studio Debugger, x64dbg, OllyDbg та інших засобів аналізу. У сучасних системах захисту цей механізм зазвичай комбінується з іншими anti-debugging-перевірками, оскільки окреме використання `CheckRemoteDebuggerPresent()` може бути обійдене шляхом модифікації API-викликів або емуляції середовища виконання.

Більш надійним підходом є перевірка структури PEB, яка містить службову ознаку BeingDebugged [26,19].

Приклад перевірки PEB (прапорець BeingDebugged):

```
bool checkPEB()
{
    return *(BYTE *)
        (__readfsdword(0x30) + 2);
}
```

Для виявлення прихованого налагодження можуть застосовуватися таймінгові перевірки [57].

Приклад таймінгового anti-debugging:

```
DWORD t1 = GetTickCount();

Sleep(10);

DWORD t2 = GetTickCount();

if ((t2 - t1) > 50)
```

```
{
    printf("Debugger detected!");
}
```

Наведений фрагмент демонструє приклад таймінгового anti-debugging-механізму, який базується на аналізі часових затримок під час виконання програми. Основна ідея методу полягає у виявленні неприродного уповільнення виконання, що може виникати при покроковому налагодженні або встановленні точок зупину.

У наведеному прикладі:

– GetTickCount()використовується для отримання системного часу до та після виконання фрагмента коду;

– Sleep(10) створює контрольовану затримку тривалістю приблизно 10 мс;

– різниця (t2 - t1) визначає фактичний час виконання ділянки програми.

У нормальних умовах різниця часу повинна бути близькою до очікуваної затримки. Якщо ж програма виконується під керуванням налагоджувача, покрокове виконання або обробка breakpoints можуть суттєво збільшити часовий інтервал. У прикладі перевищення порогу 50 мс інтерпретується як ознака присутності налагоджувача, після чого виводиться повідомлення «Debugger detected!».

Подібні таймінгові перевірки широко використовуються у системах anti-debugging та anti-analysis, оскільки дозволяють виявляти не лише класичні налагоджувачі, а й деякі sandbox- та емуляційні середовища.

При покроковому виконанні програми або встановленні точок зупину часові характеристики змінюються, що дозволяє виявляти присутність налагоджувача.

Додатково можуть застосовуватися [19, 26, 57]:

– перевірка debug-регістрів DR0–DR7;

– аналіз виняткових ситуацій (exception-based anti-debugging);

– перевірка цілісності коду;

– контроль модифікації таблиць імпорту;

– аналіз поведінки потоків виконання.

На рис. 7.24 наведено класифікацію основних anti-debugging-перевірок.

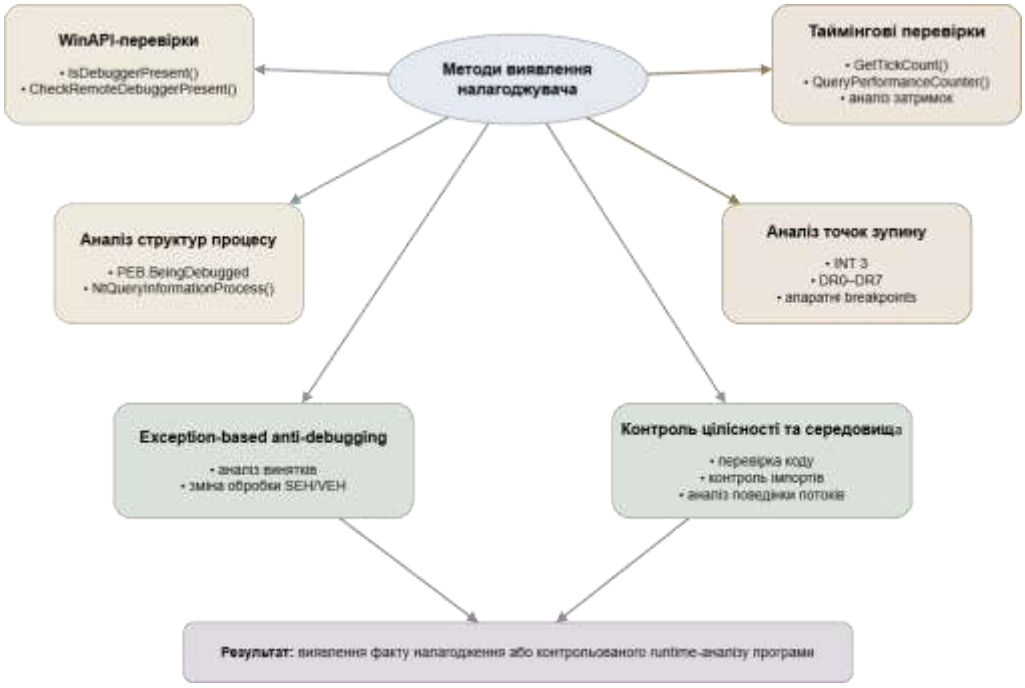


Рисунок 7.24. Класифікація методів виявлення налагоджувача

На рис. 7.24 виділено декілька груп перевірок: WinAPI-методи (IsDebuggerPresent(), CheckRemoteDebuggerPresent()), аналіз службових структур процесу (PEB, NtQueryInformationProcess()), перевірка апаратних і програмних точок зупину (INT 3, DR0–DR7), таймінгові перевірки, а також методи на основі аналізу винятків (exception-based anti-debugging). Окрему групу становлять механізми контролю цілісності та аналізу середовища виконання. У сукупності подібні перевірки дозволяють виявляти факт налагодження або контрольованого runtime-аналізу програми та є основою сучасних механізмів захисту програмного забезпечення від несанкціонованого дослідження.

Комбінування декількох anti-debugging-механізмів суттєво ускладнює приховане налагодження та автоматизований аналіз програмного забезпечення.

7.6.3. Реакції програм на виявлення налагоджувача

Після виявлення налагоджувача програма може ініціювати захисні реакції, спрямовані на ускладнення подальшого аналізу або модифікації програмного коду [57, 59].

Основні реакції наведено на рис. 7.25.

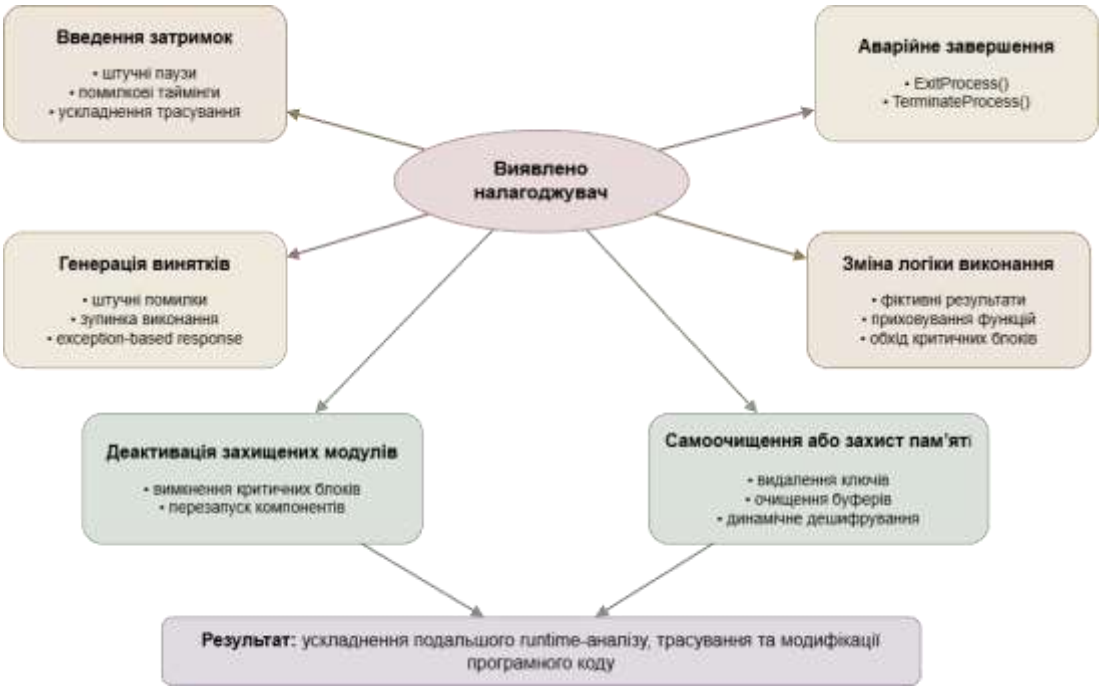


Рисунок 7.25. Основні реакції програм на виявлення налагоджувача

На рис. 7.25 наведено основні реакції програмного забезпечення на виявлення налагоджувача під час runtime-аналізу. Після фіксації факту налагодження програма може виконувати аварійне завершення процесу, генерувати виняткові ситуації, змінювати логіку виконання або вводити штучні часові затримки. Окрему групу становлять механізми деактивації захищених модулів та самоочищення пам’яті, які дозволяють приховувати критичні дані або ускладнювати подальший аналіз програми. У складних системах захисту подібні реакції часто комбінуються з обфускацією, контролем цілісності та динамічним дешифруванням коду. У результаті суттєво підвищується складність runtime-аналізу, трасування та модифікації програмного забезпечення.

Приклад простої реакції на виявлення налагоджувача:

```
#include <windows.h>

int main()
{
    if (IsDebuggerPresent())
    {
        MessageBox(
            NULL,
            "Debugging detected!",
            "Warning",
            MB_OK
        );

        ExitProcess(0);
    }

    return 0;
}
```

Наведений фрагмент демонструє приклад простої anti-debugging-реакції програми на виявлення налагоджувача. У даному випадку використовується WinAPI-функція IsDebuggerPresent(), яка дозволяє визначити, чи виконується поточний процес під контролем налагоджувача.

У прикладі:

– підключення заголовкового файла windows.h забезпечує доступ до функцій WinAPI;

– IsDebuggerPresent() повертає ненульове значення (TRUE), якщо процес виконується у режимі налагодження;

– при виявленні налагоджувача викликається MessageBox(), що виводить попереджувальне повідомлення Debugging detected!;

– після цього функція ExitProcess(0) примусово завершує роботу програми.

Подібний підхід використовується для ускладнення runtime-аналізу та перешкоджання дослідженню програми у середовищах Visual Studio Debugger, x64dbg, OllyDbg та інших налагоджувачах. У більш складних системах захисту замість простого завершення процесу можуть застосовуватися приховані реакції: зміна логіки виконання, генерація помилкових результатів, самоочищення пам’яті або динамічне дешифрування коду.

У більш складних системах захисту програма може не завершувати роботу одразу, а навмисно змінювати результати обчислень або приховувати реальну логіку виконання, що суттєво ускладнює аналіз.

У деяких реалізаціях anti-debugging комбінується з обфускацією графа керування, контролем цілісності, шифруванням окремих фрагментів коду, anti-tampering-механізмами, методами динамічного формуванням інструкцій.

Таким чином, anti-debugging є важливим елементом багаторівневого runtime-захисту програмного забезпечення.

7.6.4. Обмеження та практичні рекомендації щодо anti-debugging

Незважаючи на ефективність anti-debugging-механізмів, їх застосування має ряд практичних обмежень [57,59].

Основні обмеження наведено у табл. 7.10.

Таблиця 7.10

Обмеження anti-debugging-механізмів

Обмеження	Характеристика
Хибні спрацювання	Можливі у sandbox, CI/CD або IDE
Залежність від ОС	Частина механізмів залежить від реалізації Windows
Можливість обходу	Досвідчений аналітик може модифікувати API
Ускладнення підтримки	Захист може заважати легітимному налагодженню
Вплив на продуктивність	Додаткові перевірки уповільнюють виконання

Для підвищення ефективності anti-debugging рекомендується:

– комбінувати декілька незалежних перевірок;
– використовувати anti-debugging у критичних ділянках коду;
– поєднувати anti-debugging із обфускацією та контролем цілісності;
– виконувати тестування у Visual Studio Debugger, x64dbg та WinDbg;
– відключати anti-debugging у dev-збірках;
– здійснювати логування спроб налагодження.

На рис. 7.26 наведено узагальнену схему інтеграції anti-debugging у систему захисту програмного забезпечення.

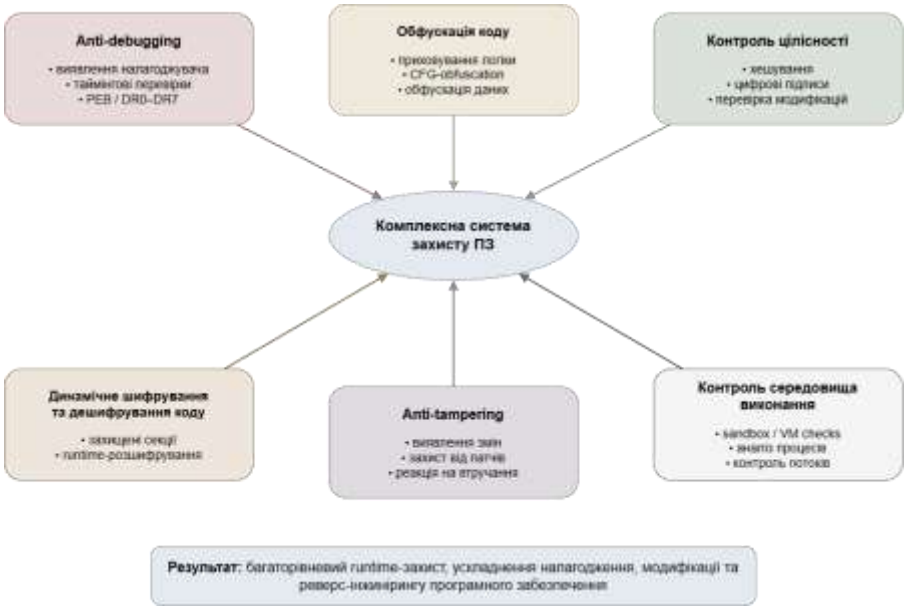


Рисунок 7.26. Інтеграція anti-debugging у комплексну систему захисту програмного забезпечення

У центрі схеми розташована багаторівнева система захисту ПЗ, яка об’єднує декілька взаємопов’язаних механізмів протидії аналізу та модифікації програмного коду.

Одним із ключових компонентів є anti-debugging, що забезпечує виявлення налагоджувачів, таймінговий аналіз, перевірку службових структур процесу (PEB) та debug-регістрів (DR0–DR7). Разом із ним застосовуються механізми обфускації програмного коду, які приховують структуру графа керування, логіку виконання та представлення даних.

Важливу роль також відіграє контроль цілісності, що базується на використанні хешування, цифрових підписів і перевірки модифікацій виконуваного файлу. Для захисту критичних фрагментів програми можуть застосовуватися механізми динамічного шифрування та дешифрування коду під час виконання. Додатково система може включати anti-tampering-механізми, спрямовані на виявлення спроб модифікації або патчингу програми, а також контроль середовища виконання, що дозволяє виявляти sandbox-, virtual machine- та емуляційні середовища.

Комплексне поєднання зазначених механізмів забезпечує багаторівневий runtime-захист програмного забезпечення та суттєво ускладнює налагодження, модифікацію й реверс-інжиніринг програмних систем. Надмірне використання anti-debugging-механізмів може призводити до ускладнення супроводу програмного забезпечення, проблем сумісності та погіршення продуктивності системи.

Таким чином, runtime-механізми протидії налагодженню є важливим компонентом сучасних систем захисту програмного забезпечення. Їх основне призначення полягає у виявленні та ускладненні процесу дослідження програми під час виконання. Найбільша ефективність досягається при комплексному поєднанні anti-debugging, обфускації, контролю цілісності та інших механізмів захисту програмного забезпечення.

ПИТАННЯ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ ЗА ТЕМОЮ 7

1. Яка різниця між автоматичними та інтерактивними дизасемблерами?
2. Опишіть ідею перемішування команд і поясніть, як вона впливає на процес дизасемблювання.
3. Як непряма адресація переходів ускладнює побудову графа керування програмою (CFG)?
4. Що таке «динамічний фуфель» і як він використовується для протидії аналізу?
5. У чому полягає суть емуляції процесора як методу захисту програмного забезпечення? Які недоліки цього підходу?
6. Дайте визначення обфускації програмного коду та назвіть її основні цілі.
7. Назвіть основні види обфускації програмного коду та коротко їх охарактеризуйте.
8. Що таке обфускація даних і які її основні підгрупи?
9. Які прийоми використовуються для обфускації графа потоку керування?

10. Що таке непрозорий предикат (opaque predicate)? Які його основні типи?

11. Які критерії використовують для оцінювання ефективності обфускації?

12. Які ризики пов’язані з надмірною обфускацією програмного коду?

13. Як поєднують обфускацію з іншими механізмами захисту програмного забезпечення?

14. Дайте визначення маскування програми та поясніть його основну ідею.

15. Назвіть і коротко опишіть етапи процесу маскування програмного забезпечення.

16. Які трансформації належать до маніпулювання функціями? Наведіть приклади.

17. У чому полягає різниця між недосяжним, мертвим і надлишковим кодом?

18. Опишіть принцип зчеплення дуг (arc hooking) та призначення «повертаючого предиката».

19. Які негативні наслідки може спричиняти надмірне маскування програмного забезпечення?

20. Наведіть приклад комбінування трансформацій, що підвищують стійкість програми до статичного аналізу.

21. Які основні механізми використовують сучасні налагоджувачі для керування виконанням програм?

22. Які методи використовуються для виявлення налагоджувача під час виконання програми?

23. Поясніть принцип роботи функцій IsDebuggerPresent() та CheckRemoteDebuggerPresent().

24. У чому полягає суть таймінгового anti-debugging?

25. Які реакції програми можуть застосовуватися після виявлення налагоджувача?

26. Які обмеження та недоліки мають anti-debugging-механізми?

27. Чому anti-debugging доцільно поєднувати з обфускацією та контролем цілісності?

ПРАКТИЧНІ ЗАВДАННЯ ДО ТЕМИ 7

ЗАВДАННЯ 1. ЗАСТОСУВАННЯ МЕТОДІВ ОБФУСКАЦІЇ ДО ПРОГРАМНОГО КОДУ

Постановка завдання

На основі підрозділів 7.4.2 – 7.4.5 посібника застосуйте методи лексичної (символьної) обфускації, обфускації даних та елементів керування до фрагменту програмного коду компонента системи ЦЗ.

Оберіть один із наведених програмних фрагментів без наявного захисту:

а) функція мовою С, що виконує перевірку пароля або ліцензійного ключа для модуля реєстрації ЦЗ-системи;

б) Python-скрипт формування та відправлення звіту про надзвичайну ситуацію (містить рядкові константи, масиви даних та цикл обробки);

в) Java-клас обробки оперативних даних про ресурси підрозділів ЦЗ (містить методи доступу до БД та логіку перевірки прав).

Для обраного фрагменту:

1) Застосуйте лексичну (символьну) обфускацію (підрозділ 7.4.2): перейменуйте щонайменше п'ять ідентифікаторів (функцій, змінних, констант) за одним із методів першого покоління (беззмістовні рядки, нумерація, нечитабельні символи); наведіть фрагменти коду «до» та «після» обфускації; вкажіть, який відсоток слів програми зазнав змін (орієнтир: 15–55%).

2) Застосуйте обфускацію даних (підрозділ 7.4.2): виберіть одну числову змінну і застосуйте до неї лінійне кодування виду $v = c1 \cdot v_new + c2$ (де c1 і c2 – константи); наведіть математичну формулу заміни, скоригований фрагмент коду та перевірочний розрахунок; або реструктуруйте одновимірний масив у двовимірний, показавши перетворення індексації.

3) Застосуйте один із методів обфускації керування (підрозділ 7.4.3):

- вставку фіктивних переходів;
- непрозорі предикати (opaque predicates);
- перемішування базових блоків;

– перетворення циклів;

– вставку мертвого коду.

Наведіть фрагменти коду до та після перетворення й коротко поясніть, як виконана трансформація ускладнює побудову графа керування програми (CFG).

4) На основі підрозділу 7.4.7 заповніть порівняльну таблицю не менш ніж чотирьох інструментів-обфускаторів, придатних для обраної платформи (мови): вкажіть назву, рівень обфускації (вихідний код / проміжний байткод / бінарний), типи підтримуваних трансформацій, наявність відкритого коду та придатність для CI/CD.

5) Оцініть ефективність виконаної обфускації за чотирма критеріями з підрозділу 7.4.6 (стійкість, продуктивність, надійність, вартість деобфускації); вкажіть також щонайменше два обмеження або ризики виконаних перетворень.

6) Для мов C/C++ або C# додайте просту anti-debugging-перевірку з підрозділу 7.6 (наприклад, IsDebuggerPresent() або таймінгову перевірку) та поясніть її призначення у комплексній системі захисту програмного забезпечення.

Приклад розв'язку

Обраний варіант: а) функція мовою C – перевірка пароля для модуля реєстрації ЦЗ-системи.

Крок 1. Лексична обфускація

Вихідний фрагмент коду:

```
int check_access(char *password) {
int attempts   = 0;
char hash_buf[32];
compute_hash(password, hash_buf);
if (strcmp(hash_buf, EXPECTED_HASH) == 0) {
return ACCESS_GRANTED;
}
attempts++;
return ACCESS_DENIED;
}
```

Застосований метод: перейменування ідентифікаторів у беззмістовні шістнадцяткові рядки (підрозділ 7.4.2).

Таблиця 7.11

Таблиця перейменувань:

Оригінальний ідентифікатор	Обфускований ідентифікатор	Тип
check_access	_0xA3F1	Функція
password	_0xB7D2 _	Параметр
attempts	_0xC9E4	Локальна змінна
hash_buf	_0xD1F5	Локальний масив
compute_hash	_0xE3A6	Функція
EXPECTED_HASH	_0xF5B7	Константа
ACCESS_GRANTED	_0xA7C8	Константа
ACCESS_DENIED	_0xB9D9	Константа

Обфускований фрагмент:

```
int _0xA3F1(char *_0xB7D2) {
int _0xC9E4   = 0;
char _0xD1F5[32];
_0xE3A6(_0xB7D2, _0xD1F5);
if (strcmp(_0xD1F5, _0xF5B7) == 0) {
return _0xA7C8;
}
_0xC9E4++;
return _0xB9D9;
}
```

Підрахунок: 8 ідентифікаторів зі орієнтовно 17 слів у фрагменті замінено → ~47% слів зазнали змін. Це відповідає діапазону 15–55%.

Обмеження: сучасні декомпілятори (IDA Hex-Rays, Ghidra) автоматично підставляють імена типу «method_1», що частково нівелює ефект символьної обфускації. Тому лексична обфускація є ефективною лише як один із шарів захисту.

Крок 2. Обфускація даних – кодування змінної

Обрана змінна: int attempts = 0 (лічильник помилкових спроб; порівнюється з порогом MAX_ATTEMPTS = 5).

Застосована трансформація: лінійне кодування (підрозділ 7.4.2).

Формула: v = c1 · v_new + c2, де c1 = 6, c2 = 3.

Отже, v_new = (v − 3) / 6.

Таблиця 7.12

Відображення значень:

v (оригінал)	v_new (кодований)	Перевірка: 6 × v_new + 3
0	0	6 × 0 + 3 = 3 → але оригінал = 0;

← код встановлює v_new = 0, а умова MAX адаптується

Скоригований фрагмент (спрощена форма, де intermed = 6 × attempts_enc + 3):

```
/* До обфускації */
int attempts = 0;
while (attempts < 5) { ... attempts++; }

/* Після обфускації (c1=6, c2=3): */
int _0xC9E4 = 3;          /* кодує attempts=0: v = 6*0 + 3 = 3 */
while (_0xC9E4 < 33) {    /* 6*5+3 = 33 – адаптований поріг */
...
_0xC9E4 += 6;         /* інкремент на c1 відповідає attempts++ */
}
```

Перевірочний розрахунок:

– attempts=0 → _0xC9E4 = 6×0+3 = 3 ✓

– attempts=1 → _0xC9E4 = 6×1+3 = 9 ✓

– attempts=5 → _0xC9E4 = 6×5+3 = 33 → вихід із циклу (33 < 33 = false) ✓

Функціональна еквівалентність збережено. Аналітик, не знаючи c1=6, c2=3, отримує незрозумілу числову прогресію 3, 9, 15, 21, 27, 33.

Крок 3. Порівняльна таблиця обфускаторів (для C/C++)

Таблиця 7.13

Порівняльна таблиця обфускаторів:

Інструмент	Рівень	Типи трансформацій	Відкритий	CI/CD
Tigress	Вихідний код C	Лексична, CFG, обфускація даних, віртуалізація	Так	Так
Obfuscator-LLVM	Проміжний (IR)	CFG (flattening, bogus CF), пакування рядків	Так	Так
Stunnix	Вихідний код C/ C++/Perl	Лексична, перестановка структур, dead code	Ні	Так
Picheta	Вихідний код C/ C++	Лексична, перестановка функцій	Так (онлайн)	Ні
Obfuscator. app	Бінарний (LLVM)	CFG, шифрування рядків, anti-debug вставки	Частково	Так

Крок 4. Оцінка ефективності

Таблиця 7.14

Критерій обфускації:

Критерій	Оцінка для виконаних трансформацій
Стійкість	Середня: символьна обфускація легко оборотна декомпіляторами; кодування змінної підвищує складність, але визначається аналізом потоку даних (DFA) через ~1-2 год роботи аналітика
Продуктивність	Висока: лексичні зміни не впливають на швидкодію; лінійне кодування змінної додає 2-3 арифметичні операції на ітерацію (вплив < 1% для функцій без tight loops)
Надійність	Висока: перевірочний розрахунок підтвердив еквівалентність для всіх значень attempts ∈ [0; 5]; ризик помилки – нульовий при правильному виборі c1, c2 (c1 > 0, цілочисельні)
Вартість деобфускаці	Низька окремо, але комбінація з CFG і anti-debug підвищить її до прийнятного рівня; лише символьна обфускація оборотна за 10–30 хв автоматичними засобами

Обмеження та ризики:

1. Символьна обфускація в .NET/Java-збірках не захищає рефлексивні компоненти – виклики через Reflection.Invoke() залишають повні імена у метаданих Assembly.

2. Надмірне кодування змінних з великими c1 може спричинити переповнення цілочисельного типу при великих значеннях – необхідно обирати c1, c2 з урахуванням діапазону змінної та типу даних.

ЗАВДАННЯ 2. ОБФУСКАЦІЯ ГРАФА ПОТОКУ КЕРУВАННЯ ТА МАСКУЮЧІ ПЕРЕТВОРЕННЯ

Постановка завдання

На основі підрозділів 7.4.3, 7.5.1, 7.5.2 та 7.4.6 посібника застосуйте методи обфускації графа потоку керування (CFG) та маскуючі трансформації до програмного фрагменту компонента ЦЗ.

Оберіть один із наведених фрагментів коду:

а) цикл обробки масиву ресурсів (наприклад, підрахунок доступних одиниць техніки у підрозділі ЦЗ) з простим лічильником від 0 до N;

б) дві окремі функції, що виконуються послідовно (наприклад, валідація вхідних даних звіту та їх запис у журнал);

в) умовна гілка перевірки рівня доступу з поверненням константних значень (granted/denied).

Для обраного фрагменту:

1) Застосуйте одне з перетворень потоку керування (підрозділ 7.4.3): розгортання циклу (unrolling), пониження розмірності індексного простору або переплетення функцій (interleaving); перемішування базових блоків. Наведіть код «до» та «після»; поясніть, яким чином перетворення ускладнює аналіз CFG.

2) Вставте непрозорий предикат (підрозділ 7.4.3): оберіть тип (PF або PT); наведіть математичне обгрунтування його сталого значення; покажіть фрагмент коду з вбудованим предикатом; продемонструйте використання недосяжного або мертвого коду.

3) Опишіть і вставте фрагмент мертвого (dead) або надлишкового (redundant) коду (підрозділ 7.5.2): поясніть, чому вибраний блок не впливає на функціональний результат програми; вкажіть тип (мертвий / надлишковий / недосяжний) та спосіб генерації (алгебраїчна тотожність, фіктивна змінна недосяжна умова тощо).

4) Оцініть сукупну ефективність виконаних маскуючих перетворень за критеріями з підрозділу 7.4.6: визначте вплив на статичний та динамічний аналіз; вкажіть, які засоби аналізу (статичні / динамічні) будуть ускладнені або введені в оману і які залишаться ефективними; запропонуйте один додатковий механізм захисту для підвищення стійкості програмного забезпечення.

5) Додайте простий runtime-механізм захисту з підрозділу 7.6 (наприклад, anti-debugging-перевірку або таймінговий контроль) та поясніть його взаємодію з методами обфускації й маскування.

Приклад розв'язку

Обраний варіант: б) дві функції – validate_report() і log_event() – виконуються послідовно при обробці звіту про надзвичайну ситуацію в ЦЗ-системі.

Крок 1. Переплетення функцій (interleaving)

Вихідний код (дві окремі функції):

```
void validate_report(int *status) {
    if (*status < 0 || *status > 10) {
        *status = -1;    /* некоректний статус */
    }
}

void log_event(int *counter) {
    *counter *= 2;       /* подвоєння лічильника подій */
}

int main() {
    int s = 5, c = 3;
    validate_report(&s);
    log_event(&c);
    return 0;
}
```

Трансформація: переплетення (interleaving) – об'єднання обох функцій в одну з параметром state (підрозділ 7.4.3, interleaving як метод обфускації керування).

```
Код після interleaving:
void _0xD3F7(int *_0xA1, int *_0xB2) {
    int _state = 0;
    while (1) {
        switch (_state) {
            case 0:
                /* логіка validate_report */
                if (*_0xA1 < 0 || *_0xA1 > 10) {
                    *_0xA1 = -1;
                }
                _state = 1;
                break;
            case 1:
                /* логіка log_event */
                *_0xB2 *= 2;
                _state = 2;
                break;
            default:
                return;
        }
```

```
    }
}

int main() {
    int _0xC3 = 5, _0xD4 = 3;
    _0xD3F7(&_0xC3, &_0xD4);
    return 0;
}
```

Вплив на CFG: після перетворення граф потоку керування має нескінченний цикл (while(1)) із оператором switch і трьома вузлами (case 0, 1, default). Статичний аналізатор (IDA Pro, Ghidra) побачить лише єдину функцію _0xD3F7 з нерозрізнюваними «шляхами» обробки двох незалежних логік. Аналітикові необхідно відновити семантику змінної _state, щоб зрозуміти, яка логіка виконується в кожному case – це значно підвищує трудовитрати реверс-інжинірингу.

Крок 2. Непрозорий предикат (opaque predicate)

Тип предиката: PT (завжди TRUE).

Математичне обґрунтування: для будь-якого цілого x вираз $(x^2 + x)$ завжди парний, тому $(x^2 + x) \% 2 == 0$ є завжди істинним (підрозділ 7.4.3, interleaving як метод обфускації керування).

Доведення:

– якщо x парне: x^2 парне, x парне → сума парна → ділення на 2 без остатку;

– якщо x непарне: x^2 непарне, x непарне → сума (непарне + непарне) = парне → ділення без остатку.

Отже, $\forall x \in \mathbb{Z}$: $(x^2 + x) \% 2 == 0$.

Вбудовування PT-предиката у функцію validate_report:

```
void _0xD3F7(int *_0xA1, int *_0xB2) {
    int _state = 0;
    int _x = 7;    /* довільне ціле, значення неважливе */

    while (1) {
        switch (_state) {
            case 0:
                /* непрозорий предикат PT: завжди TRUE */
                if ((_x * _x + _x) % 2 == 0) {
```

```
                    /* реальна логіка: validate_report */
                    if (*_0xA1 < 0 || *_0xA1 > 10) {
                        *_0xA1 = -1;
                    }
                } else {
                    /* НЕДОСЯЖНИЙ КОД – аналітик витрачає час на аналіз */
                    *_0xA1 = 0xFF;
                    return;
                }
                _state = 1;
                break;
            case 1:
                *_0xB2 *= 2;
                _state = 2;
                break;
            default:
                return;
        }
    }
}
```

Пояснення: гілка else після непрозорого предиката ніколи не виконується (недосяжний код). Але автоматичний статичний аналізатор не може тривіально довести константність виразу (_x * _x + _x) % 2 – він бачить залежність від змінної _x і витрачає ресурси на аналіз.

Крок 3. Мертвий код (dead code)

Тип вставки: надлишковий код (redundant code) – виконується, але не впливає на результат.

Метод: алгебраїчна тотожність: додавання та віднімання однієї константи (підрозділ 7.5.2 – вставка мертвого та надлишкового коду).

```
/* Надлишковий код: результат обчислення не використовується */
int _dummy = (_x * 3 - _x * 3) + 0;  /* завжди 0, не впливає ні на що */
_x = _x + 17 - 17;                   /* тотожне перетворення: x = x */
```

Розміщення: між case 0 і case 1 у switch-блоці. Аналітик бачить обчислення з _dummy і _x та витрачає час на визначення їх впливу на стан програми.

Додатково – вставка недосяжного коду (unreachable):

```
if (0) {
```

```
/* блок, що ніколи не виконується – заповнений «шумовими» операціями */
int _noise = *_0xA1 ^ 0xDEADBEEF;
*_0xA1 = _noise;
return;
}
```

Хоча умова if(0) очевидно хибна, деякі автоматичні аналізатори звертають увагу на цей блок як потенційно модифікувальний *_0xA1, підвищуючи «шум» у звіті аналізу.

Крок 4. Сукупна оцінка ефективності

Таблиця 7.15

Застосований комплекс перетворень:

Перетворення	Тип	Вплив на статичний аналіз	Вплив на динамічний аналіз
Interleaving функцій	CFG-ускладнення	Руйнує граф підграфів, ускладнює декомпіляцію двох логік	OllyDbg/x64dbg бачать єдину функцію; state-трасування ускладнено
Непрозорий предикат PT	Недосяжний блок	Псевдогілка «else» вводить помилкові шляхи у CFG	Апаратна точка зупину не зупиняється на else
Надлишковий код	Algebraic identity	Шум у DFA (data flow analysis)	Виконується, але не змінює стан – мінімальний час виконання
Недосяжний блок	if(0)-гілка	Декомпілятор видає помилковий шлях *_0xA1	Не виконується; не фіксується у трасуванні

Додатковий рекомендований механізм захисту (підрозділи 7.5, 7.6): клонування базових блоків (block cloning) як елемент маскуючих перетворень графа керування. Це створить кілька еквівалентних шляхів виконання і унеможливить однозначну побудову CFG навіть інтерактивним дизасемблером (IDA Pro).

На практиці подібні маскуючі трансформації доцільно поєднувати з anti-debugging, контролем цілісності та динамічним дешифруванням коду, що

дозволяє суттєво підвищити стійкість програмного забезпечення до реверс-інжинірингу.

ЗАВДАННЯ 3. ПРОЄКТУВАННЯ АНТИДЕБАГ-ЗАХИСТУ ДЛЯ КОМПОНЕНТА СИСТЕМИ ЦИВІЛЬНОГО ЗАХИСТУ

Постановка завдання

На основі підрозділів 7.6.1 - 7.6.4, а також матеріалів підрозділів 7.4.3 та 7.5.2 посібника розробіть систему захисту від несанкціонованого налагодження для програмного компонента підрозділу цивільного захисту.

Оберіть один із захищуваних модулів:

а) 32-бітний Windows-модуль реєстрації та активації ліцензії ЦЗ-системи;

б) мережевий компонент, що надсилає зашифровані звіти на сервер координації ЦЗ;

в) криптографічний модуль зберігання та формування ключів доступу до оперативних даних.

Для обраного модуля:

1) Оберіть щонайменше три методи виявлення налагоджувача (підрозділ 7.6.2) та заповніть таблицю із зазначенням: назви методу, використовуваного API або структури Windows, обґрунтування вибору для обраного модуля та очікуваної надійності виявлення.

2) Наведіть псевдокод або фрагмент коду C/C++ для кожного з обраних методів; вкажіть умову виявлення; контрольований параметр (прапорець, часовий інтервал, значення структури тощо); призначення відповідної перевірки у системі runtime-захисту.

3) Спроектуйте реакцію програми на виявлений факт налагодження (підрозділ 7.6.3): складіть таблицю з чотирма різними видами реакцій (завершення, генерація винятку, зміна логіки виконання; введення затримок); для кожної реакції вкажіть: місце інтеграції у програмний код; використовувані

API або механізми; мету застосування з точки зору захисту програмного забезпечення.

4) Проаналізуйте обмеження застосованих anti-debugging-механізмів: можливі способи їх обходу; запропонуйте додатковий механізм захисту; обґрунтуйте доцільність комбінування декількох методів runtime-захисту.

5) Запропонуйте інтеграцію anti-debugging із механізмами обфускації або контролю цілісності (підрозділи 7.4-7.5) та поясніть, як це впливає на стійкість програмного забезпечення до реверс-інжинірингу.

Приклад розв'язку

Обраний модуль: а) 32-бітний Windows-модуль реєстрації ліцензії ЦЗ-системи «license_reg.exe», що зберігає ліцензійний ключ у зашифрованому вигляді та перевіряє активацію при запуску.

Крок 1. Вибір методів виявлення налагоджувача

Таблиця 7.16

Порівняльна таблиця методів виявлення:

Метод виявлення	API / Структура	Обґрунтування вибору	Надійність
WinAPI-виклик	IsDebuggerPresent()	Найшвидший і найпростіший метод;	Середня:
(IsDebuggerPresent)	(kernel32.dll)	для реєстраційного модуля важлива перевірка при старті до будь-яких криптографічних операцій	легко обійти через NOP- патч
Перевірка PEB	PEB.BeingDebugged	Прямий доступ до структури ОС без	Середня:
(структура Windows)	(FS:[0x30]+2 у x86)	виклику публічного API; ускладнює автоматичне патчування через hooking Dbgapi	обходиться через патч PEB-прапорець
Таймінгова перевірка	GetTickCount() або RDTSC (інструкція)	Налагоджувач суттєво збільшує час між інструкціями при покроковому виконанні; для криптоопераций – відхилення > 50 мс є аномалією	Висока: складно підробити затримку точно

Крок 2. Псевдокод / код с для кожного методу

Метод 1. IsDebuggerPresent():

```
#include <windows.h>

void check_debugger_api() {
if (IsDebuggerPresent()) {
/* Виявлено налагоджувач – реакція (крок 3) */
trigger_protection();
}
}
```

```
/* Умова виявлення: IsDebuggerPresent() повертає != 0,
якщо процес виконується під налагоджувачем (підрозділ
7.6.2) */
```

Метод 2. Перевірка PEB.BeingDebugged:

```
bool check_peb_flag() {
/* Читання прапорця BeingDebugged з PEB напряму (x86):
PEB знаходиться за адресою FS:[0x30];
поле BeingDebugged – зміщення +2 від початку PEB */
bool being_debugged = *(BYTE *)(__readfsdword(0x30) + 2);
return being_debugged;
}

void check_debugger_peb() {
if (check_peb_flag()) {
trigger_protection();
}
}
```

```
/* Умова виявлення: *(FS:[0x30]+2) != 0 —
прапорець встановлюється системою при підключенні
налагоджувача */
```

Метод 3. Таймінгова перевірка (GetTickCount):

```
void check_debugger_timing() {
DWORD t1 = GetTickCount();

/* Виконання критичної ділянки: наприклад, один раунд AES */
perform_crypto_round();

DWORD t2 = GetTickCount();
DWORD elapsed = t2 - t1;
```

```
/* Порогове значення: нормальне виконання < 5 мс,
при покроковому налагодженні – зазвичай > 100 мс */
if (elapsed > 50) {
trigger_protection();
}
}
```

```
/* Умова виявлення: (t2 - t1) > 50 мс свідчить про покрокове
виконання (підрозділ 7.6.2, методи виявлення
налагоджувача) */
```

Крок 3. Реакції на виявлення налагоджувача

Таблиця 7.17

Методи виявлення налагоджувача:

Тип реакції	Розміщення у коді	API / Механізм	Захисна мета
Негайне завершення процесу	Одразу після виявлення (усі три методи виявлення)	ExitProcess(0) або TerminateProcess(GetCurrentProcess(), 0)	Унеможливити будь-який подальший аналіз; мінімізує час доступу до ліцензійних даних у пам'яті
Підміна логіки (false positive)	Після перевірки PEB-прапорця перед операцією перевірки ліцензійного ключа	Повернення хибного результату validate_license() → return ACCESS_GRANTED; (реальна перевірка пропускається	Створення ілюзії успішної активації за допомогою штучного ключа без розкриття реального алгоритму перевірки ліцензії
Введення затримок (timing decoy)	Перед виведенням повідомлення або в циклі обробки введення	Sleep(rand() % 5000 + 3000); або цикл активнного очікування	Уповільнення трасування та збільшення тривалості сеансу покрокового аналізу
Знищення ключового матеріалу	Після виявлення налагоджувача за часовими характеристиками або після завершення використання ключа	Безпечне очищення буфера пам’яті, наприклад SecureZeroMemory(key_buffer, key_len)	Унеможливити вилучення ключа із дампу пам'яті, навіть якщо налагоджувач продовжить виконання

Функція trigger_protection() – центральний диспетчер реакцій:

```
void trigger_protection() {
/* Рівень 1: знищення ключового матеріалу */
memset(g_key_buffer, 0, sizeof(g_key_buffer));

/* Рівень 2: підміна логіки (фіктивна відповідь) */
g_license_valid = 1;
g_license_key_id = 0xDEADBEEF;  /* фіктивне значення */

/* Рівень 3: затримка */
Sleep(rand() % 4000 + 2000);

/* Рівень 4: завершення процесу */
ExitProcess(0);
}
```

Крок 4. Обмеження методів та компенсація

Таблиця 7.18

Обмеження anti-debugging механізмів та компенсуючі заходи:

Метод виявлення	Відомий спосіб обходу зловмисником	Компенсуючий захисний шар
IsDebuggerPresent	Патчування коду (NOP-ін'єкція на виклик або hooking ntdll.IsDebuggerPresent → return 0); налагоджувачі ScyllaHide автоматично патчують цю функцію	Перевірка PEB напряму без виклику WinAPI – ускладнює hooking; або верифікація кількох методів одночасно (AND-умова) |
Перевірка PEB	Патчування PEB.BeingDebugged = 0 через WriteProcessMemory() або через плагін для OllyDb (HideDebugger, ScyllaHide)	Перехресна перевірка: NtQueryInformation Process() з ProcessDebugPort (клас 7) та перевірка NtGlobalFlag у PEB+0x68
Таймінгова перевірка	Налаштування «детермінованого» часу у VM або патчування GetTickCount → return const; автоматичні sandbox'и підробляють час	Комбінування з RDTSC (CPU-цикли): навіть при підробці GetTickCount, RDTSC відображає реальні цикли процесора; різниця виявляється при порівнянні двох вимірювань

Загальний висновок: жоден з антидебаг-методів не забезпечує абсолютного захисту (7.6.3 – реакції на виявлення налагоджувача посібника). Жоден із anti-debugging-механізмів не забезпечує абсолютного захисту програмного забезпечення. Окремі методи можуть бути нейтралізовані шляхом

модифікації середовища виконання, емуляції системних функцій або приховування налагоджувача. Тому ефективна стратегія захисту license_reg.exe для ЦЗ-системи повинна базуватися на комплексному підході: поєднанні anti-debugging-механізмів (розділ 7.6), обфускації графа керування (підрозділ 7.4.3), контролю цілісності виконуваного модуля (наприклад, перевірка контрольної суми секції .text) та ізольованого зберігання критичного ключового матеріалу. Комплексне використання декількох взаємодоповнювальних механізмів суттєво підвищує стійкість програмного забезпечення до реверс-інжинірингу та несанкціонованого дослідження.

У сучасних системах захисту anti-debugging-механізми рідко використовуються ізольовано. Найбільша ефективність досягається при їх комбінуванні з обфускацією графа керування, контролем цілісності, маскуванням коду та динамічним дешифруванням критичних ділянок програми.

ЗАВДАННЯ 4. КЛАСИФІКАЦІЯ ТА АНАЛІЗ МЕХАНІЗМІВ ВПРОВАДЖЕННЯ ЗАХИСНОГО КОДУ У PE-ФАЙЛ

Постановка завдання

На основі підрозділів 5.5 та 5.6 посібника дослідить механізми впровадження захисного коду (X-коду) у виконувані файли.

Розгляньте чотири нижченаведені сценарії впровадження захисного X-коду в гіпотетичний модуль auth.exe – компонент автентифікації програмної системи ЦЗ:

а) X-код розміщується у вільному просторі між кінцем PE-заголовку та початком першої секції (~256 байтів);

б) X-код дописується в кінець секції .text, збільшуючи її фізичний розмір понад SectionAlignment, що вимагає розширення секції та коригування адресацій;

в) X-код виноситься в нову секцію .prot, яку додають у таблицю секцій, і точка входу переноситься туди;

г) Захисна DLL реєструється у ключі реєстру HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\AppCertDLLs і завантажується у адресний простір автоматично при кожному запуску auth.exe.

Розгляд сценаріїв здійснюється виключно з навчальною метою для дослідження механізмів модифікації програмних реалізацій та методів їх виявлення засобами статичного аналізу.

Для кожного сценарію:

1) Визначте категорію механізму впровадження за класифікацією підрозділу 5.6 (A, B, C або Z) та коротко поясніть, які структури PE-файлу змінюються і чи змінюється довжина файлу.

2) Складіть порівняльну таблицю чотирьох сценаріїв за характеристиками: категорія, зміна довжини файлу, зміна адресацій, можливість виявлення засобами статичного аналізу, ризик порушення працездатності.

3) Перевірте кожен сценарій на відповідність вимогам до X-коду з підрозділу 5.5.2 (переміщуваність, компактність, відсутність затримок, незалежність від середовища).

4) Оберіть найбільш доцільний сценарій для захисту модуля автентифікації системи ЦЗ та обґрунтуйте вибір з урахуванням вимог надійності та прихованості захисту.

Приклад розв'язку

Аналіз сценаріїв:

а) Вільний простір у PE-заголовку – категорія A.

PE-заголовок займає ~0x300 байтів, перша секція починається після вирівнювання на FileAlignment (0x200). Між ними завжди є ~0x100 вільних байтів (нулів). X-код розміщується тут без зміни будь-яких адрес чи розмірів. Єдина модифікація: встановлення нової точки входу (AddressOfEntryPoint) на адресу X-коду.

б) Розширення секції .text понад FileAlignment – категорія B.

Хвіст секції .text розширюється завдяки фізичному образу. Довжина файлу збільшується. Фізичні зсуви всіх наступних секцій зміщуються, що потребує

повного перерахунку PointerToRawData у таблиці секцій. Помилка під час перерахунку адрес може призвести до порушення завантаження або працездатності файлу.

в) Нова секція .prot – категорія C.

До таблиці секцій додається новий запис. Збільшуються як довжина файлу, так і обсяг пам’яті, що виділяється під час завантаження. Точка входу переноситься в нову секцію.

Такий спосіб є одним із найбільш помітних під час статичного аналізу, оскільки нетипові імена секцій або нестандартні атрибути легко виявляються інструментами аналізу PE-файлів.

г) Реєстр AppCertDLLs – категорія Z.

Файл auth.exe не змінюється. Захисна DLL підвантажується операційною системою автоматично через механізм AppCertDLLs.

Метод не залишає ознак безпосередньо у структурі PE-файлу, однак може бути виявлений засобами аудиту системного реєстру та контролю автозавантаження.

Таблиця 7.19

Порівняльна таблиця:

Характеристика	Сценарій а (A)	Сценарій б (B)	Сценарій в (C)	Сценарій г (Z)
Категорія	A	B	C	Z
Зміна довжини файлу	Ні	Так	Так	Ні (файл не змінюється)
Зміна фізичних адресацій	Ні	Так	Так	Ні
Зміна віртуальних адресацій	Ні	Ні	Так	Ні
Можливість виявлення засобами статичного аналізу	Низька (нульовий блок у заголовку)	Середня (розмір секції перевищує VirtualSize)	Висока (нова секція, нетипове ім'я)	Відсутня для PE-файлу
Ризик порушення працездатності	Низький	Високий (помилка в адресах)	Середній	Середній

Перевірка вимог підрозділу 5.5.2:

а) Вільний простір у заголовку – відповідає всім вимогам: Х-код використовує відносну адресацію (переміщуваний), вільних байтів достатньо (~256) за умови компактності коду, виконується на старті (затримка мінімальна), не змінює середовище файлу.

б) Розширення секції .text – Частково порушує вимоги до коректності модифікації, оскільки потребує складного перерахунку фізичних адрес і вирівнювань.

в) Нова секція .prot – Відповідає вимогам щодо функціонування, однак порушує вимогу прихованості – нова секція є помітною під час статичного аналізу.

г) Реєстр AppCertDLLs – Відповідає вимозі прихованості від аналізу PE-файлу, однак залежить від конфігурації середовища та наявності адміністративних привілеїв.

Для захисту модуля автентифікації системи ЦЗ найбільш доцільним є сценарій а (категорія А – використання вільного простору PE-заголовка).

Такий підхід не змінює довжину файлу, не потребує складного коригування адресацій і має найменшу помітність під час статичного аналізу. Крім того, він найбільш повно відповідає вимогам до Х-коду, визначеним у підрозділі 5.5.2.

Після внесення змін контрольна сума файлу (CheckSum) повинна бути перерахована, щоб забезпечити коректне завантаження та перевірку цілісності образу.

РОЗДІЛ 8. АПАРАТНО-ПРОГРАМНИЙ ЗАХИСТ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

8.1. АПАРАТНО-ПРОГРАМНІ МЕХАНІЗМИ ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

8.1.1. Особливості апаратно-програмного захисту програмного забезпечення

У сучасних умовах програмне забезпечення є одним з основних об’єктів атак, пов’язаних із модифікацією програмного коду, компрометацією вбудованого програмного забезпечення (firmware), викраденням криптографічних ключів, аналізом внутрішньої структури програм і несанкціонованим копіюванням. Використання лише програмних механізмів захисту не завжди забезпечує необхідний рівень безпеки, оскільки зловмисник може виконувати аналіз пам’яті, здійснювати reverse engineering, використовувати засоби налагодження або втручатися у процес виконання програм.

У зв’язку з цим дедалі ширше застосовуються апаратно-програмні механізми захисту, які поєднують програмні засоби безпеки зі спеціалізованими апаратними компонентами. Такий підхід дозволяє перенести частину критичних функцій безпеки на нижчі рівні архітектури системи, що суттєво ускладнює компрометацію програмних компонентів.

До основних функцій апаратно-програмного захисту належать:

– захищене зберігання криптографічних ключів;

– контроль цілісності програмних компонентів;

– формування довіреного середовища виконання;

– автентифікація програмних і апаратних компонентів;

– захист вбудованого ПЗ та процесу завантаження;

–протидія несанкціонованому копіюванню ПЗ.

Місце апаратно-програмного захисту у загальній системі безпеки програмного забезпечення наведено на рис. 8.1.



Рисунок 8.1. Місце апаратно-програмного захисту у загальній системі безпеки програмного забезпечення

Наведена схема демонструє, що апаратно-програмний захист займає проміжне положення між програмними та апаратними механізмами безпеки і забезпечує їх взаємодію у межах багаторівневого підходу Defense in Depth.

Основні переваги та обмеження апаратно-програмного захисту наведено у табл. 8.1.

Таблиця 8.1

Переваги та обмеження апаратно-програмного захисту

Переваги	Обмеження
Висока стійкість до модифікації ПЗ	Залежність від апаратної платформи
Захищене зберігання ключів	Підвищення вартості системи
Контроль цілісності	Складність інтеграції
Протидія reverse engineering	Необхідність підтримки виробника
Підтримка довіреного завантаження	Можливість апаратних вразливостей
Ускладнення несанкціонованого копіювання	Обмежена універсальність

Наведені характеристики демонструють, що апаратно-програмний захист дозволяє підвищити стійкість програмного забезпечення до модифікації та

компрометації критичних компонентів, однак його ефективність залежить від правильного поєднання програмних і апаратних механізмів.

8.1.2. Trusted Platform Module у захисті програмного забезпечення

Одним із найбільш поширених механізмів апаратного захисту є Trusted Platform Module (TPM), який використовується для формування апаратного кореня довіри та захищеного зберігання криптографічних ключів.

Використання TPM дозволяє реалізувати:

– перевірку стану системи під час запуску;

– контроль цілісності вбудованого програмного забезпечення;

– підтримку механізмів Secure Boot;

– захищене зберігання ключів;

– автентифікацію пристроїв.

Принцип роботи TPM наведено на рис. 8.2.

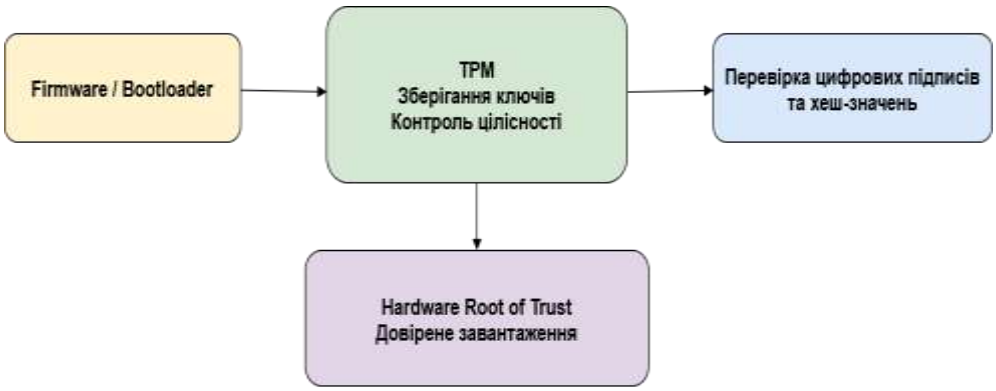


Рисунок 8.2. Принцип роботи TPM та формування апаратного кореня довіри

Наведена схема демонструє взаємодію TPM із компонентами завантаження системи та механізмами перевірки цифрових підписів. Основні функції TPM наведено у табл. 8.2.

Таблиця 8.2

Основні функції TPM у захисті програмного забезпечення

Функція TPM	Сутність функції	Значення для захисту ПЗ
Захищене зберігання ключів	Зберігання криптографічних ключів у межах TPM без можливості їх прямого зчитування	Запобігання компрометації ключів
Перевірка цілісності системи	Контроль хеш-значень firmware, bootloader та системних компонентів	Виявлення несанкціонованої модифікації
Підтримка Secure Boot	Перевірка цифрових підписів під час запуску системи	Блокування запуску модифікованого ПЗ
Формування апаратного кореня довіри	Створення довіреної основи для запуску компонентів системи	Підвищення загальної стійкості системи
Автентифікація пристрою	Використання криптографічних механізмів підтвердження автентичності	Захист від підміни обладнання
Захист каналів зв’язку	Використання ключів TPM у криптографічних протоколах	Захист службового обміну даними
Підтримка encrypted storage	Шифрування конфіденційних даних і системних параметрів	Захист службової інформації
Контроль довіреного стану системи	Перевірка цілісності програмного середовища перед виконанням критичних операцій	Запобігання роботі скомпрометованої системи

Таким чином, TPM забезпечує не лише захист криптографічних ключів, а й контроль цілісності програмного забезпечення, підтримку secure boot та формування апаратного кореня довіри, що є особливо важливим для систем цивільного захисту й критичної інфраструктури й використовується не як окремий засіб захисту, а як складова комплексної системи безпеки програмного забезпечення у мобільних пунктах управління, embedded-комплексах моніторингу та ситуаційних центрах, де необхідно захищати криптографічні

ключі, перевіряти автентичність вузлів і запобігати запуску модифікованого програмного забезпечення.

8.1.3. Secure Boot та формування ланцюга довіри

Для запобігання запуску модифікованого програмного забезпечення використовується механізм Secure Boot. Його принцип роботи базується на послідовній перевірці цифрових підписів компонентів системи під час завантаження. Кожний наступний етап завантаження виконується лише після підтвердження цілісності та автентичності попереднього компонента.

Логіку формування ланцюга довіри наведено на рис. 8.3.

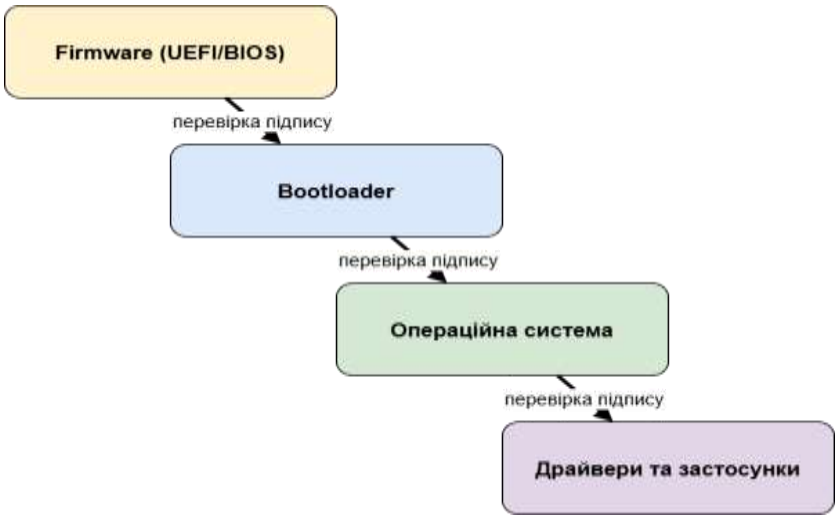


Рисунок 8.3. Ланцюг довіри під час Secure Boot

Наведена схема демонструє, що кожний компонент системи перевіряє автентичність наступного компонента перед його запуском.

Secure Boot дозволяє запобігти bootkit-атакам, підміні системних компонентів і запуску неавторизованого вбудованого програмного забезпечення. У системах цивільного захисту Secure Boot доцільно застосовувати в серверних платформах ситуаційних центрів, контролерах систем оповіщення, телеметричних вузлах і мобільних комплексах реагування, оскільки компрометація bootloader або firmware може призвести до запуску недовіреного програмного забезпечення.

8.1.4. Trusted Execution Environment та захист середовища виконання

Trusted Execution Environment (TEE) – це ізольоване середовище виконання, призначене для обробки критичних даних та виконання захищеного коду незалежно від основної операційної системи.

TEE дозволяє ізолювати криптографічні операції, захищати ключові дані та виконувати критичні функції у спеціальному захищеному середовищі. Навіть у випадку компрометації операційної системи зловмисник не отримує безпосереднього доступу до даних і процесів, що виконуються всередині TEE.

Прикладами таких технологій є ARM TrustZone, Intel SGX та AMD SEV. У системах цивільного захисту подібні механізми можуть використовуватись для захисту телеметричних даних, автентифікації компонентів системи та роботи з криптографічними ключами.

Технології Trusted Execution Environment (TEE) забезпечують ізольоване виконання критичних програмних компонентів і захист конфіденційних даних навіть у випадку компрометації основної операційної системи. Порівняльну характеристику найбільш поширених технологій TEE наведено у табл. 8.3.

Таблиця 8.3

Порівняння технологій Trusted Execution Environment

Технологія / Платформа	Принцип роботи	Призначення	Переваги	Обмеження
ARM TrustZone/ ARM-процесори	Поділ системи на Secure World і Normal World	Захист embedded- та mobile-систем	Висока продуктивність, інтеграція у SoC	Залежність від реалізації виробника
Intel SGX/ Intel x86	Ізольовані enclaves у пам’яті процесора	Захист критичних обчислень і даних	Високий рівень ізоляції процесів	Обмежений обсяг enclave-пам’яті
AMD SEV/ AMD EPYC та Ryzen	Апаратне шифрування пам’яті віртуальних машин	Захист віртуалізованих середовищ	Захист VM від гіпервізора	Орієнтація переважно на серверні системи

Технологія / Платформа	Принцип роботи	Призначення	Переваги	Обмеження
Apple Secure Enclav Apple Silicon	Виділений криптографічний співпроцесор	Захист біометричних і криптографічних даних	Високий рівень інтеграції та безпеки	Закритість архітектури
Google Titan M/ Android-пристрої	Апаратний модуль безпеки для Android	Захист boot-процесу та ключів	Підтримка verified boot	Обмежене застосування поза Android
Microsoft Pluton/ ПК та серверні платформи	Інтеграція апаратного кореня довіри у CPU	Захист firmware і ключових компонентів	Поглиблена інтеграція з ОС	Залежність від підтримки виробників

Trusted Execution Environment відрізняються архітектурою, рівнем ізоляції та сферами застосування, однак усі вони спрямовані на забезпечення довіреного виконання критичних компонентів програмного забезпечення, захист криптографічних ключів і підвищення стійкості систем до компрометації.

У системах цивільного захисту TEE може використовуватися для захищеної обробки телеметричних даних, виконання криптографічних операцій, автентифікації вузлів і захисту критичних службових функцій навіть у разі компрометації основної ОС.

8.1.5. Апаратний захист від несанкціонованого копіювання

Однією із загроз для програмного забезпечення є його несанкціоноване копіювання та подальша модифікація. Для протидії таким загрозам використовуються механізми прив’язки програмного забезпечення до конкретного обладнання, засоби захищеного зберігання ліцензійних параметрів та спеціалізовані апаратні засоби автентифікації.

Методи апаратного захисту від несанкціонованого копіювання програмного забезпечення наведено на рис. 8.4.

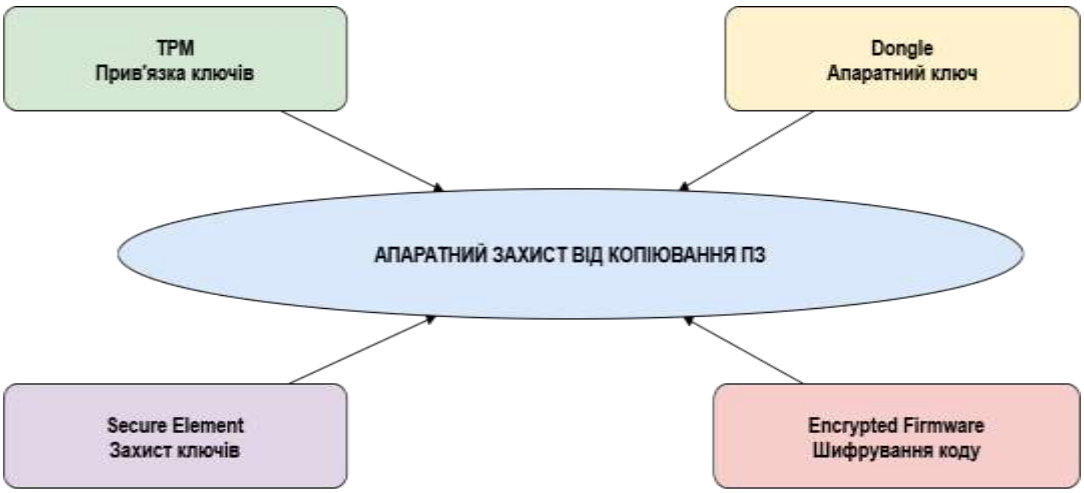


Рисунок 8.4. Методи апаратного захисту від несанкціонованого копіювання ПЗ

Для цього використовуються апаратні ключі (dongle), прив’язка ліцензії до обладнання, encrypted firmware та механізми апаратної автентифікації компонентів системи.

Dongle-захист передбачає використання спеціального апаратного пристрою, без якого запуск програмного забезпечення неможливий.

Програма перевіряє наявність апаратного ключа та його криптографічних параметрів перед виконанням критичних функцій.

Іншим підходом є прив’язка програмного забезпечення до параметрів конкретної платформи, наприклад TPM, серійного номера процесора або ідентифікатора embedded-пристрою.

У сучасних embedded-системах широко використовується encrypted firmware – зберігання вбудованого програмного забезпечення (firmware) у зашифрованому вигляді з автоматичним дешифруванням під час запуску системи. Це ускладнює аналіз коду, несанкціоноване копіювання та модифікацію firmware.

Наведена схема демонструє, що сучасні методи захисту від копіювання програмного забезпечення базуються не на одному окремому механізмі, а на їх поєднанні. Використання TPM, апаратних ключів, Secure Element та механізмів шифрування вбудованого програмного забезпечення дозволяє підвищити стійкість системи до копіювання, модифікації програмного коду та

компрометації критичних компонентів. Для систем цивільного захисту апаратний захист від копіювання важливий насамперед у embedded-пристроях моніторингу, контролерах систем оповіщення, мобільних пунктах управління та спеціалізованому ПЗ, де копіювання або модифікація firmware може порушити роботу критичних сервісів.

8.2. ПОТОКОВІ МЕХАНІЗМИ ТА СКРЕМБЛЕРИ У ЗАХИСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У сучасних програмних системах питання безпеки пов’язане не лише із захистом програмного коду, а й із необхідністю забезпечення цілісності службових потоків даних, каналів оновлення програмного забезпечення, firmware, телеметричної інформації та службових повідомлень. Для таких задач поряд із традиційними криптографічними механізмами використовуються потокові механізми захисту, які забезпечують послідовне перетворення інформації під час її передавання або обробки.

На відміну від блокових механізмів, що виконують обробку інформації фіксованими блоками, потокові механізми здійснюють безперервне перетворення потоку даних у режимі реального часу. Такий підхід дозволяє зменшити затримки обробки, знизити вимоги до обчислювальних ресурсів та забезпечити ефективну роботу у системах з обмеженою продуктивністю.

У системах захисту програмного забезпечення потокові механізми застосовуються для захисту службових каналів зв’язку, передавання телеметричних даних, захисту вбудованого програмного забезпечення, безпечного оновлення програмних компонентів, захисту обміну даними між embedded-пристроями, підтримки роботи IoT-компонентів.

У сучасних системах захисту програмного забезпечення потокові механізми використовуються не лише для перетворення інформаційного потоку, а й як допоміжний механізм ускладнення аналізу службових даних, приховування структури обміну, захисту вбудованого програмного забезпечення та зменшення передбачуваності інформаційних послідовностей. Особливого

значення подібні механізми набувають у lightweight- та embedded-системах, де використання повноцінних криптографічних алгоритмів може бути обмежене вимогами до продуктивності, енергоспоживання або апаратних ресурсів.

У системах цивільного захисту використовуються телеметричні комплекси, системи оповіщення населення, мобільні пункти управління та мережі сенсорного моніторингу. Для подібних систем критично важливими є швидкодія, безперервність функціонування та можливість роботи в умовах обмежених апаратних ресурсів.

Практичне використання скремблерів і потокових механізмів регламентується низкою галузевих стандартів та специфікацій у сфері цифрового зв'язку. Зокрема, механізми скремблювання застосовуються у професійних системах радіозв'язку стандарту TETRA (ETSI EN 300 392-7), цифрових телекомунікаційних мережах, супутникових системах зв'язку, телеметричних комплексах та засобах передавання службових даних. Використання скремблерів у таких системах дозволяє покращувати статистичні характеристики інформаційного потоку, ускладнювати несанкціоноване прослуховування службового трафіку, зменшувати вплив повторюваних послідовностей та підвищувати надійність передавання даних у каналах зв'язку [60]. Для систем цивільного захисту подібні механізми становлять інтерес насамперед як засіб захисту телеметричних потоків, службового радіозв'язку, каналів обміну між мобільними пунктами управління та компонентами розподіленої інфраструктури реагування на надзвичайні ситуації.

8.2.1. Особливості використання скремблерів

Одним із різновидів потокових механізмів є скремблери. Скремблером називають механізм перетворення інформаційного потоку, призначений для зміни його структури та зменшення передбачуваності переданих даних.

На відміну від повноцінних криптографічних алгоритмів, основною метою скремблювання не завжди є забезпечення високої криптостійкості. У більшості випадків його призначення полягає в усуненні довгих повторюваних

послідовностей, покращенні статистичних характеристик потоку, зменшенні можливості прогнозування даних, ускладненні аналізу інформаційного потоку та покращенні характеристик передавання сигналів.

Скремблювання широко використовується у системах передавання інформації, цифрових мережах зв'язку, телеметричних системах та embedded-платформах. У системах захисту програмного забезпечення скремблери застосовуються для приховування структури службового обміну, ускладнення аналізу вбудованого програмного забезпечення, маскування службових параметрів та зменшення інформативності дампів пам'яті.

У деяких системах захисту програмного забезпечення скремблери використовуються як допоміжний anti-analysis механізм. Зокрема, потокове перетворення службових даних, конфігураційних параметрів або фрагментів вбудованого програмного забезпечення (firmware) дозволяє ускладнювати статичний аналіз програмного забезпечення, приховувати структуру службового обміну та зменшувати інформативність дампів пам'яті у випадку компрометації системи.

Загальний принцип потокового перетворення інформації наведено на рис. 8.5.

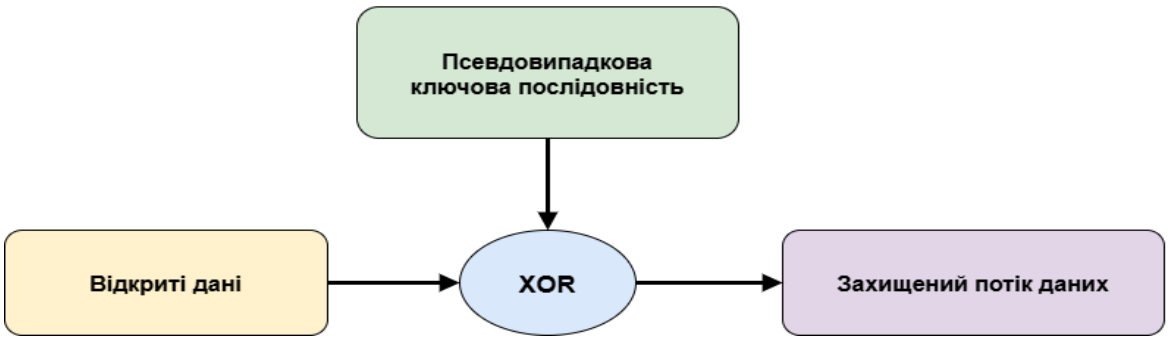


Рисунок 8.5. Загальний принцип потокового перетворення даних

Наведена схема демонструє загальний принцип роботи потокових механізмів. Псевдовипадкова послідовність використовується для послідовного перетворення інформаційного потоку шляхом виконання операції XOR.

До основних переваг скремблерів належать висока швидкодія, невеликі вимоги до апаратних ресурсів, простота програмної реалізації та можливість

використання у режимі реального часу. Крім того, скремблери є ефективними для обробки безперервних потоків даних, придатними для lightweight- та embedded-систем, а також можуть застосовуватись у системах телеметрії та захисту вбудованого програмного забезпечення.

Разом із перевагами ефективність скремблерів значною мірою залежить від механізму формування псевдовипадкових послідовностей, синхронізації сторін обміну та захисту параметрів системи.

Основні переваги та обмеження використання скремблерів наведено у табл. 8.4.

Таблиця 8.4

Переваги та обмеження скремблерів

Переваги	Обмеження
Висока швидкодія	Залежність від синхронізації
Низькі вимоги до ресурсів	Можливість прогнозування ПСП
Простота реалізації	Невисока криптостійкість простих схем
Можливість роботи у реальному часі	Чутливість до помилок реалізації
Придатність для embedded-систем	Необхідність додаткового контролю цілісності
Ефективність для телеметрії та firmware	Обмеженість простих реалізацій

Наведені характеристики демонструють, що скремблери є ефективним допоміжним засобом захисту інформації, однак у сучасних системах вони зазвичай використовуються у поєднанні з криптографічними механізмами, засобами контролю цілісності та апаратно-програмними механізмами захисту.

Практичне значення скремблерів у захисті програмного забезпечення наведено у табл. 8.5.

Таблиця 8.5

Практичне значення скремблерів у захисті програмного забезпечення

Напрям використання	Роль скремблерів
Захист firmware	Ускладнення аналізу та модифікації
Embedded-системи	Lightweight-захист потоків даних

Напрям використання	Роль скремблерів
Телеметрія	Маскування службового обміну
IoT-компоненти	Зниження навантаження на ресурси
Захист пам’яті	Зменшення інформативності дампів
Канали оновлення	Ускладнення аналізу службових пакетів
Anti-analysis	Приховування структури службових даних

Нижче наведено код спрощеного програмного скремблера, який демонструє принцип маскування голосового потоку за допомогою XOR із псевдовипадковою послідовністю. Для реального криптографічного захисту службового зв’язку слід використовувати стандартизовані криптографічні алгоритми та протоколи [61; 68].

```
#include <iostream>
#include <vector>
#include <cstdint>

uint8_t lfsr_next(uint16_t &state)
{
   uint8_t new_bit = ((state >> 15) ^ (state >> 13) ^
                      (state >> 12) ^ (state >> 10)) & 1;

   state = (state << 1) | new_bit;

   return state & 0xFF;
}

std::vector<uint8_t> scramble_voice(
   const std::vector<uint8_t> &voice_stream,
   uint16_t seed
)
{
   std::vector<uint8_t> result;
   uint16_t lfsr_state = seed;

   for (uint8_t sample : voice_stream)
   {
      uint8_t key_byte = lfsr_next(lfsr_state);
      result.push_back(sample ^ key_byte);
   }
```

```
    return result;
}

int main()
{
    std::vector<uint8_t> voice_data = {
        120, 122, 125, 130, 128, 126, 124, 121
    };

    uint16_t seed = 0xACE1;

    auto scrambled = scramble_voice(voice_data, seed);
    auto restored = scramble_voice(scrambled, seed);

    std::cout << "Original: ";
    for (auto b : voice_data) std::cout << (int)b << " ";

    std::cout << "\nScrambled: ";
    for (auto b : scrambled) std::cout << (int)b << " ";

    std::cout << "\nRestored: ";
    for (auto b : restored) std::cout << (int)b << " ";

    return 0;
}
```

Наведені приклади демонструють, що скремблери використовуються не лише для потокового перетворення даних, а й як допоміжний механізм підвищення стійкості програмного забезпечення до аналізу, модифікації та компрометації службових компонентів.

У наведеному прикладі голосовий потік подано як послідовність байтів, що умовно відповідають дискретизованим аудіосемплам. Генератор LFSR формує псевдовипадкову ключову послідовність, яка використовується для побітового перетворення голосових даних за допомогою операції XOR. Оскільки операція XOR є симетричною, той самий алгоритм може застосовуватися як для скремблювання, так і для відновлення початкового голосового потоку за умови використання однакового початкового стану генератора [68].

Такий підхід може використовуватися як навчальна модель захисту голосового трафіку від несанкціонованого прослуховування, однак у практичних

системах цивільного захисту скремблювання має поєднуватися з криптографічним захистом, автентифікацією абонентів, захищеним зберіганням ключів та контролем цілісності каналів зв'язку.

8.2.2. LFSR як механізм формування псевдовипадкових послідовностей

LFSR-механізми – це регістри зсуву зі зворотним зв'язком (Linear Feedback Shift Register), призначені для формування псевдовипадкових послідовностей, які використовуються у потокових механізмах захисту програмного забезпечення, системах маскування службових даних, firmware-захисті та embedded-платформах. Завдяки високій швидкодії, простоті програмної реалізації та невеликим вимогам до апаратних ресурсів LFSR-механізми широко застосовуються у системах, де критичними є безперервність функціонування та робота у режимі реального часу. LFSR використовується у генераторах псевдовипадкових послідовностей, тестуванні цифрових схем, потокових механізмах захисту, телеметричних системах, embedded-пристроях, системах передавання даних [62; 64].

Принцип роботи LFSR наведено на рис. 8.6.

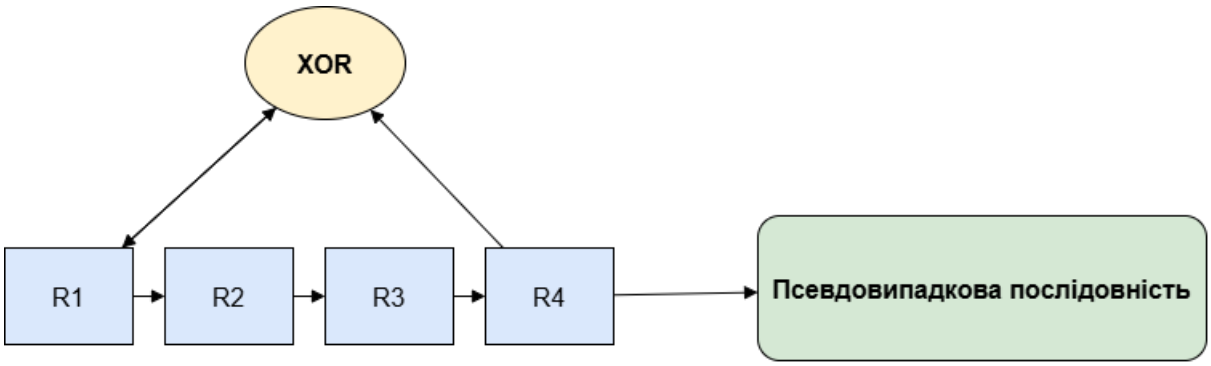


Рисунок 8.6. Принцип роботи LFSR

Наведена схема демонструє, що новий біт формується на основі функції зворотного зв'язку, після чого відбувається зміщення вмісту регістра та генерація чергового елемента псевдовипадкової послідовності.

Основні характеристики LFSR наведено у табл. 8.6.

Таблиця 8.6

Основні характеристики LFSR-механізмів

Характеристика	Опис
Принцип роботи	Послідовний зсув регістра з формуванням нового біта за допомогою XOR між визначеними розрядами
Тип генерованої послідовності	Псевдовипадкова бітова послідовність
Основна операція	XOR (додавання за модулем 2)
Структура	Набір тригерів (розрядів) та ліній зворотного зв’язку
Швидкодія	Висока завдяки простоті реалізації
Вимоги до ресурсів	Мінімальні апаратні та програмні витрати
Криптостійкість	Обмежена через лінійність структури
Основна перевага	Простота реалізації та ефективність у потокових системах
Основний недолік	Вразливість до криптоаналізу та прогнозування послідовності
Типові сфери застосування	Потокові шифри, embedded-системи, телеметрія, тестування цифрових схем
Використання у сучасних системах	Переважно як компонент складніших криптографічних механізмів

Незважаючи на простоту реалізації та високу швидкодію, класичні LFSR не забезпечують достатньої криптостійкості для самостійного використання у сучасних системах безпеки [63, 64]. Основний недолік LFSR полягає у лінійності структури, що за наявності достатньої кількості вихідних бітів дозволяє відновлювати параметри генератора за допомогою спеціалізованих методів криптоаналізу, зокрема алгоритму Берлекемпа-Мессі [64].

У зв’язку з цим у сучасних системах LFSR використовуються переважно як складові більш складних криптографічних механізмів або для допоміжного формування псевдовипадкових послідовностей.

8.2.3. Самосинхронізуючі скремблери

Для потокових систем захисту критично важливим є забезпечення синхронізації між передавальною та приймальною сторонами. Одним із підходів до вирішення цієї задачі є використання самосинхронізуючих скремблерів. На відміну від звичайних потокових механізмів, самосинхронізуючі скремблери використовують попередні біти зашифрованого потоку для формування наступних елементів ключової послідовності.

Такий підхід дозволяє автоматично відновлювати синхронізацію після втрати окремих символів або короткочасних помилок передавання [68].

Процес роботи самосинхронізуючого скремблера наведено на рис. 8.7.

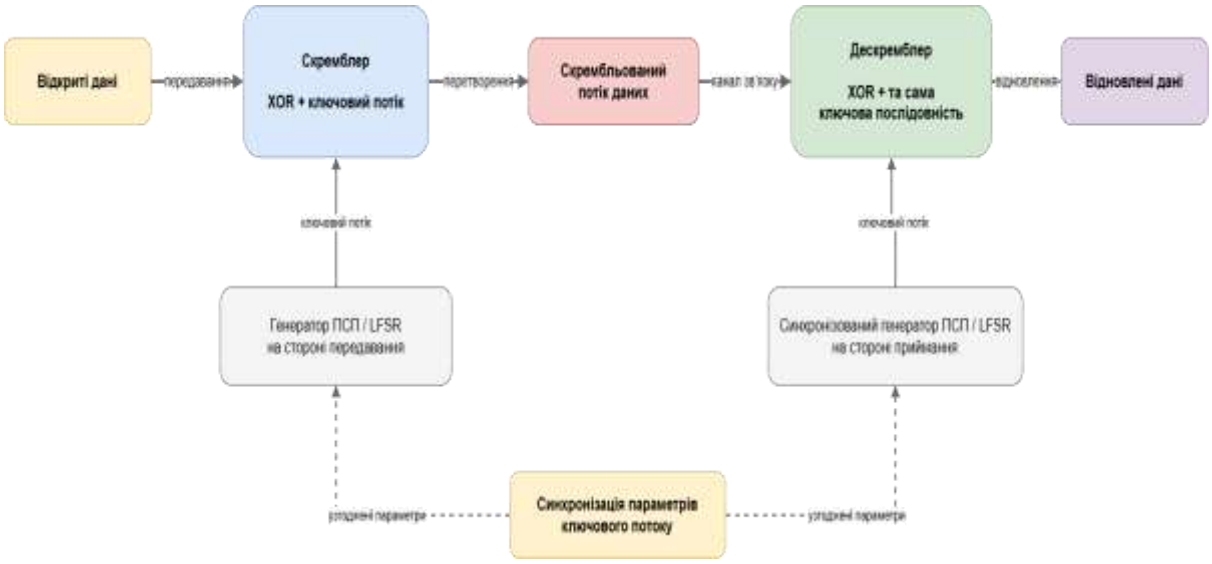


Рисунок 8.7. Процес скремблювання та дескремблювання даних

Наведена схема демонструє взаємозв’язок процесів скремблювання та дескремблювання, а також роль синхронізації параметрів під час передавання даних.

Перевагами самосинхронізуючих скремблерів є автоматичне відновлення синхронізації, стійкість до короткочасних помилок передавання, можливість використання у потокових системах, невеликі обчислювальні витрати.

Водночас для таких механізмів характерні певні обмеження:

– можливість поширення помилок у потоці;

– залежність від параметрів регістра;

– необхідність захисту ключових параметрів;

– обмежена криптостійкість простих реалізацій.

Особливого значення самосинхронізуючі скремблери набувають у системах цивільного захисту, де використовуються телеметричні потоки, сенсорні мережі, системи моніторингу та канали службового зв’язку. У подібних системах короткочасна втрата пакетів або порушення синхронізації не повинні призводити до повної втрати інформації, тому механізми автоматичного відновлення синхронізації дозволяють підвищити надійність функціонування системи.

Таким чином, потокові механізми та скремблери є важливими складовими сучасних систем захисту програмного забезпечення і можуть використовуватися як допоміжні засоби підвищення захищеності інформаційних потоків, вбудованого програмного забезпечення та службових каналів зв’язку.

8.3. ПРОГРАМНА РЕАЛІЗАЦІЯ ПОТОКОВИХ МЕХАНІЗМІВ ЗАХИСТУ

Потокові механізми захисту широко використовуються у програмних системах для забезпечення безперервного перетворення інформації, захисту службових потоків даних та підтримки безпечного обміну інформацією між компонентами системи. На відміну від блокових механізмів, потокові алгоритми орієнтовані на послідовну обробку інформації, що дозволяє зменшити затримки та ефективно працювати у системах реального часу.

Програмна реалізація потокових механізмів має важливе значення для сучасних систем захисту програмного забезпечення, оскільки дозволяє забезпечити захист службових повідомлень, вбудованого програмного забезпечення, каналів оновлення та телеметричних даних. Особливо важливим це є для систем цивільного захисту, де використовуються embedded-пристрої, автономні сенсорні вузли, мобільні пункти управління та комплекси моніторингу.

8.3.1. Загальні принципи реалізації

Програмна реалізація потокових механізмів захисту передбачає виконання послідовного перетворення інформаційного потоку за допомогою ключової псевдовипадкової послідовності.

У більшості випадків таке перетворення виконується шляхом побітової операції XOR між відкритими даними та елементами ключового потоку. Оскільки операція XOR є оборотною, одна й та сама операція може використовуватися як для перетворення інформації, так і для її відновлення.

Основними компонентами програмної реалізації потокового захисту є генератор псевдовипадкової послідовності, механізм синхронізації, алгоритми скремблювання та дескремблювання, а також засоби сихрнізації й контролю цілісності даних. У найпростішому випадку кожен біт інформаційного потоку послідовно обробляється ключовою послідовністю, сформованою генератором ПСП.

Загальна структура програмної реалізації потокового механізму захисту включає генерацію псевдовипадкової послідовності, формування ключового потоку, виконання потокового перетворення даних та контроль цілісності отриманих результатів. Загальну схему програмної реалізації потокового захисту даних наведено на рис. 8.8.



Рисунок 8.8. Загальна схема програмної реалізації потокового захисту даних

Наведена схема демонструє послідовність основних етапів потокового захисту інформації. Генератор псевдовипадкової послідовності формує ключовий потік, який використовується для виконання операції XOR з інформаційним потоком. Після перетворення даних можуть застосовуватися механізми контролю цілісності, що дозволяють виявляти несанкціоновані модифікації або помилки передавання.

Перевагами програмної реалізації потокового захисту є висока швидкодія, простота реалізації, низькі вимоги до апаратних ресурсів, можливість роботи у режимі реального часу й придатність для embedded-платформ [65].

У сучасних системах потокові механізми використовуються не як окремі засоби захисту, а як компоненти комплексної системи безпеки.

8.3.2. Програмна реалізація LFSR-скремблера

Одним із найбільш поширених підходів до програмної реалізації потокового захисту є використання лінійних регістрів зсуву зі зворотним зв’язком (LFSR).

Принцип роботи такого механізму полягає у послідовному зміщенні стану регістра та формуванні нового біта за допомогою функції зворотного зв’язку. У процесі генерації ключового потоку система виконує (рис. 8.9):

– зчитування визначених розрядів регістра;
– виконання операції XOR;
– формування нового біта;
– зміщення вмісту регістра;
– формування елемента псевдовипадкової послідовності.

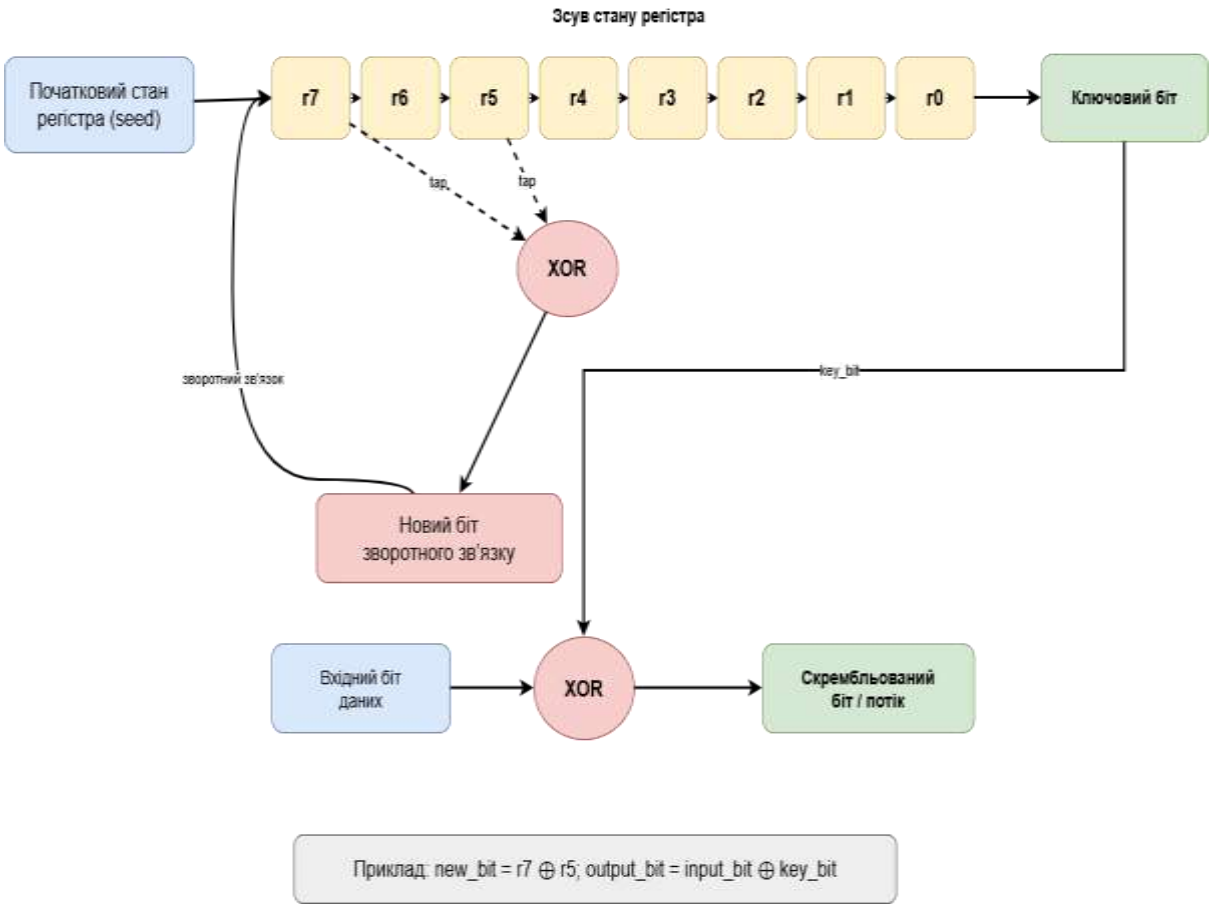


Рисунок 8.9. Принцип роботи програмного LFSR-скремблера

Наведена схема демонструє процес генерації ключової послідовності та її використання для перетворення інформаційного потоку.

Нижче наведено приклад спрощеної програмної реалізації LFSR-скремблера мовою C++.

```
unsigned char lfsr = 0b10101101;

for(int i=0;i<8;i++)
{
    int new_bit=((lfsr>>7)^(lfsr>>5))&1;

    lfsr=(lfsr<<1)|new_bit;

    int key=lfsr&1;

    output[i]=input[i]^key;
}
```

У наведеному прикладі новий біт формується шляхом XOR між визначеними розрядами регістра. Після цього виконується зсув регістра, формується ключовий біт та здійснюється побітове перетворення вхідних даних. Джерело [65] містить безпосередньо програмний файл **scramble.c**, який включає повну реалізацію скремблера та дескремблера на основі LFSR. Автор прямо зазначає, що проєкт присвячений самосинхронізуючим скремблерам та містить програму для демонстрації роботи скремблера.

У роботі [66] наведено готові приклади реалізації Fibonacci LFSR на C, Galois LFSR на C, Xorshift LFSR на C. Також окремо пояснюється використання LFSR для scrambling/descrambling у телекомунікаційних системах.

Наукова стаття [67] включає схему скремблера та дескремблера, у якій розглядаються архітектура скремблера, архітектура дескремблера, використання XOR, LFSR з моделлю полінома, приклади реалізації на FPGA.

Основні етапи роботи LFSR-скремблера наведено у табл. 8.7.

Таблиця 8.7

Основні етапи роботи LFSR-скремблера

Етап роботи	Опис етапу
Ініціалізація регістра	Заповнення регістра початковим станом (seed)
Визначення бітів зворотного зв’язку	Вибір розрядів, що беруть участь у формуванні нового біта
Обчислення нового біта	Формування нового біта за допомогою операції XOR
Зсув регістра	Послідовний зсув вмісту регістра на один розряд
Формування ключового біта	Використання одного з розрядів регістра як елемента ПСП
Перетворення даних	Виконання XOR між відкритими даними та ключовим потоком
Формування зашифрованого потоку	Отримання скрембльованих даних
Оновлення стану регістра	Підготовка регістра до наступного такту генерації

Наведені етапи демонструють, що LFSR-скремблер забезпечує послідовне потокове перетворення даних із мінімальними апаратними та обчислювальними витратами, що робить його придатним для embeddedсистем, телеметричних комплексів і засобів зв’язку реального часу.

8.3.3. Програмна реалізація scramble()/descramble()

У більшості потокових механізмів реалізуються дві взаємопов’язані функції:

– scramble() – перетворення відкритих даних;

– descramble() – відновлення початкової інформації.

Процес роботи цих функцій наведено на рис. 8.10.

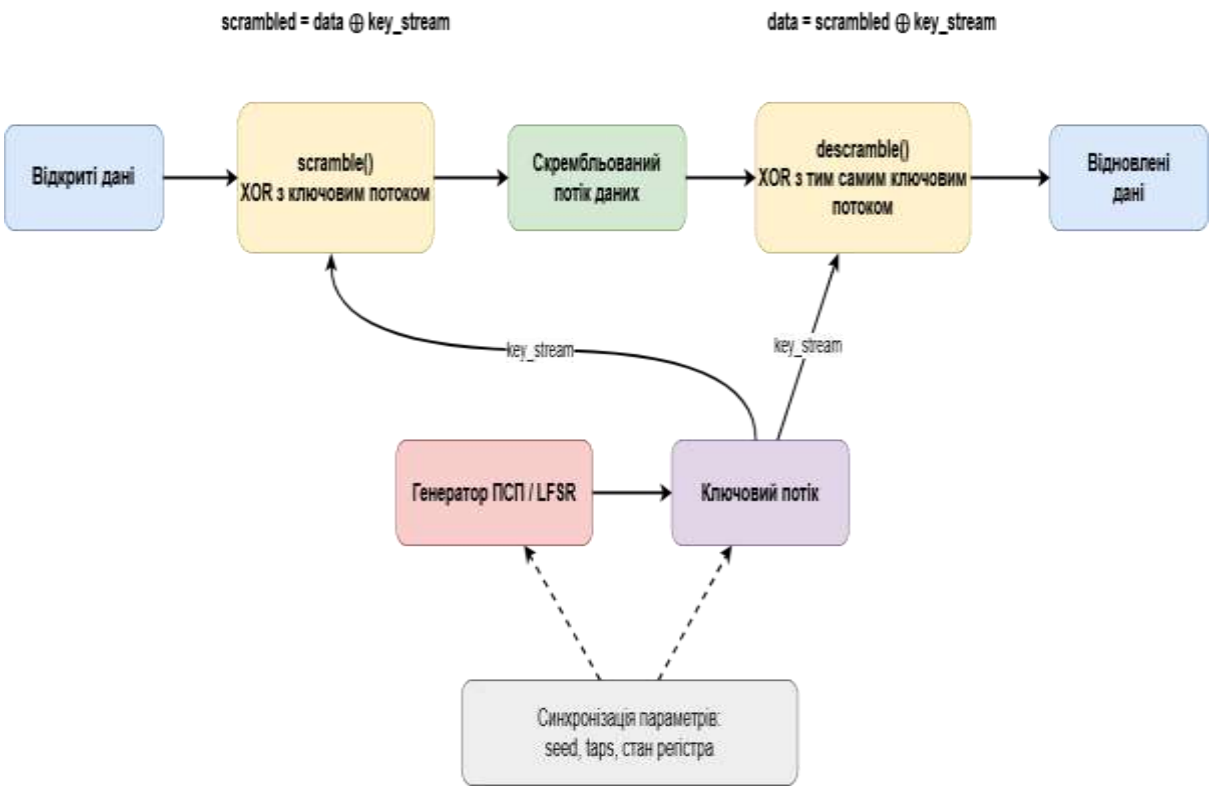


Рисунок 8.10. Процес роботи функцій scramble() та descramble()

Наведена схема демонструє взаємозв’язок процесів перетворення та відновлення інформації.

Приклад програмної реалізації:

```
char scramble(char input,char key)
{
    return input^key;
}

char descramble(char input,char key)
{
    return input^key;
}
```

Оскільки операція XOR є симетричною, одна й та сама функція може використовуватися як для перетворення інформації, так і для її відновлення.

Основні характеристики функцій scramble() та descramble() наведено у табл. 8.8.

Таблиця 8.8

Порівняння функцій scramble() та descramble()

Характеристика	scramble()	descramble()
Призначення	Перетворення	Відновлення
Вхідні дані	Відкриті	Скрембльовані
Операція	XOR	XOR
Результат	Захищений потік	Початкові дані

Наведене порівняння демонструє, що ефективність роботи потокових механізмів безпосередньо залежить від коректної синхронізації ключових параметрів.

8.3.4. Аналіз безпеки реалізації

Безпека програмної реалізації потокових механізмів залежить не лише від використаного алгоритму, а й від способу зберігання параметрів, організації синхронізації та захисту програмних компонентів.

До найбільш поширених загроз належать reverse engineering, аналіз дампів пам’яті, компрометація ключових параметрів, повторне використання ключового потоку, порушення синхронізації, модифікація програмної реалізації, атаки на канали оновлення.

Однією з основних загроз є можливість відновлення параметрів генератора псевдовипадкових послідовностей шляхом аналізу фрагментів вихідного потоку. Для реалізації подібних атак можуть використовуватись криптоаналітичні методи, зокрема алгоритм Берлекемпа–Мессі [64].

Для підвищення рівня захищеності використовуються механізми обфускації, anti-debugging, encrypted firmware, Secure Boot та засоби захищеного зберігання критичних параметрів [62]. Основні загрози наведено у табл. 8.9.

Таблиця 8.9

Основні загрози для програмної реалізації потокового захисту

Загроза	Сутність загрози	Можливі наслідки
Reverse engineering	Аналіз програмного коду та алгоритму захисту	Відновлення логіки роботи механізму
Відновлення параметрів LFSR	Визначення структури генератора ПСП	Компрометація ключового потоку
Повторне використання ключового потоку	Використання однакової ПСП для різних потоків	Можливість криптоаналізу даних
Компрометація ключових параметрів	Отримання доступу до ключів або seed-значень	Повне порушення конфіденційності
Порушення синхронізації	Втрата узгодженості між сторонами обміну	Неможливість коректного відновлення даних
Ін’єкція шкідливого коду	Модифікація програмної реалізації захисту	Обхід або деактивація механізму захисту
Аналіз дампів пам’яті	Отримання ключових параметрів із RAM	Компрометація криптографічних даних
Атаки на канали оновлення	Підміна оновлень програмного забезпечення	Встановлення модифікованого firmware
Відсутність контролю цілісності	Неможливість виявлення модифікації ПЗ	Приховане втручання у роботу системи
Помилки реалізації	Некоректна реалізація алгоритму	Зниження криптостійкості системи

Наведені загрози демонструють, що використання навіть криптографічно стійкого алгоритму не гарантує безпечної роботи системи [62].

Для підвищення рівня захищеності програмного забезпечення можуть використовуватися апаратні модулі довіри TPM, механізми безпечного завантаження (Secure Boot), зашифроване вбудоване програмне забезпечення (encrypted firmware), механізми протидії налагодженню (anti-debugging), обфускація програмного коду, засоби контролю цілісності та механізми захищеного зберігання криптографічних ключів.

У системах цивільного захисту особливе значення має забезпечення безпечного функціонування програмних компонентів телеметричних систем, систем моніторингу та службових каналів зв’язку, оскільки компрометація ключових параметрів або модифікація вбудованого програмного забезпечення можуть призвести до спотворення критичних даних або порушення роботи системи.

Таким чином, програмна реалізація потокових механізмів повинна розглядатися як складова комплексної системи захисту програмного забезпечення, що поєднує криптографічні, програмні та апаратно-програмні механізми безпеки.

8.4. ВИКОРИСТАННЯ АПАРАТНО-ПРОГРАМНИХ МЕХАНІЗМІВ У СИСТЕМАХ ЦИВІЛЬНОГО ЗАХИСТУ

Сучасні системи цивільного захисту функціонують у середовищі, яке характеризується високими вимогами до доступності, достовірності та цілісності інформації. До складу таких систем входять автоматизовані системи оповіщення населення, телеметричні комплекси, ситуаційні центри, сенсорні мережі моніторингу, системи радіаційного та хімічного контролю, мобільні пункти управління, а також спеціалізовані програмно-апаратні комплекси підтримки прийняття рішень [69, 72, 75].

Особливістю програмного забезпечення систем цивільного захисту є необхідність безперервної роботи навіть за умов надзвичайних ситуацій, нестабільного зв’язку або зовнішніх впливів. Порушення цілісності програмних компонентів або компрометація службових даних можуть призводити до помилок під час прийняття рішень, спотворення телеметричної інформації або відмови критичних сервісів [69, 72, 75].

До основних загроз програмному забезпеченню систем цивільного захисту належать [62, 68, 69, 74]:

– модифікація вбудованого програмного забезпечення (firmware);

– компрометація криптографічних ключів;

– підміна службових повідомлень;

– порушення процесу завантаження системи;

– несанкціоноване копіювання програмних компонентів;

– компрометація вузлів телеметричних мереж;

– впровадження шкідливого коду;

– порушення цілісності програмного забезпечення.

Для протидії таким загрозам використовуються апаратно-програмні механізми захисту, які дозволяють реалізувати багаторівневий підхід до безпеки [68, 70, 74].

Під час проєктування та експлуатації програмно-апаратних комплексів цивільного захисту важливо враховувати не лише окремі механізми безпеки, а й загальні вимоги до побудови систем захисту. Основні вимоги до апаратно-програмного захисту систем цивільного захисту наведено у табл. 8.10.

Таблиця 8.10

Вимоги до апаратно-програмного захисту систем цивільного захисту

Вимога	Характеристика	Значення для систем цивільного захисту
Захист цілісності програмного забезпечення	Контроль змін firmware, bootloader і системних компонентів	Запобігання несанкціонованій модифікації ПЗ
Довірене завантаження (Secure Boot)	Перевірка цифрових підписів компонентів перед запуском	Блокування запуску скомпрометованих модулів
Захищене зберігання ключів	Використання TPM, HSM або Secure Element	Захист криптографічних ключів і службових даних
Автентифікація компонентів системи	Перевірка справжності вузлів та пристроїв	Запобігання підміні обладнання
Захист firmware	Шифрування та контроль цілісності firmware	Протидія копіюванню та модифікації коду
Захист каналів зв’язку	Використання криптографічного захисту передавання даних	Запобігання перехопленню або спотворенню інформації
Відмовостійкість	Збереження працездатності при збоях або атаках	Підвищення доступності критичних сервісів

Вимога	Характеристика	Значення для систем цивільного захисту
Контроль оновлень ПЗ	Перевірка цифрових підписів і джерел оновлення	Запобігання встановленню модифікованого ПЗ
Моніторинг стану безпеки	Контроль цілісності та журналювання подій	Своєчасне виявлення інцидентів безпеки
Безпечне оновлення програмного забезпечення	Контроль джерел і цілісності оновлень	Запобігання встановленню скомпрометованих компонентів
Масштабованість і підтримка	Можливість інтеграції нових компонентів	Забезпечення довготривалої експлуатації систем

Наведені вимоги демонструють, що ефективний захист програмного забезпечення систем цивільного захисту передбачає не лише використання окремих механізмів безпеки, а й комплексне забезпечення цілісності, автентичності, доступності та відмовостійкості системи протягом усього життєвого циклу її експлуатації [62, 68, 69].

Загальну структуру використання апаратно-програмних механізмів у системах цивільного захисту наведено на рис. 8.11.

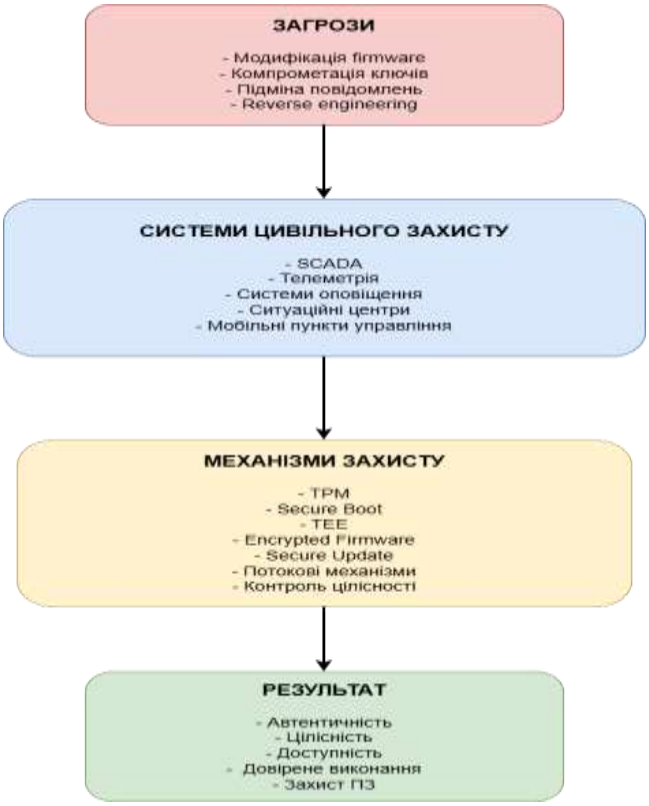


Рисунок 8.11. Використання апаратно-програмних механізмів у системах цивільного захисту

Наведена схема демонструє взаємозв’язок між основними загрозами програмному забезпеченню, компонентами систем цивільного захисту та механізмами захисту, які використовуються для підвищення стійкості програмних систем.

Одним із найбільш поширених механізмів є використання Trusted Platform Module (TPM). У системах цивільного захисту TPM застосовується для захищеного зберігання криптографічних ключів, підтримки механізмів автентифікації та контролю цілісності системи [70, 71].

Використання механізмів безпечного завантаження (Secure Boot) дозволяє реалізувати контроль цифрових підписів компонентів системи під час запуску. Такий підхід запобігає виконанню модифікованого firmware або неавторизованого програмного забезпечення [70, 76].

Trusted Execution Environment (TEE) використовується для ізольованого виконання критичних функцій, зокрема обробки службових повідомлень, виконання криптографічних операцій та захисту ключових параметрів системи [68, 73].

Шифрування вбудованого програмного забезпечення (Encrypted firmware) застосовується для захисту програмних компонентів embedded-пристроїв і перешкоджає несанкціонованому копіюванню або модифікації коду [70, 73, 77].

Потокові механізми та скремблери можуть використовуватися для захисту службових каналів зв’язку та передавання телеметричної інформації [60, 75].

Основні приклади використання апаратно-програмних механізмів у системах цивільного захисту наведено у табл. 8.11.

Таблиця 8.11

Використання апаратно-програмних механізмів у системах цивільного захисту

Компонент системи ЦЗ	Основні загрози	Механізми захисту	Призначення
Системи оповіщення населення	Підміна повідомлень	TPM, Secure Boot, цифровий підпис	Автентичність повідомлень
Embedded-системи моніторингу	Модифікація firmware	Encrypted firmware, Secure Element	Захист ПЗ
Телеметричні комплекси	Перехоплення даних	Потокові механізми, TPM	Захист потоків
Мобільні пункти управління	Компрометація вузлів	TPM, Secure Boot	Контроль цілісності
SCADA	Підміна команд керування	TEE, криптографічна автентифікація	Захист керування
Сенсорні мережі	Підміна вузлів	TEE, захист ключів	Достовірність даних
Радіаційний моніторинг	Підміна вимірювань	Потоковий захист + контроль цілісності	Захист телеметрії
Ситуаційні центри	Компрометація серверного ПЗ	TPM, HSM, Secure Boot	Захист критичних сервісів

Наведені приклади демонструють, що вибір конкретного механізму захисту залежить від функціонального призначення компонента системи та характеру потенційних загроз [69, 75, 77].

Важливою особливістю сучасних систем захисту програмного забезпечення є використання комплексного підходу, який поєднує програмні та апаратно-програмні механізми безпеки. У такій архітектурі механізми безпечного завантаження забезпечують контроль цілісності компонентів під час запуску системи, TPM використовується для захищеного зберігання криптографічних ключів, зашифроване вбудоване програмне забезпечення ускладнює аналіз і модифікацію коду, а механізми протидії налагодженню та обфускації перешкоджають дослідженню внутрішньої логіки програмного

забезпечення. Поєднання декількох механізмів захисту дозволяє знизити ризик компрометації окремих компонентів та підвищити загальну стійкість системи до атак [68, 70, 73, 74, 76].

Архітектуру комплексного використання механізмів захисту у системах цивільного захисту наведено на рис. 8.12.



Рисунок 8.12. Комплексне використання механізмів захисту у системах цивільного захист

Наведена схема демонструє багаторівневий підхід до забезпечення безпеки програмного забезпечення, у межах якого різні механізми захисту взаємодіють між собою та компенсують недоліки окремих засобів [68, 69, 74].

Для систем цивільного захисту застосування комплексного підходу має особливе значення, оскільки компрометація програмного забезпечення може призводити до порушення роботи систем оповіщення, втрати або спотворення телеметричних даних, відмови систем моніторингу та ускладнення координації підрозділів під час реагування на надзвичайні ситуації. Саме тому найбільш ефективним є поєднання механізмів контролю цілісності, захисту ключів, безпечного завантаження та захисту вбудованого програмного забезпечення у межах єдиної багаторівневої системи безпеки.

8.5. ПРАКТИЧНІ ПРИКЛАДИ ЗАСТОСУВАННЯ АПАРАТНО-ПРОГРАМНОГО ЗАХИСТУ У СИСТЕМАХ ЦИВІЛЬНОГО ЗАХИСТУ

У попередніх підрозділах були розглянуті основні апаратні, програмні та апаратно-програмні механізми захисту програмного забезпечення. Проте ефективність таких механізмів найбільш повно проявляється під час їх практичного застосування у реальних системах цивільного захисту. Особливості функціонування таких систем пов’язані з необхідністю безперервної роботи, високою відповідальністю за достовірність інформації, використанням розподіленої інфраструктури та значною кількістю вбудованих пристроїв і телекомунікаційних компонентів.

У цьому підрозділі розглядаються приклади використання апаратно-програмного захисту в системах оповіщення населення, embedded-системах моніторингу, телеметричних і SCADA-комплексах, мобільних пунктах управління та системах службового зв’язку цивільного захисту

8.5.1. Захист систем оповіщення населення

Сучасні системи цивільного захисту значною мірою базуються на цифрових засобах оповіщення населення, які використовують серверні компоненти, телекомунікаційні мережі, мобільні застосунки та embedded-пристрої керування. Надійність функціонування таких систем безпосередньо впливає на своєчасність реагування на надзвичайні ситуації, а їх компрометація може призводити до формування фальшивих повідомлень, блокування процесу оповіщення або порушення роботи інфраструктури реагування.

Одним із прикладів модернізації систем оповіщення в Україні стало впровадження технології Cell Broadcast, яка застосовується для екстреного інформування населення у багатьох країнах світу для інформування населення про загрозу надзвичайних ситуацій та підтримується специфікаціями 3GPP [78]. Використання такого підходу дозволяє здійснювати масове розповсюдження повідомлень через інфраструктуру мобільних операторів навіть за умов нестабільного зв’язку або значного навантаження на мережу.

Для забезпечення захищеності подібних систем використовуються механізми Secure Boot серверних компонентів, криптографічна автентифікація вузлів, TPM-модулі для захисту ключів, контроль цілісності firmware, резервування каналів зв’язку та моніторинг змін конфігурацій. Їх застосування дозволяє знизити ризик компрометації критичних програмних компонентів і забезпечити стабільність функціонування системи.

Схема (рис. 8.13) демонструє взаємодію серверних компонентів, центрів формування повідомлень, телекомунікаційної інфраструктури та абонентських пристроїв із використанням механізмів контролю цілісності та автентифікації.

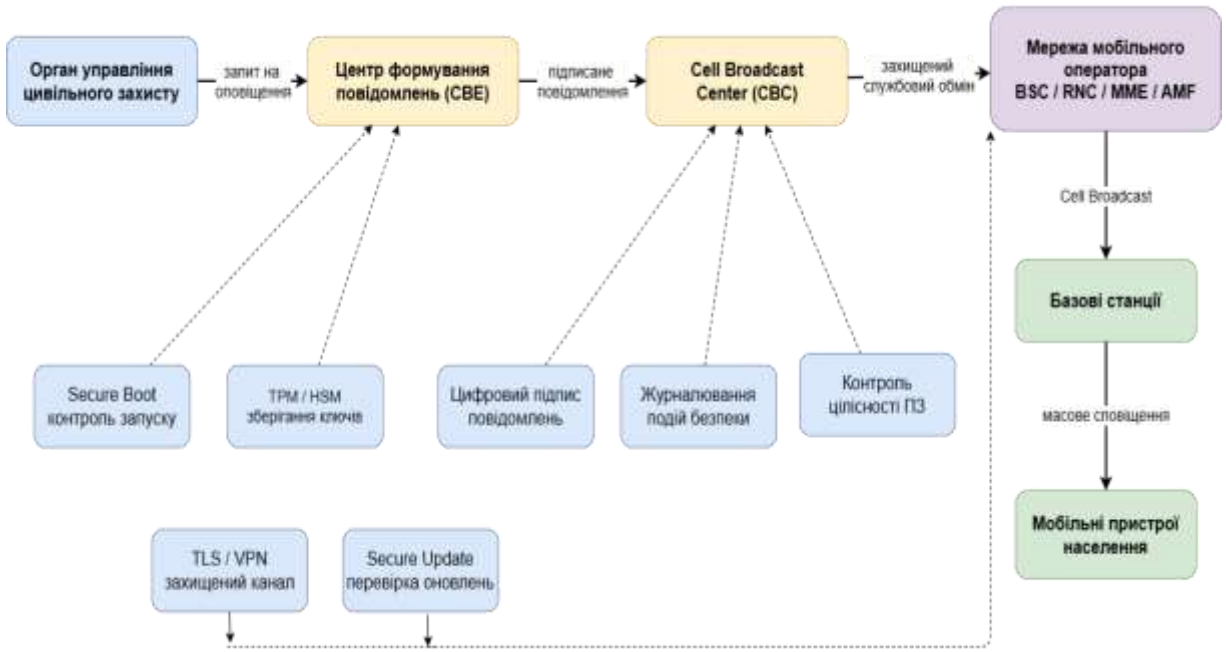


Рисунок 8.13. Архітектура захищеної системи оповіщення населення на основі Cell Broadcast

Особливу небезпеку для цифрових систем оповіщення становлять підміна серверних компонентів, компрометація каналів оновлення, несанкціоноване втручання у firmware контролерів та атаки на канали передавання повідомлень. У сучасних архітектурах оповіщення використовується багаторівневий підхід до захисту, за якого криптографічні механізми поєднуються з апаратною автентифікацією компонентів системи, контролем цілісності та засобами моніторингу безпеки.

Системи оповіщення населення є критично важливими компонентами цивільного захисту, оскільки забезпечують своєчасне інформування про надзвичайні ситуації та координацію дій населення і відповідних служб.

Компрометація таких систем може призвести до поширення неправдивих повідомлень, блокування процесу оповіщення або дезорганізації роботи служб реагування. Основні загрози для систем оповіщення та механізми їх захисту наведено у табл. 8.12.

Таблиця 8.12

Основні загрози та механізми захисту систем оповіщення населення

Загроза	**Характеристика загрози**	**Основні механізми захисту**
Підміна повідомлень	Формування або розсилання неправдивих повідомлень	Цифровий підпис повідомлень, криптографічна автентифікація
Компрометація серверного ПЗ	Модифікація програмних компонентів системи	Контроль цілісності, secure boot
Модифікація firmware-контролерів	Несанкціонована зміна firmware обладнання	Encrypted firmware, TPM, Secure Element
Перехоплення службового трафіку	Отримання доступу до інформації під час передавання	TLS, VPN, потокове шифрування
Атаки на канали оновлення	Завантаження модифікованого ПЗ або firmware	Перевірка цифрових підписів оновлень
DDoS-атаки	Перевантаження серверів або каналів зв’язку	Резервування, балансування навантаження

Наведені приклади демонструють, що ефективний захист систем оповіщення потребує комплексного підходу, який поєднує програмні, криптографічні та апаратні механізми безпеки. Для систем цивільного захисту особливого значення набувають забезпечення достовірності повідомлень, безперервності роботи та стійкості до зовнішніх і внутрішніх загроз.

8.5.2. Захист embedded-систем моніторингу надзвичайних ситуацій

У системах цивільного захисту широко використовуються embedded-пристрої для моніторингу радіаційного, хімічного, сейсмічного та метеорологічного стану навколишнього середовища. Подібні системи зазвичай функціонують автономно, розміщуються на територіально розподілених об’єктах та нерідко експлуатуються у віддалених районах, що збільшує ризик фізичного доступу до обладнання та створює додаткові вимоги до захисту програмного забезпечення [79].

Архітектура таких систем зазвичай включає сенсорні вузли, embedded-контролери, телеметричні модулі, шлюзи передавання даних і центральні сервери обробки інформації (рис. 8.14). Їх взаємодія забезпечує збір параметрів середовища, передавання телеметричної інформації та формування повідомлень про потенційні загрози.

Компрометація firmware embedded-пристроїв може призводити до спотворення телеметричних даних, формування хибних повідомлень або блокування передавання інформації до ситуаційних центрів. Подібні інциденти можуть негативно впливати на процес оцінювання обстановки та прийняття рішень у системах цивільного захисту.

Для протидії зазначеним загрозам використовуються механізми encrypted firmware, Secure Boot, TPM або Secure Element, цифровий підпис оновлень, контроль цілісності пам’яті та захищене зберігання криптографічних ключів. Особливого значення набуває захист каналів оновлення firmware, оскільки механізми OTA-оновлення (Over-the-Air Update) часто є однією з основних цілей атак на embedded-пристрої через можливість несанкціонованої модифікації програмного коду.

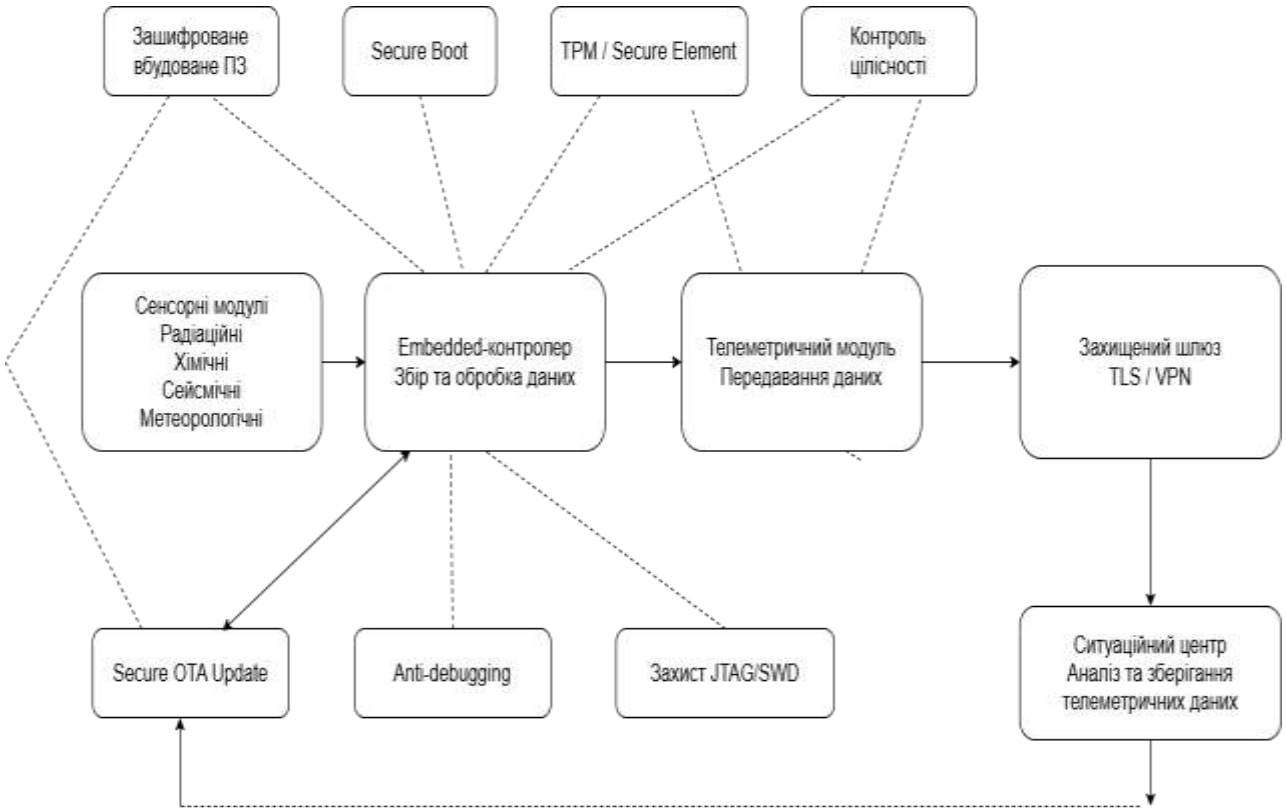


Рисунок 8.14 – Захист embedded-систем моніторингу надзвичайних ситуацій

Схема демонструє використання довіреного завантаження, контролю цілісності та захисту криптографічних ключів у вбудованих системах цивільного захисту.

Embedded-системи у цивільному захисті застосовуються у сенсорних мережах, системах моніторингу, телеметричних комплексах, мобільних пунктах управління та засобах службового зв’язку. Особливістю таких систем є обмежені обчислювальні ресурси, автономність функціонування та підвищений ризик фізичного доступу до обладнання. У зв’язку з цим embedded-платформи потребують використання спеціалізованих апаратно-програмних механізмів безпеки. Основні загрози для embedded систем цивільного захисту та методи їх протидії наведено у табл. 8.13.

Таблиця 8.13

Загрози для embedded-систем ЦЗ та методи протидії

Загроза	**Характеристика загрози**	**Методи протидії**
Компрометація firmware	Несанкціонована модифікація програмного коду пристрою	Encrypted firmware, контроль цілісності, цифровий підпис
Reverse engineering	Аналіз логіки роботи embedded-компонентів	Обфускація, anti-debugging, anti-tamper
Несанкціоноване копіювання ПЗ	Копіювання firmware або конфігурацій системи	Апаратне ліцензування, Secure Element, TPM
Компрометація криптографічних ключів	Отримання доступу до ключових параметрів системи	TPM, HSM, захищене зберігання ключів
Підміна вузлів системи	Підключення неавторизованих пристроїв	Апаратна автентифікація, цифрові сертифікати
Атаки через debug-інтерфейси	Використання JTAG або SWD для отримання доступу	Блокування debug-інтерфейсів, secure configuration
Ін’єкція шкідливого коду	Впровадження стороннього програмного коду	Контроль цілісності, secure boot
Підміна оновлень firmware	Встановлення модифікованого оновлення	Перевірка цифрових підписів, secure update
Компрометація каналів зв’язку	Перехоплення або зміна переданих даних	VPN, TLS, потокове шифрування

Наведені приклади демонструють, що захист embedded-систем цивільного захисту потребує комплексного підходу, який поєднує криптографічний захист, контроль цілісності, механізми довіреного завантаження та засоби апаратної автентифікації компонентів системи. Це дозволяє підвищити стійкість систем до зовнішнього втручання та забезпечити надійність функціонування критичних компонентів цивільного захисту.

8.5.3. Захист телеметричних і SCADA-систем

SCADA-системи широко використовуються у системах енергозабезпечення, інженерній інфраструктурі, водопостачанні, системах

раннього виявлення загроз та комплексах моніторингу критичної інфраструктури. У системах цивільного захисту вони забезпечують збір, передавання та обробку телеметричних даних про стан об’єктів критичної інфраструктури, а також підтримують керування технологічними процесами у режимі реального часу. Значна частина критичних об’єктів цивільного захисту використовує SCADA-системи для моніторингу та керування інженерною інфраструктурою, системами енергозабезпечення та засобами спостереження [69]

Особливістю SCADA-систем є необхідність безперервного функціонування та використання значної кількості віддалених RTU (Remote Terminal Unit) і PLC (Programmable Logic Controller) пристроїв, які здійснюють збір даних та керування технологічними процесами. Така архітектура дозволяє забезпечувати моніторинг розподілених об'єктів, проте одночасно збільшує поверхню потенційних атак.

Компрометація firmware, конфігураційних параметрів або каналів передавання даних може призвести до підміни команд керування, формування хибної телеметричної інформації, порушення функціонування критичних об'єктів та відмови систем моніторингу. Особливо небезпечними є ситуації, коли помилкові команди або спотворені дані впливають на процес прийняття рішень у системах цивільного захисту.

Для підвищення рівня захищеності сучасних SCADA-архітектур використовуються механізми Secure Boot для RTU-пристроїв, шифрування телеметричних потоків, криптографічний захист каналів зв'язку із застосуванням VPN, TPM або HSM-модулів, апаратна автентифікація вузлів та контроль цілісності firmware. Особливу роль у забезпеченні стійкості таких систем відіграє захист телеметричних потоків і резервування джерел даних, що дозволяє підтримувати безперервність функціонування критичних сервісів навіть у випадку компрометації окремих компонентів (рис. 8.15).

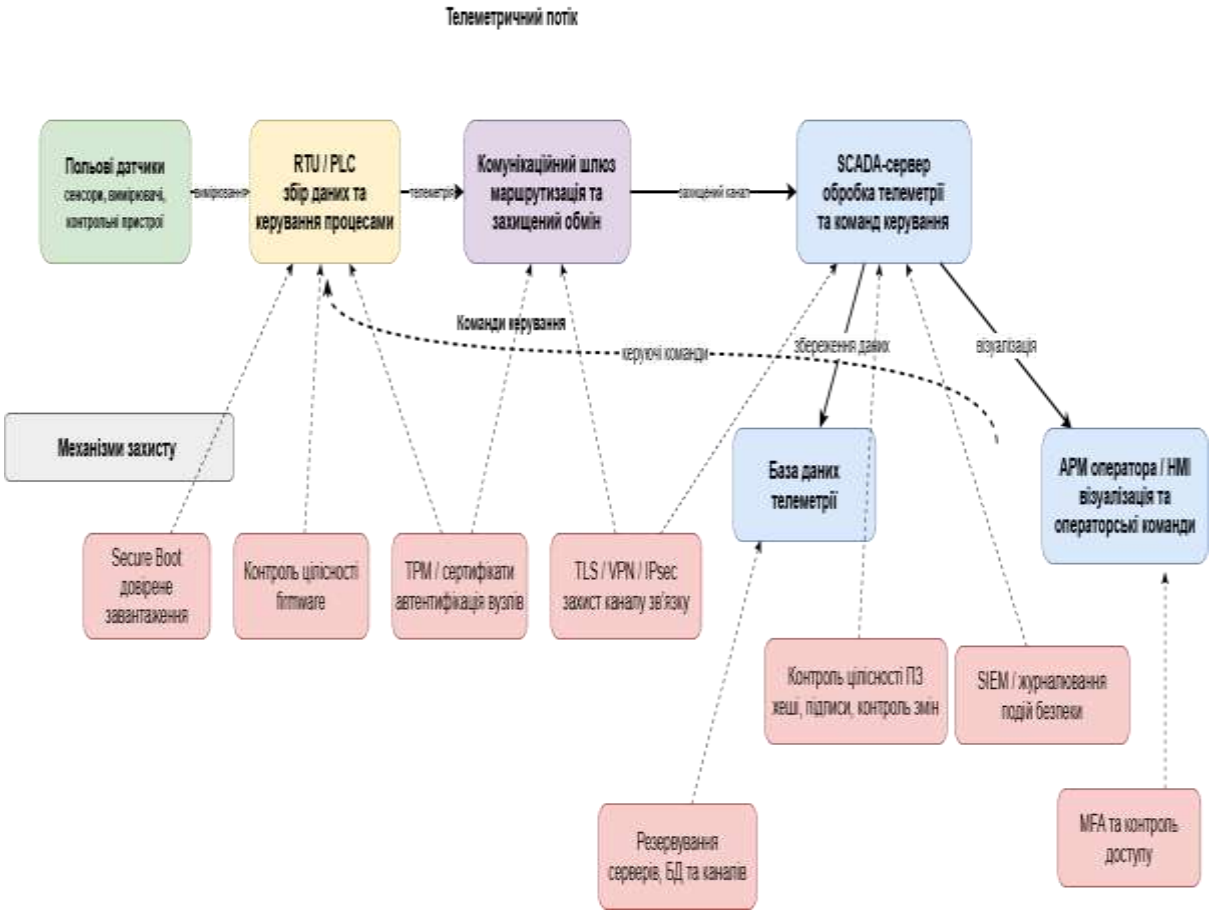


Рисунок 8.15 – Захист телеметричних потоків у SCADA-системі

SCADA-системи широко використовуються у критичній інфраструктурі та системах цивільного захисту для моніторингу, керування технологічними процесами та збору телеметричних даних. Компрометація окремих компонентів SCADA може призвести до порушення процесів керування, спотворення даних або втрати доступності критичних сервісів. У зв’язку з цим важливого значення набуває застосування спеціалізованих механізмів захисту програмних і апаратних компонентів таких систем. Приклади механізмів захисту SCADA-компонентів наведено у табл. 8.14.

Таблиця 8.14

Приклади механізмів захисту SCADA-компонентів

Компонент SCADA-системи	**Основні загрози**	**Механізми захисту**	**Основне призначення**

PLC (програмований логічний контролер)	Модифікація firmware, підміна команд керування	Secure Boot, encrypted firmware, контроль цілісності	Захист програмного коду та процесу завантаження
RTU (Remote Terminal Unit)	Компрометація конфігурації, підміна даних	TPM, цифрові сертифікати, автентифікація вузлів	Забезпечення автентичності компонентів
HMI (людино-машинний інтерфейс)	Несанкціонований доступ оператора	Багатофакторна автентифікація, контроль доступу	Захист операторських станцій
Сервер SCADA	Ін'єкція шкідливого коду, компрометація ПЗ	Контроль цілісності, SIEM, антивірусний захист	Захист серверних компонентів
Канали передавання даних	Перехоплення або модифікація інформації	TLS, VPN, потокове шифрування	Захист службового трафіку
База даних SCADA	Несанкціонована зміна або видалення інформації	Шифрування, контроль доступу, резервування	Захист критичних даних
Сенсорні вузли	Підміна вузлів або спотворення показників	Апаратна автентифікація, цифрові підписи	Підтвердження достовірності даних
Система оновлення	Підміна firmware або ПЗ	Secure Update, цифровий підпис оновлень	Запобігання встановленню змінених компонентів

Наведені приклади демонструють, що ефективний захист SCADA-систем потребує комплексного підходу, який поєднує криптографічний захист, контроль цілісності, автентифікацію компонентів та апаратний корінь довіри. Для систем цивільного захисту це особливо важливо через критичність безперервного функціонування та достовірності телеметричних даних.

8.5.4. Захист ситуаційних центрів та систем підтримки прийняття рішень

Ситуаційні центри є ключовими елементами сучасної системи цивільного захисту, оскільки забезпечують збір телеметричних даних, аналіз інформації, підтримку процесів прийняття рішень, координацію дій підрозділів та управління реагуванням на надзвичайні ситуації. Ефективність функціонування

таких центрів безпосередньо залежить від стійкості інформаційної інфраструктури та рівня захищеності програмного забезпечення [69, 70, 72, 76].

Особливу небезпеку для ситуаційних центрів становлять компрометація серверного firmware, атаки на SCADA-компоненти, підміна телеметричних даних, несанкціонований доступ до систем управління, а також атаки на системи візуалізації та аналітичні компоненти. Подібні загрози можуть призводити до спотворення оперативної інформації, порушення процесу прийняття рішень та зниження ефективності реагування на надзвичайні ситуації.

Для підвищення рівня захищеності ситуаційних центрів застосовуються механізми Secure Boot серверних платформ, TPM або HSM для захищеного зберігання криптографічних ключів, сегментація мереж, контроль цілісності системних компонентів, криптографічний захист телеметричних потоків та резервування критичних сервісів. Додатково у сучасних архітектурах використовуються механізми апаратної автентифікації мережевих вузлів і контролю довіреного стану компонентів інфраструктури, що дозволяє зменшити ризик компрометації критичних елементів системи (рис. 8.16).

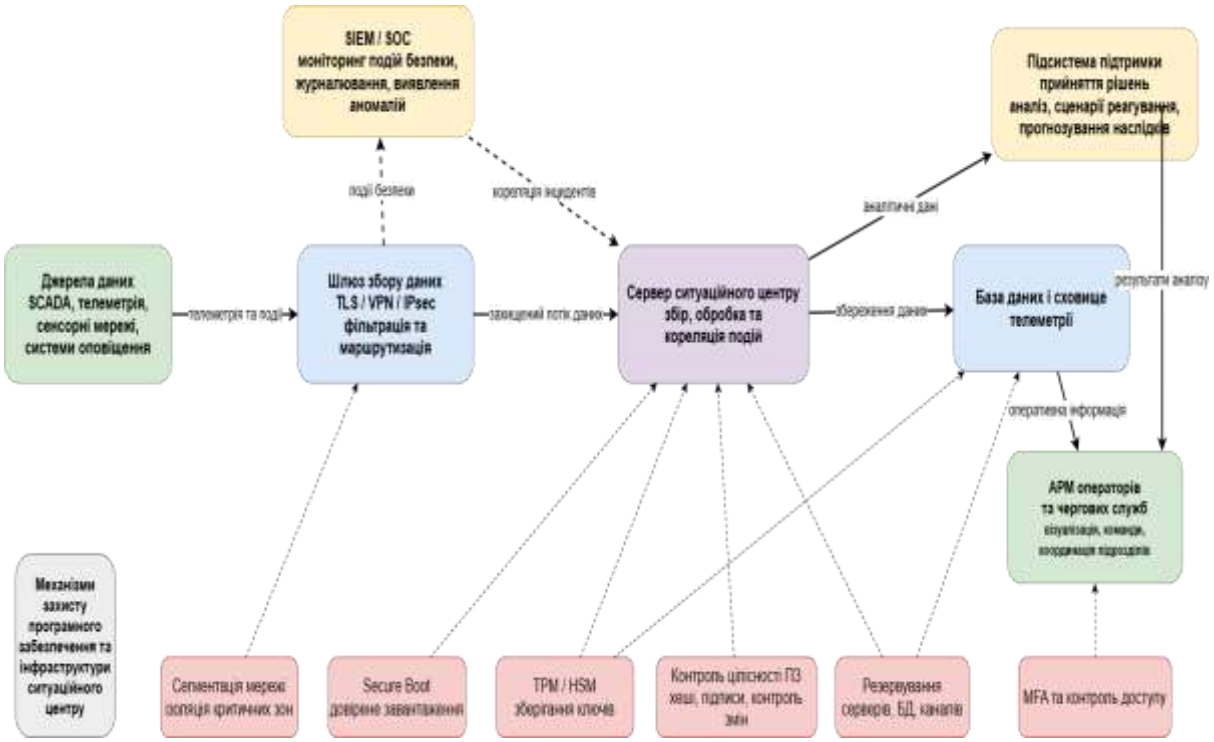


Рисунок 8.16 – Захищена архітектура ситуаційного центру цивільного захисту

Ситуаційні центри є ключовими елементами систем цивільного захисту, оскільки забезпечують збір, обробку та аналіз інформації про стан об’єктів, координацію дій підрозділів і підтримку процесу прийняття рішень під час надзвичайних ситуацій. Компрометація програмного забезпечення, серверної інфраструктури або каналів передачі даних у таких системах може призводити до втрати достовірності інформації та порушення керованості процесами реагування. Основні механізми захисту ситуаційних центрів наведено у табл. 8.15.

Таблиця 8.15

Механізми захисту ситуаційних центрів

Механізм захисту	**Призначення**	**Приклад реалізації**
Secure Boot	Перевірка цілісності компонентів під час запуску системи	Контроль цифрових підписів firmware і bootloader
TPM (Trusted Platform Module)	Захищене зберігання ключів і формування апаратного кореня довіри	Захист криптографічних ключів серверів
Контроль цілісності ПЗ	Виявлення несанкціонованої модифікації програмних компонентів	Контроль хеш-значень системних файлів
Багатофакторна автентифікація	Захист доступу до критичних сервісів	Пароль + токен + біометрія
Криптографічний захист каналів зв’язку	Захист службового обміну інформацією	TLS, VPN, IPsec
Сегментація мережі	Ізоляція критичних компонентів інфраструктури	Поділ мережі на окремі захищені сегменти
Системи моніторингу подій безпеки	Виявлення інцидентів і аномальної активності	SIEM-системи
Резервування компонентів	Забезпечення безперервності функціонування	Резервні сервери й канали зв’язку

Trusted Execution Environment	Захищене виконання критичних операцій	Ізольоване виконання криптографічних функцій
Захищене оновлення ПЗ	Запобігання встановленню модифікованих компонентів	Перевірка цифрових підписів оновлень

Наведені механізми демонструють, що захист ситуаційних центрів повинен базуватись на комплексному підході, який поєднує апаратний корінь довіри, криптографічний захист, контроль цілісності програмного забезпечення та засоби моніторингу безпеки. Для систем цивільного захисту це є необхідною умовою забезпечення достовірності інформації, безперервності роботи та стійкості до кібератак.

8.5.5. Захист мобільних комплексів управління та реагування

У сучасних системах цивільного захисту використовуються мобільні пункти управління та польові комплекси реагування, які забезпечують координацію дій підрозділів у зоні надзвичайної ситуації, організацію службового зв’язку, оперативний обмін інформацією та підтримку процесу прийняття рішень. До таких систем належать мобільні пункти управління ДСНС, мобільні ситуаційні центри реагування, мобільні комплекси радіаційного та хімічного контролю, мобільні вузли захищеного зв’язку та мобільні телеметричні комплекси моніторингу.

Архітектура подібних комплексів зазвичай поєднує серверні вузли, телекомунікаційне обладнання, супутникові канали зв’язку, embedded-системи, сенсорні модулі та засоби відеомоніторингу. Прикладом таких рішень є сучасні системи екстреного зв’язку та реагування (emergency response communication systems), які інтегрують технології SDR, mesh-мережі, супутниковий зв’язок і польові вузли управління для забезпечення безперервного функціонування під час надзвичайних ситуацій (рис. 8.17).

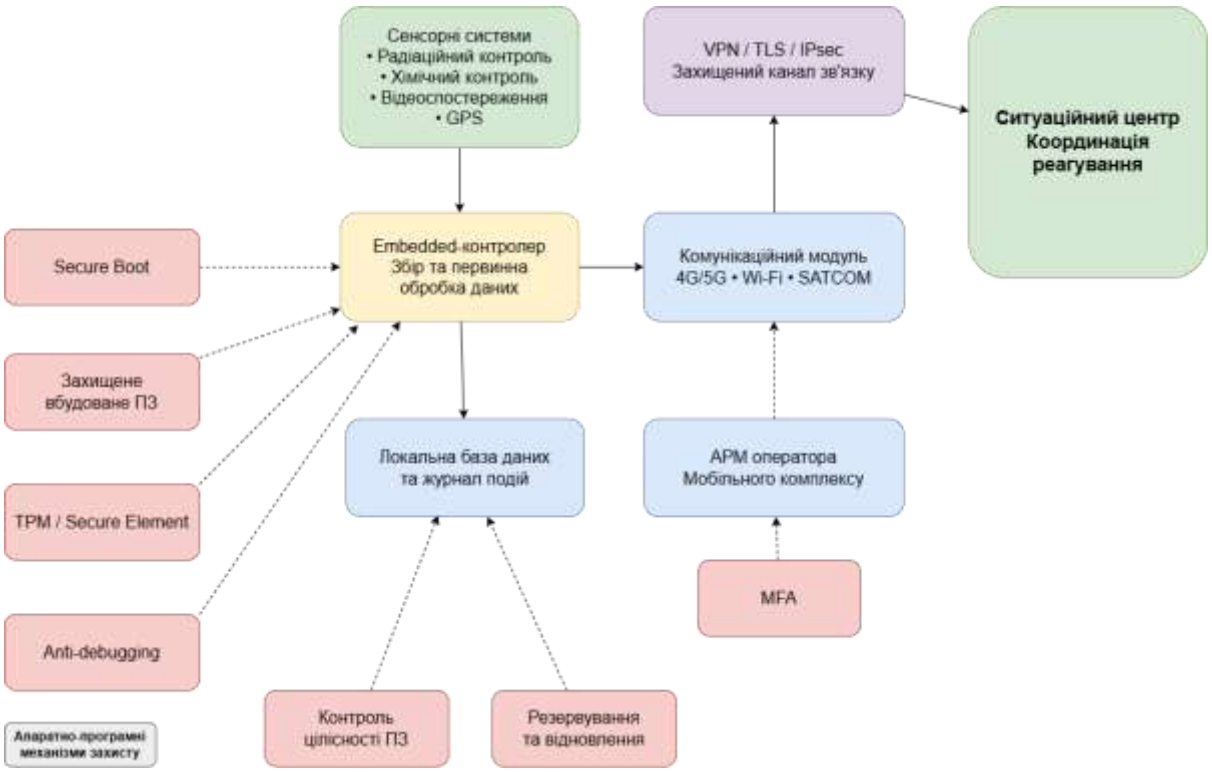


Рисунок 8.17 – Приклад архітектури мобільного комплексу реагування з апаратно-програмним захистом

Особливості функціонування мобільних комплексів створюють додаткові вимоги до безпеки програмного забезпечення, оскільки такі системи працюють у розподіленому середовищі та можуть піддаватися як мережевим атакам, так і фізичному втручанню. У зв’язку з цим особливого значення набувають захист firmware польових контролерів, реалізація механізмів Secure Boot у мобільних вузлах, захист каналів передавання телеметричної інформації, перевірка автентичності обладнання та забезпечення стійкості систем до фізичного доступу.

Для реалізації зазначених вимог використовуються механізми апаратно-програмного захисту, зокрема TPM, encrypted firmware, захищені VPN-з’єднання, hardware root of trust, механізми anti-tamper та багаторівневий контроль доступу. Їх комплексне використання дозволяє підвищити стійкість мобільних систем цивільного захисту до несанкціонованого втручання та забезпечити безпечне функціонування критичних програмних компонентів.

Мобільні комплекси цивільного захисту використовуються для організації управління, координації дій підрозділів та оперативного обміну інформацією під час реагування на надзвичайні ситуації [76]. Такі системи функціонують в умовах підвищених ризиків, пов’язаних із використанням бездротових каналів зв’язку, мобільністю обладнання, можливістю фізичного доступу та необхідністю забезпечення безперервної роботи. У зв’язку з цим важливого значення набуває застосування комплексних апаратно-програмних механізмів захисту. Основні засоби захисту мобільних комплексів управління та реагування цивільного захисту наведено у табл. 8.16.

Таблиця 8.16

Засоби захисту мобільних комплексів управління та реагування

Засіб захисту	**Призначення**	**Приклад застосування**
Secure Boot	Перевірка цілісності компонентів під час запуску системи	Блокування запуску модифікованого firmware
TPM (Trusted Platform Module)	Захищене зберігання криптографічних ключів	Захист службових ключів і сертифікатів
VPN та TLS	Захист службового обміну інформацією	Шифрування каналів зв’язку між мобільними вузлами
Контроль цілісності ПЗ	Виявлення змін програмних компонентів	Перевірка хеш-значень системних модулів
Багатофакторна автентифікація	Захист доступу до службових ресурсів	Пароль, токен та біометричні параметри
Encrypted firmware	Захист firmware від копіювання й модифікації	Захищене зберігання програмного коду пристроїв
Anti-tamper механізми	Виявлення несанкціонованого втручання в обладнання	Контроль відкриття корпусу або зміни конфігурації
Trusted Execution Environment	Ізольоване виконання критичних операцій	Захист криптографічних функцій
Резервування каналів зв’язку	Забезпечення безперервної передачі інформації	Автоматичне перемикання між каналами

Наведені засоби демонструють, що захист мобільних комплексів цивільного захисту потребує поєднання криптографічних, програмних та апаратних механізмів безпеки. Такий підхід дозволяє забезпечити конфіденційність, цілісність і доступність інформації, а також підвищити стійкість систем до кібератак і несанкціонованого втручання в умовах реального функціонування.

8.5.6. Використання скремблерів у системах службового зв'язку та телеметрії

Потокові механізми та скремблери можуть використовуватися для захисту службового голосового зв’язку, телеметричних потоків та оперативної інформації, яка передається між компонентами систем цивільного захисту.

Основною перевагою таких механізмів є можливість роботи в режимі реального часу за мінімальних вимог до апаратних ресурсів.

Скремблери можуть застосовуватися у службовому радіозв’язку, мобільних пунктах управління, телеметричних каналах, системах дистанційного моніторингу, embedded-пристроях збору даних. (рис.8.18, рис. 8.19).



Рисунок 8.18 – Апаратний скремблер для захищеного голосового зв’язку PrivacyLiteVoice PLV2.0 [80]

На рисунку наведено приклад апаратного скремблера голосового зв’язку, призначеного для захисту аналогових радіоканалів від несанкціонованого прослуховування. Подібні пристрої можуть застосовуватись у системах службового зв’язку, мобільних пунктах управління та польових комплексах реагування цивільного захисту, зокрема у програмно-апаратному комплексі PrivacyLite Voice



Рисунок 8.19 – Захищені радіосистеми для служб екстреного реагування [81]

Сучасні цифрові радіосистеми поєднують потокові механізми перетворення даних із криптографічними алгоритмами та засобами автентифікації користувачів, забезпечуючи захищений службовий зв’язок підрозділів цивільного захисту. У професійних системах для підрозділів екстреного реагування широко використовуються цифрові стандарти TETRA та APCO P25, які поєднують механізми захисту голосового трафіку, автентифікації та криптографічного захисту для забезпечення конфіденційного службового зв’язку [71].

Розглянуті приклади демонструють, що ефективний захист програмного забезпечення у системах цивільного захисту неможливо забезпечити за допомогою одного механізму безпеки. Найбільш доцільним є комплексний підхід, який поєднує механізми безпечного завантаження, контролю цілісності,

захисту криптографічних ключів, захищеного оновлення програмного забезпечення, апаратної автентифікації та потокових механізмів захисту даних. Це дозволяє підвищити стійкість систем до кібератак, забезпечити достовірність інформації та підтримувати безперервність функціонування критичних сервісів цивільного захисту.

ПИТАННЯ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ ЗА ТЕМОЮ 8

1. Які основні переваги апаратно-програмного захисту програмного забезпечення порівняно з виключно програмними механізмами захисту?
2. Для чого призначений модуль TPM (Trusted Platform Module) та які функції він виконує у системах цивільного захисту?
3. У чому полягає принцип роботи механізму Secure Boot та від яких загроз він захищає програмне забезпечення?
4. Які особливості потокових механізмів захисту відрізняють їх від блокових криптографічних механізмів?
5. Що таке скремблер і які задачі він вирішує у системах передавання інформації та захисту програмного забезпечення?
6. Поясніть принцип роботи LFSR-механізму (Linear Feedback Shift Register) та його роль у формуванні псевдовипадкових послідовностей.
7. У чому полягає різниця між синхронними та самосинхронізуючими скремблерами?
8. Які загрози можуть виникати під час програмної реалізації потокових механізмів захисту та які засоби використовуються для їх нейтралізації?
9. Які апаратно-програмні механізми захисту доцільно використовувати для забезпечення безпеки ситуаційних центрів цивільного захисту?
10. Наведіть приклади використання апаратно-програмних механізмів захисту у системах оповіщення населення, телеметричних комплексах або SCADA-системах цивільного захисту та поясніть їх призначення.

11. Поясніть, чому для критичних систем цивільного захисту найбільш ефективним є комплексне поєднання механізмів TPM, Secure Boot, захисту firmware, потокових механізмів та криптографічного захисту каналів зв'язку.

ПРАКТИЧНІ ЗАВДАННЯ ДО ТЕМИ 8

Практичне завдання 1. Проєктування лінійного регістра зсуву зі зворотними зв'язками для захисту каналу передачі даних ЦЗ-системи

Постановка завдання

На основі підрозділів 8.2 та 8.3 посібника спроєктуйте скремблер на базі лінійного регістра зсуву зі зворотними зв'язками (LFSR) для захисту цифрового потоку даних компонента цивільного захисту.

Оберіть один із варіантів захищуваного каналу:

а) канал передачі оперативних звітів між мобільними підрозділами ЦЗ і центром координації (розрядність регістра n = 4, примітивний поліном: $x^4 + x + 1$);

б) канал синхронізації системи оповіщення населення (розрядність регістра n = 5, примітивний поліном: $x^5 + x^2 + 1$);

в) канал передавання телеметричних даних між постами дозиметричного контролю та ситуаційним центром (розрядність регістра n = 6, примітивний поліном: $x^6 + x + 1$).

Для обраного варіанта:

1) За двійковим поданням обраного примітивного полінома (без старшого розряду) визначте схему зворотних зв'язків регістра; побудуйте ASCII-схему LFSR із позначенням вузлів зворотного зв'язку та операції XOR; поясніть, чому саме примітивний поліном забезпечує максимальну довжину послідовності $2^n - 1$.

2) Задайте початковий стан регістра (довільне ненульове значення); пропрацюйте не менше 8 тактів і зафіксуйте стан регістра та біт, що генерується ПСП, у табличній формі; переконайтеся, що жоден стан не повторюється впродовж перших $2^n - 1$ тактів.

3) Скремблюйте восьмибітовий байт даних (оберіть довільне значення, відмінне від 0x00 і 0xFF) покроково: застосуйте операцію XOR між кожним бітом вхідного байта і відповідним бітом ПСП; наведіть таблицю «вхідний біт – біт ПСП – вихідний біт» і вкажіть отриманий скремблований байт у двійковому та шістнадцятковому форматах.

4) Оцініть криптостійкість спроєктованого скремблера: вкажіть довжину ПСП, а також два відомих обмеження LFSR-скремблерів (наприклад, лінійна складність, атака Берлекемпа–Мессі); запропонуйте один спосіб підвищення криптостійкості з посиланням на матеріал теми.

5) Реалізуйте функції scramble() та descramble() мовою C, C++ або Python. Продемонструйте процес скремблювання та відновлення початкового байта даних, підтвердивши коректність роботи реалізації.

6) Запропонуйте один програмний та один апаратно-програмний механізм підвищення захищеності реалізації скремблера (наприклад, обфускація програмного коду, anti-debugging, TPM, Secure Boot або захищене зберігання ключів) та поясніть доцільність їх використання у системах цивільного захисту.

Приклад розв'язку

Обраний варіант: а) канал передачі оперативних звітів, n = 4, поліном $x^4 + x + 1$.

Алгоритм формування біта зворотного зв’язку визначається вибраним примітивним поліномом. Для полінома $x^4 + x + 1$ новий біт формується як результат операції XOR між вмістом розрядів r_4 та r_1 : $new_bit = r_4 \oplus r_1$.

Згенерований біт подається на вхід регістра, після чого виконується зсув усіх розрядів на одну позицію. Вихідний біт використовується для формування псевдовипадкової послідовності.

Крок 1. Схема зворотних зв'язків LFSR

Двійкове подання полінома $x^4 + x + 1 = 1 \cdot x^4 + 0 \cdot x^3 + 0 \cdot x^2 + 1 \cdot x + 1 \rightarrow 10011$.

Після відкидання старшого розряду: 0011.

Зворотні зв'язки наявні у розрядах r_1 та r_0 (позиції з одиницями в поданні 0011).

Алгоритм генерації нового біта: $new_bit = r_4 \oplus r_1$.

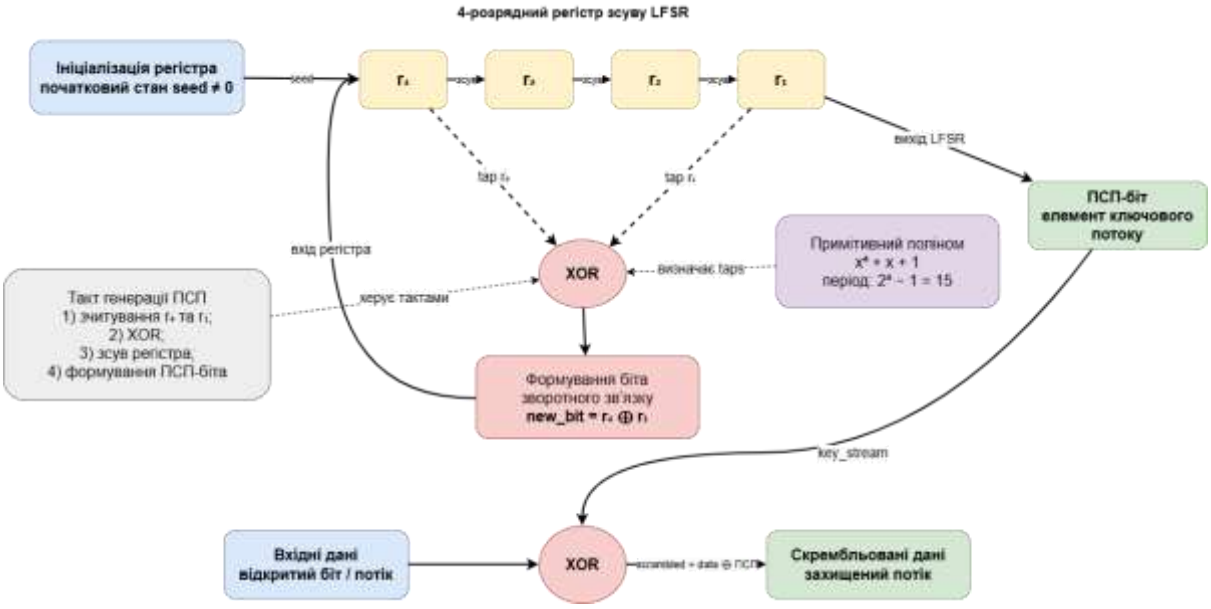


Рисунок 8.20 – Принцип роботи програмного LFSR-скремблера

Пояснення. Примітивний поліном над полем GF(2) забезпечує максимальний період роботи регістра зсуву зі зворотним зв'язком (LFSR), тому відповідний регістр проходить через всі 2^n-1 ненульових станів перед поверненням до початкового. Для чотирирозрядного регістра, наведеного на рис. 8.20, максимальна довжина псевдовипадкової послідовності становить $2^4-1=15$ бітів. Сформована LFSR послідовність використовується як ключовий потік (ПСП-біти), який за допомогою операції XOR виконує скремблювання вхідних даних. Це забезпечує максимальну довжину псевдовипадкової послідовності, добрі статистичні характеристики сформованого потоку та ускладнює прогнозування вихідних даних.

Крок 2. Трасування станів регістра

Початковий стан: $r_4r_3r_2r_1 = 1001$ (довільне ненульове).

Таблиця 8.17

Трасування станів регістра:

Такт	Стан $r_4r_3r_2r_1$	new_bit = $r_4 \oplus r_1$ |	ПСП-біт (r_1)
0	1 0 0 1	$1 \oplus 1 = 0$	1
1	0 1 0 0	$0 \oplus 0 = 0$	0
2	0 0 1 0	$0 \oplus 0 = 0$	0
3	0 0 0 1	$0 \oplus 1 = 1$	1
4	1 0 0 0	$1 \oplus 0 = 1$	0
5	1 1 0 0	$1 \oplus 0 = 1$	0
6	1 1 1 0	$1 \oplus 0 = 1$	0
7	1 1 1 1	$1 \oplus 1 = 0$	1
8	0 1 1 1	$0 \oplus 1 = 1$	1

Усі 8 зафіксованих станів є унікальними. Повний цикл становить $2^4 - 1 = 15$ тактів – жоден стан не повторюється раніше 15-го кроку.

Крок 3. Скремблювання байта даних

Вхідний байт: 0xA5 = 10100101_2. ПСП-послідовність (біти з ПСП-стовпця кроку 2, такти 0–7): 1, 0, 0, 1, 0, 0, 0, 1.

Таблиця 8.18

Таблиця скремблювання

Позиція	**b_7**	**b_6**	**b_5**	**b_4**	**b_3**	**b_2**	**b_1**	**b_0**	**Позиція**
Вхідний біт	1	0	1	0	0	1	0	1	Вхідний біт
ПСП-біт	1	0	0	1	0	0	0	1	ПСП-біт
Вихідний біт (XOR)	0	0	1	1	0	1	0	0	Вихідний біт (XOR)

Примітка: (біти обробляються від старшого до молодшого)

Скремблований байт: 00110100_2 = 0x34.

Перевірка дескремблювання: застосувавши ту саму ПСП-послідовність повторно до 0x34, отримаємо 0xA5 – вихідне значення відновлено. Функціональна оборотність підтверджена.

Крок 4. Оцінка криптостійкості

Довжина ПСП: $2^4 - 1 = 15$ бітів – надзвичайно коротка для криптографічних застосувань.

Таблиця 8.19

Обмеження LFSR:

Обмеження	Пояснення
Лінійна складність	Знаючи 2n = 8 послідовних вихідних бітів, алгоритм Берлекемпа–Мессі однозначно відновлює структуру регістра (поліном і початковий стан) – ключовий простір зловмисника зводиться до перебору $2^n - 1 = 15$ початкових станів
Детермінованість за початковим станом	При відомому початковому стані вся ПСП відтворюється повністю без будь-яких додаткових відомостей; відсутність справжньої випадковості робить LFSR непридатним для самостійного криптографічного використання

Спосіб підвищення захищеності: використання механізмів захищеного зберігання ключів і початкових параметрів скремблера (наприклад, TPM або Secure Element), а також їх передавання захищеними криптографічними каналами. Такий підхід ускладнює компрометацію початкового стану регістра та відповідає концепції комплексного апаратно-програмного захисту, розглянутій у підрозділі 8.4 посібника. Це усуває проблему детермінованості: навіть при перехопленні зашифрованого потоку зловмисник не може відновити початковий стан без знання приватного ключа.

Практичне завдання 2. Реалізація та дослідження самосинхронізуючого скремблера для захисту потоку даних підрозділу ЦЗ

Постановка завдання

На основі підрозділу 8.2.3 та 8.3 посібника реалізуйте і дослідіть самосинхронізуючий (АД) скремблер для захисту потоку оперативних даних компонента цивільного захисту.

Оберіть один із варіантів вхідних даних для захищеної передачі:

а) рядок виклику аварійно-рятувальної служби у форматі «[ПРІОРИТЕТ]:[АДРЕСА], [ТИП НС]» (наприклад: «HIGH: вул. Шевченка 15, пожежа»);

б) повідомлення про статус ресурсів підрозділу у форматі «[ПІДРОЗДІЛ]:[РЕСУРС]=[КІЛЬКІСТЬ]» (наприклад: «ПЧ-7:АЦ=3,ЦНС=1»);

в) короткий звіт дозиметричного поста у форматі «[ПОСТ]:[РІВЕНЬ мкЗв/год]:[СТАТУС]» (наприклад: «ДП-12:0.23:НОРМА»).

Для обраного варіанта:

1) Реалізуйте або доопрацюйте алгоритм скремблювання мовою C++ або Python: клас (або функція) має приймати вхідний рядок, значення ключа (key) і початковий стан регістра; виконувати побітове скремблювання кожного символу операцією XOR із зсувом регістра; повертати результат у шістнадцятковому форматі. Наведіть ключові фрагменти коду з поясненнями.

2) Застосуйте реалізований скремблер до обраного рядка; зафіксуйте вхідний рядок, значення key та shift_register, і отриману шістнадцяткову послідовність у таблиці; поясніть, чому виведення здійснюється в шістнадцятковому форматі, а не як ASCII-рядок.

3) Виконайте дескремблювання отриманої послідовності; підтвердіть, що відновлений рядок повністю збігається з вхідним; продемонструйте у таблиці різницю між алгоритмами scramble() і descramble() з погляду оновлення регістра.

4) Проаналізуйте явище лавинного ефекту (розмноження помилок): змініть один байт у зашифрованому потоці (наприклад, третій байт) та виконайте дескремблювання; зафіксуйте у таблиці, скільки символів результату виявились спотвореними та чим цей ефект загрожує надійності каналу ЦЗ-системи.

5) Запропонуйте спосіб підвищення захищеності реалізації самосинхронізуючого скремблера шляхом використання одного програмного та одного апаратно-програмного механізму захисту (наприклад, обфускація, anti-

debugging, TPM, Secure Boot або захищене зберігання ключів). Поясніть доцільність їх застосування для систем цивільного захисту.

Приклад розв'язку

Обраний варіант: а) виклик «HIGH: вул. Шевченка 15, пожежа».

Крок 1. Ключові фрагменти реалізації

Використано клас self_synchronizing_scrambler, реалізований на основі принципів самосинхронізуючих скремблерів, розглянутих у підрозділах 8.2.3 та 8.3.3 посібника.

key = 5 (00000101_2), shift_register = 17 (00010001_2).

Принцип scramble():

for кожного біта $b_7..b_0$ вхідного символу:

```
      new_bit = b ⊕ (XOR усіх бітів: key[j] AND shift_register[j])
           output[i] = new_bit
           key = (key >> 1) | (new_bit << 7)   // регістр оновлюється
вихідним бітом
```

Принцип descramble():

for кожного біта $b_7..b_0$ зашифрованого символу:

```
      old_bit = b ⊕ (XOR усіх бітів: key[j] AND shift_register[j])
           output[i] = old_bit
           key = (key >> 1) | (b << 7)         // регістр оновлюється вхідним
(зашифрованим) бітом
```

Ключова різниця: при скремблюванні регістр оновлюється вихідним бітом; при дескремблюванні – вхідним (зашифрованим) бітом. Саме це забезпечує самосинхронізацію.

Крок 2. Скремблювання рядка

Вхідний рядок: "HIGH: пожежа" (скорочено для прикладу; повний рядок обробляється аналогічно) key = 5, shift_register = 17.

Таблиця 8.20

Скремлювання символів:

Символ	ASCII (hex)	Скремблований (hex)
H	0x48	0xfc
I	0x49	0x5a
G	0x47	0x80
H	0x48	0xef
:	0x3a	0x75
(пробіл	| 0x20	0x43

Пояснення виведення в hex-форматі: операція XOR може породжувати байти, що відповідають непечатним або керуючим ASCII-символам (значення < 0x20 та > 0x7E), які неможливо коректно відобразити або передати як текст. Шістнадцятковий формат гарантує однозначне відображення будь-якого байта. Саме тому більшість програмних реалізацій скремблерів та дескремблерів використовують шістнадцяткове або двійкове подання результатів перетворення.

Крок 3. Дескремблювання та порівняння алгоритмів

Відновлений рядок: "HIGH: пожежа" – повністю збігається з вхідним.

Таблиця 8.21

Порівняння алгоритмів з погляду оновлення регістра:

Крок	**scramble()**	**descramble()**
Обчислення	new_bit = in_bit ⊕ f(key, reg)	old_bit = in_bit ⊕ f(key, reg)
Оновлення регістра	key ← (key >> 1) | (new_bit << 7) | (зашифрований біт)	key ← (key >> 1) | (in_bit << 7) (прийнятий зашифрований біт)
Результат	вихідний зашифрований біт	відновлений відкритий біт

Обидва пристрої (скремблер і дескремблер) синхронізують свій регістр через зашифровані біти, що передаються каналом. Завдяки цьому при втраті синхронізації вона відновлюється автоматично після n = 8 прийнятих бітів без додаткових керуючих повідомлень. Ця властивість є основною перевагою самосинхронізуючих скремблерів у каналах зв'язку з підвищеною ймовірністю виникнення помилок або короткочасних втрат даних.

Крок 4. Лавинний ефект (розмноження помилок)

Умова експерименту: третій байт зашифрованого потоку (0x80) замінено на 0xFF.

Таблиця 8.22

Результат дескремблювання спотвореного потоку:

Позиція	Очікуваний символ	Отриманий символ	Спотворено?
1	H	H	Ні
2	I	I	Ні
3	G	X (довільний)	Так
4	H	? (довільний)	Так
5	:	: (відновлено)	Ні
6	(пробіл)	(пробіл)	Ні

Спостереження: помилка в одному зашифрованому байті спотворює рівно два суміжних символи відкритого тексту (той, що відповідає пошкодженому байту, і наступний). Після цього регістр самосинхронізується і подальші символи відновлюються правильно.

Загроза для ЦЗ-системи: у каналі передачі оперативних викликів навіть спотворення двох символів у повідомленні типу «HIGH: вул. Шевченка 15, пожежа» може призвести до неправильної адреси або типу надзвичайної ситуації. Для критичних каналів ЦЗ необхідно поєднувати скремблювання з кодами виявлення помилок (CRC або HMAC), щоб пошкоджені пакети відкидались на рівні протоколу ще до декодування.

Практичне завдання 3. Порівняльний аналіз засобів захисту мовної інформації та їх застосування у підрозділах цивільного захисту

Постановка завдання

На основі підрозділів 8.1 та 8.2 посібника проведіть порівняльний аналіз апаратних засобів захисту мовних комунікацій і визначте найбільш доцільний засіб для застосування у конкретному сценарії підрозділу цивільного захисту.

Оберіть один із сценаріїв оперативної комунікації:

а) захищений зв'язок між черговим диспетчером обласного центру управління та рухомими групами реагування під час ліквідації хімічної аварії;

б) захищена передача даних між стаціонарним постом радіаційного та хімічного контролю і мобільним штабом під час навчань з ядерної безпеки;

в) захист внутрішнього телефонного зв'язку оперативного штабу пожежно-рятувального підрозділу від прослуховування через підключення до АТС.

Для обраного сценарію:

1) Визначте тип і характеристики каналу зв'язку (аналоговий / цифровий, дротовий / бездротовий, тривалість сесій, вимоги до мобільності); за матеріалом підрозділу 8.1 вкажіть, який з відомих методів перехоплення телефонних переговорів є найбільш реалістичною загрозою для обраного сценарію та обґрунтуйте вибір.

2) Складіть порівняльну таблицю трьох типів засобів захисту мовної інформації (аналоговий скремблер, цифровий скремблер (TETRA/P25), криптофон) за такими критеріями: принцип захисту, тип каналу, мобільність, рівень криптостійкості, стійкість до перехоплення цифровим аналізатором, придатність для обраного сценарію.

3) Обґрунтуйте вибір конкретного типу пристрою для обраного сценарію; запишіть дві технічних вимоги, яким повинен відповідати обраний пристрій у контексті підрозділів ЦЗ (наприклад, щодо ключового обміну, синхронізації, резервування живлення).

4) Проаналізуйте роль апаратно-програмних механізмів захисту (TPM, Secure Element, Secure Boot або захищене зберігання ключів) у забезпеченні

безпеки обраного засобу зв'язку та поясніть, які загрози вони дозволяють нейтралізувати.

5) Спираючись на матеріал підрозділу 8.1, наведіть один історичний приклад розвитку засобів захисту мовної інформації (системи А-3 або SIGSALY), проаналізуйте їх переваги та недоліки й поясніть, які сучасні технології усунули виявлені обмеження.

Приклад розв'язку

Обраний сценарій: а) захищений зв'язок диспетчера з рухомими групами під час ліквідації хімічної аварії.

Крок 1. Характеристика каналу та актуальна загроза

Характеристики каналу:

– Тип: бездротовий радіоканал (УКХ/КХ-діапазон або стандарт TETRA);

– Тривалість сесій: від 30 секунд до кількох годин безперервно;

– Мобільність: висока – абоненти переміщуються в зоні аварії;

– Вимоги до надійності: критичні – обрив зв'язку неприпустимий під час активної фази ліквідації.

Найбільш реалістична загроза – безконтактне підключення до каналу (метод 3 із підрозділу 8.1 посібника): зловмисник або розвідувальний засіб перехоплює радіосигнал пасивно, без фізичного доступу до інфраструктури. В умовах хімічної аварії отримані в такий спосіб відомості про маршрути евакуації та дії рятувальних підрозділів можуть становити значну оперативну цінність для зловмисника. Реалізація не потребує доступу до АТС або фізичного наближення до проводки.

Крок 2. Порівняльна таблиця пристроїв

Таблиця 8.23

Порівняння таблиця пристроїв:

Критерій	Аналоговий скремблер	Цифровий скремблер (TETRA/P25)	Криптофон

Принцип закриття сигналу	Частотна інверсія або перестановка смуг спектру (підрозділ 8.1)	Цифрове шифрування мовного сигналу (AES-256, Twofish); ключ 256 біт (підрозділ 8.2)	Накладання згенерованого шуму на аналоговий сигнал у лінію зв'язку
Тип каналу	Аналоговий	Цифровий радіоканал	Цифровий IP-канал
Мобільність	Середня (зовнішній пристрій до радіостанції)	Висока (вбудовано у смартфон)	Низька (стаціонарний пристрій)
Рівень криптостійкості	Низький: метод частотної інверсії зламано у 1941 р.	Високий: AES-256 з обміном ключами Діффі–Хеллман (4096-бітний ключ)	Мінімальний: не є засобом шифрування; лише маскує сигнал від прямого прослуховування
Стійкість до цифрового аналізатора спектру	Вкрай низька: сучасний SDR-приймач відновлює відкритий сигнал за секунди	Висока: для дешифрування потрібен 256-бітний ключ	Відсутня: фільтрація шуму сучасними засобами обробки сигналів є тривіальним завданням
Придатність для сценарію а)	Непридатний (бездротовий канал, криптостійкість недостатня	Придатний (бездротовий/TETRA, висока | мобільність, сильний захист)	Непридатний (лише стаціонарний | дротовий зв'язок)

Крок 3. Обґрунтування вибору та технічні вимоги

Обраний тип пристрою: криптофон (наприклад, клас пристроїв типу GSMK CryptoPhone або аналогічні сертифіковані засоби).

Обґрунтування: єдиний із трьох типів, що поєднує високу мобільність, стійкість до пасивного перехоплення в бездротовому каналі та достатній рівень криптостійкості (AES-256). Для рухомих груп реагування вирішальним є форм-фактор смартфона без зовнішніх модулів, що можуть відмовити в умовах хімічного забруднення.

Технічні вимоги для застосування у підрозділах ЦЗ:

1. Захищений обмін ключами без попередньої інфраструктури (Out-of-Band або ZRTP): в умовах аварії неможливо гарантувати доступність центрального сервера ключів – пристрій має підтримувати автономний обмін сеансовим ключем безпосередньо між абонентами, без залежності від PKI-сервера (аналогічно протоколу Діффі–Хеллмана, зазначеному у підрозділі 8.2).

2. Захист від розряду та робота у несприятливих умовах: пристрій має забезпечувати не менш ніж 8 годин роботи від батареї в режимі передачі та

відповідати стандарту захисту не нижче IP54 – обов'язкові умови для роботи особового складу в зоні хімічної аварії.

3. Захищене зберігання криптографічних ключів. Для запобігання компрометації службового зв'язку ключовий матеріал повинен зберігатися у TPM, Secure Element або іншому апаратному модулі безпеки.

4. Контроль цілісності програмного забезпечення засобу зв'язку. Для захисту від модифікації firmware доцільно використовувати механізми Secure Boot та перевірку цифрових підписів оновлень.

Крок 4. Історичний приклад провалу та аналіз

Приклад: система А-3 (США, 1930-ті роки).

Суть провалу: система А-3 закривала телефонні переговори методом поділу мовного спектра на 5 частотних смуг із наступною їх перестановкою та частотною інверсією за ключем, що змінювався кожні 20 секунд. У 1941 році німецька дешифрувальна служба (B-Dienst) успішно відновила алгоритм перестановки і отримала доступ до переговорів вищого військового командування США і Великобританії, включно з розмовами Рузвельта та Черчілля.

Причина компрометації: перестановка частотних смуг є лінійним аналоговим перетворенням із вкрай малим простором ключів (число можливих перестановок 5 смуг – 5! = 120, і зміна ключа кожні 20 секунд не компенсувала алгоритмічну слабкість). Збір достатньої кількості шифртекстів дозволяв аналітику відновити тип перестановки методом частотного аналізу спектра.

Технічне рішення, що усунуло вразливість: апаратура SIGSALY (1943 р., підрозділ 8.1) замінила аналогове перетворення спектра на цифрове шифрування мовного сигналу, попередньо оцифрованого вокодером. Ключова послідовність записувалась на одноразові вінілові шифродиски (принцип одноразового блокнота): кожен диск використовувався рівно один раз і знищувався. Це повністю усунуло можливість кріптоаналітичного відновлення ключа – навіть

при перехопленні всього трафіку, оскільки ключ не мав статистичних закономірностей.

У сучасних системах службового зв'язку цивільного захисту додатково використовуються апаратно-програмні механізми захисту, які забезпечують захищене зберігання ключів, контроль цілісності firmware та автентифікацію компонентів системи зв'язку.

Зв'язок із сучасністю: той самий принцип (замінити аналогове перетворення стійким цифровим шифруванням із захищеним обміном ключами) реалізовано в сучасних криптофонах через алгоритм AES-256 у поєднанні з протоколом Діффі-Хеллмана – саме тому криптофон є єдиним обґрунтованим вибором для захисту мовного каналу підрозділів ЦЗ у наведеному сценарії.

ЛІТЕРАТУРА

1. McGraw G. Software Security: Building Security In. Boston : Addison-Wesley Professional, 2006. 448 p. ISBN 978-0-321-35670-3. URL: https://www.oreilly.com/library/view/software-security-building/0321356705/
2. Howard M., Lipner S. The Security Development Lifecycle. Redmond : Microsoft Press, 2006. 352 p. ISBN 978-0-7356-2214-2.
3. Stallings W., Brown L. Computer Security: Principles and Practice. 5th ed. Hoboken : Pearson, 2023. 864 p. ISBN 978-1-292-43060-6.
4. Shostack A. Threat Modeling: Designing for Security. Indianapolis *:* Wiley, 2014. 624 p. ISBN 978-1-118-80999-0.
5. Souppaya M., Scarfone K., Dodson D. Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities (NIST SP 800-218). Gaithersburg : NIST, 2022. 25 p. DOI: 10.6028/NIST.SP.800-218. URL: https://csrc.nist.gov/pubs/sp/800/218/final
6. Open Worldwide Application Security Project (OWASP). OWASP Top 10:2021. The Ten Most Critical Web Application Security Risks. 2021. URL: https://owasp.org/Top10/2021/ (дата звернення: 03.06.2026).
7. European Union Agency for Cybersecurity (ENISA). ENISA Threat Landscape 2024. Heraklion : ENISA, 2024. 132 p. URL: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2024
8. Kim D., Solomon M. Fundamentals of Information Systems Security. *4th* ed. Burlington : Jones & Bartlett Learning, 2022. 850 p. ISBN 978-1-284-22430-5.
9. Кодекс цивільного захисту України : Закон України від 02.10.2012 № 5403-VI. URL: https://zakon.rada.gov.ua/laws/show/5403-17
10. Про основні засади забезпечення кібербезпеки України : Закон України від 05.10.2017 № 2163-VIII. URL: https://zakon.rada.gov.ua/laws/show/2163-19
11. Про захист інформації в інформаційно-комунікаційних системах : Закон України від 05.07.1994 № 80/94-ВР. URL: https://zakon.rada.gov.ua/laws/show/80/94-%D0%B2%D1%80

12. НД ТЗІ 3.7-003-2005. Порядок проведення робіт зі створення комплексної системи захисту інформації в інформаційно-комунікаційній системі (у редакції зі змінами, затвердженими наказом Адміністрації Держспецзв'язку від 28.10.2023 № 924). Київ : Держспецзв'язку, 2023.
13. ДСТУ ISO/IEC 27001:2023. Інформаційна безпека, кібербезпека та захист конфіденційності. Системи управління інформаційною безпекою. Вимоги. Київ : ДП «УкрНДНЦ», 2023.
14. ISO/IEC 27034-1:2011. Information technology – Security techniques – Application security – Part 1: Overview and concepts. Geneva : International Organization for Standardization, 2011.
15. ISO/IEC 27005:2022. Information security, cybersecurity and privacy protection – Guidance on managing information security risks. Geneva : International Organization for Standardization, 2022. 92 p.
16. National Institute of Standards and Technology. Guide for Conducting Risk Assessments (SP 800-30 Rev. 1). Gaithersburg : NIST, 2012. DOI: 10.6028/NIST.SP.800-30r1. URL: https://csrc.nist.gov/pubs/sp/800/30/r1/final
17. Eilam E. Reversing: Secrets of Reverse Engineering. Indianapolis : Wiley Publishing, 2005. 624 p. ISBN 978-0-7645-7481-8.
18. Sikorski M., Honig A. Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. San Francisco : No Starch Press, 2012. 800 p. ISBN 978-1-59327-290-6.
19. Dang T., Gazet A., Bachaalany E., Josse S. Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools and Obfuscation. Indianapolis : Wiley, 2014. 384 p. ISBN 978-1-118-78731-1.
20. Kim G., Humble J., Debois P., Willis J., Forsgren N. The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations. 2nd ed. Portland : IT Revolution Press, 2021. 528 p. ISBN 978-1950508402.

21. Saltzer J. H., Schroeder M. D. The Protection of Information in Computer Systems // Proceedings of the IEEE. 1975. Vol. 63, No. 9. P. 1278–1308. DOI: 10.1109/PROC.1975.9939
22. OWASP Foundation. Mobile Application Security Testing Guide: MASTG-KNOW-0118 Runtime Application Self-Protection. URL: https://mas.owasp.org/MASTG-KNOW-0118/
23. Bejtlich R. The Practice of Network Security Monitoring: Understanding Incident Detection and Response. San Francisco : No Starch Press, 2013. 376 p. ISBN 978-1-59327-509-9.
24. National Telecommunications and Information Administration (NTIA). The Minimum Elements for a Software Bill of Materials (SBOM). Washington : U.S. Department of Commerce, 2021. 14 p. URL: https://www.ntia.gov/report/2021/minimum-elements-software-bill-materials-sbom
25. National Institute of Standards and Technology. Secure Software Development Framework (SSDF): Attestation Form. Gaithersburg : NIST, 2024. URL: https://csrc.nist.gov/projects/ssdf
26. Yosifovich P., Ionescu A., Russinovich M., Solomon D. Windows Internals, Part 1: System Architecture, Processes, Threads, Memory Management, and More. 7th ed. Redmond : Microsoft Press, 2017. 800 p. ISBN 978-0-7356-8418-8.
27. Ferraiolo D. F., Kuhn D. R., Chandramouli R. Role-Based Access Control. 2nd ed. Boston : Artech House, 2007. 312 p. ISBN 978-1-59693-113-8.
28. National Institute of Standards and Technology. Digital Identity Guidelines (SP 800-63-4). Gaithersburg : NIST, 2025. URL: https://pages.nist.gov/800-63-4/
29. Rosenblatt B. Digital Rights Management: Business and Technology. Indianapolis : Wiley Publishing, 2002. 432 p. ISBN 978-0764548895.
30. Каплун В. А., Дмитришин О. В., Баришев Ю. В. Захист програмного забезпечення. Частина 2 : навчальний посібник. Вінниця : ВНТУ, 2014. 105 с. URL: https://pdf.lib.vntu.edu.ua/books/2024/Kaplun_P2_2014_105.pdf

31. National Institute of Standards and Technology. Security and Privacy Controls for Information Systems and Organizations (NIST SP 800-53 Rev. 5). Gaithersburg: NIST, 2020. 492 p. DOI: 10.6028/NIST.SP.800-53r5. URL: https://csrc.nist.gov/pubs/sp/800/53/r5/final
32. National Institute of Standards and Technology. Zero Trust Architecture (NIST SP 800-207). Gaithersburg: NIST, 2020. 59 p. DOI: 10.6028/NIST.SP.800-207. URL: https://csrc.nist.gov/pubs/sp/800/207/final
33. Collberg C., Nagra J. Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Upper Saddle River : Addison-Wesley Professional, 2009. 608 p. ISBN 978-0-321-54925-9.
34. International Organization for Standardization. ISO 31000:2018. Risk Management – Guidelines. Geneva : ISO, 2018. 16 p. URL: https://www.iso.org/standard/65694.html
35. Common Weakness Enumeration (CWE). CWE Top 25 Most Dangerous Software Weaknesses. MITRE Corporation, 2024. URL: https://cwe.mitre.org/top25/ .
36. National Institute of Standards and Technology. Guide to Software and Systems Vulnerability Analysis (NIST IR 8397). Gaithersburg : NIST, 2022. DOI: 10.6028/NIST.IR.8397. URL: https://csrc.nist.gov/publications/detail/nistir/8397/final
37. Erickson J. Hacking: The Art of Exploitation. 2nd ed. San Francisco : No Starch Press, 2008. 488 p. ISBN 978-1-59327-144-2.
38. Skoudis E., Liston T. Malware: Fighting Malicious Code. Upper Saddle River : Prentice Hall PTR, 2003. 624 p. ISBN 978-0131014053.
39. Szor P. The Art of Computer Virus Research and Defense. Boston : Addison-Wesley Professional, 2005. 744 p. ISBN 978-0321304544.
40. Hoglund G., McGraw G. Exploiting Software: How to Break Code. Boston : Addison-Wesley Professional, 2004. 512 p. ISBN 978-0201786958.
41. HHD Software Ltd. Free Hex Editor Neo. URL: https://freehexeditorneo.com/

42. Microsoft Corporation. PE Format. Portable Executable and Common Object File Format Specification. Redmond : Microsoft, 2025. URL: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format
43. Mares M. CFF Explorer and Explorer Suite Documentation. NTCore. URL: https://ntcore.com/exsuite.php
44. Eagle C., Nance K. The Ghidra Book: The Definitive Guide. San Francisco : No Starch Press, 2020. 608 p. ISBN 978-1-71850-102-7.
45. National Security Agency (NSA). Ghidra Software Reverse Engineering Framework. 2025. URL: https://ghidra-sre.org
46. VirusTotal. YARA Documentation. 2025. URL: https://yara.readthedocs.io
47. Hex-Rays SA. IDA Pro User Guide. Liège : Hex-Rays, 2025. URL: https://docs.hex-rays.com
48. Noonan C., Barnum S. Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly. San Francisco : No Starch Press, 2019. 504 p. ISBN 978-1-59327-912-7.
49. NIST. Guide to Malware Incident Prevention and Handling for Desktops and Laptops (SP 800-83 Rev.1). Gaithersburg : NIST, 2013. 153 p. DOI: 10.6028/NIST.SP.800-83r1. URL: https://csrc.nist.gov/pubs/sp/800/83/r1/final
50. Pilon M. x64dbg Documentation. x64dbg Project, 2025. URL: https://help.x64dbg.com
51. Microsoft Corporation. Windows Debugging Tools (WinDbg) Documentation. Redmond: Microsoft, 2025. URL: https://learn.microsoft.com/windows-hardware/drivers/debugger/
52. Sysinternals. Process Monitor Documentation. Microsoft, 2025. URL: https://learn.microsoft.com/sysinternals/downloads/procmon
53. Combs G. Wireshark User's Guide. Wireshark Foundation, 2025. URL: https://www.wireshark.org/docs/wsug_html_chunked/
54. Oleavr. Frida Dynamic Instrumentation Toolkit Documentation. 2025. URL: https://frida.re/docs/home/
55. Cuckoo Foundation. Cuckoo Sandbox Documentation. 2025. URL:

https://cuckoo.readthedocs.io

56. 0xd4d. dnSpy Documentation and Source Code. GitHub Project, 2025. URL: https://github.com/dnSpyEx/dnSpy
57. Gagnon M., Taylor S., Ghosh A. *Software Protection through Anti-Debugging*. IEEE Security & Privacy. 2004. Vol. 2, No. 3. P. 82–84. URL: https://www.researchgate.net/publication/3437933_Software_Protection_through_Anti-Debugging
58. Schrittwieser S., Katzenbeisser S., Kinder J., Merzdovnik G., Weippl E. *Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis?* ACM Computing Surveys. 2016. Vol. 49, No. 1. Article 4. URL^ https://eprints.cs.univie.ac.at/6372/1/ProtectingSoftwareThroughObfuscation.pdf .
59. Abrath B., Coppens B., Nevolin I., De Sutter B. *Resilient Self-Debugging Software Protection*. 2020. URL: https://arxiv.org/abs/2004.06417
60. ETSI EN 300 392-7 V3.5.1. Terrestrial Trunked Radio (TETRA); Voice plus Data (V+D); Part 7: Security. Sophia Antipolis : European Telecommunications Standards Institute, 2019. 112 p. URL: https://www.etsi.org/deliver/etsi_en/300300_300399/30039207/03.05.01_60/en_30039207v030501p.pdf
61. Rahman M. M., Islam M. S. Designing of Real Time Voice Scrambler/Descrambler Based on Microcontroller. International Journal of Engineering Research and Technology. 2020. 8 p. URL: https://repository.sustech.edu/jspui/bitstream/123456789/26343/1/DesigningofReal%20%20.....pdf?utm_source=chatgpt.com
62. Stallings W. Cryptography and Network Security: Principles and Practice. 8th ed. Hoboken : Pearson, 2020. 880 p. ISBN 978-0-13-670722-6
63. Schneier B. Applied Cryptography: Protocols, Algorithms and Source Code in C. 20th Anniversary Edition. Indianapolis : Wiley, 2015. 784 p. ISBN 978-1119096726.
64. Menezes A., van Oorschot P., Vanstone S. Handbook of Applied Cryptography. Boca Raton : CRC Press, 1996. 816 p. URL: https://cacr.uwaterloo.ca/hac/

65.Hamsternz. *Experiments with self-synchronizing LFSR scramblers* [Electronic resource]. Available at: https://github.com/hamsternz/lfsr_scramblers

66. Linear-feedback shift register, https://en.wikipedia.org/wiki/Linear-feedback_shift_register

67. Manda Lurina, Sugondo Hadiyoso, Rina Pudji Astuti. Scrambling and De-Scrambling Implementation Using Linear Feedback Shift Register Method on FPGA. International Journal of Applied Information Technology, 2017. DOI: 10.25124/ijait.v1i02.876

68. Anderson R. Security engineering: a guide to building dependable distributed systems. 3rd ed. Indianapolis : Wiley, 2020. 1248 p. URL: https://www.cl.cam.ac.uk/~rja14/book.html (дата звернення: 03.06.2026).

69. National Institute of Standards and Technology. Guide to operational technology (ot) security : nist special publication 800-82 revision 3. Gaithersburg : NIST, 2024. 408 p. DOI: 10.6028/NIST.SP.800-82r3. URL: https://doi.org/10.6028/NIST.SP.800-82r3 (дата звернення: 03.06.2026).

70. National Institute of Standards and Technology. Hardware-enabled security: enabling a layered approach to platform security with trusted platform modules (TPMS) : NIST IR 8320. Gaithersburg : NIST, 2020. 52 p. DOI: 10.6028/NIST.IR.8320. URL: https://doi.org/10.6028/NIST.IR.8320 (дата звернення: 03.06.2026).

71. Trusted Computing Group. TPM 2.0 Library specification. Family 2.0. Level 00. Revision 01.59. Beaverton : Trusted Computing Group, 2019. 1300 p. URL: https://trustedcomputinggroup.org/resource/tpm-library-specification/ (дата звернення: 03.06.2026).

72. European Union Agency for Cybersecurity (ENISA). Good practices for cyber crisis management. Athens : ENISA, 2023. 79 p. URL: https://www.enisa.europa.eu/publications (дата звернення: 03.06.2026).

73. Yeluri R., Castro-Leon E. Building the infrastructure for cloud security: a solutions view. Berkeley, CA : Apress, 2014. 244 p. ISBN 978-1-4302-6145-2. DOI:

10.1007/978-1-4302-6146-9. URL: https://link.springer.com/book/10.1007/978-1-4302-6146-9.

74. Kizza J. M. Guide to Computer Network Security. 5th ed. Cham : Springer, 2020. 595 p. DOI: 10.1007/978-3-030-38141-7. URL: https://link.springer.com/book/10.1007/978-3-030-38141-7
75. Boyes H., Isbell R., Watson T. Cyber Security of Operational Technology. *Stevenage* : Institution of Engineering and Technology, 2022. 336 p. ISBN 978-1-83953-315-0.
76. National Security Agency. Server Security Guidance. Fort Meade, MD : NSA, 2024. 51 p. URL: https://www.nsa.gov/press-room/cybersecurity-advisories-guidance/
77. ENISA. Guidelines for securing the internet of things. Athens : European Union Agency for Cybersecurity, 2020. 94 p. URL: https://www.enisa.europa.eu/publications/guidelines-for-securing-the-internet-of-things
78. 3GPP TS 23.041. Technical realization of Cell Broadcast Service (CBS). Release 18. Valbonne, France : 3rd Generation Partnership Project, 2024, URL:https://www.etsi.org/deliver/etsi_ts/123000_123099/123041/18.06.00_60/ts_123041v180600p.pdf
79. International Atomic Energy Agency. Design of Radiation Monitoring Systems for Nuclear Facilities. Vienna : IAEA, 2023. 126 p., UrL: https://www.iaea.org/publications/search
80. Griesmayr Business Consulting GmbH. Privacylite voice – secure voice communication. URL:https://www.griesmayr.de/en/products/privacylitevoice/
81. Motorola Solutions. APX P25 two-way radio cybersecurity solutions. Schaumburg : Motorola Solutions, Inc. URL:https://www.motorolasolutions.com/en_us/products/project-25-systems/project-25-radios/p25-two-way-radio-cybersecurity.html

********** НЕ РОЗСТАВЛЕНО посилання**********

82. CRYPTOGRAPHIC ANALYSIS OF RIJNDAEL LIKE CHIPHER Cryptographic analysis of rijndael like chipher Scientific and Practical Cyber Security Journal (SPCSJ) 2(3): 42-46 ISSN 2587-4667 Scientific Cyber Security Association *(SCSA), № 3 (02), September 2018. – P. 42-46 Kovalov S., Maslova N., Psel V, URL:* https://journal.scsa.ge/papers/cryptographic-analysis-of-rijndael-like-cipher/

83. Maslova, N. (2023). Абстракції зворотного інжинірингу при контролі та побудові захищених програмних реалізацій. *Фізико-математичне моделювання та інформаційні технології*, - 2023, (37), 12-16. URL https://www.fmmit.lviv.ua/index.php/fmmit/article/view/296

84. ТКАЧУК Р. Л., МАСЛОВА Н. О., ІВАНУСА А. І. Основи інформаційної безпеки та кіберзахисту: Навчальний посібник. – Львів: Львівський державний університет безпеки життєдіяльності, 2026. – 271 с. – ISBN 978-617-8654-19-1 https://sci.ldubgd.edu.ua/handle/123456789/17461

85. Yashchuk, V., Ivanusa, A., Maslova, N., Tkachuk, R., & Brych, T. (2025). Концептуалізація інтегративного використання баз даних вразливостей у контексті системного менеджменту інформаційної безпеки. *Вісник Львівського державного університету безпеки життєдіяльності*, Т.*31*, 126-139. https://doi.org/10.32447/20784643.31.2025.13

86. Кутник Н., Маслова Н. Застосування віртуального середовища для дослідження шкідливого програмного забезпечення // Інформаційна безпека та інформаційні технології: збірник тез доповідей VI Всеукраїнської науково-практичної конференції молодих учених, студентів і курсантів, м. Львів, 30 листопада 2023 року. Львів, ЛДУ БЖД, 2023, 489 с. , с.104-106, https://indico.ldubgd.edu.ua/event/42/attachments/512/720/Конференція%20ІБІТ%20%28збірник%29%20-%202023.pdf, https://indico.ldubgd.edu.ua/event/42/

87. МАТЕМАТИЧНІ МЕТОДИ КРИПТОГРАФІЇ Ковальчук Л.В., Маслова Н.О. Математичні методи криптографії. Електронний навчальний посібник. –

Дрогобич: ДВНЗ «ДонНТУ», 2024. – 146с. (7 у.д.а), ISBN 978-966-377-258-5, DOI: https://doi.org/10.5281/zenodo.14655234

88 Балацька В.С., Ткачук Р.Л., Івануса А.І., Ящук В.І., Маслова Н.О. Інтеграція технологій блокчейн у комплексні системи захисту інформації для підвищення безпеки державних реєстрів. Вісник ЛДУБЖД, No32, 2025. С. 90-99. DOI: https://doi.org/10.32447/20784643.32.2025.07

89. МАСЛОВА Н. О., ІВАНУСА А. І., БАЛАЦЬКА В. С., НИКОЛАЙЧУК М. І. Оцінювання захищеності програмних систем цивільного захисту з урахуванням пріоритету доступності // *X Міжнародна науково-практична конференція «Міжнародна та національна безпека: теоретичні і прикладні аспекти»* (16 березня 2026 р., м. Дніпро). Дніпро: Дніпропетровський державний університет внутрішніх справ, 2026. С. XX–XX